```
(* Last modified on Wed Jan 14 09:02:17 1987 by mcjones *)

(* Base definition module for per-process state of Firefly Topaz. *)

SAFE DEFINITION MODULE Ux;

IMPORT NubTypes, OS, System, Text, User;

TYPE
  UxProT = REF; (* process control *)
  UxAST = REF; (* address space *)
  UxWST = REF; (* window system *)
  UxFileT = REF; (* file system *)
  UxTimeT = REF; (* interval timer *)

CONST
  SuperUID = 0; (* user ID of 'super user' *)
  NullPID = 0; (* default PID *)
  InitPID = 1; (* PID of the Init process *)

TYPE
  T = REF TObject;
  RefText = REF Text.T;
  TObject = RECORD
    p: UxProT;
    a: UxAST;
    w: UxWST;
    f: UxFileT;
    i: UxTimeT; (* NIL until first call to getitimer/setitimer *)
    (* The following are immutable in Topaz programs. In Ultrix programs they
       may be updated only by the watcher thread. *)
    pid: OS.PID; (* process id *)
    pgrp: OS.PGRP; (* process id of process group leader *)
    space: NubTypes.Space; (* address space *)
    euser, ruser: User.T; (* effective and real user *)
    password: RefText; (* password (inside "courtesy privacy screen") *)
  END;

VAR
  uTaos: T;
  map: ARRAY NubTypes.Space OF T;

TYPE
  ArgList = ARRAY [0 .. 6] OF UNSIGNED;
  EntryProc = PROCEDURE(T, VAR ArgList): INTEGER;
  (* Such procedures are not expected to raise exceptions other than
     OS.Error, OS.Failure, UxPro.Error, and Thread.Alerted *)

PROCEDURE Init() RAISES {};
(* Must be called exactly once, before calling any procedure or referencing
   any variable in the Ultrix emulator. *)

END Ux.
```

```
(* Last modified on Tue Jan 27 18:14:35 1987 by mcjones *)

SAFE DEFINITION MODULE UxAS;

IMPORT
  NubTypes, OS, OSSpecial, SBuf, System, Text, Thread, Time,
  TPFriends, TPSpecial, User, Ux, UxTypes;

TYPE
  T = Ux.UxAST;

  EventKind =
    (KernelCall,      (* kernel call from process *)
     Exception,       (* synchronous error within process *)
     Alert,           (* asynchronous alert from another process *)
     HandlerReturn); (* signal handler returned *)

  Event = RECORD
    CASE kind: EventKind OF
    | KernelCall: entry: UxTypes.KernelEntry; argList: Ux.ArgList;
    | Exception: sig: OS.Signal; code: INTEGER;
    | Alert:
    | HandlerReturn: mask: OS.Signals; onStack: BOOLEAN;
    END;
  END;

  Addr = UNSIGNED; (* client-space address (trick Loupe) *)

CONST
  AlertPriority = TPFriends.PriIOLow;

PROCEDURE AlertHandler(u: Ux.T) RAISES {};
(* ONLY FOR OLD OS *)
(* Sends an alert to the handler thread specified by a prior call
   'SetHandler(u)'. *)

PROCEDURE BackupKernelCall(
    u: Ux.T;
    entry: UxTypes.KernelEntry)
  RAISES {SBuf.Fault};
(* Backs up the program counter of the current thread of 'u' so it will
   reexecute the current kernel call, 'entry', when restarted. *)

PROCEDURE Copy(u, uNew: Ux.T; vFork: BOOLEAN) RAISES {SBuf.Fault};
(* Callable by watcher thread only. *)
(* Assigns an actual VM.Space to 'uNew'. (If 'vFork=TRUE', this is just the
   space of 'u'; otherwise it is a newly allocated space. Copies the
   registers, and, if 'vFork=FALSE', the contents of the mapped pages,
   including write-protect status, from 'u' to 'uNew'. Assumes no threads are
   running in the address space 'u'. *)

PROCEDURE Dispose(u: Ux.T; freeSpace: BOOLEAN) RAISES {};
(* Destroys all resources, including threads and virtual memory (unless
   'freeSpace = FALSE'), associated with the address space of process 'u'.
   Assumes all threads are already stopped. *)

PROCEDURE Get(
    u: Ux.T;
    a: Addr;
    VAR v: ARRAY OF System.Byte;
    offset: CARDINAL := 0;
    count: CARDINAL := System.MaxCard)
  RAISES {SBuf.Fault};
(* 'a' is interpreted as an address in the address space of 'u'. 'v' is
   filled, starting at 'v[offset]', with 'MIN(count, NUMBER(v)-offset)' bytes
   read starting at 'a'. Raises SBuf.Fault if an address fault is encountered.
*)

PROCEDURE GetArrayOfText(
    u: Ux.T;
    a: Addr)
    : Text.RefArray
  RAISES {SBuf.Fault};
(* 'a' is interpreted as an address in the address space of 'u'. The pointers
   starting at 'a' and terminated by a nil pointer are interpreted as pointing
   to null-terminated character strings. A REF to an open array of Text.T's is
   returned containing equal strings (without the terminating null characters.
   Raises SBuf.Fault if there is an address fault. *)

PROCEDURE GetClientPriority(u: Ux.T): TPFriends.Priority RAISES {};
(* Return highest priority of all threads in address space of 'u'. *)

PROCEDURE GetClientTime(
    u: Ux.T;
    alreadyStopped: BOOLEAN := FALSE)
    : Time.T
  RAISES {};
(* Returns the amount of cpu time used so far by all client threads in address
   space of 'u'. Temporarily stops all threads in the address space, unless
   'alreadyStopped=TRUE'. *)

PROCEDURE GetInfo(
    u: Ux.T;
    VAR (*out *) mappedBytes, residentBytes: CARDINAL)
  RAISES {};
(* Returns information about address space of 'u': approximations to number of
   mapped bytes and number of resident bytes. *)

PROCEDURE GetRLimit(
    u: Ux.T;
    resource: UxTypes.Resource;
    VAR (*out*) rLimit: UxTypes.RLimit)
  RAISES {};
(* Returns the current resource limit values for address space of 'u' when
   'resource' is one of UxTypes.rLimitData or UxTypes.rLimitStack. No-ops for
   other values of 'resource'. *)

PROCEDURE GetText(u: Ux.T; a: Addr): Text.T RAISES {SBuf.Fault};
(* Returns a Text with the contents of the null-terminated character string
   starting at the given location 'a' within the address space of 'u'. Raises
   SBuf.Fault if an address fault is encountered. *)

PROCEDURE HideThread(u: Ux.T) RAISES {};
(* Callable by watcher thread only. *)
(* Hides client thread of address space 'u' in such a way that it will not be
   affected by StopAll, StartAll, and Release applied to any Ux.T sharing the
   same VM.Space. RestoreThread should be the next operation applied to the
   address space of 'u'. Intended to be used for parent of VFork kernel call,
   since parent's address space is shared by child. Assumes only one thread in
   'u'. *)

PROCEDURE Init() RAISES {};
(* Initializes UxAS. To be called once, before any other procedure in this
   interface. *)

PROCEDURE IsTopaz(u: Ux.T): BOOLEAN RAISES {};
(* Checks whether address space of 'u' is a Topaz address space, i.e. whether
   it contains more than one thread and/or it has used RPC. CURRENTLY ONLY
```

```
   CHECKS NUMBER OF THREADS. *)

PROCEDURE Load(
    u: Ux.T;
    file: OS.File;
    relDir: OS.Dir;
    euser: User.T;
    name: Text.T;
    argv, envp: Text.RefArray;
    uArgs: Ux.T;
    argva, envpa: Addr;
    vFork: BOOLEAN) (* if true, must create new space *)
    : BOOLEAN
  RAISES {OS.Error, SBuf.Fault, Thread.Alerted};
(* Arranges for 'u' to execute 'file', which is either an a.out file or a
   script file whose first line names--relative to 'relDir' and authorized by
   'euser'--the a.out file of an interpreter. Load always closes 'file'; if it
   must open an additional file then it closes that one too. The arguments and
   environment strings passed to the new program come from 'argv' and 'envp'
   if 'argv' is non-nil, else are fetched from address space of 'uArgs'
   (possibly 'uArgs = u') using addresses 'argva' and 'envpa' respectively.
   'name' is only used as part of the argument list if the original file is an
   interpreter script. Assumes the current thread of 'u' is the only thread,
   and is stopped. Leaves the address space and thread in a state such that
   WaitEvent will start it executing the new program. If any problems are
   encountered before the address space is disturbed, OS.Error or SBuf.Fault
   is raised. After that point, any errors are reflected by returning FALSE,
   or raising Thread.Alerted. Success is reflected by returning TRUE. *)

PROCEDURE MakeCoreImage(
    u: Ux.T;
    sig: OS.Signal)
  RAISES {OS.Error, OS.Failure, Thread.Alerted};
(* Writes out a file named 'core' in the working directory of 'u' containing
   an image of its memory (data, stack, and registers for all threads) and
   terminating signal for use by the Loupe debugger. See UxCoreFile.def for
   format--not guaranteed compatible with adb and dbx. Assumes no threads are
   running in the address space of 'u'. *)

PROCEDURE New(u, uNew: Ux.T) RAISES {};
(* Initializes the 'a' and 'space' fields of 'uNew^'. If 'u=NIL', it is
   assumed that the process descriptor for the root itself is being created,
   otherwise certain fields are copied from u^.a^. Before the new space can be
   started (by calling WaitEvent), Copy or Load must be called to assign and
   fill in the address space and initial thread. *)

PROCEDURE PushHandlerCall(
    u: Ux.T;
    signal: OS.Signal;
    code: INTEGER;
    handler: UxTypes.SignalHandler;
    mask: OS.Signals;
    onStack: UxTypes.SignalOnStack;
    switchStacks: BOOLEAN;
    sp: Addr)
  RAISES {SBuf.Fault};
(* Callable by watcher thread only. *)
(* Pushes context onto stack of current thread of address space of 'u' so that
   it is ready to execute 'handler' with arguments 'signal', 'code', and
   'scp', where 'scp' is a pointer to a UxTypes.SigContext record fabricated
   by PushHandlerCall. 'mask' and 'onStack' are saved on the stack of 'u' in
   such a way that they will be returned by a subsequent HandlerReturn
   returned by WaitEvent. If 'switchStacks' is TRUE, causes handler to execute
   on signal stack 'sp'. Raises SBuf.Fault if pushing context would extend
   stack beyond rlimit. *)

PROCEDURE Put(
    u: Ux.T;
    a: Addr;
    VAR v: ARRAY OF System.Byte;
    offset: CARDINAL := 0;
    count: CARDINAL := System.MaxCard)
  RAISES {SBuf.Fault};
(* Inverse of 'Get'. *)

PROCEDURE PutText(
    u: Ux.T;
    a: Addr;
    t: Text.T;
    count: CARDINAL := System.MaxCard)
    : CARDINAL
  RAISES {SBuf.Fault};
(* 'a' is interpreted as an address in the address space of process 'u'. The
   bytes of 't' are written into a buffer starting at 'a' and continuing for
   at most 'count' bytes. NOTE: No null termination is supplied. Returns the
   number of characters actually written, which is the minimum of 'count' and
   'Text.Length(t)'. Raises SBuf.Fault if there is an address fault. *)

PROCEDURE R0Result(
    u: Ux.T;
    result: INTEGER;
    error: BOOLEAN := FALSE)
  RAISES {};
(* Callable by watcher thread only. *)
(* Sets register R0 in the current thread of address space of 'u' to 'result',
   which is either a return value or error code. In the latter case, either
   'error' should be TRUE, or a separate call to 'SetError' should be
   performed. *)

PROCEDURE R1Result(u: Ux.T; result: INTEGER) RAISES {};
(* Callable by watcher thread only. *)
(* Causes register R1 to contain 'result' when this kernel call returns to the
   application address space. Used for returning second result of kernel
   entries like 'pipe'. *)

PROCEDURE RestoreThread(u: Ux.T) RAISES {};
(* Callable by watcher thread only. *)
(* Restores thread of 'u' to the VM.Space associated with 'u'. Used to reverse
   the effect of a preceding HideThread (for the parent of a VFork). *)

PROCEDURE SetClientPriority(u: Ux.T; priority: TPFriends.Priority) RAISES {};
(* Callable from watcher thread only. *)
(* Sets priority of all threads in address space of 'u'. *)

PROCEDURE SetError(u: Ux.T) RAISES {};
(* Callable from watcher thread only. *)
(* Sets error flag of 'u', which causes cause code following the kernel call
   to set the global variable 'errno' (when using the C library) or raise an
   exception (when using the Topaz library). *)

PROCEDURE SetHandler(u: Ux.T) RAISES {OS.Error};
(* ONLY FOR OLD OS *)
(* If handler thread of 'u' is currently null, sets it to current thread, else
   if handler thread is different than current thread raises
   OS.Error(OS.InvalidArgumentEC). Should only be called directly or
   indirectly from a kernel entry procedure. See AlertHandler. *)

PROCEDURE SetRLimit(
```

```
    u: Ux.T;
    resource: UxTypes.Resource;
    rLimit: UxTypes.RLimit)
  RAISES {};
(* Sets the limit for 'resource' within the address space of 'u'. No-ops
   unless 'resource' is UxTypes.rlimitData or UxTypes.rLimitStack. Assumes
   checking ('cur' <= 'max', etc.) done by caller, but silently enforces that
   limits are less than 'maxSegSize'. *)

PROCEDURE SetSTrace(u: Ux.T; strace: BOOLEAN) RAISES {};
(* Sets the strace bit of the process 'u' to the specified value. If
   'strace=TRUE' when the next call to Load succeeds, the process will wait to
   be debugger via UserTTD. *)

PROCEDURE SetThreadPriority(
    thread: Thread.T;
    priority: TPFriends.Priority)
  RAISES {};
(* Sets priority of specified thread, possibly self. *)

PROCEDURE StartAll(u: Ux.T) RAISES {};
(* Starts all threads in the address space of 'u'. To be used after a call to
   StopAll. *)

PROCEDURE StopAll(u: Ux.T) RAISES {};
(* Stops all threads in the address space of 'u'. *)

PROCEDURE WaitEvent(u: Ux.T; VAR (*out*) event: Event) RAISES {SBuf.Fault};
(* Callable by watcher thread only. *)
(* Starts the current thread of 'u', i.e. the one reported by the result of
   the last call to WaitEvent, or the initial thread after a CopySpace or
   Load. Waits for the next interesting event from some thread in 'u'. Returns
   in 'event' the description of what happened. Could raise SBuf.Fault if it
   has trouble fetching the arguments to a kernel call. *)


END UxAS.
```

```
(* Last modified on Thu Jul 24 10:19:02 1986 by mcjones *)

DEFINITION MODULE UxCoreFile;
IMPORT System, ThreadsPort, TPSpecial, Unix42;

(* These declarations define the variant of core files interpreted by the
   Loupe debugger for the Topaz Ultrix emulator.

   The structure is based on the information in the core(5) manpage and the
   <sys/user.h> file, and has been tested to some extent with adb (but not
   with dbx). *)

TYPE
  Byte = System.Byte;

(* The layout of a core file is as follows. Note that page refers to a VAX
   page of 512 bytes.

                      Length
                      ------
       Header         10 pages
       Data image     header.dataPages
       Stack image    header.stackPages
       Thread states  # of threads * System.TByteSize(ThreadMapEntry) bytes

*)

CONST (* offsets, lengths of header fields *)
  oPcbKsp = 0;
  lPcbKsp = 4;
  oPcbUsp = 12;
  lPcbUsp = 4;
  oStatus = 388;
  lStatus = 2;
  oTextPages = 536;
  lTextPages = 4;
  oDataPages = 540;
  lDataPages = 4;
  oStackPages = 544;
  lStackPages = 4;
  oAp = 5036;
  lAp = 4;
  oFp = 5040;
  lFp = 4;
  oRegs = 5048;
  lRegs = 48;
  oSp = 5100;
  lSp = 4;
  oPc = 5112;
  lPc = 4;
  oPsw = 5116;
  lPsw = 4;

(* For compatibility with possible future versions, unused fields in the
   header (xxNNN) contain zero. *)

TYPE
  Header = RECORD
    pcbKsp: UNSIGNED;
    xx4: ARRAY [oPcbKsp + lPcbKsp .. oPcbUsp - 1] OF Byte;
    pcbUsp: UNSIGNED;
    xx16: ARRAY [oPcbUsp + lPcbUsp .. oStatus - 1] OF Byte;
    status: Unix42.Short;
    xx390: ARRAY [oStatus + lStatus .. oTextPages - 1] OF Byte;
    textPages: CARDINAL;
    dataPages: CARDINAL;
    stackPages: CARDINAL;
    xx548: ARRAY [oStackPages + lStackPages .. oAp - 1] OF Byte;
    ap: UNSIGNED;
    fp: UNSIGNED;
    xx5044: ARRAY [oFp + lFp .. oRegs - 1] OF Byte;
    regs: ARRAY [0 .. 11] OF UNSIGNED;
    xx5096: ARRAY [oRegs + lRegs .. oSp - 1] OF Byte;
    sp: UNSIGNED;
    xx5104: ARRAY [oSp + lSp .. oPc - 1] OF Byte;
    pc: UNSIGNED;
    psw: UNSIGNED;
  END;

(* ASSERT( System.TByteSize( Header ) = 5120 *)

TYPE
  ThreadMapEntry = RECORD
    handle: ThreadsPort.Handle;
    state: TPSpecial.State;
  END;
  ThreadMap = ARRAY OF ThreadMapEntry;

END UxCoreFile.
```

```
(* Last modified on Wed Jan 14 08:55:50 1987 by mcjones *)
(*      modified on Wed Nov 19 00:10:06 1986 by swart *)
(* Convert OS errors into Ultrix errors. *)

SAFE DEFINITION MODULE UxError;

IMPORT OS, UxTypes;

TYPE
  SysError = UxTypes.SysError;

TYPE
  EC = OS.EC;
  FC = OS.FC;

VAR
  failTran: ARRAY FC OF SysError;
  errTran: ARRAY EC OF SysError;

END UxError.
```

```
(* Last modified on Wed Jan 21 11:17:44 1987 by mcjones *)
(*      modified on Mon Nov 24 21:52:02 1986 by swart *)
(*      modified on Tue Sep 17  9:31:00 1985 by mds *)

SAFE DEFINITION MODULE UxPro;

IMPORT
  OS, OSSpecial, Text, Thread, ThreadFriends, Time, User, Ux, UxTypes;

TYPE
  T = Ux.UxProT;

EXCEPTION
  Error(UxTypes.SysError);

PROCEDURE AcquireProcess(u: Ux.T) RAISES {};
(* A section of code that is to be atomic with respect to the execution of
   other process management operations should begin with AcquireProcess and
   end with ReleaseProcess (q.v.). *)

PROCEDURE AcquireProcessTemplate(
    t: OS.ProcessTemplate)
  RAISES {OS.Error};
(* A section of code that is to be atomic with respect to the execution of
   StartProcess should begin with AcquireProcessTemplate and end with
   ReleaseProcessTemplate (q.v.). Raises Error(InvalidArgumentEC) if the
   process template has already been passed to StartProcess. *)

PROCEDURE EnsureRunning(u: Ux.T) RAISES {Thread.Alerted};
(* Blocks if 'u' has been stopped. *)

PROCEDURE GetWTime(u: Ux.T): Time.T RAISES {};
(* Returns amount of cpu time spent so far by watcher thread for 'u'. *)

PROCEDURE Import() RAISES {};
(* Should be called exactly once, when it is permissible to import RPC
   interfaces used by the implementation of UxPro. Import forks a worker
   thread and returns immediately. *)

PROCEDURE Init() RAISES {};
(* Initializes UxPro. To be called once, before any other procedure in this
   interface. *)

PROCEDURE IsWatcher(u: Ux.T): BOOLEAN RAISES {};
(* Returns TRUE iff the calling thread is the watcher thread for process 'u',
   i.e. if the calling thread is working on a kernel call trap. *)

PROCEDURE PIDsPGRP(pid: OS.PID): OS.PID RAISES {OS.Error};
(* Returns the process group id of the process identified by 'pid'. Raises
   OS.Error(BadPIDEC) if process 'pid' does not exist. *)

PROCEDURE RaiseError(e: UxTypes.SysError) RAISES {Error};
(* Equivalent to RAISE(UxPro.Error, e). *)

PROCEDURE Register(entry: UxTypes.KernelEntry; proc: Ux.EntryProc) RAISES {};
(* Registers the procedure 'proc' as the implementation of the ultrix kernel
   entry 'entry'. See below. *)

PROCEDURE ReleaseProcess(u: Ux.T) RAISES {};
(* A section of code that is to be atomic with respect to the execution of
   other process management operations should begin with AcquireProcess (q.v.)
   and end with ReleaseProcess. *)

PROCEDURE ReleaseProcessTemplate(t: OS.ProcessTemplate) RAISES {};
(* A section of code that is to be atomic with respect to the execution of
   StartProcess should begin with AcquireProcessTemplate (q.v.) and end with
   ReleaseProcessTemplate. *)

PROCEDURE SetEUser(u: Ux.T; eUser: User.T) RAISES {};
(* Sets the effective user of the process 'u'. *)

PROCEDURE SigEffective(u: Ux.T; signal: OS.Signal): BOOLEAN RAISES {};
(* Returns TRUE iff for process 'u', 'signal' is not ignored nor masked, and
   if the id of the parent of process 'u' is not InitPID, and the process is
   not in the middle of a vfork/exec sequence. Intended for use by the tty
   driver on SigTTIn and SigTTOu. *)

PROCEDURE Signal(u: Ux.T; signal: OS.Signal) RAISES {};
(* Posts 'signal' for delivery to process 'u'. Intended for use by interval
   timer on SigAlrm, SigVTAlrm, and SigProf. *)


(*****************************************************)
(* Procedures exported through OSFriends via RemoteOS *)
(*****************************************************)

PROCEDURE CloseTemplate(u: Ux.T; t: OS.ProcessTemplate) RAISES {};

PROCEDURE GetProcessInfo(
    u: Ux.T;
    pid: OS.PID;
    VAR processInfoOUT: OS.ProcessInfo;
    getUser: BOOLEAN)
  RAISES {OS.Error};

PROCEDURE NewProcess(u: Ux.T): OS.ProcessTemplate RAISES {};

PROCEDURE NextProcess(u: Ux.T; pid: OS.PID): OS.PID RAISES {};

PROCEDURE SendSignal(
    u: Ux.T;
    pid: OS.PID;
    signal: OS.SignalOrNone;
    euser: User.T)
  RAISES {OS.Error};

PROCEDURE SendSignalToGroup(
    u: Ux.T;
    pgrp: OS.PGRP;
    signal: OS.SignalOrNone;
    euser: User.T)
  RAISES {OS.Error};

PROCEDURE SetMySignalState(
    u: Ux.T;
    signal: OS.Signal;
    state: OS.SignalState)
    : OS.SignalState
  RAISES {OS.Error};

PROCEDURE SetPGRP(
    u: Ux.T;
    t: OS.ProcessTemplate)
  RAISES {OS.Error};

PROCEDURE SetSignalState(
    u: Ux.T;
    t: OS.ProcessTemplate;
```

```
    signal: OS.Signal;
    state: OS.SignalState)
  RAISES {OS.Error};

PROCEDURE SetUser(
    u: Ux.T;
    t: OS.ProcessTemplate;
    effective, real: User.T;
    pswd: Text.T)
  RAISES {OS.Error};

PROCEDURE StartProcess(
    u: Ux.T;
    file: OS.File;
    relDir: OS.Dir;
    euser: User.T;
    name: Text.T;
    argv: Text.RefArray;
    t: OS.ProcessTemplate;
    relationship: OS.Relationship;
    env: Text.RefArray)
    : OS.PID
  RAISES {OS.Error, OS.Failure};

PROCEDURE WaitForChild(
    u: Ux.T;
    pid: OS.PID;
    VAR rUsageOUT: OS.RUsage)
    : OS.WStatus
  RAISES {OS.Error, Thread.Alerted};

PROCEDURE WaitForSignal(
    u: Ux.T;
    allowed: OS.Signals)
    : OS.Signal
  RAISES {OS.Error, Thread.Alerted};


(*************************************************)
(* Procedures exported thru OSFriends via RemoteOS *)
(*************************************************)

PROCEDURE GetPriority(
    u: Ux.T;
    pid: OS.PID)
    : ThreadFriends.Priority
  RAISES {Error};

PROCEDURE SetPriority(
    u: Ux.T;
    t: OS.ProcessTemplate;
    priority: ThreadFriends.Priority)
  RAISES {OS.Error};

PROCEDURE SetPriorityPID(
    u: Ux.T;
    pid: OS.PID;
    priority: ThreadFriends.Priority)
  RAISES {OS.Error};

PROCEDURE SetPriorityPGRP(
    u: Ux.T;
    pgrp: OS.PGRP;
    priority: ThreadFriends.Priority)
  RAISES {OS.Error};

PROCEDURE SetPriorityUser(
    u: Ux.T;
    userName: Text.T;
    priority: ThreadFriends.Priority)
  RAISES {OS.Error};

PROCEDURE SetMyUser(
    u: Ux.T;
    effective, real: User.T;
    pswd: Text.T)
  RAISES {OS.Error};


(*************************************************)
(* Procedures exported thru OSSpecial via RemoteOS *)
(*************************************************)

PROCEDURE Exit(
    u: Ux.T;
    status: OS.ExitStatus) RAISES {};

END UxPro.
```

M. D. Schroeder -- started June, 1985

IMPLEMENTING ULTRIX KERNEL ENTRIES

This module dispatches kernel calls from a application process to the procedures that implement them. A procedure that implements a kernel entry has type Ux.EntryProc, and is registered with UxPro as part of initialization. To prevent initialization confusion, the 'Init' procedures of all the modules of the Ultrix implementation are called from Ux.mod. These 'Init' procedures call back to UxPro.Register to report each kernel entry procedure. An attempt by the process to invoke a kernel entry with no registered procedure will cause the signal OS.SigSys to be posted to that process.

The first argument to an EntryProc allows access to the state information of the process. For a particular process, the procedures implementing kernel calls always are executed by the same "watcher thread".

The second argument to an EntryProc allows access to the arguments of the kernel entry. An Ux.ArgList is an array containing the argument values provided by the caller in the application. ArgList elements not provided by the caller are set to 0. Each EntryProc knows the number, position, type, and semantics of its arguments, as defined in Unix42.def and the Ultrix system documentation. The EntryProc reads the ArgList directly, applying LOOPHOLEs as necessary to produce the appropriate Modula types. Writes into ArgList are not copied back to the application address space. In cases where these arguments are to be treated as addresses in the application address space, the EntryProc apply procedures UxAS.Get, UxAS.Put, etc. to indirectly access the application address space. These procedures copy the addressed storage from/to the application address space and do the indicated type conversion.

The R0 result of a EntryProc is always returned as an INTEGER. If the result is some other type then a LOOPHOLE is used to return it. Kernel entries that produce no result return 0. The few kernel entries that produce a second result in R1 call UxAS.R1Result to provide this value.

When an EntryProc needs to produce an error return, it raises UxPro.Error. The procedure UxPro.RaiseError is useful for doing the raise, since it

constructs the exception's argument record. If the signal 's' in Error's argument record is OS.SigNone, then the error code 'e' will be returned as the result of the kernel call. If some signal value is given for 's', then the signal is synchronously delivered to the process. Following delivery the error code 'e' is returned as the result of the kernel call, unless 'e' is UxTypes.ok, in which case the kernel call is reexecuted.

An EntryProc is prepared for Thread.Alert(watcherThread) to be performed at any time. This alert will cause waits on CONDITIONs done with Thread.AlertWait to RAISE the exception Thread.Alerted. Such an alert means that an asynchronous signal is awaiting delivery to the application. If the alert should cause the kernel call to abort, then it is handled by the EntryProc and RaiseError (UxTypes.EINTR) called in its place. If the alert should cause the kernel call to be retried, then the alert is not handled: UxPro interprets the alert as an instruction to redo the kernel call after the signal is delivered to the application. A kernel entry procedure that needs to restart itself raises Thread.Alerted directly: this causes any posted, unmasked signals to be delivered and the kernel entry then to be recalled.

```
(* Last modified on Sun Dec 21 10:36:02 1986 by mcjones *)
(*      modified on Fri Oct 11 14:09:54 1985 by price *)
SAFE DEFINITION MODULE UxTime;

(* This module provides interval timer and time of day support to Ux. *)

IMPORT Ux;

PROCEDURE Init() RAISES {};
(* This procedure is called only once at Ux startup. *)

PROCEDURE Exit(u: Ux.T) RAISES {};
(* This procedure is called when an Ux process is terminated. It cancels any
   pending timers on u^.i. *)

END UxTime.
```

```
(* Last modified on Tue Jan 27 17:53:05 1987 by mcjones *)
(*      modified on Fri Jan 23 00:07:05 1987 by swart *)
(*      modified on Tue Dec 31 13:38:47 1985 by roberts *)
(*      modified on Fri Dec 13 11:44:00 1985 by price *)
(*      modified on Thu Jul 18 11:18:51 1985 by mds *)
(*      modified on Mon Jan  7 09:58:52 1985 by birrell *)
(*      modified on Sat Nov 24 14:40:41 1984 by stewart *)

(* Types and constants for the Ultrix(TM) 1.1 kernel call trap interface, with
   Taos and Ultrix 2.0 extensions.

   UxTypes.def was derived from the earlier Unix42.def by passing through
   relevant definitions from OS.def (e.g. Signal) and adding some additional
   kernel entry names from the old UnsafeOS.def.

   Added in extensions from Ultrix 2.0. These new types or enumeration
   elements are marked where they differ from Ultrix 1.1. The new kernel calls
   themselves have not been added yet. -- Garret Swart *)

DEFINITION MODULE UxTypes;

FROM System IMPORT Address, Byte, MaxInt, TByteSize;
IMPORT OS, Time, ThreadsPort, TPSpecial;

TYPE

(*************************************************************)
(*                                                           *)
(*    Types used by kernel calls.  Derived from "C" types    *)
(*    described in include files in /usr/include.            *)
(*    These are intended to be bitwise identical to "C" types.  *)
(*                                                           *)
(*************************************************************)


(*************************************************************)
(*                                                           *)
(*    SYScall numbers - for use with syscall(2)              *)
(*    From:  syscall.h                                       *)
(*                                                           *)
(*************************************************************)

  KernelEntry =
    (SYSspare0, SYSexit, SYSfork, SYSread, SYSwrite, SYSopen, SYSclose,
     SYSspare7, SYScreat, SYSlink, SYSunlink, SYSexecv, SYSchdir, SYSspare13,
     SYSmknod, SYSchmod, SYSchown, SYSsbreak (* was SYSspare17 *), SYSspare18,
     SYSlseek, SYSgetpid, SYSmount, SYSumount, SYSspare23, SYSgetuid,
     SYSspare25, SYSptrace, SYSspare27, SYSspare28, SYSspare29, SYSspare30,
     SYSspare31, SYSspare32, SYSaccess, SYSspare34, SYSspare35, SYSsync,
     SYSkill, SYSstat, SYSspare39, SYSlstat, SYSdup, SYSpipe, SYSspare43,
     SYSprofil, SYSspare45, SYSspare46, SYSgetgid, SYSspare48, SYSspare49,
     SYSspare50, SYSacct, SYSspare52, SYSspare53, SYSioctl, SYSreboot,
     SYSspare56, SYSsymlink, SYSreadlink, SYSexecve, SYSumask, SYSchroot,
     SYSfstat, SYSspare63, SYSgetpagesize, SYSmremap, SYSvfork
       (* was SYSspare66 *), SYSvread (* was SYSspare67 *), SYSvwrite
       (* was SYSspare68 *), SYSsbrk, SYSsstk, SYSmmap, SYSvadvise
       (* was SYSspare72 *), SYSmunmap, SYSmprotect, SYSmadvise, SYSvhangup,
     SYSspare77, SYSmincore, SYSgetgroups, SYSsetgroups, SYSgetpgrp,
     SYSsetpgrp, SYSsetitimer, SYSwait, SYSswapon, SYSgetitimer,
     SYSgethostname, SYSsethostname, SYSgetdtablesize, SYSdup2, SYSgetdopt,
     SYSfcntl, SYSselect, SYSsetdopt, SYSfsync, SYSsetpriority, SYSsocket,
     SYSconnect, SYSaccept, SYSgetpriority, SYSsend, SYSrecv, SYSspare103,
     SYSbind, SYSsetsockopt, SYSlisten, SYSspare107, SYSsigvec, SYSsigblock,
     SYSsigsetmask, SYSsigpause, SYSsigstack, SYSrecvmsg, SYSsendmsg,
     SYSspare115, SYSgettimeofday, SYSgetrusage, SYSgetsockopt, SYSresuba
       (* was SYSspare119 *), SYSreadv, SYSwritev, SYSsettimeofday, SYSfchown,
     SYSfchmod, SYSrecvfrom, SYSsetreuid, SYSsetregid, SYSrename, SYStruncate,
     SYSftruncate, SYSflock, SYSspare132, SYSsendto, SYSshutdown,
     SYSsocketpair, SYSmkdir, SYSrmdir, SYSutimes, SYShandlerreturn
       (* was SYSspare139 *), SYSrevoke, SYSgetpeername, SYSgethostid,
     SYSsethostid, SYSgetrlimit, SYSsetrlimit, SYSkillpg, SYSspare147,
     SYSsetquota, SYSquota, SYSgetsockname,

(* new with Unix 4.3 and/or Ultrix 2.0: *)

     SYSmsgctl, SYSmsgget, SYSmsgrcv, SYSmsgsnd, SYSsemctl, SYSsemget,
     SYSsemop, SYSuname, SYSshmsys, SYSplock, SYSlockf, SYSustat, SYSgetmnt,
     SYSgetdirents, SYSnfsbiod, SYSnfsgetfh, SYSnfssvc, SYSspare168,
     SYSgetdomainname, SYSsetdomainname);

(* SYScall numbers for Taos extensions. *)

CONST
  SYScopy = SYSspare7; (* Ux only *)
  SYSnsenumerate = SYSspare13; (* Ux only *)
  SYScheckpw = SYSspare18; (* Ux only *)
  SYSterminate = SYSspare63; (* Ux only *)
  SYSgetpw = SYSspare77; (* Ux only *)
  SYSdummy = SYSspare103; (* Reserved for Taos testing purposes *)
  SYSreservetraps = SYSspare107; (* Ux only *)
  SYSgetsignal = SYSspare115; (* Ux only *)
  SYSgetspaceinfo = SYSresuba; (* Ux only *) (* oops! *)
  SYSstartnewspace = SYSspare132; (* Ux only *)
  SYSstrace = SYSspare147; (* Ux only *)

(*************************************************************)
(*                                                           *)
(*    Return codes left in global variable "errno"           *)
(*    From:  errno.h                                         *)
(*                                                           *)
(*************************************************************)

TYPE
  SysError =
    (ok, EPERM, ENOENT, ESRCH, EINTR, EIO, ENXIO, E2BIG, ENOEXEC, EBADF,
     ECHILD, EAGAIN, ENOMEM, EACCES, EFAULT, ENOTBLK, EBUSY, EEXIST, EXDEV,
     ENODEV, ENOTDIR, EISDIR, EINVAL, ENFILE, EMFILE, ENOTTY, ETXTBSY, EFBIG,
     ENOSPC, ESPIPE, EROFS, EMLINK, EPIPE, EDOM, ERANGE, EWOULDBLOCK,
     EINPROGRESS, EALREADY, ENOTSOCK, EDESTADDRREQ, EMSGSIZE, EPROTOTYPE,
     ENOPROTOOPT, EPROTONOSUPPORT, ESOCKTNOSUPPORT, EOPNOTSUPP, EPFNOSUPPORT,
     EAFNOSUPPORT, EADDRINUSE, EADDRNOTAVAIL, ENETDOWN, ENETUNREACH,
     ENETRESET, ECONNABORTED, ECONNRESET, ENOBUFS, EISCONN, ENOTCONN,
     ESHUTDOWN, E59, ETIMEDOUT, ECONNREFUSED, ELOOP, ENAMETOOLONG, EHOSTDOWN,
     EHOSTUNREACH, ENOTEMPTY, EPROCLIM, EUSERS, EDQUOT,

(* New with 2.0 *)
     ESTALE, EREMOTE, ENOMSG, EIDRM, EALIGN);


(*************************************************************)
(*                                                           *)
(*    Basic data types                                       *)
(*    From:  sys/types.h                                     *)
(*                                                           *)
(*************************************************************)

(* From Unix.def *)
  CBuffer = ARRAY @NOCOUNT OF Byte;
```

```
  CString = ARRAY @NOCOUNT OF CHAR;
  CStringPointer = POINTER TO ARRAY [0 .. MaxInt - 2] OF CHAR;
  CParamPointer = POINTER TO ARRAY [0 .. MaxInt - 2] OF CStringPointer;

(* Home-grown *)
  Short = BITS 16 FOR [-32768 .. 32767];
  Long = INTEGER;
  Int = INTEGER;
  UChar = BITS 8 FOR [0 .. 255];
  UShort = BITS 16 FOR [0 .. 65535];
  UInt = UNSIGNED;
  ULong = UNSIGNED;
  DAddrT = Long;
  CAddrT = Address;
  InoT = ULong;
  SwBlkT = Long;
  SizeT = Int;
  TimeT = Int;
  DevT = RECORD minor, major: BITS 8 FOR [0 .. 255] END;
  OffT = Int;
  StringArray = Address; (* pointer to an array of strings *)


(*************************************************************)
(*                                                           *)
(*    Types related to files and file descriptors            *)
(*    From:  sys/file.h                                      *)
(*                                                           *)
(*************************************************************)

  PathName = Address; (* of null-terminated character sequence *)
  FileDesc = INTEGER; (* index in per-address-space table *)
  FileDescPair = ARRAY [0 .. 1] OF FileDesc;

(* file descriptor status flags for open and fcntl *)
  FileDescFlag =
    (fWrOnly,
       (* open only: write but don't read; default is read-only *)

     fReadWrite,
       (* open only: both read and write; default is read-only *)

     fNDelay,
       (* don't block: signal if operation would block *)

     fAppend,
       (* each write appends to file *)

     fMark,
       (* not public: used by kernel *)

     fDefer, (* not public: used by kernel *)

     fAsync, (* fcntl only: signal PID or pgrp when data is ready *)

     fShLock, (* not public: used by kernel *)

     fExLock, (* not public: used by kernel *)

     fCreat, (* open only: create file if it doesn't exist *)

     fTrunc, (* open only: truncate previous size to 0 *)

     fExcl, (* open only: error if "creat" and file does not exist *)

(* New with Ultrix 2.0 *)

     fBlkInUse, (* open only, block if in use *)

     fSetInUse, (* Mark in use *)

     fIllegal,

     fSyncron, (* Write file syncronously. *)

     fNBuf (* file used for n buffering *)

       );

  FileDescFlags = BITS 32 FOR SET OF FileDescFlag;

CONST
  fBlkAndSet = FileDescFlags{fBlkInUse, fSetInUse};

TYPE

  FcntlCmd =
    (fDupFD, (* duplicate file descriptor *)

     fGetFD, (* get the close-on-exec flag:  low bit = 1 iff close wanted *)

     fSetFD, (* set the close-on-exec flag *)

     fGetFL, (* get the descriptor status flags, see below *)

     fSetFL, (* set the descriptor status flags *)

     fGetOwn, (* get the PID or pgrpID (-ve) for SIGIO and SIGURG signals *)

     fSetOwn, (* set the PID of pgrpID (-ve) for SIGIO and SIGURG signals *)

(* New with Ultrix 2.0 *)

     fGetLk, (* get file lock *)

     fSetLk, (* set file lock *)

     fSetLkW, (* Set file lock and wait *)

     fSetSyn, (* Set syncronous write *)

     fClrSyn (* Clear sync write *)

       );

  LSeekCmd = OS.SeekCmd;

CONST
  LSet = OS.SeekSet;

  LIncr = OS.SeekIncr;

  LXtnd = OS.SeekExtend;

TYPE
  FLockType = (FLockNotUsed, fRdLck, fWrLck, fUnLck);

  FLock = RECORD
```

```
    type: BITS 16 FOR FLockType;
    whence: BITS 16 FOR LSeekCmd; (* For interpreting start *)
    start: CARDINAL;
    len: CARDINAL; (* =0 means until EOF *)
    pid: INTEGER;
  END;

  FileLockMode = (lockSh, lockEx, lockNB, lockUn);
  FileLockModes = BITS 32 FOR SET OF FileLockMode;

  FileAccess = OS.FileAccess;
  FileAccessMode = BITS 32 FOR OS.FileAccessMode;

  FileMode = BITS 32 FOR OS.FileMode;

(* From sys/gnode.h *)

  FileType =
    (FTPipe, (* Unix manual doesn't say this but it's true! *)

     FTPort, (* Port (named pipe), Ultrix 2.0 only *)

     FTChr, (* Is really a character (unstructured) device *)

     FTill3,

     FTDir, (* File is a directory *)

     FTill5,

     FTBlk, (* Is really a block device *)

     FTil7,

     FTReg, (* Regular File *)

     FTill9,

     FTLnk, (* Symbolic Link *)

     FTillll,

     FTSock, (* Socket *)

     FTill13, FTill14, FTill15);

  StFileMode = RECORD (* This is a word generated by the stat calls *)
    mode: BITS 12 FOR OS.FileMode;
    type: BITS 4 FOR FileType;
    (* WARNING!! THIS WON'T WORK ON THE 68000! *)
  END;


(*****************************************************************)
(*                                                               *)
(*    Structures for stat, lstat, fstat                          *)
(*    From:  sys/stat.h                                          *)
(*                                                               *)
(*****************************************************************)

  FileStat = RECORD
    dev: DevT;
    ino: ULong;
    mode: StFileMode;
    nlink: Short;
    uid: Short;
    gid: Short;
    rdev: DevT;
    size: Int;
    atime: Time.T; (* microseconds field is garbage on Ultrix 1.1 *)
    mtime: Time.T;
    ctime: Time.T;
    blksize: Long;
    blocks: Long; (* Number of 512 byte blocks *)
    spare4: ARRAY [1 .. 2] OF Long;
  END;


(*****************************************************************)
(*                                                               *)
(*    Structures for readv and writev                            *)
(*    From:  sys/uio.h                                           *)
(*                                                               *)
(*****************************************************************)

  IOVec = RECORD iovBase: Address; iovLen: CARDINAL END;


(*****************************************************************)
(*                                                               *)
(*    Structures for sockets                                     *)
(*    From:  sys/socket.h                                        *)
(*                                                               *)
(*****************************************************************)

  Socket = FileDesc; (* communication socket *)
  SocketType =
    (sockUnused, sockStream, sockDatagram, sockRaw, sockRDM, sockSeqPacket);
  SocketOption =
    (soDebug (*1*), (* turn on debugging info recording *)

     soAcceptConn (*2*), (* socket has had "listen()" done *)

     soReuseAddr (*4*), (* allow local address reuse *)

     soKeepAlive (*8*), (* send keep-alives on connection *)

     soDontRoute (*16*), (* just use interface addresses *)

     so5 (*32*),

     soUseLoopback (*64*), (* bypass hardware when possible *)

     soLinger (*128*)
       (* linger on close if data is present *)

       );

  SocketOptions = BITS 32 FOR SET OF SocketOption;

  SocketShut = (noReceives, noSends, noSendsReceives);

  AddressFamily =
    (afUnspecified, afUnix, (* local to host: pipes *)

     afInet, (* arpa internet: TCP *)

     afIMPLink, (* arpanet IMP addresses *)
```

```
    afPUP, (* PARC PUP protocols *)

    afChaos, (* MIT CHAOS protocols *)

    afNS, (*6*)
      (* Xerox NS protocols *)

    afNBS (*7*), (* NBS protocols *)

    afECMA (*8*), (* European Computer Manufacturers Association *)

    afDatakit (*9*), (* datakit protocols *)

    afCCITT (*10*), (* CCITT protocols: X.25 *)

    afSNA (*11*) (* IBM protocols: SNA *)

      );

  SocketAddressFamily = BITS 8 FOR AddressFamily;
  SocketInetAddr = ARRAY [0 .. 3] OF UChar;
  SocketOtherAddr = ARRAY [0 .. 13] OF BITS 8 FOR CHAR;
  SocketAddr = RECORD
    CASE family: SocketAddressFamily OF
      afInet:
        filla: UChar; (* Must be 0 *)
        port: UShort;
        address: SocketInetAddr;
        zero: ARRAY [0 .. 7] OF UChar
    ELSE
      fillb: UChar; (* must be 0 *)
      addr: SocketOtherAddr
    END;
  END;
  SocketMsg = (msgPeek, msgOOB);
  SocketFlags = BITS 32 FOR SET OF SocketMsg;
  MsgHdr = RECORD
    name: POINTER TO SocketAddr (*???*);
    namelen: INTEGER;
    iov: Address (* pointer to array [0..iovlen-1] OF IOVec *);
    iovLen: INTEGER;
    accRights: Address;
    accRightsLen: INTEGER
  END;


(*************************************************************)
(*                                                           *)
(*    Structures for timers                                  *)
(*    From:  sys/time.h                                      *)
(*                                                           *)
(*************************************************************)

  TimeVal = Time.T;
  TimeValPair = ARRAY [0 .. 1] OF TimeVal;
  TimeValPtr = POINTER TO Time.T;
  DSTTime =
    (DSTnone (*0*), (* not on DST *)

     DSTusa (*1*), (* USA style DST *)

     DSTaust (*2*), (* Australian style *)

     DSTwet (*3*), (* Western European style *)

     DSTmet (*4*), (* Middle European style *)

     DSTeet (*5*) (* Eastern European style *)

       );

  TimeZone = Time.TimeZone;
  ITimer = (iTimerReal, iTimerVirtual, iTimerProf);
  ITimerVal = RECORD interval: TimeVal; value: TimeVal END;


(*************************************************************)
(*                                                           *)
(*    Resource usage info                                    *)
(*    From:  sys/resource.h                                  *)
(*                                                           *)
(*************************************************************)

TYPE
  WhichPrio = (prioProcess (*0*), prioPgrp (*1*), prioUser (*2*));

CONST
  rUsageSelf = 0;
  rUsageChildren = -1;

TYPE
  Resource =
    ( (* for use by get/setrlimit *) rLimitCPU (*0*), rLimitFSize (*1*),
     rLimitData (*2*), rLimitStack (*3*), rLimitCore (*4*), rLimitRSS (*5*));
  RLimit = RECORD cur, max: INTEGER END;

  RUsage = RECORD
    uTime: TimeVal;
    sTime: TimeVal;
    CASE BOOLEAN OF
    | FALSE:
        old: RECORD
          maxRSS, ixRSS, idRSS, isRSS: INTEGER;
          minflt, majflt, nswap: INTEGER;
          inBlock, outBlock: INTEGER;
          msgSnd, msgRcv: INTEGER;
          nSignals: INTEGER;
          nvcsw, nivcsw: INTEGER
        END;
    | TRUE:
        new: RECORD
          maxRSS, ixRSS, ismRSS, idRSS, isRSS: INTEGER;
          minflt, majflt, nswap: INTEGER;
          inBlock, outBlock: INTEGER;
          msgSnd, msgRcv: INTEGER;
          nSignals: INTEGER;
          nvcsw, nivcsw: INTEGER
        END;
    END;
  END;
  RUsagePtr = POINTER TO RUsage;

CONST
  U11RUsageLen = TByteSize(RUsage) - 4;

(*************************************************************)
(*                                                           *)
```

```
(*    Signals and codes                                          *)
(*    From:  signal.h                                            *)
(*                                                               *)
(***************************************************************)

TYPE
  SignalOnStack = BITS 32 FOR BOOLEAN;
  SignalStack = RECORD sp: Address; onStack: SignalOnStack END;
  SignalContext = RECORD
    onStack: SignalOnStack;
    mask: OS.Signals;
    sp: Address;
    pc: INTEGER;
    ps: INTEGER
  END;
  SignalHandler = RECORD
    CASE BOOLEAN OF
    | FALSE: h: UNSIGNED;
    | TRUE: p: PROCEDURE(OS.Signal, INTEGER, VAR SignalContext);
    END;
  END;

CONST
  SignalDefault = 0;
  SignalIgnore = 1;
  SignalAlert = 2; (* ONLY FOR OLD OS *)

TYPE
  SignalVectorEntry = RECORD
    handler: SignalHandler;
    mask: OS.Signals;
    onStack: SignalOnStack;
  END;
  SignalVector =
    ARRAY [FIRST(OS.Signal) .. LAST(OS.Signal)] OF SignalVectorEntry;

CONST
  (* code values for SigFPE *)
  FpeIntovfTrap = 1; (* integer overflow *)
  FpeIntdivTrap = 2; (* integer divide by 0 *)
  FpeFltovfTrap = 3; (* floating overflow *)
  FpeFltdivTrap = 4; (* floating/decimal divide by 0 *)
  FpeFltundTrap = 5; (* floating underflow *)
  FpeDecovfTrap = 6; (* decimal overflow *)
  FpeSubrngTrap = 7; (* subscript out of range *)
  FpeFltovfFault = 8; (* floating overflow fault *)
  FpeFltdivFault = 9; (* divide by 0 floating fault *)
  FpeFltundFault = 10; (* floating underflow fault *)
  (* code values for SigIll *)
  IllResadFault = 0; (* reserved address fault *)
  IllPrivinFault = 1; (* privileged instruction fault *)
  IllResopFault = 2; (* reserved operand fault *)

(* Deprecated spellings of signal types. (For compatiblity with Unix42.def. *)
TYPE Signal = OS.SignalOrNone; (* but sometimes you want OS.Signal *)
CONST FirstSig = FIRST(OS.Signal); LastSig = LAST(OS.Signal);
TYPE
  SigMask = OS.Signals;
  SigOnStack = SignalOnStack;
  SigStackRec = SignalStack;
  SigContext = SignalContext;
  SigHandler = PROCEDURE(OS.Signal, INTEGER, VAR SignalContext);
  SigVecRec = SignalVectorEntry;
```

```
(***************************************************************)
(*                                                             *)
(*    Types for Wait(2) and Wait3(2)                           *)
(*    From:  wait.h                                            *)
(*                                                             *)
(***************************************************************)

TYPE
  WStopVal = (WTerm, WStop);
  WStatus = RECORD
    CASE WStopVal OF
    | WTerm: (* use this case when wStopVal # WSTOPPED *)
        wTermSig: BITS 7 FOR OS.SignalOrNone;
        wCoreDump: BITS 1 FOR BOOLEAN;
        wRetCode: BITS 8 FOR [0 .. 255];
    | WStop: (* use this case when wStopVal = WSTOPPED *)
        wStopVal: BITS 8 FOR [0 .. 255];
        wStopSig: BITS 8 FOR OS.Signal;
    END;
  END;
  WOption = (WNOHANG, WUNTRACED);
  WOptions = BITS 32 FOR SET OF WOption;

CONST
  WSTOPPED = 177B;

TYPE
  (* Result record for old SYSgetspaceinfo call: *)
  SpaceState = (SSRunning, SSStopped, SSTerminating);
  TrapSpaceInfo = RECORD
    pid, ppid, pgrp: INTEGER;
    uid: INTEGER;
    space: [0 .. 127];
    ctlTty: ARRAY [0 .. 1] OF CHAR;
    runState: SpaceState;
    topaz: BOOLEAN;
    ws: WStatus;
    commandString: ARRAY [0 .. 59] OF CHAR;
    cTime, wTime: Time.T;
  END;
  SpaceInfoPtr = POINTER TO ARRAY [0 .. MaxInt] OF TrapSpaceInfo;

CONST
  AlwaysHandledTraps = (* constant used by SYSreservetraps kernel call *)
    TPSpecial.TrapKinds{TPSpecial.TKFromDebugger, TPSpecial.TKIllegalNubCall};


(***************************************************************)
(*                                                             *)
(*    Executable object files                                  *)
(*    From:  a.out.h                                           *)
(*                                                             *)
(***************************************************************)

CONST
  OMagic = 407B; (* old impure format *)
  NMagic = 410B; (* read-only text *)
  ZMagic = 413B; (* demand load format *)

TYPE
  Exec = RECORD
    magic: CARDINAL; (* magic number *)
    text: CARDINAL; (* size of text segment *)
```

```
    data: CARDINAL; (* size of initialized data *)
    bss: CARDINAL; (* size of uninitialized data *)
    syms: CARDINAL; (* size of symbol table *)
    entry: CARDINAL; (* entry point *)
    tRSize: CARDINAL; (* size of text relocation *)
    dRSize: CARDINAL; (* size of data relocation *)
  END;

(*****************************************************************)
(*                                                               *)
(*    Core dump files                                            *)
(*    From:  core(5) manpage and sys/user.h and machine/param.h  *)
(*                                                               *)
(*****************************************************************)

(* These declarations define the variant of core files interpreted by the
   Loupe debugger for the Topaz Ultrix emulator. The structure has been tested
   to some extent with adb (but not with dbx). *)

(* The layout of a core file is as follows. Note that page refers to a VAX
   page of 512 bytes.

                        Length
                        ------
        Header          10 pages
        Data image      header.dataPages
        Stack image     header.stackPages
        Thread states   # of threads * System.TByteSize(ThreadMapEntry) bytes

*)

CONST (* offsets, lengths of header fields *)
  oPcbKsp = 0;
  lPcbKsp = 4;
  oPcbUsp = 12;
  lPcbUsp = 4;
  oStatus = 388;
  lStatus = 2;
  oTextPages = 536;
  lTextPages = 4;
  oDataPages = 540;
  lDataPages = 4;
  oStackPages = 544;
  lStackPages = 4;
  oAp = 5036;
  lAp = 4;
  oFp = 5040;
  lFp = 4;
  oRegs = 5048;
  lRegs = 48;
  oSp = 5100;
  lSp = 4;
  oPc = 5112;
  lPc = 4;
  oPsw = 5116;
  lPsw = 4;

(* For compatibility with possible future versions, unused fields in the
   header (xxNNN) contain zero. *)
TYPE
  CoreFileHeader = RECORD
    pcbKsp: UNSIGNED;
    xx4: ARRAY [oPcbKsp + lPcbKsp .. oPcbUsp - 1] OF Byte;
    pcbUsp: UNSIGNED;
    xx16: ARRAY [oPcbUsp + lPcbUsp .. oStatus - 1] OF Byte;
    status: Short;
    xx390: ARRAY [oStatus + lStatus .. oTextPages - 1] OF Byte;
    textPages: CARDINAL;
    dataPages: CARDINAL;
    stackPages: CARDINAL;
    xx548: ARRAY [oStackPages + lStackPages .. oAp - 1] OF Byte;
    ap: UNSIGNED;
    fp: UNSIGNED;
    xx5044: ARRAY [oFp + lFp .. oRegs - 1] OF Byte;
    regs: ARRAY [0 .. 11] OF UNSIGNED;
    xx5096: ARRAY [oRegs + lRegs .. oSp - 1] OF Byte;
    sp: UNSIGNED;
    xx5104: ARRAY [oSp + lSp .. oPc - 1] OF Byte;
    pc: UNSIGNED;
    psw: UNSIGNED;
  END;

(* ASSERT( System.TByteSize( Header ) = 5120 *)

TYPE
  CoreFileThreadMapEntry = RECORD
    handle: ThreadsPort.Handle;
    state: TPSpecial.State;
  END;
  CoreFileThreadMap = ARRAY OF CoreFileThreadMapEntry;


(*****************************************************************)
(*                                                               *)
(*    Raw kernel calls (alphabetic order)                        *)
(*    Unless otherwise specified, these return -1 iff error.     *)
(*                                                               *)
(*****************************************************************)

PROCEDURE @EXTERNAL accept(
    s: Socket;
    (* out *)
    VAR addr: SocketAddr;
    (* inout *)
    VAR addrlen: INTEGER)
    : Socket;

PROCEDURE @EXTERNAL access(path: PathName; mode: FileAccessMode): INTEGER;

PROCEDURE @EXTERNAL acct(file: PathName): INTEGER;

PROCEDURE @EXTERNAL bind(
    s: Socket;
    VAR name: SocketAddr;
    namelen: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL brk(
    addr: Address)
    : INTEGER
    (* see also sbrk *);

PROCEDURE @EXTERNAL chdir(path: PathName): INTEGER;

PROCEDURE @EXTERNAL chmod(path: PathName; mode: FileMode): INTEGER;

PROCEDURE @EXTERNAL chown(path: PathName; owner, group: INTEGER): INTEGER;
```

```
PROCEDURE @EXTERNAL chroot(dirname: PathName): INTEGER;

PROCEDURE @EXTERNAL close(d: FileDesc): INTEGER;

PROCEDURE @EXTERNAL connect(
    s: Socket;
    VAR name: SocketAddr;
    namelen: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL creat(name: PathName; mode: FileMode): FileDesc;
(* use "open" instead of "creat". *)

PROCEDURE @EXTERNAL dup(oldd: FileDesc): FileDesc;

PROCEDURE @EXTERNAL dup2(oldd, newd: FileDesc): INTEGER;

PROCEDURE @EXTERNAL execve(
    name: PathName;
    argv: StringArray;
    envp: StringArray)
    : INTEGER;

(* Use underbarexit to call _exit *)
PROCEDURE @EXTERNAL underbarexit(status: INTEGER);

PROCEDURE @EXTERNAL fchmod(fd: FileDesc; mode: FileMode): INTEGER;
(* see "chmod". *)

PROCEDURE @EXTERNAL fchown(fd: FileDesc; owner, group: INTEGER): INTEGER;
(* see "chown". *)

PROCEDURE @EXTERNAL fcntl(fd: FileDesc; cmd: FcntlCmd; arg: INTEGER): INTEGER;
(* for cmd=fDupFD, result is a FileDesc *)
(* for cmd=fGetFD, result is bottom bit of the integer *)
(* for cmd=fSetFD, arg is bottom bit of integer, result is not -1 *)
(* for cmd=fGetFl, result is FileDescFlags *)
(* for cmd=fSetFl, argument is FileDescFlags, result is not -1 *)
(* for cmd=fGetOwn, result is PID (+ve) or pgrpID (-ve) *)
(* for cmd=fSetOwn, argument is PID (+ve) or pgrpID (-ve), result # -1 *)

PROCEDURE @EXTERNAL flock(fd: FileDesc; operation: FileLockModes): INTEGER;

PROCEDURE @EXTERNAL fork(): INTEGER;
(* see also vfork. On error, no child is created and -1 is returned. Otherwise
   returns 0 to the child and the child's PID to the parent *)

PROCEDURE @EXTERNAL fstat(fd: FileDesc; VAR buf: FileStat): INTEGER;
(* see documentation for "stat" *)

PROCEDURE @EXTERNAL fsync(fd: FileDesc): INTEGER;

PROCEDURE @EXTERNAL ftruncate(fd: FileDesc; length: INTEGER): INTEGER;

PROCEDURE @EXTERNAL getdtablesize(): INTEGER;

PROCEDURE @EXTERNAL getegid(): INTEGER;

PROCEDURE @EXTERNAL geteuid(): INTEGER;

PROCEDURE @EXTERNAL getgid(): INTEGER;

PROCEDURE @EXTERNAL getgroups(ngroups: INTEGER; gidset: Address): INTEGER;

PROCEDURE @EXTERNAL gethostid(): INTEGER;
(* see also sethostid *)

PROCEDURE @EXTERNAL gethostname(name: Address; namelen: INTEGER): INTEGER;
(* see also sethostname. *)

PROCEDURE @EXTERNAL getitimer(which: ITimer; VAR value: ITimerVal): INTEGER;
(* see also settimer. *)

PROCEDURE @EXTERNAL getpagesize(): INTEGER;

PROCEDURE @EXTERNAL getpeername(
    s: Socket;
    VAR name: SocketAddr;
    VAR namelen: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL getpgrp(pid: INTEGER): INTEGER;

PROCEDURE @EXTERNAL getpid(): INTEGER;

PROCEDURE @EXTERNAL getppid(): INTEGER;

PROCEDURE @EXTERNAL getpriority(
    which: WhichPrio;
    who: INTEGER (* pid, pgrpid, or uid *))
    : INTEGER;
(* see also setpriority. Beware: this can return -1 as a valid result. To
   check for errors, you must first clear errno (!) *)

PROCEDURE @EXTERNAL getrlimit(resource: Resource; VAR rlp: RLimit): INTEGER;
(* see also setrlimit. *)

PROCEDURE @EXTERNAL getrusage(who: INTEGER; usage: RUsagePtr): INTEGER;
(* who=0 for self, who=-1 for terminated children . Caution depends on which
   version of Ultrix is being used. *)

PROCEDURE @EXTERNAL getsockname(
    s: Socket;
    VAR name: SocketAddr;
    VAR namelen: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL getsockopt(
    s: Socket;
    level, optname: INTEGER;
    optval: Address;
    VAR optlen: INTEGER)
    : INTEGER;
(* see also setsockopt *)

PROCEDURE @EXTERNAL gettimeofday(VAR tp: TimeVal; VAR tzp: TimeZone): INTEGER;
(* see also settimeofday *)

PROCEDURE @EXTERNAL getuid(): INTEGER;

PROCEDURE @EXTERNAL ioctl(
    d: FileDesc;
    request: INTEGER;
    argp: Address)
    : INTEGER;

PROCEDURE @EXTERNAL kill(pid: INTEGER; sig: OS.SignalOrNone): INTEGER;
```

```
PROCEDURE @EXTERNAL killpg(pgrp: INTEGER; sig: OS.SignalOrNone): INTEGER;

PROCEDURE @EXTERNAL link(name1, name2: PathName): INTEGER;

PROCEDURE @EXTERNAL listen(s: Socket; backlog: INTEGER): INTEGER;

PROCEDURE @EXTERNAL lseek(
    d: INTEGER;
    offset: INTEGER;
    whence: LSeekCmd)
    : INTEGER;

PROCEDURE @EXTERNAL lstat(path: PathName; VAR buf: FileStat): INTEGER;

PROCEDURE @EXTERNAL mkdir(path: PathName; mode: FileMode): INTEGER;

PROCEDURE @EXTERNAL mknod(
    path: PathName;
    mode: FileMode;
    dev: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL mount(special, name: PathName; rwflag: BOOLEAN): INTEGER;

PROCEDURE @EXTERNAL open(
    name: PathName;
    flags: FileDescFlags;
    mode: FileMode)
    : INTEGER;

PROCEDURE @EXTERNAL pipe(VAR fildes: FileDescPair): INTEGER;

PROCEDURE @EXTERNAL profil(
    buff: Address;
    bufsiz, offset, scale: INTEGER)
    : INTEGER;
(* Always returns 0 *)

PROCEDURE @EXTERNAL ptrace(
    request, pid: INTEGER;
    addr: Address;
    data: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL quota(cmd, uid, arg: INTEGER; addr: Address);

PROCEDURE @EXTERNAL read(
    d: FileDesc;
    buf: Address;
    nbytes: CARDINAL)
    : INTEGER;

PROCEDURE @EXTERNAL readlink(
    path: PathName;
    buf: Address;
    bufsiz: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL readv(
    d: FileDesc;
    buf: Address;
    iovcnt: CARDINAL)
    : INTEGER;
(* "buf" is address of array [0..iovcnt-1] of IOVec *)

PROCEDURE @EXTERNAL reboot(howto: BITSET): INTEGER;

PROCEDURE @EXTERNAL recv(
    s: Socket;
    buf: Address;
    len: INTEGER;
    flags: SocketFlags)
    : INTEGER;

PROCEDURE @EXTERNAL recvfrom(
    s: Socket;
    buf: Address;
    len: INTEGER;
    flags: SocketFlags;
    VAR from: SocketAddr;
    VAR fromLen: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL recvmsg(
    s: Socket;
    VAR msg: MsgHdr;
    flags: SocketFlags)
    : INTEGER;

PROCEDURE @EXTERNAL rename(from, to: PathName): INTEGER;

PROCEDURE @EXTERNAL rmdir(path: PathName): INTEGER;

PROCEDURE @EXTERNAL sbrk(incr: INTEGER): INTEGER;

PROCEDURE @EXTERNAL select(
    nfds: INTEGER;
    VAR readfds: ARRAY @NOCOUNT OF BITSET;
    VAR writefds: ARRAY @NOCOUNT OF BITSET;
    VAR exceptfds: ARRAY @NOCOUNT OF BITSET;
    timeout: TimeValPtr)
    : INTEGER;
(* nfds must be less than 32 on Ultrix 1.1, and less than 64 on Ultrix 2.0 *)

PROCEDURE @EXTERNAL send(
    s: Socket;
    msg: Address;
    len: INTEGER;
    flags: SocketFlags)
    : INTEGER;

PROCEDURE @EXTERNAL sendto(
    s: Socket;
    msg: Address;
    len: INTEGER;
    flags: SocketFlags;
    VAR to: SocketAddr;
    tolen: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL sendmsg(
    s: Socket;
    VAR msg: MsgHdr;
    flags: SocketFlags)
    : INTEGER;

PROCEDURE @EXTERNAL setgroups(ngroups: INTEGER; gidset: Address): INTEGER;
```

```
PROCEDURE @EXTERNAL sethostid(hostid: INTEGER): INTEGER;

PROCEDURE @EXTERNAL sethostname(name: Address; namelen: INTEGER): INTEGER;

PROCEDURE @EXTERNAL setitimer(
    which: ITimer;
    VAR value, ovalue: ITimerVal)
    : INTEGER;

PROCEDURE @EXTERNAL setpgrp(pid: INTEGER; pgrp: INTEGER): INTEGER;

PROCEDURE @EXTERNAL setpriority(
    which: WhichPrio;
    who: INTEGER;
    prio: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL setquota(special, file: PathName): INTEGER;

PROCEDURE @EXTERNAL setregid(rgid, egid: INTEGER): INTEGER;

PROCEDURE @EXTERNAL setreuid(ruid, euid: INTEGER): INTEGER;

PROCEDURE @EXTERNAL setrlimit(resource: Resource; VAR rlp: RLimit): INTEGER;

PROCEDURE @EXTERNAL setsockopt(
    s: Socket;
    level, optname: INTEGER;
    optval: Address;
    optlen: INTEGER)
    : INTEGER;

PROCEDURE @EXTERNAL settimeofday(VAR tp: TimeVal; VAR tzp: TimeZone): INTEGER;

PROCEDURE @EXTERNAL shutdown(s: Socket; how: SocketShut): INTEGER;

PROCEDURE @EXTERNAL sigblock(mask: OS.Signals): OS.Signals;
(* returns old mask; no errors; SigKill, SigStop and SigCont cannot be
   blocked, and this is imposed silently by the system *)

PROCEDURE @EXTERNAL sigpause(mask: OS.Signals): INTEGER;
(* Always terminates by being interrupted by a signal; therefore, always gives
   error EINTR *)

PROCEDURE @EXTERNAL sigsetmask(mask: OS.Signals): OS.Signals;
(* returns old mask; no errors; SigKill, SigStop and SigCont cannot be
   blocked, and this is imposed silently by the system *)

PROCEDURE @EXTERNAL sigstack(
    VAR ss: SignalStack;
    VAR oss: SignalStack)
    : INTEGER;

PROCEDURE @EXTERNAL sigvec(
    sig: OS.Signal;
    VAR vec: SignalVectorEntry;
    VAR ovec: SignalVectorEntry)
    : INTEGER;

PROCEDURE @EXTERNAL socket(
    af: AddressFamily;
    type: SocketType;
    protocol: INTEGER)
    : FileDesc;

PROCEDURE @EXTERNAL socketpair(
    af: AddressFamily;
    type: SocketType;
    protocol: INTEGER;
    sv: Address)
    : INTEGER;

PROCEDURE @EXTERNAL stat(name: PathName; VAR buf: FileStat): INTEGER;

PROCEDURE @EXTERNAL swapon(special: PathName): INTEGER;

PROCEDURE @EXTERNAL symlink(name1, name2: PathName): INTEGER;

PROCEDURE @EXTERNAL sync();

PROCEDURE @EXTERNAL syscall(number: KernelEntry): INTEGER;
(* See the manual. Since this really takes a variable number of arguments, you
   need to loophole it into something useful *)

PROCEDURE @EXTERNAL truncate(path: PathName; length: INTEGER): INTEGER;

PROCEDURE @EXTERNAL umask(numask: FileMode): FileMode;

PROCEDURE @EXTERNAL umount(special: PathName): INTEGER;

PROCEDURE @EXTERNAL unlink(name: PathName): INTEGER;

PROCEDURE @EXTERNAL utimes(name: PathName; VAR tvp: TimeValPair): INTEGER;

PROCEDURE @EXTERNAL vfork(): INTEGER;

PROCEDURE @EXTERNAL vhangup();

PROCEDURE @EXTERNAL wait(VAR status: WStatus): INTEGER;

PROCEDURE @EXTERNAL wait3(
    VAR status: WStatus;
    options: WOptions;
    rusage: RUsagePtr)
    : INTEGER;

PROCEDURE @EXTERNAL write(
    d: FileDesc;
    buffer: Address;
    nbytes: CARDINAL)
    : INTEGER;

PROCEDURE @EXTERNAL writev(
    d: FileDesc;
    iov: Address;
    ioveclen: CARDINAL)
    : INTEGER;
(* "iov" is the address of an array [0..ioveclen-1] of IOVec *)

END UxTypes.
```

```
(* Last modified on Mon Apr  7 09:55:17 1986 by mcjones *)

SAFE IMPLEMENTATION MODULE Ux;
  IMPORT UxAS, UxFile, UxPro, UxTime;


(***********************)
(* Exported procedures *)
(***********************)

PROCEDURE Init() RAISES {};
  BEGIN
    UxPro.Init(); (* must be first, so others may call UxPro.Register *)
    UxAS.Init();
    UxFile.Init();
    UxTime.Init();
  END Init;

END Ux.
```

```
(* DIGITAL EQUIPMENT CORPORATION  CONFIDENTIAL AND PROPRIETARY *)
(* Last modified on Mon Jul 20 10:27:23 1987 by mcjones *)
(*      modified on Wed May  6 16:52:24 1987 by swart *)
(*      modified on Tue Oct 21 13:17:16 1986 by mds *)

MODULE UxAS IMPLEMENTS UxAS, RemoteOSAS;

IMPORT
  Hardware, OSFriends, RPCLocal, RdV, TDState, TDTraps, ThreadFriends,
  ThreadsPort, TPSpecial, User, UxFile, UxPro, VM, VMFriends, VMFriends2,
  WrV;

CONST

(* Constants determined by VAX architecture. *)
  BytesPerRegion = Hardware.PagesPerRegion * Hardware.BytesPerPage;
  BytesPerUserSpace = 2(*regions*) * BytesPerRegion;

(* Constants determined by Ultrix implementation. *)
  UltrixBytesPerPage = 1024;
  TextAddr = 0; (* start of text (code) *)
  TextPage = TextAddr DIV Hardware.BytesPerPage;
  HoleAddr = BytesPerUserSpace - TPSpecial.PreferredStackHole;
  (* Stack starts here and grows towards smaller addresses. There is one more
     page at this address, containing the handler linkage code. *)
  HandlerEntry = HoleAddr + 6cH;  (* start of handler linkage *)
  initRLimDataCur = 1578000H;
  initRLimDataMax = 1578000H;
  initRLimStackCur = 1578000H;
  initRLimStackMax = 1578000H;
  initRLimCoreCur = 7fffffffH;
  initRLimCoreMax = 7fffffffH;

(* Constants determined by VM implementation. *)
(* VMBytesPerPage = 512; *) (* variable while experimenting *)

VAR
  initialized: BOOLEAN;

VAR (* but logically constant *)
  VMBytesPerPage: CARDINAL;
  initialStackAddr: Addr;
  nullTHandle: ThreadsPort.Handle;
  handlerLinkage: ARRAY [0 .. 3] OF INTEGER;
  osSpace: NubTypes.Space; (* address space Taos is in *)

(* Traps handled by Ux. *)
CONST
  AllTraps =
    TPSpecial.TrapKinds{ (* all but TKNoTrap *)
                        TPSpecial.TKBadParameter .. LAST(TPSpecial.TrapKind)};
  MyTraps = AllTraps - TDTraps.HandledTraps;

(* Private state for UxAS. *)
(* Fields below marked (*!*) are inherited by forked process. In case of
   state, only some subfields are inherited. *)
TYPE
  T = REF TObj;
  Continuation = (Resume, Start, Nothing);
  TObj = RECORD
    firstWritableAddr: Addr; (* lowest not write protected *)
    traps: TPSpecial.TrapKinds; (* ones we are catching *)
    handlerThread: ThreadsPort.Handle; (* ONLY FOR OLD OS *)
    straceExec: OSFriends.STrace; (* whether to trap after execve *)
    straceFork: OSFriends.STrace; (* whether to trap after fork *)
    rLimData: UxTypes.RLimit; (* <= BytesPerRegion *)  (*!*)
    rLimStack: UxTypes.RLimit; (* <= BytesPerRegion *) (*!*)
    rLimCore: UxTypes.RLimit;                          (*!*)
    currentThread: ThreadsPort.Handle; (* thread for current action *)
    CASE continuation: Continuation OF
    | Resume: sState: TPSpecial.SState;
    END;
  END;

CONST
  MaxIdleSpaces = 7;

(* Global variables under protection of mutex x. *)
VAR
  x: Thread.Mutex;
  idleSpaces:
    ARRAY [1 .. MaxIdleSpaces] OF RECORD
      s: VMFriends.Space; (* s # VMFriends.NullSpace => s is idle space *)
      b: VM.PageNumber; (* s is idle space => b = GetBoundary(, VM.Low) *)
    END;
  maxIdleSpaces: CARDINAL; (* maxIdleSpaces <= MaxIdleSpaces *)
  maxIdleBreakAddr: Addr; (* zero disables space caching *)


(***********************)
(* Internal procedures *)
(***********************)

PROCEDURE AllocSpace(u: Ux.T; bssAddr, breakAddr, stackAddr: Addr);
  (* Assign space to u, with low and high segment boundaries equal to
     breakAddr and stackAddr respectively; ensure bytes from bssAddr to
     breakAddr-1 are zero. AllocSpace assumes the initial thread for u has
     already been allocated. UserTTD won't see this space until
     TDAdvise.SpaceLoaded is called. *)
  VAR
    space: VMFriends.Space;
    i, j: CARDINAL;
    delta, bestDelta: INTEGER;
    minBreakAddr: Addr;
    pages: CARDINAL;
  BEGIN
    space := VMFriends.NullSpace;

    (* Use an idle space if one exists and this request is not too large. *)
    LOCK x DO
      bestDelta := LAST(INTEGER);
      IF breakAddr <= maxIdleBreakAddr THEN
        FOR i := 1 TO maxIdleSpaces DO
          WITH idleSpaces[i] DO
            IF s # VMFriends.NullSpace THEN
              (* ASSERT((b <= MaxCard) AND (breakAddr <= MaxCard)); *)
              delta :=
                ABS(LOOPHOLE(b, CARDINAL) - LOOPHOLE(breakAddr, CARDINAL));
              IF delta < bestDelta THEN j := i; bestDelta := delta; END;
            END;
          END;
        END;
        IF bestDelta # LAST(INTEGER) THEN
          WITH idleSpaces[j] DO space := s; s := VMFriends.NullSpace; END;
          u^.space := space;
          RestoreThread(u);
          minBreakAddr := MIN(GetBoundary(u, VM.Low), breakAddr);
          SetBoundary(space, VM.Low, breakAddr);
```

```
        IF bssAddr < minBreakAddr THEN
          (* We only need to zero from bssAddr to minBreakAddr; any new
             pages will be zeroed by VM. *)
          SBuf.Zero(SBuf.SNew(space, bssAddr, minBreakAddr - bssAddr));
        END;
        SetBoundary(space, VM.High, stackAddr);
      END;
    END;
  END;

  (* Else create and initialize new space. *)
  IF space = VMFriends.NullSpace THEN

    space := VMFriends.AllocSpace();
    IF space = VMFriends.NullSpace THEN
      RAISE(OS.Error, OS.TooManyProcessesEC);
    END;

    u^.space := space;
    RestoreThread(u);
    SetBoundary(space, VM.Low, breakAddr);
    SetBoundary(space, VM.High, stackAddr);

    (* Copy handlerLinkage to page above stack. *)
    Put(u, HandlerEntry, handlerLinkage);

    (* Make the hole above the stack read-only to trap erroneous stores.
       Avoid finding out what MakePagesReadOnly does when the count is zero
       but the page number is up in system space. *)
    pages := TPSpecial.PreferredStackHole DIV VMBytesPerPage;
    IF 0 < pages THEN
      VMFriends.MakePagesReadOnly(space,
        BytesPerUserSpace DIV VMBytesPerPage - pages, pages);
    END;
  END;
  Ux.map[u^.space] := u;
END AllocSpace;

PROCEDURE AllocThread(u: Ux.T);
  (* Create the initial thread to run in a new space. UserTTD won't see this
     thread until TDAdvise.SpaceLoaded is called. *)
  VAR a: T; state: TPSpecial.State;
  BEGIN
    a := u^.a;
    a^.currentThread := TPSpecial.Create(NIL, NIL, 0);
    a^.continuation := Resume;
    WITH a^.sState DO
      setR1 := FALSE;
      error := FALSE;
    END;
    TPSpecial.GetState(a^.currentThread, state);
    state.priority := ThreadFriends.GetPriority();
    (* state.processor is default *)
    state.stackHigh := HoleAddr; (* for the garbage collector *)
    state.trapInfo.reported := TRUE;
    state.trapInfo.kind := TPSpecial.TKIllegalNubCall;
    TPSpecial.SetState(a^.currentThread, state);
  END AllocThread;

PROCEDURE CloseQuietly(f: OS.File) RAISES {};
  BEGIN
    TRY OS.Close(f); EXCEPT OS.Error: (* e.g. RemoteFileEC *) END;
  END CloseQuietly;
```

```
PROCEDURE ExcludedStop(r: REFANY);
  (* Called via RPCLocal.Exclude. *)
  VAR a: T;
  BEGIN
    a := NARROW(r, T);
    TPSpecial.Stop(a^.currentThread);
  END ExcludedStop;

PROCEDURE ExcludedStopAll(r: REFANY);
  (* Called via RPCLocal.Exclude. *)
  VAR u: Ux.T;
  BEGIN
    u := NARROW(r, Ux.T);
    TPSpecial.StopAll(u^.space);
    (* this is at least the 2nd stop for a^.currentThread *)
  END ExcludedStopAll;

PROCEDURE GetBoundary(u: Ux.T; seg: VM.Segment): Addr RAISES {};
  VAR page: VM.PageNumber; ok: BOOLEAN;
  BEGIN
    ok := VMFriends.AllocPages(u^.space, seg, 0, page);
    RETURN page * VMBytesPerPage;
  END GetBoundary;

PROCEDURE GrowStack(u: Ux.T; addr: Addr): BOOLEAN;
  (* Attempt to grow stack of address space of u so as to include addr.
     Result is TRUE if successful; FALSE if unsuccessful (because stack has
     reached current rlimit). *)
  BEGIN
    IF NOT IsTopaz(u) AND (HoleAddr - Addr(u^.a^.rLimStack.cur) <= addr)
       AND (addr < HoleAddr) THEN
      SetBoundary(u^.space, VM.High, addr);
      RETURN TRUE;
    ELSE
      RETURN FALSE;
    END;
  END GrowStack;

PROCEDURE SetBoundary(space: VMFriends.Space; seg: VM.Segment; addr: Addr);
  (* Change length of seg in space to smallest multiple of VMBytesPerPage not
     less than that implied by new "boundary address" addr (see GetBoundary
     for definition of "boundary address"). Assume caller has done appropriate
     checking. *)
  VAR
    tempAddr: Addr;
    delta: INTEGER; (* alloc if > 0; free if < 0 *)
    ok: BOOLEAN;
    page: VM.PageNumber;
  BEGIN
    ok := VMFriends.AllocPages(space, seg, 0, page);
    tempAddr := page * VMBytesPerPage;

    IF seg = VM.Low THEN
      delta := addr - tempAddr;
    ELSE (* VM.High *)
      delta := tempAddr - addr;
    END;

    IF delta > 0 THEN (* round up *)
      ok :=
        VMFriends.AllocPages(space, seg,
          (delta + VMBytesPerPage - 1) DIV VMBytesPerPage, page);
      ASSERT(ok, "UxAS.SetBoundary confused");
    ELSIF delta <= -VMBytesPerPage THEN (* round down *)
```

```
      VMFriends.FreePages(space, seg, (-delta) DIV VMBytesPerPage);
    END;
  END SetBoundary;

PROCEDURE UltrixRoundUp(addr: Addr): Addr;
  (* Result is the smallest multiple of UltrixBytesPerPage not less than the
     argument. *)
  BEGIN
    RETURN
      (addr + UltrixBytesPerPage - 1) DIV UltrixBytesPerPage *
        UltrixBytesPerPage;
  END UltrixRoundUp;

PROCEDURE VMRoundUp(addr: Addr): Addr;
  (* Result is the smallest multiple of VMBytesPerPage not less than the
     argument. *)
  BEGIN
    RETURN (addr + VMBytesPerPage - 1) DIV VMBytesPerPage * VMBytesPerPage;
  END VMRoundUp;


(***********************************)
(* Procedures exported through UxAS *)
(***********************************)

PROCEDURE AlertHandler(u: Ux.T) RAISES {};
  (* The argument to this procedure is in general not a thread currently
     executing a kernel call. *)
  (* ONLY FOR OLD OS *)
  VAR a: T; state: TPSpecial.State;
  BEGIN
    a := u^.a;
    ASSERT(NOT ThreadsPort.Equal(a^.handlerThread, nullTHandle),
      "UxAS.AlertHandler confused");

    (* The Thread package currently recycles threads within the same address
       space; for safety we check this here. *)
    TPSpecial.GetState(a^.handlerThread, state);
    ASSERT(state.space = u^.space); (* ***** or just terminate space? *)

    ThreadsPort.Alert(a^.handlerThread);
  END AlertHandler;

PROCEDURE BackupKernelCall(
    u: Ux.T;
    entry: UxTypes.KernelEntry)
  RAISES {SBuf.Fault};
  (* Only applied to thread that has done a TKIllegalNubCall trap. *)
  CONST
    opChmk = 0bcH;
    immMode = 08fH;
    maxLit = 03fH;
    regR0 = 050H;
    regR15 = 05fH;
  TYPE Byte = BITS 8 FOR CARDINAL;
  VAR state: TPSpecial.State; pc: Addr; bytes: ARRAY [0 .. 3] OF Byte;
  BEGIN
    TPSpecial.GetState(u^.a^.currentThread, state);
    pc := state.machineState.regs[Hardware.PCReg].val;
    Get(u, pc - 4, bytes, 0, 4);
    IF (bytes[0] = opChmk) AND (bytes[1] = immMode)
       AND (bytes[2] + 256 * bytes[3] = ORD(entry)) THEN
      pc := pc - 4;
    ELSIF
      (bytes[2] = opChmk)
      AND ((bytes[3] <= maxLit) AND (bytes[3] = ORD(entry))
           OR (regR0 <= bytes[3]) AND (bytes[3] <= regR15)
              AND (state.machineState.regs[bytes[3] - regR0].val = ORD(entry)))
    THEN
      pc := pc - 2;
    ELSE
      ASSERT(FALSE, "UxAS.BackupKernelCaller confused");
    END;
    state.machineState.regs[Hardware.PCReg].val := pc;
    TPSpecial.SetState(u^.a^.currentThread, state);
  END BackupKernelCall;

PROCEDURE Copy(u, uNew: Ux.T; vFork: BOOLEAN) RAISES {OS.Error, SBuf.Fault};
  VAR
    a, aNew: T;
    lowAddr, breakAddr, stackAddr: Addr;
    pageState: VM.PageState;
    state, stateNew: TPSpecial.State;
  BEGIN
    AllocThread(uNew);
    a := u^.a; aNew := uNew^.a;

    (* New space gets copy of registers of old. *)
    TPSpecial.GetState(a^.currentThread, state);
    TPSpecial.GetState(aNew^.currentThread, stateNew);
    stateNew.machineState.regs := state.machineState.regs;
    TPSpecial.SetState(aNew^.currentThread, stateNew);

    (* Allocate appropriate space. *)
    ASSERT(uNew^.space = VMFriends.NullSpace, "UxAS.Copy: old=null");
    aNew^.firstWritableAddr := a^.firstWritableAddr;
    IF vFork THEN
      uNew^.space := u^.space;
      RestoreThread(uNew);
    ELSE

      (* The first page may have been unmapped by the Topaz runtime. *)
      lowAddr := TextAddr;
      pageState := VMFriends.StatePage(u^.space, 0);
      IF pageState.contents = VM.Nonexistent THEN
        lowAddr := MAX(lowAddr, LOOPHOLE(VMBytesPerPage, Addr));
      END;

      breakAddr := GetBoundary(u, VM.Low); (* lowest free *)
      ASSERT(breakAddr > 0, "UxAS.Copy: break=0");
      stackAddr := GetBoundary(u, VM.High); (* lowest allocated *)
      ASSERT(stackAddr < HoleAddr, "UxAS.Copy: bad stack");
      AllocSpace(uNew, breakAddr, breakAddr, stackAddr);

      (* Copy low and high regions. *)
      IF NOT VMFriends.CopyBlock(u^.space, lowAddr, uNew^.space, lowAddr,
               breakAddr - lowAddr)
         OR NOT VMFriends.CopyBlock(u^.space, stackAddr, uNew^.space,
                  stackAddr, HoleAddr - stackAddr) THEN
        RAISE(SBuf.Fault);
      END;

      (* Unmap first page if appropriate. *)
      IF lowAddr # 0 THEN
        VMFriends.UnmapPages(uNew^.space, 0, lowAddr DIV VMBytesPerPage);
      END;

      (* Write-protect prefix of low region. *)
```

```
      VMFriends.MakePagesReadOnly(uNew^.space, lowAddr DIV VMBytesPerPage,
        (MIN(aNew^.firstWritableAddr, breakAddr) - lowAddr) DIV
          VMBytesPerPage); (* round size down *)
    END;
  END Copy;

PROCEDURE Dispose(u: Ux.T; freeSpace: BOOLEAN) RAISES {};
  (* This always follows a call to TDAdvise.SpaceGone to avoid surprising
     UserTTD. *)
  VAR idle, ok: BOOLEAN; nHandles, i: CARDINAL; breakAddr: Addr;
  BEGIN
    IF u^.space # VMFriends.NullSpace THEN
      IF freeSpace THEN TDState.DisposeTarget(u^.pid.i, u^.space); END;
      Ux.map[u^.space] := NIL;
      nHandles := TPFriends.NHandles(u^.space);
      TPSpecial.DestroyAll(u^.space);
      IF freeSpace THEN
        breakAddr := GetBoundary(u, VM.Low);
        LOCK x DO
          i := 1;
          WHILE
            (idleSpaces[i].s # VMFriends.NullSpace) AND (i < maxIdleSpaces) DO
            INC(i);
          END;
          idle :=
            (nHandles = 1) AND (breakAddr <= maxIdleBreakAddr)
            AND (i <= maxIdleSpaces)
            AND (idleSpaces[i].s = VMFriends.NullSpace);
          (* ***** Really need count of unmapped, readonly pages. *)
          IF idle THEN
            VMFriends.MakePagesReadWrite(u^.space, TextPage,
              (MIN(u^.a^.firstWritableAddr, breakAddr) - TextAddr) DIV
                VMBytesPerPage); (* round size down *)
            WITH idleSpaces[i] DO s := u^.space; b := breakAddr; END;
          END;
        END;
        IF NOT idle THEN
          ok := VMFriends.FreeSpace(u^.space);
          ASSERT(ok, "UxAS.Dispose confused");
        END;
      END;
    ELSIF NOT ThreadsPort.Equal(u^.a^.currentThread, nullTHandle) THEN
      TPSpecial.Destroy(u^.a^.currentThread);
    END;
  END Dispose;

PROCEDURE Get(
    u: Ux.T;
    addr: Addr;
    VAR v: ARRAY OF System.Byte;
    offset: CARDINAL := 0;
    count: CARDINAL := System.MaxCard)
  RAISES {SBuf.Fault};
  VAR actualCount: INTEGER;
  BEGIN
    actualCount := MIN(count, NUMBER(v) - offset);
    IF actualCount > 0 THEN
      IF NOT VMFriends.ReadBlock(u^.space, addr, System.Adr(v[offset]),
                actualCount) THEN
        RAISE(SBuf.Fault);
      END;
    END;
  END Get;
```

```
PROCEDURE GetArrayOfText(
    u: Ux.T;
    addr: Addr)
    : Text.RefArray
  RAISES {SBuf.Fault};
  VAR
    t: Text.RefArray;
    i, bufPos, bufCount: INTEGER;
    addr1: Addr;
    buf: ARRAY [0 .. 31] OF Addr;
  PROCEDURE NextBufPos();
    BEGIN
      INC(bufPos);
      IF bufPos >= bufCount THEN
        bufPos := 0;
        INC(addr1, bufCount * System.TByteSize(Addr));
        TRY
          Get(u, addr1, buf);
          bufCount := NUMBER(buf);
        EXCEPT
          SBuf.Fault: Get(u, addr, buf[0]); bufCount := 1;
        END;
      END;
    END NextBufPos;
  BEGIN
    IF addr = 0 THEN
      NEW(t, 0);
    ELSE
      addr1 := addr;
      bufCount := 0;
      bufPos := -1;
      i := 0;
      LOOP NextBufPos(); IF buf[bufPos] = 0 THEN EXIT; END; INC(i); END;
      NEW(t, i);
      IF addr1 # addr THEN addr1 := addr; bufCount := 0; END;
      bufPos := -1;
      FOR i := 0 TO HIGH(t^) DO
        NextBufPos();
        t^[i] := GetText(u, buf[bufPos]);
      END;
    END;
    RETURN t;
  END GetArrayOfText;

PROCEDURE GetClientPriority(u: Ux.T): TPFriends.Priority RAISES {};
  VAR
    space: CARDINAL;
    maxPriority: TPFriends.Priority;
    handle: ThreadsPort.Handle;
    state: TPSpecial.State;
  BEGIN
    space := u^.space;
    maxPriority := FIRST(TPFriends.Priority);
(*  TPSpecial.StopAll(space); *)
    handle := nullTHandle;
    LOOP
      handle := TPFriends.NextHandleInSpace(space, handle);
      IF ThreadsPort.Equal(handle, nullTHandle) THEN EXIT; END;
      TPSpecial.GetState(handle, state);
      maxPriority := MAX(maxPriority, state.priority);
    END;
(*  TPSpecial.StartAll(space); *)
    RETURN maxPriority;
  END GetClientPriority;
```

```
PROCEDURE GetClientTime(
    u: Ux.T;
    alreadyStopped: BOOLEAN := FALSE)
    : Time.T
  RAISES {};
  VAR
    space: CARDINAL;
    handle: ThreadsPort.Handle;
    time, totalTime: Time.T;
    us: CARDINAL;
  BEGIN
    space := u^.space;
    totalTime.seconds := 0;
    totalTime.microseconds := 0;
(*  IF NOT alreadyStopped THEN TPSpecial.StopAll(space); END; *)
    handle := nullTHandle;
    LOOP
      handle := TPFriends.NextHandleInSpace(space, handle);
      IF ThreadsPort.Equal(handle, nullTHandle) THEN EXIT; END;
      TPFriends.GetCPUTime(handle, time);
      ASSERT((time.seconds >= 0) AND (time.microseconds >= 0),
        "UxAS.GetClientTime: Negative CPU time (bad FPU chip?)");
      (* totalTime := Time.Add(totalTime, time); *)
      us := totalTime.microseconds + time.microseconds;
      INC(totalTime.seconds, time.seconds);
      IF us >= 1000000 THEN
        totalTime.microseconds := us - 1000000;
        INC(totalTime.seconds, 1);
      ELSE
        totalTime.microseconds := us;
      END;
    END;
(*  IF NOT alreadyStopped THEN TPSpecial.StartAll(space); END; *)
    RETURN totalTime;
  END GetClientTime;

PROCEDURE GetInfo(
    u: Ux.T;
    VAR (*out*) mappedBytes, residentBytes: CARDINAL)
  RAISES {};
  VAR spc: VMFriends2.SegmentPageCounts;
  BEGIN
    IF NOT VMFriends2.GetSegmentPageCounts(u^.space, VM.Low, spc) THEN
      ASSERT(FALSE, "I have a hold on a bad space");
    END;
    mappedBytes := spc.created * VMBytesPerPage;
    residentBytes := spc.resident * VMBytesPerPage;
    IF NOT VMFriends2.GetSegmentPageCounts(u^.space, VM.High, spc) THEN
      ASSERT(FALSE, "I have a hold on a bad space");
    END;
    INC(mappedBytes, spc.created * VMBytesPerPage);
    INC(residentBytes, spc.resident * VMBytesPerPage);
  END GetInfo;

PROCEDURE GetRLimit(
    u: Ux.T;
    resource: UxTypes.Resource;
    VAR (*out*) rLimit: UxTypes.RLimit)
  RAISES {};
  VAR a: T;
  BEGIN
    a := u^.a;
    CASE resource OF
    | UxTypes.rLimitData: rLimit := a^.rLimData;
    | UxTypes.rLimitStack: rLimit := a^.rLimStack;
    | UxTypes.rLimitCore: rLimit := a^.rLimCore;
    END;
  END GetRLimit;

PROCEDURE GetText(u: Ux.T; addr: Addr): Text.T RAISES {SBuf.Fault};
  VAR
    bufPos, bufCount: INTEGER;
    buf: ARRAY [0 .. 127] OF CHAR; (* ASSERT(NUMBER(buf) <= VMPageSize *)
    t: Text.T;
  BEGIN
    IF addr = 0 THEN
      RETURN NIL;
    ELSE
      t := NIL;
      LOOP
        bufCount := NUMBER(buf);
        IF NOT VMFriends.ReadBlock(u^.space, addr, System.Adr(buf), bufCount)
        THEN
          IF addr MOD VMBytesPerPage + NUMBER(buf) <= VMBytesPerPage THEN
            RAISE(SBuf.Fault);
          END;
          bufCount := VMBytesPerPage - (addr MOD VMBytesPerPage);
          IF NOT VMFriends.ReadBlock(u^.space, addr, System.Adr(buf), bufCount)
          THEN
            RAISE(SBuf.Fault);
          END;
        END;
        bufPos := 0;
        WHILE (bufPos < bufCount) AND (buf[bufPos] # 0C) DO INC(bufPos) END;
        IF t = NIL THEN (* the first buffer *)
          t := Text.FromSub(buf, 0, bufPos);
        ELSE (* text comes from multiple buffers *)
          t := Text.Cat(t, Text.FromSub(buf, 0, bufPos));
        END;
        IF (bufPos < bufCount) THEN EXIT END;
        INC(addr, bufCount);
      END;
      RETURN t;
    END;
  END GetText;

PROCEDURE HideThread(u: Ux.T) RAISES {};
  VAR a: T; state: TPSpecial.State;
  BEGIN
    a := u^.a;
    TPSpecial.GetState(a^.currentThread, state);
    state.space := osSpace;
    TPSpecial.SetState(a^.currentThread, state);
  END HideThread;

PROCEDURE Init() RAISES {};
  VAR i: CARDINAL;
  BEGIN
    ASSERT(NOT initialized, "UxAS.Init: called twice");
    initialized := TRUE;

    ASSERT(
      (Hardware.FPReg = Hardware.SPReg - 1)
      AND (Hardware.SPReg = Hardware.PCReg - 1)
      AND (Hardware.PCReg = LAST(Hardware.RegNum)),
      "UxAS.Init: Unexpected register numbers");
```

```
    (* ASSERT(VMBytesPerPage = VM.PagesToBytes(1)); *)
    VMBytesPerPage := VM.PagesToBytes(1);

    ASSERT(TextAddr MOD VMBytesPerPage = 0, "UxAS.Init: bad TextAddr");
      (* if not TextAddr = 0 *)
    ASSERT(
      System.TByteSize(UxTypes.CoreFileHeader) = TPSpecial.PreferredStackHole,
      "UxAS.Init: bad UxTypes.CoreFileHeader");
    initialStackAddr :=
      HoleAddr - TPSpecial.PreferredStackSize - VMBytesPerPage;
    (* extra page will be unmapped by Modula-2+ runtime *)

    nullTHandle := ThreadsPort.Null();

    osSpace := TPFriends.GetSpace();

    handlerLinkage[0] := 005af04fbH; (*         calls $4,.+5 ;helper    *)
    handlerLinkage[1] := 0008b8fbcH; (*         chmk  $139              *)
    handlerLinkage[2] := 0fa007f02H; (*         rei                     *)
    (*                                  helper: .word 0x7f ;r0-r6       *)
    (*                                          callg ...               *)
    handlerLinkage[3] := 00410bc6cH; (*           ... (ap),*10(ap)      *)
    (*                                          ret                     *)

    Thread.InitMutex(x);
    FOR i := 1 TO MaxIdleSpaces DO
      idleSpaces[i].s := VMFriends.NullSpace;
    END;
    maxIdleSpaces := 0; (* ***** tune this *)
    maxIdleBreakAddr := 90000; (* ***** and this *)

    UxPro.Register(UxTypes.SYSgetpagesize, UGetPageSize);
    UxPro.Register(UxTypes.SYSreservetraps, UReserveTraps); (* Ux only *)
    UxPro.Register(UxTypes.SYSsbreak, USBreak);
    UxPro.Register(UxTypes.SYSvadvise, UVAdvise);
  END Init;

PROCEDURE IsTopaz(u: Ux.T): BOOLEAN RAISES {};
  BEGIN RETURN (1 < TPFriends.NHandles(u^.space)); END IsTopaz;

PROCEDURE Load(
    u: Ux.T;
    file: OS.File;
    dir: OS.Dir;
    euser: User.T;
    name: Text.T;
    argv, envp: Text.RefArray;
    uArgs: Ux.T;
    argva, envpa: Addr;
    vFork: BOOLEAN)
    : BOOLEAN
  RAISES {OS.Error, SBuf.Fault, Thread.Alerted};
  VAR
    a: T; (* address space of u *)
    argsFetched: BOOLEAN;
    buf: ARRAY [0 .. 511] OF CHAR;
    bufCount, bufIndex: CARDINAL;
    e: UxTypes.Exec; (* first few bytes from file *)
    extraCount: INTEGER; (* > 0 if named interpreter is found *)
    extras: Text.RefArray; (* [name [, arg]] *)
  PROCEDURE IsADotOut(VAR mode: OS.FileMode; VAR owner: Text.T): BOOLEAN;
    (* Reads first few bytes of file and sets, mode, owner, and e. If file
       appears to contain an executable object program (i.e. an a.out file),
       returns TRUE and leaves file positioned to the beginning of the text
       segment. *)
    VAR headerSize: CARDINAL; fileInfo: OS.FileInfo;
    BEGIN
      OS.GetInfo(file, fileInfo, (*returnNames:*) TRUE);
      mode := fileInfo.mode;
      owner := fileInfo.owner;
      bufCount := UxFile.ReadF(file, SBuf.New(buf));
      bufIndex := 0;
      System.Copy(System.Adr(buf), System.Adr(e), System.ByteSize(e));
      IF (bufCount < System.ByteSize(e))
         OR (e.magic # UxTypes.ZMagic) AND (e.magic # UxTypes.NMagic)
            AND (e.magic # UxTypes.OMagic) THEN
        RETURN FALSE;
      END;
      headerSize := System.ByteSize(e);
      IF e.magic = UxTypes.ZMagic THEN
        headerSize := UltrixRoundUp(headerSize);
        e.text := UltrixRoundUp(e.text);
        e.data := UltrixRoundUp(e.data);
      END;
      IF OS.Seek(file, headerSize) = 0 THEN END;

      (* Check that lengths in header are consistent with length of file. *)
      IF fileInfo.length < headerSize + e.text + e.data THEN
        CloseQuietly(file);
        RAISE(OS.Error, OS.ShortExecutableEC);
      END;
      RETURN TRUE;
    END IsADotOut;
  PROCEDURE FetchArgs;
    BEGIN
      IF NOT argsFetched THEN
        IF argv = NIL THEN
          argv := GetArrayOfText(uArgs, argva);
          envp := GetArrayOfText(uArgs, envpa);
        END;
        IF NUMBER(argv^) = 0 THEN NEW(argv, 1); argv^[0] := name; END;
        argsFetched := TRUE;
      END;
    END FetchArgs;
  PROCEDURE OpenInterpreter;
    (* File is assumed to be open to an executable file other than an object
       program. If its first line is of the form:
       '#![blanks]interpreter[blanks[args]]' and the named interpreter appears
       to contain an executable object program, we use it. Otherwise we raise
       ENOENT, ENOEXEC, or EACCES as appropriate. *)
    VAR
      c: CHAR; (* look ahead *)
      w: WrV.T; (* gathers interpreter name, arg *)
      iname, iarg: Text.T; (* interpreter name, arg *)
      mode: OS.FileMode;
      owner: Text.T;
    PROCEDURE GetChar(): CHAR;
      BEGIN
        IF bufIndex = bufCount THEN
          bufCount := UxFile.ReadF(file, SBuf.New(buf));
          bufIndex := 0;
        END;
        IF bufCount = 0 THEN RAISE(OS.Error, OS.BadExecutableEC); END;
        INC(bufIndex);
        RETURN (buf[bufIndex - 1]);
      END GetChar;
    BEGIN
      TRY
```

```
      IF (GetChar() # "#") OR (GetChar() # "!") THEN
        RAISE(OS.Error, OS.BadExecutableEC);
      END;
      c := GetChar(); (* prime the look ahead *)
      WHILE c = " " DO c := GetChar(); END;
      WrV.New(w);
      WHILE (c # "\n") AND (c # " ") DO
        WrV.PutChar(w, c);
        c := GetChar();
      END;
      iname := WrV.ToText(w); (* interpreter *)
      WHILE c = " " DO c := GetChar(); END;
      WHILE c # "\n" DO WrV.PutChar(w, c); c := GetChar(); END;
      (* Ultrix seems to include blanks in and after arg *)
    FINALLY
      CloseQuietly(file);
    END;
    iarg := WrV.ToText(w); (* arg *)
    file := UxFile.OpenForLoading(u, dir, iname, euser);
    IF IsADotOut(mode, owner) THEN (* ignore this mode, owner *)
      IF Text.IsEmpty(iarg) THEN extraCount := 1; ELSE extraCount := 2; END;
      NEW(extras, extraCount);
      FetchArgs;
      extras^[0] := argv^[0];
      argv^[0] := name;
      IF extraCount = 2 THEN extras^[1] := iarg; END;
    ELSE
      CloseQuietly(file);
      RAISE(OS.Error, OS.BadExecutableEC);
    END;
  END OpenInterpreter;
PROCEDURE LoadChunk(addr: Addr; length: CARDINAL);
  (* Copies length bytes from file to address addr in address space of u. *)
  VAR n: CARDINAL;
  BEGIN
    n := UxFile.ReadF(file, SBuf.SNew(u^.space, addr, length));
    IF n # length THEN RAISE(OS.Error, OS.ShortExecutableEC); END;
  END LoadChunk;
VAR
  zero: ARRAY [0 .. 6] OF CHAR;
  mode: OS.FileMode;
  owner: Text.T;
  user: User.T;
  argc: CARDINAL; (* number of argument strings (including extras) *)
  i: INTEGER; (* loop index *)
  sumArgSizes, sumEnvSizes, padSize: CARDINAL; (* in bytes *)
  dataAddr, bssAddr, minBreakAddr, breakAddr, addr, stringAddr, stackAddr:
    Addr; (* user-space addresses in u^.space *)
  execVE: BOOLEAN; (* TRUE iff execve not following vfork *)
  state: TPSpecial.State;
  handle: ThreadsPort.Handle; (* to strace *)
BEGIN

  System.Zero(System.Adr(zero), System.ByteSize(zero));

  a := u^.a;

  argsFetched := FALSE;

  extraCount := 0; (* assume no interpreter name, arg *)

  (* Set file (leaving it positioned ready to read the text portion),
     length, e, extraCount, extras, mode, and owner. *)
  IF NOT IsADotOut(mode, owner) THEN OpenInterpreter; END;
```

```
TRY (* FINALLY *)

  FetchArgs;

  (* Lay out the new address space. For details see Ultrix Programmer's
     manuals, sections execve(2) and a.out(5), plus the memo "Ultrix User/
     Kernel Interface" by Paul McJones. *)

  dataAddr := TextAddr + e.text;

  IF e.magic = UxTypes.NMagic THEN
    dataAddr := UltrixRoundUp(dataAddr);
  END;

  bssAddr := dataAddr + e.data;
  breakAddr := VMRoundUp(bssAddr + e.bss);

  argc := extraCount + NUMBER(argv^);

  sumArgSizes := 0;
  FOR i := 0 TO extraCount - 1 DO
    INC(sumArgSizes, Text.Length(extras^[i]) + 1);
  END;
  FOR i := 0 TO NUMBER(argv^) - 1 DO
    INC(sumArgSizes, Text.Length(argv^[i]) + 1);
  END;

  sumEnvSizes := 0;
  FOR i := 0 TO NUMBER(envp^) - 1 DO
    INC(sumEnvSizes, Text.Length(envp^[i]) + 1);
  END;

  padSize := (4 - (sumArgSizes + sumEnvSizes) MOD 4) MOD 4;

  stackAddr :=
    HoleAddr -
      (4 + 4 * argc + 4 + 4 * NUMBER(envp^) + 4 + sumArgSizes +
         sumEnvSizes + padSize + 4);

  (* Make sure neither low nor high segment is too large. *)
  IF (Addr(a^.rLimData.cur) < breakAddr)
     OR (stackAddr < Addr(HoleAddr - TPSpecial.PreferredStackSize)) THEN
    RAISE(OS.Error, OS.NotEnoughVMEC);
  END;

  (* This is the "point of no return" as far as concerns the Ultrix
     process which performed the execve. From here on, we will not raise
     exceptions. *)

  TRY (* EXCEPT *)

    execVE := FALSE;
    IF u^.space = VMFriends.NullSpace THEN (* StartProcess *)
      AllocThread(u);
      AllocSpace(u, bssAddr, breakAddr, initialStackAddr);
    ELSE (* ExecVE *)
      IF vFork THEN (* following VFork *)
        (* Serialize w.r.t. GetProcessInfo from another process *)
        UxPro.AcquireProcess(u);
        AllocSpace(u, bssAddr, breakAddr, initialStackAddr);
        UxPro.ReleaseProcess(u);
      ELSE (* ExecVE not following VFork *)
        execVE := TRUE;
```

```
    TDState.PreExec(u^.pid.i, u^.space);

    (* [Re]map the first page in case it had been unmapped to detect
       dereferences of NIL pointers. *)
    VMFriends.MapPages(u^.space, 0, 1);

    minBreakAddr := MIN(GetBoundary(u, VM.Low), breakAddr);
    SetBoundary(u^.space, VM.Low, breakAddr);

    (* Make writable again any pages that had been made read-only to
       detect wild stores into the text. *)
    VMFriends.MakePagesReadWrite(u^.space, TextPage,
      (MIN(a^.firstWritableAddr, minBreakAddr) - TextAddr +
         VMBytesPerPage - 1) DIV VMBytesPerPage);

    IF bssAddr < minBreakAddr THEN
      (* We only need to zero from bssAddr to minBreakAddr; any new
         pages will be zeroed by VM. *)
      SBuf.Zero(SBuf.SNew(u^.space, bssAddr, minBreakAddr - bssAddr));
    END;
    SetBoundary(u^.space, VM.High, initialStackAddr);
  END;
END;

(* Set up the low segment, which contains the code and data. *)

(* Copy text to u^.space. *)
LoadChunk(TextAddr, e.text);

(* Copy initialized data to u^.space. Note that if e.magic =
   UxTypes.NMagic, it is not guaranteed that dataAddr = TextAddr+
   e.text. *)
LoadChunk(dataAddr, e.data);

IF (e.magic = UxTypes.ZMagic) OR (e.magic = UxTypes.NMagic) THEN
  (* Write-protect up to start of data. *)
  a^.firstWritableAddr := dataAddr;
  VMFriends.MakePagesReadOnly(u^.space, TextPage,
    (a^.firstWritableAddr - TextAddr) DIV VMBytesPerPage);
  (* round size down *)
ELSE (* e.magic = UxTypes.OMagic *)
  a^.firstWritableAddr := 0;
END;

(* Set up the stack. *)

addr := stackAddr;

(* Copy argument count longword. *)
Put(u, addr, argc, 0, 4);
INC(addr, 4);

(* Copy pointers to argument strings. *)
stringAddr := addr + Addr(4 * argc + 4 + 4 * NUMBER(envp^) + 4);
FOR i := 0 TO extraCount - 1 DO
  Put(u, addr, stringAddr, 0, 4);
  INC(addr, 4);
  INC(stringAddr, Text.Length(extras^[i]) + 1);
END;
FOR i := 0 TO NUMBER(argv^) - 1 DO
  Put(u, addr, stringAddr, 0, 4);
  INC(addr, 4);
  INC(stringAddr, Text.Length(argv^[i]) + 1);
END;
Put(u, addr, zero, 0, 4);
INC(addr, 4);

(* Copy pointers to environment strings. *)
FOR i := 0 TO NUMBER(envp^) - 1 DO
  Put(u, addr, stringAddr, 0, 4);
  INC(addr, 4);
  INC(stringAddr, Text.Length(envp^[i]) + 1);
END;
Put(u, addr, zero, 0, 4);
INC(addr, 4);

(* Copy null-terminated argument strings. *)
FOR i := 0 TO extraCount - 1 DO
  INC(addr, PutText(u, addr, extras^[i]));
  Put(u, addr, zero, 0, 1);
  INC(addr, 1);
END;
FOR i := 0 TO NUMBER(argv^) - 1 DO
  INC(addr, PutText(u, addr, argv^[i]));
  Put(u, addr, zero, 0, 1);
  INC(addr, 1);
END;

(* Copy null-terminated environment strings. *)
FOR i := 0 TO NUMBER(envp^) - 1 DO
  INC(addr, PutText(u, addr, envp^[i]));
  Put(u, addr, zero, 0, 1);
  INC(addr, 1);
END;

(* Copy padding (to longword boundary) and final zero longword. *)
Put(u, addr, zero, 0, padSize + 4);
ASSERT(addr + padSize + 4 = HoleAddr, "UxAS.Load confused");

(* Set up thread state. The minimum expected by /lib/crt0.o (the first
   code executed in a normal Unix program) is that PC=2+address of
   entry point and SP=address of the arg count longword on the stack.
   Zeroing FP gives Loupe a reliable way of determining that there is
   no previous frame when the strace feature is used. Zeroing the
   other registers helps avoid nondeterministic results from faulty
   programs and may ease debugging. Note that if we are being called
   as part of an execve kernel call, a zero result code will also
   later be stored in register 0. See the ASSERT in Init regarding
   register numbering. *)
TPSpecial.GetState(a^.currentThread, state);
state.machineState.regs[Hardware.PCReg].val := e.entry + 2;
state.machineState.regs[Hardware.SPReg].val := stackAddr;
System.Zero(System.Adr(state.machineState.regs[0]),
  4 * (Hardware.FPReg + 1));
TPSpecial.SetState(a^.currentThread, state);

(* If file being executed has the appropriate mode, its owner becomes
   the effective user. *)
IF (OS.SetUIDonExec IN mode.fileState) THEN
  user := User.LookupUserByName(owner);
  IF user # NIL THEN UxPro.SetEUser(u, user); END;
END;

(* If the straceExec flag is set, clear it and cause the process to
   appear to have gotten a "help" trap right after the execve. *)
IF a^.straceExec # OSFriends.STNever THEN
  handle := a^.currentThread;
  a^.continuation := Nothing;
```

```
        IF a^.straceExec = OSFriends.STOnce THEN
          a^.straceExec := OSFriends.STNever;
        END;
      ELSE
        handle := nullTHandle;
      END;
      IF vFork THEN
        TDState.NewTarget(u^.pid.i, u^.space, handle);
      ELSIF execVE THEN
        TDState.PostExec(u^.pid.i, u^.space, handle);
      END;
    EXCEPT
      SBuf.Fault, OS.Error:
        IF vFork AND (u^.space # VMFriends.NullSpace) THEN
          TDState.NewTarget(u^.pid.i, u^.space, nullTHandle);
        END;
        RETURN FALSE;
    END;
  FINALLY
    CloseQuietly(file);
  END;
  RETURN TRUE; (* success *)
END Load;

PROCEDURE MakeCoreImage(
    u: Ux.T;
    sig: OS.Signal)
  RAISES {OS.Error, Thread.Alerted};
  VAR space: VMFriends.Space; f: OS.File;
  PROCEDURE W(p: CARDINAL; v: UNSIGNED; l: CARDINAL := 4);
    (* Write l bytes of v at position p in f. *)
    BEGIN
      IF OS.Seek(f, p) = 0 THEN END;
      UxFile.WriteF(f, SBuf.New(v, 0, l));
    END W;
  VAR pos: CARDINAL; (* position in f, for use with Seek *)
  PROCEDURE CarefullyWrite(addr, limitAddr: Addr);
    (* Writes bytes from addr through limitAddr-1 of space to f, substituting
       zeros for unmapped or undefined pages. Maintains pos. *)
    TYPE StateSet = SET OF VM.Contents;
    CONST Unreal = StateSet{VM.Nonexistent, VM.Undefined};
    VAR pageState: VM.PageState; runAddr: Addr; count: CARDINAL;
    BEGIN
      WHILE addr < limitAddr DO
        (* Although the page counts in the core file format are defined in
           terms of hardware pages, we can only ask VM about its pages. To
           keep things simple, we ask about each hardware page in a VM page.
        *)
        pageState :=
          VMFriends.StatePage(space, addr DIV VMBytesPerPage);
        IF pageState.contents IN Unreal THEN
          (* Seek forward one page, leaving zeros. *)
          INC(addr, Hardware.BytesPerPage);
          INC(pos, Hardware.BytesPerPage);
          IF OS.Seek(f, pos) = 0 THEN END;
        ELSE
          (* Write run of mapped pages. *)
          runAddr := addr;
          LOOP
            INC(addr, Hardware.BytesPerPage);
            IF addr = limitAddr THEN EXIT; END;
            pageState :=
              VMFriends.StatePage(space, addr DIV Hardware.BytesPerPage);
            IF pageState.contents IN Unreal THEN EXIT; END;
          END;
          count := addr - runAddr;
          UxFile.WriteF(f, SBuf.SNew(space, runAddr, count));
          INC(pos, count);
        END;
      END;
    END CarefullyWrite;
  VAR
    a: T;
    count: CARDINAL;
    i: CARDINAL;
    dataAddr, breakAddr, stackAddr: Addr;
    dataPages, stackPages: CARDINAL; (* of Hardware.BytesPerPage *)
    tME: UxTypes.CoreFileThreadMapEntry;

  BEGIN
    a := u^.a;
    space := u^.space;
    IF space = VMFriends.NullSpace THEN RETURN; END;
    breakAddr := GetBoundary(u, VM.Low);
    dataAddr := MIN(a^.firstWritableAddr, breakAddr);
    stackAddr := GetBoundary(u, VM.High);

    dataPages := (breakAddr - dataAddr) DIV Hardware.BytesPerPage;
    stackPages := (HoleAddr - stackAddr) DIV Hardware.BytesPerPage;

    count :=
      System.TByteSize(UxTypes.CoreFileHeader) +
        Hardware.BytesPerPage * (dataPages + stackPages) +
        System.ByteSize(tME) * TPFriends.NHandles(space);
    IF a^.rLimCore.cur < count THEN
      (* Raising any error clears core dump bit in wait status. *)
      RAISE(OS.Error, OS.NotEnoughRoomEC);
    END;

    TPSpecial.GetState(a^.currentThread, tME.state);

    f := UxFile.OpenCoreFile(u);
    OS.SetSize(f, count);

    TRY (* FINALLY *)

      (* Write header. *)
      W(UxTypes.oPcbKsp, 7fffffffH); (* kernel stack pointer *)
      W(UxTypes.oPcbUsp, tME.state.machineState.regs[Hardware.SPReg].val);
      (* user stack pointer *)
      W(UxTypes.oStatus, ORD(sig), UxTypes.lStatus);
      W(UxTypes.oTextPages, a^.firstWritableAddr DIV Hardware.BytesPerPage);
      W(UxTypes.oDataPages, dataPages);
      W(UxTypes.oStackPages, stackPages);
      W(UxTypes.oAp, tME.state.machineState.regs[Hardware.APReg].val);
      W(UxTypes.oFp, tME.state.machineState.regs[Hardware.FPReg].val);
      FOR i := 0 TO Hardware.APReg - 1 DO
        W(UxTypes.oRegs + 4 * i, tME.state.machineState.regs[i].val);
      END;
      W(UxTypes.oSp, tME.state.machineState.regs[Hardware.SPReg].val);
      W(UxTypes.oPc, tME.state.machineState.regs[Hardware.PCReg].val);
      W(UxTypes.oPsw, LOOPHOLE(tME.state.machineState.status, UNSIGNED));
      pos := System.TByteSize(UxTypes.CoreFileHeader);
      IF OS.Seek(f, pos) = 0 THEN END;

      (* Write data pages. *)
      count := breakAddr - dataAddr;
      TRY
```

```
        UxFile.WriteF(f, SBuf.SNew(space, dataAddr, count));
        INC(pos, count);
      EXCEPT
        SBuf.Fault: CarefullyWrite(dataAddr, breakAddr);
      END;

      (* Write stack pages, watching for unmapped pages. Recall core(5) is
         defined in terms of hardware pages. *)
      CarefullyWrite(stackAddr, HoleAddr);
      (* After this point, pos is not maintained. *)

      (* Dump thread states, starting with a^.currentThread. *)
      tME.handle := a^.currentThread;
      UxFile.WriteF(f, SBuf.New(tME));
      tME.handle := nullTHandle;
      LOOP
        tME.handle := TPFriends.NextHandleInSpace(space, tME.handle);
        IF ThreadsPort.Equal(tME.handle, nullTHandle) THEN EXIT; END;
        IF NOT ThreadsPort.Equal(tME.handle, a^.currentThread) THEN
          TPSpecial.GetState(tME.handle, tME.state);
          UxFile.WriteF(f, SBuf.New(tME));
        END;
      END;
    FINALLY
      OS.Close(f);
    END;
  END MakeCoreImage;

PROCEDURE New(u, uNew: Ux.T) RAISES {};
  VAR a: T;
  BEGIN
    NEW(uNew^.a);
    uNew^.space := VMFriends.NullSpace; (* see Copy, Load *)
    WITH uNew^.a^ DO
      traps := MyTraps;
      handlerThread := nullTHandle;
      currentThread := nullTHandle;
      IF u = NIL THEN (* first process *)
        straceExec := OSFriends.STNever; straceFork := OSFriends.STNever;
        rLimData.cur := initRLimDataCur;
        rLimData.max := initRLimDataMax;
        rLimStack.cur := initRLimStackCur;
        rLimStack.max := initRLimStackMax;
        rLimCore.cur := initRLimCoreCur;
        rLimCore.max := initRLimCoreMax;
      ELSE (* subsequent processes *)
        a := u^.a;
        straceExec := a^.straceExec; straceFork := a^.straceFork;
        rLimData := a^.rLimData;
        rLimStack := a^.rLimStack;
        rLimCore := a^.rLimCore;
      END;
    END;
  END New;

PROCEDURE NewTarget(u, uNew: Ux.T) RAISES {};
  VAR handle: ThreadsPort.Handle;
  BEGIN
    IF uNew^.a^.straceExec # OSFriends.STNever THEN
      handle := u^.a^.currentThread;
    ELSE
      handle := nullTHandle;
    END;
    TDState.NewTarget(uNew^.pid.i, uNew^.space, handle);

(* Something like this would be necessary for straceFork.
   But if the parent has done a StartProcess, continuation is irrelevant.
    IF u^.a^.straceFork # OSFriends.STNever THEN
      IF u^.a^.straceFork = OSFriends.STOnce THEN
        u^.a^.straceFork = OSFriends.STNever;
      END;
      TDState.PostFork(u^.pid.i, u^.space, u^.currentThread,
        uNew^.pid.i, uNew^.space);
      u^.a^.continuation := Nothing;
    END;
*)
  END NewTarget;

PROCEDURE PushHandlerCall(
    u: Ux.T;
    signal: OS.Signal;
    code: INTEGER;
    handler: UxTypes.SignalHandler;
    mask: OS.Signals;
    onStack: UxTypes.SignalOnStack;
    switchStacks: BOOLEAN;
    sp: Addr)
  RAISES {SBuf.Fault};
  (* To avoid surprising UserTTD, this procedure need to run under a new mutex
     shared by Taos and UserTTD. *)
  VAR a: T; state: TPSpecial.State;
  PROCEDURE Push(w: System.Word); (* onto stack of u *)
    VAR sp: Addr;
    BEGIN
      sp := state.machineState.regs[Hardware.SPReg].val - 4;
      TRY
        Put(u, sp, w);
      EXCEPT
        SBuf.Fault:
          IF GrowStack(u, sp) THEN Put(u, sp, w); ELSE RAISE(SBuf.Fault); END;
      END;
      state.machineState.regs[Hardware.SPReg].val := sp;
    END Push;
  VAR scp: Addr; (* SigContext pointer *)
  BEGIN
    a := u^.a;
    TPSpecial.GetState(a^.currentThread, state);
    IF a^.continuation = Resume THEN
      (* Make sure correct carry bit is pushed in SigContext. *)
      state.machineState.status.cc.c := a^.sState.error;
    END;
    TRY
      (* Push SigContext on current stack, and save pointer in scp. *)
      Push(state.machineState.status); (* PSL *)
      Push(state.machineState.regs[Hardware.PCReg].val);
      Push(state.machineState.regs[Hardware.SPReg].val); (* = .+4 *)
      Push(mask);
      Push(onStack);
      scp := state.machineState.regs[Hardware.SPReg].val;

      (* If appropriate, switch to signal stack. *)
      IF switchStacks THEN
        state.machineState.regs[Hardware.SPReg].val := sp;
      END;

      Push(scp); (* argument for subsequent 'chmk $139' kernel call *)

      Push(handler); (* real handler entry point for handler linkage routine *)
```

```
      Push(scp); (* handler arguments ... *)
      Push(code); (* in case SIGFPE or SIGILL *)
      Push(signal); (* ... rightmost through leftmost *)

      (* Set program counter to handler linkage routine. *)
      state.machineState.regs[Hardware.PCReg].val := HandlerEntry;

      (* Clean out PSL. *)
      state.machineState.status.cc.c := FALSE;
      state.machineState.status.cc.v := FALSE;
      state.machineState.status.cc.z := FALSE;
      state.machineState.status.cc.n := FALSE;
      (* t *)
      (* iv, fu, dv *)
      state.machineState.status.fpd := FALSE; (* important! *)
      IF a^.continuation = Resume THEN
        (* Make sure correct carry bit is passed to ResumeKernelCaller. *)
        a^.sState.error := state.machineState.status.cc.c;
      END;
    FINALLY
      TPSpecial.SetState(a^.currentThread, state);
    END;
  END PushHandlerCall;

PROCEDURE Put(
    u: Ux.T;
    a: Addr;
    VAR v: ARRAY OF System.Byte;
    offset: CARDINAL := 0;
    count: CARDINAL := System.MaxCard)
  RAISES {SBuf.Fault};
  VAR actualCount: INTEGER;
  BEGIN
    actualCount := MIN(count, NUMBER(v) - offset);
    IF actualCount > 0 THEN
      IF (a < u^.a^.firstWritableAddr)
         OR NOT VMFriends.WriteBlock(u^.space, a, System.Adr(v[offset]),
                  actualCount) THEN
        RAISE(SBuf.Fault);
      END;
    END;
  END Put;

PROCEDURE PutText(
    u: Ux.T;
    a: Addr;
    t: Text.T;
    count: CARDINAL := System.MaxCard)
    : CARDINAL
  RAISES {SBuf.Fault};
  VAR buf: ARRAY [0 .. 127] OF CHAR; rd: RdV.T; n: CARDINAL; aNow: Addr;
  BEGIN
    count := MIN(count, Text.Length(t));
    RdV.FromText(rd, t);
    aNow := a;
    WHILE aNow - a < count DO
      n := RdV.GetSub(rd, buf);
      ASSERT(n # 0, "UxAS.PutText confused");
      Put(u, aNow, buf, 0, n);
      INC(aNow, n);
    END;
    RETURN count;
  END PutText;
```

```
PROCEDURE R0Result(
    u: Ux.T;
    result: INTEGER;
    error: BOOLEAN := FALSE)
  RAISES {};
  VAR a: T; state: TPSpecial.State;
  BEGIN
    a := u^.a;
    IF a^.continuation = Resume THEN
      a^.sState.r0Val.val := result;
    ELSE (* a^.continuation = Nothing *)
      ASSERT(a^.continuation = Nothing, "UxAS.R0Result confused");
      TPSpecial.GetState(a^.currentThread, state);
      state.machineState.regs[0].val := result;
      TPSpecial.SetState(a^.currentThread, state);
    END;
    IF error THEN SetError(u); END;
  END R0Result;

PROCEDURE R1Result(u: Ux.T; result: INTEGER) RAISES {};
  VAR a: T;
  BEGIN
    a := u^.a;
    ASSERT(a^.continuation = Resume, "UxAS.R1Result confused");
    a^.sState.r1Val.val := result;
    a^.sState.setR1 := TRUE;
  END R1Result;

PROCEDURE RestoreThread(u: Ux.T) RAISES {};
  VAR a: T; state: TPSpecial.State;
  BEGIN
    a := u^.a;
    TPSpecial.GetState(a^.currentThread, state);
    state.space := u^.space;
    TPSpecial.SetState(a^.currentThread, state);
  END RestoreThread;

PROCEDURE SetClientPriority(u: Ux.T; priority: TPFriends.Priority) RAISES {};
  (* To avoid surprising UserTTD, this procedure needs to run under the
     same mutex as PushHandlerCall (q.v.). *)
  VAR space: CARDINAL; handle: ThreadsPort.Handle; state: TPSpecial.State;
  BEGIN
    space := u^.space;
    TPSpecial.StopAll(space);
    handle := nullTHandle;
    LOOP
      handle := TPFriends.NextHandleInSpace(space, handle);
      IF ThreadsPort.Equal(handle, nullTHandle) THEN EXIT; END;
      TPSpecial.GetState(handle, state);
      state.priority := priority;
      TPSpecial.SetState(handle, state);
    END;
    TPSpecial.StartAll(space);
  END SetClientPriority;

PROCEDURE SetError(u: Ux.T) RAISES {};
  VAR a: T;
  BEGIN
    a := u^.a;
    ASSERT(a^.continuation = Resume, "UxAS.SetError confused");
    a^.sState.error := TRUE;
  END SetError;

PROCEDURE SetHandler(u: Ux.T) RAISES {OS.Error};
```

```
  (* ONLY FOR OLD OS *)
  VAR a: T;
  BEGIN
    a := u^.a;
    IF ThreadsPort.Equal(a^.handlerThread, nullTHandle) THEN
      a^.handlerThread := a^.currentThread;
    ELSIF NOT ThreadsPort.Equal(a^.handlerThread, a^.currentThread) THEN
      RAISE(OS.Error, OS.InvalidArgumentEC);
    END;
  END SetHandler;

PROCEDURE SetRLimit(
    u: Ux.T;
    resource: UxTypes.Resource;
    rLimit: UxTypes.RLimit)
  RAISES {};
  VAR a: T;
  BEGIN
    a := u^.a;
    IF resource # UxTypes.rLimitCore THEN
      rLimit.max := MIN(rLimit.max, BytesPerRegion);
      rLimit.cur := MIN(rLimit.cur, BytesPerRegion);
    END;
    CASE resource OF
    | UxTypes.rLimitData: a^.rLimData := rLimit;
    | UxTypes.rLimitStack: a^.rLimStack := rLimit;
    | UxTypes.rLimitCore: a^.rLimCore := rLimit;
    END;
  END SetRLimit;

PROCEDURE SetThreadPriority(
    thread: Thread.T;
    priority: TPFriends.Priority)
  RAISES {};
  VAR ok: BOOLEAN; handle: ThreadsPort.Handle; state: TPSpecial.State;
  BEGIN
    IF thread = Thread.Self() THEN
      IF 0 = ThreadFriends.SetPriority(priority) THEN END;
    ELSE
      ok := ThreadFriends.Stop(thread, handle);
      ASSERT(ok, "UxAS.SetThreadPriority confused");
      TPSpecial.GetState(handle, state);
      state.priority := priority;
      TPSpecial.SetState(handle, state);
      TPSpecial.Start(handle);
    END;
  END SetThreadPriority;

PROCEDURE StartAll(u: Ux.T) RAISES {};
  BEGIN
    TPSpecial.StartAll(u^.space);
    (* at least one more start required to get u^.a^.currentThread going *)
  END StartAll;

PROCEDURE StopAll(u: Ux.T) RAISES {};
  BEGIN RPCLocal.Exclude(u^.space, ExcludedStopAll, u); END StopAll;

PROCEDURE WaitEvent(u: Ux.T; VAR (*out*) event: Event) RAISES {SBuf.Fault};
  VAR a: T; handledHere: BOOLEAN; state: TPSpecial.State;
  PROCEDURE ClassifyTrap();
    VAR
      scp: Addr; (* pointer to SigContext in client address space *)
      sigContext: UxTypes.SignalContext;
    BEGIN
      event.kind := Exception; (* default *)
      CASE state.trapInfo.kind OF
      | TPSpecial.TKVmFailure: event.sig := OS.SigSegV;
      | TPSpecial.TKIllegalNubCall: (* = TKCHMKInstruction *)
          ASSERT(state.trapInfo.arg = ORD(UxTypes.SYShandlerreturn),
            "UxAS.WaitEvent confused");
          event.kind := HandlerReturn;

          (* Pop pointer to SigContext record and fetch record. *)
          Get(u, state.machineState.regs[Hardware.SPReg].val, scp);
          Get(u, scp, sigContext);

          (* Restore process state from SigContext record. *)
          event.mask := sigContext.mask;
          event.onStack := sigContext.onStack;
          state.machineState.regs[Hardware.SPReg].val := sigContext.sp;
          TPSpecial.SetState(a^.currentThread, state);
      | TPSpecial.TKFromDebugger:
          state.trapInfo := TDTraps.FromDebugger(a^.currentThread);
          (* ***** What does the TDState protocol say about the following: *)
          TPSpecial.SetState(a^.currentThread, state);
          ClassifyTrap();
      | TPSpecial.TKIllegalAddress:
          handledHere := GrowStack(u, state.trapInfo.addr);
          IF NOT handledHere THEN event.sig := OS.SigSegV; END;
      | TPSpecial.TKIllegalAccess: event.sig := OS.SigBus;
      | TPSpecial.TKArithmetic:
          event.sig := OS.SigFPE;
          event.code := ORD(state.trapInfo.code);
      | TPSpecial.TKReservedInstruction, TPSpecial.TKCompatibilityMode:
          event.sig := OS.SigIll;
          event.code := UxTypes.IllPrivinFault;
      | TPSpecial.TKReservedOperand:
          event.sig := OS.SigIll;
          event.code := UxTypes.IllResopFault;
      | TPSpecial.TKReservedAddressingMode:
          event.sig := OS.SigIll;
          event.code := UxTypes.IllResadFault;
          (* TKBreakpoint, TKTrace handled by debugger *)
      | TPSpecial.TKXFCInstruction: event.sig := OS.SigEMT;
      | TPSpecial.TKCHMEInstruction, TPSpecial.TKCHMSInstruction,
            TPSpecial.TKCHMUInstruction:
          event.sig := OS.SigSegV;
      |
      ELSE (* TKBadParameter or ? *)
          event.sig := OS.SigIll;
          event.code := UxTypes.IllPrivinFault;
      END; (* CASE trapKind *)
    END ClassifyTrap;
  VAR tState: TPSpecial.TState; ap: Addr;
  BEGIN

    a := u^.a;

    REPEAT (* UNTIL NOT handledHere *)
      handledHere := FALSE;

      (* Start the current thread and wait for some thread in u to trap or for
         us to be alerted. There is a potential race condition: we are
         alerted, but before we can stop the client thread, it traps itself.
         We detect this case and jiggle things so it looks like the trap came
         before the alert. This logic depends on the fact that with the
         current implementation of Thread, a given ThreadsPort.Handle stays in
         a single address space until that address space terminates. By
```

```
    remembering the initial ThreadsPort.Handle, we could relax this
    dependency.

    In any case, it is only single-threaded address spaces that are
    allowed to have Ultrix-style signal handlers; it is to prepare for
    such a handler that we must stop the address space. *)
  TRY
    IF a^.continuation = Resume THEN
      a^.currentThread :=
        TPSpecial.ResumeKernelCaller(a^.currentThread, a^.sState, u^.space,
          a^.traps, tState);
    ELSE
      IF a^.continuation = Nothing THEN
        (* It is highttd's responsibility to start this thread. *)
      ELSE (* a^.continuation = Start *)
        TPSpecial.Start(a^.currentThread);
      END;
      a^.currentThread := TPSpecial.GetKernelCaller(u^.space, a^.traps,
        tState);
    END;
    IF (tState.trapInfo.kind = TPSpecial.TKIllegalNubCall)
       AND (tState.trapInfo.arg # ORD(UxTypes.SYShandlerreturn)) THEN
      a^.continuation := Resume;
    ELSE
      a^.continuation := Start;
      TPSpecial.GetState(a^.currentThread, state);
    END;
  EXCEPT
    Thread.Alerted: (* asynchronous signal to handle *)
      RPCLocal.Exclude(u^.space, ExcludedStop, a);
      a^.continuation := Start;
      TPSpecial.GetState(a^.currentThread, state);
      tState.trapInfo := state.trapInfo;
      IF NOT tState.trapInfo.reported
         AND (tState.trapInfo.kind IN a^.traps) THEN
        TPSpecial.Start(a^.currentThread);
        Thread.Alert(Thread.Self()); (* pretend trap came first *)
        IF (tState.trapInfo.kind = TPSpecial.TKIllegalNubCall)
           AND (tState.trapInfo.arg # ORD(UxTypes.SYShandlerreturn)) THEN
          ap := state.machineState.regs[Hardware.APReg].val;
          TRY
            Get(u, ap, tState.arguments);
          EXCEPT
            SBuf.Fault:
              Get(u, ap, tState.arguments.n);
              Get(u, ap + 4, tState.arguments.a, 0, tState.arguments.n);
          END;
          tState.validArguments := TRUE;
          state.trapInfo.reported := TRUE;
          TPSpecial.SetState(a^.currentThread, state);
          a^.continuation := Resume;
        END;
      ELSE
        (* The thread does not have a trap waiting for us to handle. *)
        tState.trapInfo.kind := TPSpecial.TKNoTrap;
      END;
  END;

  IF a^.continuation = Resume THEN
    a^.sState.setR1 := FALSE;
    a^.sState.error := FALSE;
    IF NOT tState.validArguments THEN
      (* The C library routine signal(3) calls sigvec(2) with AP < SP,
         which worries ThreadsPort. Let's try to fetch the arguments
         ourselves. (In the case noted above, AP points to a zero argument
         count, so use the maximum. *)
        TPSpecial.GetState(a^.currentThread, state);
        Get(u, state.machineState.regs[Hardware.APReg].val,
          tState.arguments);
      END;
      event.kind := KernelCall;
      IF (ORD(FIRST(UxTypes.KernelEntry)) <= tState.trapInfo.arg)
         AND (tState.trapInfo.arg <= ORD(LAST(UxTypes.KernelEntry))) THEN
        event.entry := VAL(UxTypes.KernelEntry, tState.trapInfo.arg);
      ELSE
        event.entry := UxTypes.SYSspare0; (* illegal, in range *)
      END;
      System.Zero(System.Adr(event.argList),
        System.ByteSize(event.argList));
      System.Copy(System.Adr(tState.arguments.a), System.Adr(event.argList),
        MIN(tState.arguments.n, NUMBER(tState.arguments.a)) *
          System.ByteSize(tState.arguments.a[0]));
      (* ***** Kludge "Dummy()" for timing purposes: *)
      IF tState.trapInfo.arg = ORD(UxTypes.SYSdummy) THEN
        handledHere := TRUE;
      END;
      (* ***** End "Dummy() kludge. *)
    ELSIF tState.trapInfo.kind IN a^.traps THEN
      ClassifyTrap(); (* Exception or HandlerReturn *)
    ELSE
      event.kind := Alert;
    END;
  UNTIL NOT handledHere;
END WaitEvent;


(***************************)
(* Kernel entry procedures *)
(***************************)

PROCEDURE UGetPageSize(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN UltrixBytesPerPage; END UGetPageSize;

PROCEDURE UReserveTraps(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  TYPE ExpandedTrapKinds = BITS 32 FOR TPSpecial.TrapKinds;
  VAR a: T; traps: ExpandedTrapKinds;
  BEGIN
    a := u^.a;
    traps := LOOPHOLE(args[0], ExpandedTrapKinds);
    IF (traps * UxTypes.AlwaysHandledTraps # TPSpecial.TrapKinds{})
       OR NOT (traps <= a^.traps) THEN
      UxPro.RaiseError(UxTypes.EINVAL);
    END;
    a^.traps := a^.traps - traps;
    RETURN 0;
  END UReserveTraps;

PROCEDURE USBreak(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR a: T; breakAddr: Addr;
  BEGIN
    a := u^.a;
    IF Addr(a^.rLimData.cur) < args[0] THEN
      UxPro.RaiseError(UxTypes.ENOMEM);
    END;
    breakAddr := UltrixRoundUp(args[0]);
    SetBoundary(u^.space, VM.Low, breakAddr);
    RETURN 0;
  END USBreak;
```

```
PROCEDURE UVAdvise(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN 0; END UVAdvise;


(*****************************************************************)
(* Procedures exported thru OSFriends via RemoteOSAS and RemoteOS *)
(*****************************************************************)

PROCEDURE SetSTrace(u: Ux.T; exec, fork: OSFriends.STrace) RAISES {};
  BEGIN
    u^.a^.straceExec := exec;
    u^.a^.straceFork := fork;
  END SetSTrace;

BEGIN initialized := FALSE; END UxAS.
```

```
(* DIGITAL EQUIPMENT CORPORATION  CONFIDENTIAL AND PROPRIETARY *)
(* Last modified on Tue Apr 28 15:07:22 1987 by mcjones *)

SAFE IMPLEMENTATION MODULE UxError;

BEGIN

  (* Initialize errTran. *)

  errTran[OkEC] := ok;
  errTran[BadExecutableEC] := ENOEXEC;
  errTran[BadFileEC] := EBADF;
  errTran[BadFileNameEC] := ENOENT;
  errTran[BadIOCtlOpEC] := ENOTTY;
  errTran[BadPIDEC] := ESRCH;
  errTran[BadStateForSignalEC] := EINVAL;
  errTran[CannotLinkToDirectoryEC] := EPERM;
  errTran[CannotWriteADirectoryEC] := EISDIR;
  errTran[CrossDeviceLinkEC] := EXDEV;
  errTran[DirectoryNotEmptyEC] := ENOTEMPTY;
  errTran[DirectoryUnlinkEC] := EISDIR;
  errTran[FileBusyEC] := ETXTBSY;
  errTran[FileExistsEC] := EEXIST;
  errTran[IOErrorEC] := EIO;
  errTran[InvalidArgumentEC] := EINVAL;
  errTran[InvalidCredentialsEC] := EPERM;
  errTran[InvalidObjectEC] := ENODEV;
  errTran[LookUpEC] := ENOENT;
  errTran[MinorDeviceDoesNotExistEC] := ENXIO;
  errTran[NameTooLongEC] := ENAMETOOLONG;
  errTran[NoDriverForDeviceEC] := ENXIO;
  errTran[NoMountedVolumesEC] := ENOENT;
  errTran[NotADirectoryEC] := ENOTDIR;
  errTran[NotATerminalEC] := ENOTTY;
  errTran[NotEnoughRoomEC] := ENOSPC;
  errTran[NotEnoughVMEC] := ENOMEM; (* see execve *)
  errTran[NotImplementedEC] := ok; (* ***** *)
  errTran[NotOwnerEC] := EPERM;
  errTran[NotSuperUserEC] := EPERM;
  errTran[NotTtyOwnerReadEC] := EWOULDBLOCK; (* ? ***** *)
  errTran[NotTtyOwnerWriteEC] := EWOULDBLOCK; (* ? ***** *)
  errTran[OperationConflictEC] := EBADF;
  errTran[PipeHasNoReaderEC] := EPIPE;
  errTran[ProtectionViolationEC] := EACCES;
  errTran[PvOfflineEC] := EIO;
  errTran[RanOutOfResourcesEC] := EMFILE;
  errTran[RemoteFileEC] := EIO;
  errTran[ServerNotAvailableEC] := ENXIO;
  errTran[ShortExecutableEC] := EFAULT; (* see execve *)
  errTran[TooManyProcessesEC] := EAGAIN;
  errTran[TooManySymbolicLinksInPathEC] := ELOOP;
  errTran[UnseekableObjectEC] := ESPIPE;
  errTran[VolumeNeedsCheckingEC] := EIO;
  errTran[WouldBlockEC] := EWOULDBLOCK;

END UxError.
```

```
(* DIGITAL EQUIPMENT CORPORATION  CONFIDENTIAL AND PROPRIETARY *)
(* Last modified on Sat Aug  1 09:18:30 1987 by mcjones *)
(*      modified on Wed May  6 23:19:00 1987 by swart *)
(*      modified on Sun Sep 17 15:45:00 1985 by mds *)

MODULE UxPro IMPLEMENTS UxPro, RemoteOSPro;

IMPORT
  DebuggerFriends, OSFriends, NS, NSUtil, NubTypes, RemoteOSAS,
  RPCLocal, RPCRuntime, SBuf, System, TDState, Time, Thread, ThreadFriends,
  ThreadsPort, TPFriends, UxAS, UxError, UxFile, UxTime, VM, VMFriends;

(* Internal TYPEs and CONSTs *)

CONST
  UltrixPriorityIncrement = 10;
  MinPID = Ux.InitPID;
  MaxPID = 32767; (* or should this be 32000? *)

(* Classification of signals according to default action: *)
  NoActionSignals =
    OS.Signals{OS.SigUrg, OS.SigCont, OS.SigChld, OS.SigIO, OS.SigWinCh};
  (* default action is ignore *)
  StopSignals = OS.Signals{OS.SigStop, OS.SigTStp, OS.SigTTIn, OS.SigTTOu};
  (* default action is stop *)
  CoreImageSignals =
    OS.Signals{OS.SigQuit, OS.SigIll, OS.SigTrap, OS.SigIOT, OS.SigEMT,
               OS.SigFPE, OS.SigBus, OS.SigSegV, OS.SigSys};
  (* default action is terminate w/ core image *)
  (* For others, the default action is termination. *)

(* Signals whose default handler doesn't require PostSignal to alert. *)
  NoAlertSignals = OS.Signals{OS.SigUrg, OS.SigChld, OS.SigIO};

(* Signals that can't be masked; must have default handler (except SigCont,
   which can also be caught. *)
  PowerfulSignals = OS.Signals{OS.SigKill, OS.SigStop, OS.SigCont};

  NoSignals = OS.Signals{};

TYPE
  ProcessTemplate = OS.ProcessTemplate;
  ProcessTemplate = Ux.T;
  State = (Nascent, Running, Stopped, Terminated);
  WatcherEvent = (WESignal, WESetPriority, WETerminate);
  WatcherEvents = SET OF WatcherEvent;
  T = REF TObj; (* private state for UxPro *)
  TObj = RECORD (* fields marked (* ! *) are inherited by forked process *)
    uNext: Ux.T;
    uParent: Ux.T;
    uTemplates: Ux.T; (* list of templates linked through uNext *)
    watcherThread: Thread.T;
    priority: ThreadFriends.Priority; (*!*)
    (* state of process *)
    state: State;
    untraced: BOOLEAN; (* state=Stopped & seen by parent *)
    terminating: BOOLEAN; (* client threads will never run again *)
    vForkParent, vForkChild: BOOLEAN;
    ws: UxTypes.WStatus; (* valid if state=Stopped or state=Terminated *)
    stateChanged: Thread.Condition; (* stopped, started, or terminated *)
    childStateChanged: Thread.Condition;
    (* stopped, terminated, or no longer vforked *)
    (* signal stuff *)
    mask: OS.Signals; (* current signal mask for process *) (*!*)
    posted: OS.Signals; (* signals waiting to be delivered *)
    signaled: Thread.Condition; (* signal state or posted changed *)
    watcherEvents: WatcherEvents;
    alerted: OS.Signals; (* ONLY FOR OLD OS: signals for GetSignal call *)
    illCode: INTEGER; (* code for last SigIll *)
    fpeCode: INTEGER; (* code for last SigFPE *)
    sigStack: UxTypes.SignalStack; (*!*)
    sigVec: UxTypes.SignalVector; (*!*)
    (* identification *)
    (* pid is in Ux.T *)
    (* pgrp is in Ux.T *) (*!*)
    (* euser, ruser, password are in Ux.T *) (*!*)
    (* ppid is uParent^.pid *)
    name: Text.T; (*!*)
    argv, (* DROP? *) (*!*) envp: Text.RefArray; (* DROP? *) (*!*)
    initWTime: Time.T; (* not necessarily 0 *)
    (* rusage of descendants (includes self after termination) *)
    descCTime: Time.T; (* client cpu time *)
    descWTime: Time.T; (* watcher cpu time *)
  END;


(* Global variables w/ initialization *)

VAR
  initialized: BOOLEAN;
  hostID: INTEGER; (* only for UGetHostID/USetHostID *)
  gx: Thread.Mutex; (* module lock *)
  never: Thread.Condition; (* never signaled; for waiting for alert *)
  terminating: Thread.Condition; (* some p^.terminating became TRUE *)
  pidLast: INTEGER; (* last pid assigned *)
  uHead: Ux.T; (* head of list of processes (for searches) *)
  uTail: Ux.T; (* last of list of processes (for additions) *)
  uRover: Ux.T; (* for NextProcess *)
  uInit: Ux.T; (* the root of the process tree *)
  uNub: Ux.T; (* placeholder for GetProcessInfo, NextProcess *)
  (* Ux.uTaos: for Taos clients in Taos address space *)
  kep: ARRAY UxTypes.KernelEntry OF Ux.EntryProc; (* entry procs *)
  kec: ARRAY UxTypes.KernelEntry OF CARDINAL; (* counters *)
  timingEnabled: BOOLEAN; (* enables ket *)
  ket: ARRAY UxTypes.KernelEntry OF Time.T; (* cumulative CPU time *)
  traceKernelCalls: BOOLEAN;


(* Some invariants on state of UxPro:
       ULIST: Either uHead=NIL and uTail=NIL, or uHead points to a
         list of records linked via the p^.next field and uTail points to the
         last record in the list (the one with p^.next=NIL.)
       ROVER: Either uRover=NIL, or uRover points to a record reachable
         from uHead.
       SIGPOSTED: If (p^.posted-p^.mask) # NoSignals, then
         WESignal IN p^watcherEvents.
       SIGWAIT: If a process is blocked on p^.signaled, then all the
         signals in its allowed set have signal state SignalHandle and are
         not posted. (See WaitForChild.)
*)


(***********************)
(* Internal procedures *)
(***********************)

PROCEDURE ActOnSignals(
    u: Ux.T;
```

```
  s: OS.SignalOrNone;
  oldMask: OS.Signals)
  : BOOLEAN;
(* Deliver all posted unmasked signals, starting with 's' if it is a
   candidate. Arrange that when execution finally resumes from the current
   kernel call (after any handlers), the mask is set to 'oldMask'.
   Maintaining SIGPOSTED is caller's responsibility (e.g. by ensuring
   oldMask contains current mask. Result is TRUE iff no signals was deferred
   due to inability to acquire the TDState mutex. *)
(* Call with gx LOCKed. *)
VAR
  p: T;
  signals, deferred: OS.Signals;
  h: UxTypes.SignalHandler;
  code: INTEGER;
  switchStacks: BOOLEAN;
BEGIN
  p := u^.p;
  deferred := NoSignals;
  LOOP
    signals := (p^.posted - p^.mask) - deferred;
    IF p^.vForkChild THEN EXCL(signals, OS.SigTStp); END;
    IF signals = NoSignals THEN EXIT; END;
    IF (s = OS.SigNone) OR NOT (s IN signals) THEN
      s := FIRST(OS.Signal);
      WHILE NOT (s IN signals) DO INC(s); END;
    END;
    EXCL(p^.posted, s);
    h := p^.sigVec[s].handler;
    IF h.h = UxTypes.SignalDefault THEN
      IF s IN NoActionSignals THEN (* nothing *)
      ELSIF
        (s IN StopSignals) AND ((p^.uParent # uInit) OR (s = OS.SigStop))
      THEN
        (* The qualification regarding orphans is not documented in section
           2 of the manual, but was described by Dave Lennert of HP in
           comp.unix.wizards. *)
        DoStop(u, s);
        s := OS.SigCont;
      ELSE
        PostTerminate(u, s, s IN CoreImageSignals);
      END;
    ELSIF h.h = UxTypes.SignalIgnore THEN (* nothing *)
    ELSIF h.h = UxTypes.SignalAlert THEN (* ONLY FOR OLD OS *)
      INCL(p^.alerted, s);
      UxAS.AlertHandler(u);
    ELSE (* Ultrix-style handler *)
      IF p^.vForkChild OR TDState.AcquireIfRunning(u^.pid.i, u^.space) THEN
        IF s = OS.SigFPE THEN
          code := p^.fpeCode;
        ELSIF s = OS.SigIll THEN
          code := p^.illCode;
        ELSE
          code := 0;
        END;
        switchStacks := p^.sigVec[s].onStack AND NOT p^.sigStack.onStack;
        IF UxAS.IsTopaz(u) THEN
          PostTerminate(u, s, TRUE);
        ELSE
          TRY
            UxAS.PushHandlerCall(u, s, code, p^.sigVec[s].handler, oldMask,
              p^.sigStack.onStack, switchStacks, p^.sigStack.sp);
          EXCEPT
            SBuf.Fault: PostTerminate(u, OS.SigSegV, TRUE);
            END;
            IF switchStacks THEN p^.sigStack.onStack := TRUE; END;
          END;
          p^.mask := p^.mask + p^.sigVec[s].mask;
          INCL(p^.mask, s);
          oldMask := p^.mask;
          IF NOT p^.vForkChild THEN TDState.Release(u^.pid.i, u^.space); END;
        ELSE
          INCL(deferred, s);
        END;
      END;
    END; (* LOOP *)
    p^.mask := oldMask;
    p^.posted := p^.posted + deferred;
    RETURN deferred = NoSignals;
  END ActOnSignals;

PROCEDURE Ancestor(u1, u2: Ux.T): BOOLEAN;
  (* Result is TRUE iff 'u1 Parent* u2', where 'Parent*' is the reflexive
     transitive closure of the the relation 'u1 Parent u2 iff u1 =
     u2^.p^.uParent'. *)
  (* Call with gx LOCKed. *)
  VAR uu: Ux.T;
  BEGIN
    uu := u2;
    LOOP
      IF uu = u1 THEN RETURN TRUE; END;
      IF uu = NIL THEN RETURN FALSE; END;
      uu := uu^.p^.uParent;
    END;
  END Ancestor;

PROCEDURE DisposeResources(u: Ux.T; template: BOOLEAN := FALSE);
  (* Release limited resources (virtual address space, threads) and perform
     exit-time file system actions (e.g. closes) required by Ultrix semantics;
     leave behind state used by UWait and WaitForChild. *)
  VAR t: OS.ProcessTemplate; ownSpace: BOOLEAN;
  BEGIN
    ownSpace := NOT u^.p^.vForkChild;
    t := u^.p^.uTemplates;
    WHILE t # NIL DO DisposeResources(t, TRUE); t := t^.p^.uNext; END;
    UxTime.Exit(u);
    UxFile.Exit(u);
    IF NOT template THEN UxAS.Dispose(u, ownSpace); END;
  END DisposeResources;

PROCEDURE DoExec(
    u: Ux.T;
    file: OS.File;
    dir: OS.Dir;
    euser: User.T;
    name: Text.T;
    argv, envp: Text.RefArray;
    uArgs: Ux.T := NIL;
    argva, envpa: UNSIGNED := 0)
    : BOOLEAN
  RAISES {OS.Error, SBuf.Fault, Thread.Alerted};
  (* If DoExec raises OS.Error or SBuf.Fault, the address space hasn't been
     modified and the error may be returned to the user process in the normal
     fashion. If DoExec returns FALSE or raises Thread.Alerted, the address
     space may have been modified so the user process should be terminated.
     Caller must eventually call UxFile.Exec2(u). *)
  VAR s: OS.Signal; p: T; ok: BOOLEAN;
  BEGIN
```

```
    p := u^.p;
    ok :=
      UxAS.Load(u, file, dir, euser, name, argv, envp, uArgs, argva, envpa,
        p^.vForkChild);
    LOCK gx DO
      IF p^.vForkChild THEN (* give address space back to parent *)
        p^.vForkChild := FALSE;
        Thread.Broadcast(p^.uParent^.p^.childStateChanged);
      END;
      FOR s := FIRST(OS.Signal) TO LAST(OS.Signal) DO
        WITH p^.sigVec[s] DO
          IF handler.h # UxTypes.SignalIgnore THEN
            handler.h := UxTypes.SignalDefault;
          END;
          onStack := FALSE;
          mask := NoSignals;
        END;
        p^.sigStack.sp := 0;
        p^.sigStack.onStack := FALSE;
      END;
      p^.name := name;
      p^.argv := argv;
      p^.envp := envp;
    END;
    UxFile.Execl(u);
    RETURN ok;
  END DoExec;

PROCEDURE DoFork(u: Ux.T; vFork: BOOLEAN): INTEGER;
  VAR p, pNew: T; uNew: Ux.T;
  BEGIN
    p := u^.p;
    IF UxAS.IsTopaz(u) THEN RAISE(OS.Error, OS.InvalidArgumentEC); END;
    New(u, uNew, vFork);
    pNew := uNew^.p;
    TRY
      UxAS.Copy(u, uNew, vFork);
    EXCEPT
      SBuf.Fault: DisposeResources(uNew); RAISE(SBuf.Fault);
    END;
    LOCK gx DO
      UxAS.R0Result(uNew, u^.pid.i);
      UxAS.R1Result(uNew, 1); (* non-zero means we're in the child *)

      DoForkCommon(u, uNew, vFork);

      IF vFork THEN
        TDState.PreVFork(u^.pid.i, u^.space);
        UxAS.HideThread(u);
        p^.vForkParent := TRUE;
        WHILE (pNew^.state # Terminated) AND pNew^.vForkChild DO
          Thread.Wait(gx, p^.childStateChanged);
        END;
        p^.vForkParent := FALSE;
        UxAS.RestoreThread(u);
        TDState.PostVFork(u^.pid.i, u^.space);
      END;
    END; (* LOCK gx *)
    UxAS.R1Result(u, 0); (* 0 means we're in the parent *)
    RETURN uNew^.pid.i;
  END DoFork;

PROCEDURE DoForkCommon(u, uNew: Ux.T; vFork: BOOLEAN := FALSE);
  (* Call with gx LOCKed. *)
  VAR pNew: T;
  BEGIN
    pNew := uNew^.p;
    pNew^.watcherThread := Thread.Fork(Watcher, uNew);
    UxAS.SetThreadPriority(pNew^.watcherThread, pNew^.priority);
    pNew^.initWTime := ThreadFriends.GetCPUTime(pNew^.watcherThread);
    Insert(uNew);

    IF NOT vFork THEN UxAS.NewTarget(u, uNew); END;
  END DoForkCommon;

PROCEDURE DoKill(
    uFrom, uTo: Ux.T;
    signal: OS.SignalOrNone;
    euser: User.T)
    : BOOLEAN;
  (* Call with gx LOCKed. *)
  VAR ok: BOOLEAN;
  BEGIN
    ok :=
      User.Equal(euser, uTo^.euser) OR User.IsSuper(euser)
      OR ((signal = OS.SigCont) AND Ancestor(uFrom, uTo));
    IF ok THEN PostSignal(uTo, signal); END;
    RETURN ok;
  END DoKill;

PROCEDURE DoNothing(u: Ux.T): INTEGER;
  BEGIN UxAS.R1Result(u, 0); RETURN 0; END DoNothing;

PROCEDURE DoSetPriorityPGRP(pgrp: OS.PGRP; priority: ThreadFriends.Priority);
  VAR found: BOOLEAN; u: Ux.T; p: T;
  BEGIN
    found := FALSE;
    LOCK gx DO
      u := uHead;
      WHILE u # NIL DO
        p := u^.p;
        IF (p^.state # Terminated) AND (u^.pgrp.i = pgrp.i) THEN
          PostSetPriority(u, priority);
          found := TRUE;
        END;
        u := p^.uNext;
      END; (* WHILE *)
    END; (* LOCK gx *)
    IF NOT found THEN RAISE(OS.Error, OS.BadPIDEC); END;
  END DoSetPriorityPGRP;

PROCEDURE DoSetPriorityUser(user: User.T; priority: ThreadFriends.Priority);
  VAR found: BOOLEAN; u: Ux.T; p: T;
  BEGIN
    found := FALSE;
    LOCK gx DO
      u := uHead;
      WHILE u # NIL DO
        p := u^.p;
        IF (p^.state # Terminated) AND User.Equal(u^.ruser, user) THEN
          PostSetPriority(u, priority);
          found := TRUE;
        END;
        u := p^.uNext;
      END; (* WHILE *)
    END; (* LOCK gx *)
    IF NOT found THEN RAISE(OS.Error, OS.BadPIDEC); END;
  END DoSetPriorityUser;
```

```
PROCEDURE DoSigEffective(u: Ux.T; signal: OS.Signal): BOOLEAN RAISES {};
  (* Called with gx LOCKed. *)
  VAR p: T;
  BEGIN
    ASSERT((signal = OS.SigTTIn) OR (signal = OS.SigTTOu));
    p := u^.p;
    RETURN
      (p^.sigVec[signal].handler.h # UxTypes.SignalIgnore)
      AND (NOT (signal IN p^.mask) OR UxAS.IsTopaz(u))
      AND (p^.uParent # uInit) AND NOT p^.vForkChild;
  END DoSigEffective;

PROCEDURE DoSignalPG(pgrp: OS.PID; signal: OS.Signal) RAISES {};
  (* Send signal to every process in pgrp. *)
  (* Call with gx LOCKed. *)
  VAR uu: Ux.T;
  BEGIN
    uu := uHead;
    WHILE uu # NIL DO
      IF (uu^.pgrp.i = pgrp.i) AND (uu^.p^.state # Terminated) THEN
        PostSignal(uu, signal);
      END;
      uu := uu^.p^.uNext;
    END;
  END DoSignalPG;

PROCEDURE DoStop(u: Ux.T; s: OS.Signal);
  (* Call with gx LOCKed. *)
  VAR p: T;
  BEGIN
    ASSERT(Ux.uTaos^.pid.i < u^.pid.i, "Attempt to stop system process");
    p := u^.p;
    UxAS.StopAll(u);
    p^.state := Stopped;
    p^.untraced := TRUE;
    p^.ws.wStopVal := UxTypes.WSTOPPED;
    p^.ws.wStopSig := s;
    Thread.Broadcast(p^.stateChanged);
    PostSignal(p^.uParent, OS.SigChld);
    Thread.Signal(p^.uParent^.p^.childStateChanged);
    LOOP
      IF OS.Signals{OS.SigKill, OS.SigCont} * p^.posted # NoSignals THEN
        EXIT;
      END;
      (* Wait for alert. *)
      TRY Thread.AlertWait(gx, never); EXCEPT Thread.Alerted: END;
    END;
    (* Remove extra stop signals, e.g. from typing an extra Control-Z. This
       doesn't apply to SigTTIn and SigTTOu, since they stem from the action
       of the process rather than the (sloppy) user. *)
    FOR s := FIRST(OS.Signal) TO LAST(OS.Signal) DO
      IF (s IN OS.Signals{OS.SigStop, OS.SigTStp}) AND (s IN p^.posted)
         AND (p^.sigVec[s].handler.h = UxTypes.SignalDefault) THEN
        EXCL(p^.posted, s);
      END;
    END;
    UxAS.StartAll(u);
    p^.state := Running;
    p^.untraced := FALSE;
    Thread.Broadcast(p^.stateChanged);
  END DoStop;

PROCEDURE DoTermination(u: Ux.T);
  VAR uu: Ux.T; p, pp: T; cTime, wTime: Time.T;
  BEGIN
    p := u^.p;
    UxAS.StopAll(u);
    LOCK gx DO p^.terminating := TRUE; Thread.Broadcast(terminating); END;
    IF u^.space # NubTypes.NullSpace THEN
      RPCLocal.Moribund(u^.space);
      (* ***** Now we must wait until all RPC calls from the given space have
         returned to the server stubs. *)
    END;
    cTime := UxAS.GetClientTime(u, TRUE);
    wTime := GetWTime(u);

    IF p^.ws.wCoreDump THEN
      TRY
        (* Since the client thread(s) will never run again, and will be
           destroyed soon, we must not deliver any alerts to it. We set things
           up so that a SigInt can still be delivered in case a core dump to a
           remote file drags on. *)
        LOCK gx DO
          InitializeSignalState(p); (* all SigDefault *)
          p^.mask := OS.AllSignals - OS.Signals{OS.SigInt};
        END;
        UxAS.MakeCoreImage(u, p^.ws.wTermSig);
      EXCEPT
        OS.Error, Thread.Alerted: p^.ws.wCoreDump := FALSE;
      END;
    END;

    LOCK gx DO
      p^.state := Terminated;

      (* If we were the child of a vFork, we must eliminate all our
         dependencies on its address space before the parent is allowed to
         proceed. Otherwise we postpone releasing the address space and other
         resources until after releasing gx. *)
      IF p^.vForkChild THEN DisposeResources(u); END;

      (* Add our rusage to that of our descendants. *)
      p^.descCTime := Time.Add(p^.descCTime, cTime);
      p^.descWTime := Time.Add(p^.descWTime, wTime);

      (* Our children become orphans. *)
      uu := uHead;
      WHILE uu # NIL DO
        pp := uu^.p;
        IF pp^.uParent = u THEN (* one of our children *)
          pp^.uParent := uInit;
          IF pp^.state = Stopped THEN
            (* Although it isn't documented in section 2 of the manual, it
               appears newly orphaned stopped processes are given a chance to
               "clean up their act": *)
            PostSignal(uu, OS.SigHUp);
            PostSignal(uu, OS.SigCont);
          ELSIF pp^.state = Terminated THEN
            (* The child is an orphan zombie; simulate Init. *)
            Remove(uu);
          END;
        END;
        uu := pp^.uNext;
      END; (* WHILE *)

      (* Update our parent's descendant-rusage. *)
      uu := p^.uParent;
```

```
      pp := uu^.p;
      pp^.descCTime := Time.Add(pp^.descCTime, p^.descCTime);
      pp^.descWTime := Time.Add(pp^.descWTime, p^.descWTime);

      (* Inform our parent of our termination. *)
      IF uu = uInit THEN
        (* We are an orphan zombie; simulate Init. *)
        Remove(u);
      ELSE
        Thread.Broadcast(p^.stateChanged);
        PostSignal(uu, OS.SigChld);
        Thread.Signal(pp^.childStateChanged);
      END;
    END; (* LOCK gx *)

    (* Now that the address space is marked as terminated, no operations
       executed by other address spaces (e.g. UGetPriority, GetProcessInfo)
       will try to look at it, so it is safe to delete the threads and space.
    *)
    IF NOT p^.vForkChild THEN DisposeResources(u); END;
  END DoTermination;

PROCEDURE GetSignalState(
    h: UxTypes.SignalHandler;
    mask: BOOLEAN)
    : OS.SignalState;
  (* Convert internal to external representation. *)
  BEGIN
    IF h.h = UxTypes.SignalIgnore THEN
      RETURN OS.SignalIgnore;
    ELSIF (h.h = UxTypes.SignalDefault) AND NOT mask THEN
      RETURN OS.SignalDefault;
    ELSE
      RETURN OS.SignalHandle;
    END;
  END GetSignalState;

PROCEDURE InitializeSignalState(p: T);
  (* Initialize mask, sigStack, and sigVec components. Set all signals to
     default signal state. *)
  VAR s: OS.Signal;
  BEGIN
    p^.mask := NoSignals;
    p^.sigStack.sp := 0;
    p^.sigStack.onStack := FALSE;
    FOR s := FIRST(OS.Signal) TO LAST(OS.Signal) DO
      WITH p^.sigVec[s] DO
        handler.h := UxTypes.SignalDefault;
        mask := NoSignals;
        onStack := FALSE;
      END;
    END;
  END InitializeSignalState;

PROCEDURE Insert(uNew: Ux.T);
  (* Set state to 'Running', assign a unique PID, and insert 'uNew' into the
     list of processes, preserving ascending order. *)
  (* Call with gx LOCKed. *)
  VAR uu, uStart: Ux.T; pp, ppPrev: T;
  BEGIN
    uNew^.p^.state := Running;
    (* We assume that the list of processes contains less than MaxPID
       processes. *)
    uStart := uHead;
    LOOP
      INC(pidLast);
      IF MaxPID < pidLast THEN pidLast := MinPID; END;
      IF (uHead = NIL) OR (pidLast < uHead^.pid.i) THEN
        uNew^.p^.uNext := uHead;
        IF uHead = NIL THEN uTail := uNew; END;
        uHead := uNew;
        EXIT;
      ELSIF uTail^.pid.i < pidLast THEN
        (* ULIST and uHead # NIL implies uTail # NIL *)
        uNew^.p^.uNext := NIL; (* in case it was on a template list *)
        uTail^.p^.uNext := uNew;
        uTail := uNew;
        EXIT;
      ELSE
        (* uHead^.pid.i <= pidlast <= uTail^.pid.i *)
        uu := uStart;
        pp := uu^.p;
        WHILE uu^.pid.i < pidLast DO
          ppPrev := pp;
          uu := pp^.uNext;
          IF uu = NIL THEN uu := uHead; END;
          pp := uu^.p;
        END;
        IF pidLast < uu^.pid.i THEN (* ppPrev -> uNew -> uu *)
          ppPrev^.uNext := uNew;
          uNew^.p^.uNext := uu;
          EXIT;
        END;
        (* pidLast = uu^.pid.i, so go around again *)
        uStart := uu;
      END; (* IF uTail^.pid.i < pidLast *)
    END; (* LOOP *)
    uNew^.pid.i := pidLast;
  END Insert;

PROCEDURE InsertTemplate(u, t: Ux.T);
  (* Add 't' to list of templates of 'u'. *)
  BEGIN
    LOCK gx DO t^.p^.uNext := u^.p^.uTemplates; u^.p^.uTemplates := t; END;
  END InsertTemplate;

PROCEDURE New(
    u: Ux.T;
    VAR uNew: Ux.T;
    vFork: BOOLEAN := FALSE;
    topaz: BOOLEAN := FALSE);
  (* Set uNew to new Ux.T based on parent u (possibly NIL). If topaz=FALSE,
     the new system state will have the same signal vector and signal mask as
     its parent and will include copies of the file descriptors from the file
     system state of u (which should be non-NIL). If topaz=TRUE, the new state
     will have no signals masked and all vector entries defaulted except it
     will ignore those its parent igonores; the file system state will have no
     open descriptors. *)
  VAR p: T; (* parent's process management state *)
  BEGIN
    NEW(uNew);
    NEW(uNew^.p);
    WITH uNew^.p^ DO (* private state of new process *)
      uParent := u;
      uTemplates := NIL;
      state := Nascent;
      untraced := FALSE;
      terminating := FALSE;
```

```
      vForkParent := FALSE;
      vForkChild := vFork;
      Thread.InitCondition(stateChanged);
      Thread.InitCondition(childStateChanged);
      InitializeSignalState(uNew^.p); (* all SignalDefault *)
      posted := NoSignals;
      Thread.InitCondition(signaled);
      watcherEvents := WatcherEvents{};
      alerted := NoSignals; (* ONLY FOR OLD OS *)
      initWTime.seconds := LAST(Time.Seconds); (* bug catcher *)
      initWTime.microseconds := LAST(Time.Microseconds); (* bug catcher *)
      descCTime.seconds := 0;
      descCTime.microseconds := 0;
      descWTime := descCTime;
      IF u = NIL THEN (* first process *)
        WITH uNew^ DO
          euser := User.SuperUser(); (* all local privileges *)
          ruser := User.FTPUser(); (* limited remote privileges *)
          NEW(password);
          password^ := "";
        END;
        priority := ThreadFriends.NormalPriority;
      ELSE (* subsequent process *)
        p := u^.p;
        LOCK gx DO (* copy relevant portions of parent's state *)
          uNew^.pgrp := u^.pgrp;
          uNew^.euser := u^.euser;
          uNew^.ruser := u^.ruser;
          uNew^.password := u^.password;
          priority := p^.priority;
          IF NOT topaz THEN
            mask := p^.mask;
            sigStack := p^.sigStack;
            sigVec := p^.sigVec;
          END;
          name := p^.name;
          argv := p^.argv;
          envp := p^.envp;
        END;
      END;
    END; (* WITH uNew^.p^ *)
    UxAS.New(u, uNew); (* sets uNew^.a, uNew^.space *)
    uNew^.f := UxFile.Fork(u, (*desc:*) NOT topaz);
    uNew^.i := NIL;
  END New;

PROCEDURE PostSetPriority(u: Ux.T; priority: ThreadFriends.Priority);
  (* Only the watcher thread for a given address space can do the SetStates to
     change the client thread priorities, so we do something almost like
     sending a signal. *)
  (* Call with gx LOCKed. *)
  VAR p: T;
  BEGIN
    p := u^.p;
    p^.priority := priority;
    PostWatcherEvent(p, WESetPriority);
  END PostSetPriority;

PROCEDURE PostSignal(u: Ux.T; s: OS.SignalOrNone);
  (* Call with gx LOCKed. *)
  VAR h: UxTypes.SignalHandler; p: T;
  BEGIN
    p := u^.p;
    ASSERT(p^.state # Terminated);
    IF s # OS.SigNone THEN
      h := p^.sigVec[s].handler;
      IF h.h # UxTypes.SignalIgnore THEN
        INCL(p^.posted, s);
        IF s IN p^.mask THEN (* maybe an OS.WaitForSignal client *)
          Thread.Broadcast(p^.signaled);
        ELSIF h.h = UxTypes.SignalAlert THEN (* ONLY FOR OLD OS *)
          EXCL(p^.posted, s);
          INCL(p^.alerted, s);
          UxAS.AlertHandler(u);
        ELSIF
          ((h.h # UxTypes.SignalDefault) OR NOT (s IN NoAlertSignals))
          AND ((s # OS.SigTStp) OR NOT p^.vForkChild)
        THEN (* an Ultrix client *)
          PostWatcherEvent(p, WESignal); (* restore SIGPOSTED *)
        END;
      END;
    END;
  END PostSignal;

PROCEDURE PostTerminate(u: Ux.T; s: OS.SignalOrNone; dump: BOOLEAN);
  (* Call with gx LOCKed. *)
  VAR p: T;
  BEGIN
    p := u^.p;
    u^.p^.ws.wTermSig := s;
    u^.p^.ws.wCoreDump := dump;
    PostWatcherEvent(p, WETerminate);
  END PostTerminate;

PROCEDURE PostWatcherEvent(p: T; watcherEvent: WatcherEvent);
  (* Call with gx LOCKed. *)
  BEGIN
    IF p^.watcherThread # NIL THEN
      INCL(p^.watcherEvents, watcherEvent);
      UxAS.SetThreadPriority(p^.watcherThread, UxAS.AlertPriority);
      Thread.Alert(p^.watcherThread);
    END;
  END PostWatcherEvent;

PROCEDURE ProcessFromPID(
    pid: OS.PID;
    esrch: BOOLEAN := TRUE;
    terminatedOK: BOOLEAN := FALSE)
    : Ux.T;
  (* Call with gx LOCKed. *)
  VAR uu: Ux.T;
  BEGIN
    uu := uHead;
    WHILE (uu # NIL) AND (uu^.pid.i < pid.i) DO uu := uu^.p^.uNext; END;
    IF (uu = NIL) OR (uu^.pid.i > pid.i)
       OR ((uu^.p^.state = Terminated) AND NOT terminatedOK) THEN
      IF esrch THEN RAISE(OS.Error, OS.BadPIDEC); END;
      ASSERT(FALSE); (* pid must exist! *)
    END;
    RETURN uu;
  END ProcessFromPID;

PROCEDURE Remove(u: Ux.T);
  (* Remove the specified process from the list of processes. *)
  (* Call with gx LOCKed. *)
  VAR uu, uuPrev: Ux.T;
  BEGIN
    IF uRover = u THEN uRover := NIL; END; (* maintain ROVER *)
```

```
    u^.p^.uParent := NIL; (* avoid circular link *)
    uu := uHead;
    uuPrev := NIL;
    WHILE uu # u DO uuPrev := uu; uu := uu^.p^.uNext; END;
    IF uu = uHead THEN
      uHead := uu^.p^.uNext;
    ELSE
      uuPrev^.p^.uNext := uu^.p^.uNext;
    END;
    IF uu = uTail THEN uTail := uuPrev; END;
  END Remove;

PROCEDURE RemoveTemplate(u: Ux.T; t: OS.ProcessTemplate);
  (* Remove 't' from the list of as-yet unstarted templates associated with
     'u'. *)
  (* Call with gx LOCKed. *)
  VAR tt: OS.ProcessTemplate;
  BEGIN
    tt := u^.p^.uTemplates;
    IF tt = t THEN
      u^.p^.uTemplates := t^.p^.uNext;
    ELSE
      WHILE tt^.p^.uNext # t DO tt := tt^.p^.uNext; END;
      tt^.p^.uNext := t^.p^.uNext;
    END;
  END RemoveTemplate;

PROCEDURE RLimit(u: Ux.T; resource: UxTypes.Resource): UxTypes.RLimit;
  CONST rLimitInfinity = 07fffffffH;
  VAR rLimit: UxTypes.RLimit;
  BEGIN
    CASE resource OF
    | UxTypes.rLimitData, UxTypes.rLimitStack, UxTypes.rLimitCore:
        UxAS.GetRLimit(u, resource, rLimit);
    |
    ELSE rLimit.cur := rLimitInfinity; rLimit.max := rLimitInfinity;
    END;
    RETURN rLimit;
  END RLimit;

PROCEDURE SigTT(
    u: Ux.T;
    ec: OS.EC;
    signal: OS.Signal;
    entry: UxTypes.KernelEntry);
  (* Convert NotTtyOwner{Read,Write}EC into appropriate synchronous signal.  *)
  BEGIN
    IF u^.p^.sigVec[signal].handler.h # UxTypes.SignalAlert THEN
      (* Retry kernel call if signal handler returns. *)
      UxAS.BackupKernelCall(u, entry);
      LOCK gx DO DoSignalPG(u^.pgrp, signal); END;
    ELSE (* ONLY FOR OLD OS *)
      (* Kernel call returns with error, not signal. *)
      UxAS.R0Result(u, ORD(UxError.errTran[ec]), TRUE);
    END;
  END SigTT;

PROCEDURE ThreadToUltrixPriority(priority: ThreadFriends.Priority): INTEGER;
  BEGIN
    IF priority < ThreadFriends.NormalPriority THEN
      RETURN UltrixPriorityIncrement;
    ELSIF priority = ThreadFriends.NormalPriority THEN
      RETURN 0;
    ELSE (* priority > ThreadFriends.NormalPriority *)
      RETURN -UltrixPriorityIncrement;
    END;
  END ThreadToUltrixPriority;

PROCEDURE UltrixToThreadPriority(
    ultrixPriority: INTEGER)
    : ThreadFriends.Priority;
  BEGIN
    IF ultrixPriority < 0 THEN
      RETURN ThreadFriends.ForegroundPriority;
    ELSIF ultrixPriority = 0 THEN
      RETURN ThreadFriends.NormalPriority;
    ELSE
      RETURN ThreadFriends.BackgroundPriority;
    END;
  END UltrixToThreadPriority;

PROCEDURE Watcher(a: Thread.ForkeeArg): Thread.ForkeeReturn;
  (* The watcher thread for a process executes this procedure until the
     process is terminated. *)
  VAR
    u: Ux.T;
    p: T;
    s: OS.SignalOrNone;
    event: UxAS.Event;
    time: Time.T;
    ec: OS.EC;
    e: UxTypes.SysError;
    deferred: BOOLEAN;
  BEGIN
    u := NARROW(a, Ux.T);
    time.seconds := 0;
    time.microseconds := 0;
    LOCK gx DO (* wait for parent to complete u^ *) p := u^.p; END;
    LOOP (* main watcher loop *)
      s := OS.SigNone;
      TRY
        (* Run client until it traps or we are alerted. *)
        UxAS.WaitEvent(u, event);
      EXCEPT
        SBuf.Fault:
          event.kind := UxAS.Alert;
          LOCK gx DO PostTerminate(u, OS.SigSegV, (*dump:*) TRUE); END;
      END;

      (* If kernel call, execute it without holding gx. *)
      IF event.kind = UxAS.KernelCall THEN
        IF timingEnabled THEN
          time := ThreadFriends.GetCPUTime(p^.watcherThread);
        END;
        IF traceKernelCalls THEN DebuggerFriends.Pause("Kernel call"); END;
        TRY
          UxAS.R0Result(u, kep[event.entry](u, event.argList));
        EXCEPT
        | OS.Error (ec):
            IF ec = OS.NotTtyOwnerReadEC THEN
              s := OS.SigTTIn;
              SigTT(u, ec, s, event.entry);
            ELSIF ec = OS.NotTtyOwnerWriteEC THEN
              s := OS.SigTTOu;
              SigTT(u, ec, s, event.entry);
            ELSIF ec = OS.PipeHasNoReaderEC THEN
              s := OS.SigPipe;
              Signal(u, s);
```

```
      ELSIF ec = OS.NotImplementedEC THEN
        s := OS.SigSys;
        Signal(u, s);
      ELSE
        UxAS.R0Result(u, ORD(UxError.errTran[ec]), TRUE);
      END;
  | SBuf.Fault: UxAS.R0Result(u, ORD(UxTypes.EFAULT), TRUE);
  | Error (e): UxAS.R0Result(u, ORD(e), TRUE);
  | Thread.Alerted:
      (* Retry kernel call if signal handler returns. *)
      UxAS.BackupKernelCall(u, event.entry);
  END;
END;

CASE event.kind OF
| UxAS.KernelCall:
    INC(kec[event.entry]);
    IF timingEnabled THEN
      ket[event.entry] :=
        Time.Add(ket[event.entry],
          Time.Subtract(ThreadFriends.GetCPUTime(p^.watcherThread),
            time));
    END;
| UxAS.Exception:
    s := event.sig; (* first signal to look for *)
    LOCK gx DO
      PostSignal(u, s);
      IF s = OS.SigFPE THEN
        p^.fpeCode := event.code;
      ELSIF s = OS.SigIll THEN
        p^.illCode := event.code;
      END;
    END;
| UxAS.Alert:
| UxAS.HandlerReturn:
    LOCK gx DO
      p^.mask := event.mask - PowerfulSignals;
      p^.sigStack.onStack := event.onStack;
      IF (p^.posted - p^.mask) # NoSignals THEN (* restore SIGPOSTED *)
        INCL(p^.watcherEvents, WESignal);
      END;
    END;
END; (* CASE *)

IF p^.watcherEvents # WatcherEvents{} THEN
  LOCK gx DO
    REPEAT
      deferred := FALSE;
      IF WETerminate IN p^.watcherEvents THEN
        EXIT; (* main watcher loop *)
      END;
      IF WESignal IN p^.watcherEvents THEN
        IF ActOnSignals(u, s, p^.mask) THEN
          EXCL(p^.watcherEvents, WESignal);
        ELSE
          deferred := TRUE;
        END;
      END;
      IF WESetPriority IN p^.watcherEvents THEN
        IF p^.vForkChild OR TDState.AcquireIfRunning(u^.pid.i, u^.space)
        THEN
          UxAS.SetClientPriority(u, p^.priority);
          IF NOT p^.vForkChild THEN
            TDState.Release(u^.pid.i, u^.space);
          END;
          EXCL(p^.watcherEvents, WESetPriority);
        ELSE
          deferred := TRUE;
        END;
      END;
      IF 0 = ThreadFriends.SetPriority(p^.priority) THEN END;
      IF Thread.TestAlert() THEN END; (* drain pending alert *)
      IF deferred THEN
        TRY
          ThreadsPort.Release(gx);
          TRY
            TDState.AwaitRunning(u^.pid.i, u^.space);
          FINALLY
            ThreadsPort.Acquire(gx);
          END;
        EXCEPT
          Thread.Alerted:
        END; (* TRY *)
      END; (* IF deferred *)
    UNTIL p^.watcherEvents = WatcherEvents{};
  END; (* LOCK *)
END; (* IF p^.watcherEvents # WatcherEvents {} *)
END; (* LOOP *)
DoTermination(u);
RETURN NIL;
END Watcher;


(***********************************)
(* Procedures exported through UxPro *)
(***********************************)

PROCEDURE AcquireProcess(u: Ux.T) RAISES {};
  BEGIN ThreadsPort.Acquire(gx); END AcquireProcess;

PROCEDURE AcquireProcessTemplate(t: OS.ProcessTemplate) RAISES {OS.Error};
  BEGIN
    ThreadsPort.Acquire(gx);
    IF (t = NIL) OR (t^.p^.state # Nascent) THEN
      RAISE(OS.Error, OS.InvalidArgumentEC);
    END;
  END AcquireProcessTemplate;

PROCEDURE EnsureRunning(u: Ux.T) RAISES {Thread.Alerted};
  BEGIN
    LOCK gx DO
      ASSERT(u^.p^.state # Terminated); (* ***** Remove later *)
      WHILE u^.p^.state = Stopped DO
        Thread.AlertWait(gx, u^.p^.stateChanged);
      END;
    END;
  END EnsureRunning;

PROCEDURE GetWTime(u: Ux.T): Time.T RAISES {};
  BEGIN
    RETURN
      Time.Subtract(ThreadFriends.GetCPUTime(u^.p^.watcherThread),
        u^.p^.initWTime);
  END GetWTime;

PROCEDURE Import RAISES {}; BEGIN END Import;

PROCEDURE Init() RAISES {};
```

```
TYPE KernelEntry = UxTypes.KernelEntry;
VAR e: KernelEntry;
PROCEDURE ForkSystemProcess(
    u: Ux.T;
    VAR (*out*) uNew: Ux.T;
    space: NubTypes.Space;
    name: Text.T);
  BEGIN
    New(u, uNew, (*vfork:*) FALSE, (*topaz:*) TRUE);
    uNew^.space := space;
    uNew^.p^.name := name;
    Insert(uNew);
    uNew^.pgrp.i := uNew^.pid.i;
  END ForkSystemProcess;
BEGIN
  ASSERT(NOT initialized, "UxPro.Init: Called twice");
  initialized := TRUE;
  hostID := 0;
  Thread.InitMutex(gx);
  Thread.InitCondition(never);
  Thread.InitCondition(terminating);
  pidLast := 0;
  uHead := NIL;
  uTail := NIL;
  uRover := NIL;

  (* uInit *must* be created first, so it gets a PID of 1 *)
  ForkSystemProcess(NIL, uInit, NubTypes.NullSpace, "Init");
  ForkSystemProcess(uInit, uNub, NubTypes.NubSpace, "Nub");
  (* Note uTaos is not an orphan--its parent is not uInit: *)
  ForkSystemProcess(uNub, Ux.uTaos, TPFriends.GetSpace(), "Taos");
  Ux.uTaos^.p^.mask := OS.AllSignals; (* i.e. all handled *)

  FOR e := FIRST(KernelEntry) TO LAST(KernelEntry) DO
    kep[e] := UUnimplemented;
    kec[e] := 0;
    ket[e].seconds := 0;
    ket[e].microseconds := 0;
  END;
  timingEnabled := FALSE;

  kep[UxTypes.SYScheckpw] := UCheckPW; (* Ux only *)
  kep[SYSexecve] := UExecVE;
  kep[SYSexit] := UExit;
  kep[SYSfork] := UFork;
  kep[SYSgetgid] := UGetGID;
  kep[SYSgethostid] := UGetHostID;
  kep[SYSgethostname] := UGetHostName;
  kep[SYSgetpgrp] := UGetPGRP;
  kep[SYSgetpid] := UGetPID;
  kep[SYSgetpriority] := UGetPriority;
  kep[UxTypes.SYSgetpw] := UGetPW; (* Ux only *)
  kep[SYSgetrlimit] := UGetRLimit;
  kep[SYSgetrusage] := UGetRUsage;
  kep[UxTypes.SYSgetsignal] := UGetSignal; (* Ux only *)
  kep[UxTypes.SYSgetspaceinfo] := UGetSpaceInfo; (* Ux only *)
  kep[SYSgetuid] := UGetUID;
  kep[SYSkill] := UKill;
  kep[SYSkillpg] := UKillPG;
  kep[SYSquota] := UQuota;
  kep[SYSsetgroups] := USetGroups;
  kep[SYSsethostid] := USetHostID;
  kep[SYSsethostname] := USetHostName;
  kep[SYSsetpgrp] := USetPGRP;
  kep[SYSsetpriority] := USetPriority;
  kep[SYSsetregid] := USetREGID;
  kep[SYSsetreuid] := USetREUID;
  kep[SYSsetrlimit] := USetRLimit;
  kep[SYSsigblock] := USigBlock;
  kep[SYSsigpause] := USigPause;
  kep[SYSsigsetmask] := USigSetMask;
  kep[SYSsigstack] := USigStack;
  kep[SYSsigvec] := USigVec;
  kep[UxTypes.SYSstartnewspace] := UStartNewSpace; (* Ux only *)
  kep[UxTypes.SYSstrace] := USTrace; (* Ux only *)
  kep[UxTypes.SYSterminate] := UTerminate; (* Ux only *)
  kep[SYSvfork] := UVFork;
  kep[SYSwait] := UWait;
  traceKernelCalls := FALSE;
END Init;

PROCEDURE PIDsPGRP(pid: OS.PID): OS.PID RAISES {OS.Error};
  VAR u: Ux.T;
  BEGIN
    LOCK gx DO u := ProcessFromPID(pid); RETURN u^.pgrp; END;
  END PIDsPGRP;

PROCEDURE PipeHasNoReader(u: Ux.T) RAISES {OS.Error, Thread.Alerted};
  BEGIN
    LOCK gx DO
      IF u^.p^.sigVec[OS.SigPipe].handler.h = UxTypes.SignalIgnore THEN
        RAISE(OS.Error, OS.PipeHasNoReaderEC);
      ELSE
        PostSignal(u, OS.SigPipe);
        WHILE NOT u^.p^.terminating DO Thread.AlertWait(gx, terminating); END;
      END;
    END;
  END PipeHasNoReader;

PROCEDURE RaiseError(e: UxTypes.SysError) RAISES {Error};
  BEGIN
    ASSERT(e # UxTypes.ok); (* unimpl. in fs *)
    RAISE(Error, e);
  END RaiseError;

PROCEDURE Register(entry: UxTypes.KernelEntry; proc: Ux.EntryProc) RAISES {};
  BEGIN
    ASSERT(kep[entry] = UUnimplemented, "UxPro.Register: Overwrite");
    kep[entry] := proc;
  END Register;

PROCEDURE ReleaseProcess(u: Ux.T) RAISES {};
  BEGIN ThreadsPort.Release(gx); END ReleaseProcess;

PROCEDURE ReleaseProcessTemplate(t: OS.ProcessTemplate) RAISES {};
  BEGIN ThreadsPort.Release(gx); END ReleaseProcessTemplate;

PROCEDURE SetEUser(u: Ux.T; euser: User.T) RAISES {};
  BEGIN LOCK gx DO u^.euser := euser; END; END SetEUser;

PROCEDURE SendTTSignal(
    u: Ux.T;
    signal: OS.Signal)
    : TTResult
  RAISES {OS.Error};
  VAR ec: OS.EC;
  BEGIN
    IF signal = OS.SigTTIn THEN
```

```
      ec := OS.NotTtyOwnerReadEC;
    ELSE
      ASSERT(signal = OS.SigTTOu, "UxPro.SendTTSignal given wrong signal");
      ec := OS.NotTtyOwnerWriteEC;
    END;
    LOCK gx DO
      IF u^.p^.watcherThread = Thread.Self() THEN
        IF DoSigEffective(u, signal) THEN
          RAISE(OS.Error, ec);
        ELSE (* not effective *)
          RETURN TTNotEffective;
        END; (* IF effective *)
      END; (* IF watcher *)
      IF u^.p^.state # Stopped THEN
        IF DoSigEffective(u, signal) THEN
          DoSignalPG(u^.pgrp, signal);
          RETURN TTSentSignal
        ELSE
          RAISE(OS.Error, ec);
        END; (* IF effective *)
      ELSE (* not running *)
        RETURN TTNotRunning;
      END; (* IF running *)
    END; (* LOCK gx *)
  END SendTTSignal;

PROCEDURE Signal(u: Ux.T; signal: OS.Signal) RAISES {};
  BEGIN
    LOCK gx DO
      IF u^.p^.state # Terminated THEN PostSignal(u, signal); END;
    END;
  END Signal;


(**************************)
(* Kernel entry procedures *)
(**************************)

PROCEDURE UCheckPW(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* OLD OS ONLY *)
  BEGIN RETURN ORD(TRUE); END UCheckPW;

PROCEDURE UExecVE(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR name: Text.T; file: OS.File; ok: BOOLEAN;
  BEGIN
    IF UxAS.IsTopaz(u) THEN RaiseError(UxTypes.EINVAL); END;
    name := UxAS.GetText(u, args[0]);
    file := UxFile.OpenForLoading(u, NIL, name);
    TRY
      ok := DoExec(u, file, NIL, NIL, name, NIL, NIL, u, args[1], args[2]);
      UxFile.Exec2(u);
    EXCEPT
      Thread.Alerted: ok := FALSE;
    END;
    IF NOT ok THEN LOCK gx DO PostTerminate(u, OS.SigKill, FALSE); END; END;
    RETURN 0;
  END UExecVE;

PROCEDURE UExit(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    LOCK gx DO
      u^.p^.ws.wRetCode := args[0] MOD 256;
      PostTerminate(u, OS.SigNone, FALSE);
    END;
    RETURN 0;
  END UExit;

PROCEDURE UFork(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN DoFork(u, (*vfork:*) FALSE); END UFork;

PROCEDURE UGetGID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN DoNothing(u); END UGetGID;

PROCEDURE UGetHostID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN hostID; END UGetHostID;

PROCEDURE UGetHostName(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR name: Text.T; count: CARDINAL; zero: CHAR;
  BEGIN
    TRY
      name := NSUtil.GetMyHost();
    EXCEPT
    | RPCRuntime.CallFailed: RaiseError(UxTypes.ETIMEDOUT);
    | NS.LabelNotFound: RaiseError(UxTypes.ENOENT);
    END;
    LOCK gx DO
      count := UxAS.PutText(u, args[0], name, LOOPHOLE(args[1], CARDINAL));
      IF count < LOOPHOLE(args[1], CARDINAL) THEN
        zero := 0C;
        UxAS.Put(u, args[0] + count, zero);
      END;
    END;
    RETURN 0;
  END UGetHostName;

PROCEDURE UGetPGRP(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR pid: OS.PID; uu: Ux.T;
  BEGIN
    pid := LOOPHOLE(args[0], OS.PID);
    LOCK gx DO
      IF pid.i = 0 THEN uu := u ELSE uu := ProcessFromPID(pid); END;
      RETURN uu^.pgrp.i;
    END;
  END UGetPGRP;

PROCEDURE UGetPID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    LOCK gx DO UxAS.R1Result(u, u^.p^.uParent^.pid.i); RETURN u^.pid.i; END;
  END UGetPID;

PROCEDURE UGetPriority(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    which: UxTypes.WhichPrio;
    who, ultrixPriority: INTEGER;
    found, match: BOOLEAN;
    uu: Ux.T;
  BEGIN
    IF ORD(LAST(UxTypes.WhichPrio)) < args[0] THEN
      RaiseError(UxTypes.EINVAL);
    END;
    which := LOOPHOLE(args[0], UxTypes.WhichPrio);
    who := LOOPHOLE(args[1], INTEGER);
    ultrixPriority := LAST(INTEGER); (* lowest Ultrix priority *)
    found := FALSE;
    LOCK gx DO
      uu := uHead;
      LOOP
        IF uu = NIL THEN EXIT; END;
```

```
        IF uu^.p^.state # Terminated THEN
          CASE which OF
          | UxTypes.prioProcess:
              match := (who = 0) AND (uu = u) OR (who = uu^.pid.i);
          | UxTypes.prioPgrp: match := who = uu^.pgrp.i;
          | UxTypes.prioUser: match := who = User.GetUserID(uu^.ruser);
          END;
          IF match THEN
            ultrixPriority :=
              MIN(ultrixPriority,
                ThreadToUltrixPriority(UxAS.GetClientPriority(uu)));
            found := TRUE;
            IF which = UxTypes.prioProcess THEN EXIT; END;
          END;
        END;
        uu := uu^.p^.uNext;
      END;
    END;
    IF NOT found THEN RaiseError(UxTypes.ESRCH); END;
    RETURN ultrixPriority;
  END UGetPriority;

PROCEDURE UGetPW(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    IF LOOPHOLE(args[1], INTEGER) < 0 THEN RaiseError(UxTypes.EINVAL); END;
    LOCK gx DO
      IF NOT User.IsSuper(u^.euser) THEN RaiseError(UxTypes.EPERM); END;
      RETURN UxAS.PutText(u, args[0], u^.password^, args[1]);
    END;
  END UGetPW;

PROCEDURE UGetRLimit(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR rLimit: UxTypes.RLimit;
  BEGIN
    rLimit := RLimit(u, LOOPHOLE(args[0], UxTypes.Resource));
    UxAS.Put(u, args[1], rLimit);
    RETURN 0;
  END UGetRLimit;

PROCEDURE UGetRUsage(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR p: T; rUsage: UxTypes.RUsage;
  BEGIN
    p := u^.p;
    System.Zero(System.Adr(rUsage), System.ByteSize(rUsage));
    (* args[0] tells 'who': self or children. *)
    IF LOOPHOLE(args[0], INTEGER) = UxTypes.rUsageSelf THEN
      rUsage.uTime := UxAS.GetClientTime(u);
      rUsage.sTime := GetWTime(u);
    ELSIF LOOPHOLE(args[0], INTEGER) = UxTypes.rUsageChildren THEN
      rUsage.uTime := p^.descCTime;
      rUsage.sTime := p^.descWTime;
    ELSE
      RaiseError(UxTypes.EINVAL);
    END;
    UxAS.Put(u, args[1], rUsage, 0, UxTypes.U11RUsageLen);
    RETURN 0;
  END UGetRUsage;

PROCEDURE UGetSignal(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* ONLY FOR OLD OS *)
  VAR p: T; s: OS.Signal;
  BEGIN
    p := u^.p;
    LOCK gx DO
      UxAS.SetHandler(u);
      FOR s := FIRST(OS.Signal) TO LAST(OS.Signal) DO
        IF s IN p^.alerted THEN
          EXCL(p^.alerted, s);
          RETURN LOOPHOLE(s, INTEGER);
        END;
      END;
      RETURN LOOPHOLE(OS.SigNone, INTEGER);
    END;
  END UGetSignal;

PROCEDURE UGetSpaceInfo(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* ONLY FOR OLD OS *)
  VAR
    pid: OS.PID;
    addr: UNSIGNED;
    nElts, nEltsSoFar: INTEGER;
    info: UxTypes.TrapSpaceInfo;
    uu: Ux.T;
    pp: T;
    j: CARDINAL;
  BEGIN
    pid.i := args[0];
    addr := args[1];
    nElts := args[2];
    IF nElts < 0 THEN RaiseError(UxTypes.EINVAL); END;
    nEltsSoFar := 0;
    LOCK gx DO
      uu := uHead;
      LOOP (* report next process *)
        IF nEltsSoFar = nElts THEN RETURN nEltsSoFar; END;
        LOOP (* find next unreported process *)
          IF uu = NIL THEN RETURN nEltsSoFar; END;
          pp := uu^.p;
          IF uu^.pid.i > pid.i THEN EXIT; END;
          uu := pp^.uNext;
        END;
        WITH info DO
          pid := uu^.pid.i;
          IF pp^.uParent # NIL THEN
            ppid := pp^.uParent^.pid.i;
          ELSE
            ppid := 0;
          END;
          pgrp := uu^.pgrp.i;
          uid := User.GetUserID(uu^.ruser);
          space := uu^.space;
          ctlTty := "  ";
          IF Text.GetSub(UxFile.GetControlTtyName(u), 0, ctlTty) = 0 THEN END;
          CASE pp^.state OF
          | Running: runState := UxTypes.SSRunning;
          | Stopped: runState := UxTypes.SSStopped;
          | Terminated: runState := UxTypes.SSTerminating;
          END;
          cTime.seconds := 0;
          cTime.microseconds := 0;
          wTime := cTime;
          topaz := FALSE;
          IF runState # UxTypes.SSTerminating THEN
            IF uu^.space # NubTypes.NullSpace THEN
              topaz := UxAS.IsTopaz(uu);
            END;
            IF (uu # uNub) AND (uu # uInit) AND (uu # Ux.uTaos) THEN
              cTime := UxAS.GetClientTime(uu);
```

```
          END;
          IF pp^.watcherThread # NIL THEN wTime := GetWTime(uu); END;
        END;
        ws := pp^.ws;
        FOR j := 0 TO HIGH(commandString) DO commandString[j] := ' '; END;
        IF Text.GetSub(pp^.name, 0, commandString) = 0 THEN END;
      END; (* WITH info *)
      UxAS.Put(u, addr, info);
      INC(addr, System.ByteSize(info));
      INC(nEltsSoFar, 1);
      pid.i := info.pid;
    END; (* report next process loop *)
  END; (* LOCK gx *)
  (* Can't get here. *)
END UGetSpaceInfo;

PROCEDURE UGetUID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    LOCK gx DO
      UxAS.R1Result(u, User.GetUserID(u^.euser));
      RETURN User.GetUserID(u^.ruser);
    END;
  END UGetUID;

PROCEDURE UKill(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR uu: Ux.T; euser: User.T; pid: OS.PID; signal: OS.SignalOrNone;
  BEGIN
    euser := u^.euser;
    IF args[1] > ORD(LAST(OS.Signal)) THEN RaiseError(UxTypes.EINVAL); END;
    signal := LOOPHOLE(args[1], OS.SignalOrNone);
    pid := LOOPHOLE(args[0], OS.PID);
    LOCK gx DO
      IF pid.i = -1 THEN (* broadcast *)
        IF NOT User.IsSuper(euser) THEN RaiseError(UxTypes.EPERM); END;
        uu := uHead;
        WHILE uu # NIL DO
          IF (uu # u) AND (uu^.p^.state # Terminated) THEN
            PostSignal(uu, signal);
          END;
          uu := uu^.p^.uNext;
        END;
      ELSIF pid.i = 0 THEN (* process group *)
        DoSignalPG(u^.pgrp, signal);
      ELSE (* specific process *)
        IF NOT DoKill(u, ProcessFromPID(pid), signal, euser) THEN
          RaiseError(UxTypes.EPERM);
        END;
      END;
    END; (* LOCK gx *)
    RETURN 0;
  END UKill;

PROCEDURE UKillPG(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR pgrp: OS.PID; uu: Ux.T; signal: OS.SignalOrNone; killed: BOOLEAN;
  BEGIN
    IF args[1] > ORD(LAST(OS.Signal)) THEN RaiseError(UxTypes.EINVAL); END;
    signal := LOOPHOLE(args[1], OS.SignalOrNone);
    pgrp := LOOPHOLE(args[0], OS.PID);
    IF pgrp.i = 0 THEN pgrp := u^.pgrp; END; (* default *)
    killed := FALSE;
    LOCK gx DO
      uu := uHead;
      WHILE uu # NIL DO
        IF (uu^.pgrp.i = pgrp.i) AND (uu^.p^.state # Terminated)
          AND DoKill(u, uu, signal, u^.euser) THEN
          killed := TRUE;
        END;
        uu := uu^.p^.uNext;
      END;
    END;
    IF NOT killed THEN RaiseError(UxTypes.ESRCH); END;
    RETURN 0;
  END UKillPG;

PROCEDURE UQuota(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN DoNothing(u); END UQuota;

PROCEDURE USetGroups(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    IF NOT User.IsSuper(u^.euser) THEN RaiseError(UxTypes.EPERM); END;
    RETURN DoNothing(u);
  END USetGroups;

PROCEDURE USetHostID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    IF NOT User.IsSuper(u^.euser) THEN RaiseError(UxTypes.EPERM); END;
    hostID := LOOPHOLE(args[0], INTEGER);
    RETURN 0;
  END USetHostID;

PROCEDURE USetHostName(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    RaiseError(UxTypes.EPERM); (* use name server instead *)
    RETURN 0;
  END USetHostName;

PROCEDURE USetPGRP(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR pid, pgrp: OS.PID; p: T; uu: Ux.T;
  BEGIN
    p := u^.p;
    pid := LOOPHOLE(args[0], OS.PID);
    pgrp := LOOPHOLE(args[1], OS.PID);
    LOCK gx DO
      IF pid.i = 0 THEN uu := u ELSE uu := ProcessFromPID(pid); END;
      IF NOT User.Equal(u^.euser, uu^.euser) AND NOT User.IsSuper(u^.euser)
         AND NOT Ancestor(u, uu) THEN
        RaiseError(UxTypes.EPERM);
      END;
      uu^.pgrp := pgrp;
    END;
    RETURN 0;
  END USetPGRP;

PROCEDURE USetPriority(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* We don't bother to check that the caller's effective user ID matches the
     real or effective user ID of the target, nor do we check that a caller
     specifying a negative Ultrix priority is the super user. *)
  VAR
    which: UxTypes.WhichPrio;
    priority: ThreadFriends.Priority;
    user: User.T;
  BEGIN
    IF ORD(LAST(UxTypes.WhichPrio)) < LOOPHOLE(args[0], UNSIGNED) THEN
      RaiseError(UxTypes.EINVAL);
    END;
    which := LOOPHOLE(args[0], UxTypes.WhichPrio);
    (* who = args[1] *)
    priority := UltrixToThreadPriority(LOOPHOLE(args[2], INTEGER));
```

```
    CASE which OF
    | UxTypes.prioProcess:
        LOCK gx DO
          IF args[1] = Ux.NullPID THEN args[1] := u^.pid.i; END; (* default *)
          PostSetPriority(ProcessFromPID(LOOPHOLE(args[1], OS.PID)),
            priority);
        END;
    | UxTypes.prioPgrp:
        DoSetPriorityPGRP(LOOPHOLE(args[1], OS.PGRP), priority);
    | UxTypes.prioUser:
        IF LAST(User.UserID) < args[1] THEN RaiseError(UxTypes.ESRCH); END;
        user := User.LookupUserByID(LOOPHOLE(args[1], User.UserID));
        IF user = NIL THEN RaiseError(UxTypes.ESRCH); END;
        DoSetPriorityUser(user, priority);
    END;
    RETURN 0;
  END USetPriority;

PROCEDURE USetREGID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN DoNothing(u); END USetREGID;

PROCEDURE USetREUID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR rNew, eNew: INTEGER; real, effective: User.T; password: Text.T;
  BEGIN
    rNew := LOOPHOLE(args[0], INTEGER);
    eNew := LOOPHOLE(args[1], INTEGER);
    IF (rNew < -1) OR (eNew < -1) THEN RaiseError(UxTypes.EINVAL); END;
    IF args[2] = 0 THEN
      password := NIL;
    ELSE
      password := UxAS.GetText(u, LOOPHOLE(args[2], UxAS.Addr));
    END;
    IF rNew = -1 THEN
      real := u^.ruser;
    ELSE
      real := User.LookupUserByID(rNew);
      IF real = NIL THEN RaiseError(UxTypes.EINVAL); END;
    END;
    IF eNew = -1 THEN
      effective := u^.euser;
    ELSE
      effective := User.LookupUserByID(eNew);
      IF effective = NIL THEN RaiseError(UxTypes.EINVAL); END;
    END;
    SetMyUser(u, effective, real, password);
    RETURN 0;
  END USetREUID;

PROCEDURE USetRLimit(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR resource: UxTypes.Resource; rLimit, oldRLimit: UxTypes.RLimit;
  BEGIN
    resource := LOOPHOLE(args[0], UxTypes.Resource);
    UxAS.Get(u, args[1], rLimit);
    oldRLimit := RLimit(u, resource);
    IF NOT User.IsSuper(u^.euser)
       AND ((oldRLimit.max < rLimit.max) OR (rLimit.cur < 0)
            OR (rLimit.max < rLimit.cur)) THEN
      RaiseError(UxTypes.EPERM);
    END;
    CASE resource OF
    | UxTypes.rLimitData, UxTypes.rLimitStack, UxTypes.rLimitCore:
        UxAS.SetRLimit(u, resource, rLimit);
    ELSE (* nothing *);
    END;
    RETURN 0;
  END USetRLimit;

PROCEDURE USigBlock(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR p: T; oldMask: OS.Signals;
  BEGIN
    p := u^.p;
    LOCK gx DO
      oldMask := p^.mask;
      p^.mask := p^.mask + (LOOPHOLE(args[0], OS.Signals) - PowerfulSignals);
    END;
    RETURN LOOPHOLE(oldMask, INTEGER)
  END USigBlock;

PROCEDURE USigPause(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR p: T; oldMask: OS.Signals;
  BEGIN
    p := u^.p;
    oldMask := p^.mask;
    p^.mask := LOOPHOLE(args[0], OS.Signals) - PowerfulSignals;
    LOCK gx DO
      WHILE (p^.posted - p^.mask) = NoSignals DO
        (* Wait for alert. *)
        TRY Thread.AlertWait(gx, never); EXCEPT Thread.Alerted: END;
      END;

      (* Set error flag (carry status bit). This must be done before the call
         to ActOnSignals, because ActOnSignals may call UxAS.PushHandlerCall,
         which saves the status bits itself. The corresponding error number,
         EINTR, is conveyed as the result of this procedure. *)
      UxAS.SetError(u);

      LOOP
        IF ActOnSignals(u, OS.SigNone, oldMask) THEN EXIT; END;
        TRY
          TDState.AwaitRunning(u^.pid.i, u^.space);
        EXCEPT
          Thread.Alerted:
        END;
      END;

      (* In case oldMask did not include mask, restore SIGPOSTED. *)
      IF (p^.posted - p^.mask) # NoSignals THEN
        INCL(p^.watcherEvents, WESignal);
      END;
    END; (* LOCK gx *)

    (* This procedure could terminate by calling RaiseError(EINTR), but it is
       sufficient to return EINTR since the error flag was already set. *)
    RETURN ORD(UxTypes.EINTR);
  END USigPause;

PROCEDURE USigSetMask(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR p: T; oldMask: OS.Signals;
  BEGIN
    p := u^.p;
    LOCK gx DO
      oldMask := p^.mask;
      p^.mask := LOOPHOLE(args[0], OS.Signals) - PowerfulSignals;
      IF (p^.posted - p^.mask) # NoSignals THEN (* restore SIGPOSTED *)
        INCL(p^.watcherEvents, WESignal);
      END;
    END;
    RETURN LOOPHOLE(oldMask, INTEGER);
```

```
  END USigSetMask;

PROCEDURE USigStack(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    IF args[1] # 0 THEN UxAS.Put(u, args[1], u^.p^.sigStack); END;
    IF args[0] # 0 THEN UxAS.Get(u, args[0], u^.p^.sigStack); END;
    RETURN 0;
  END USigStack;

PROCEDURE USigVec(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR p: T; signal: OS.Signal; vec: UxTypes.SignalVectorEntry;
  BEGIN
    p := u^.p;
    IF (args[0] < ORD(FIRST(OS.Signal))) OR (args[0] > ORD(LAST(OS.Signal)))
    THEN
      RaiseError(UxTypes.EINVAL);
    END;
    signal := LOOPHOLE(args[0], OS.Signal);
    IF (signal IN (PowerfulSignals - OS.Signals{OS.SigCont})) THEN
      RAISE(OS.Error, OS.BadStateForSignalEC);
    END;
    IF args[2] # 0 THEN UxAS.Put(u, args[2], p^.sigVec[signal]); END;
    IF args[1] # 0 THEN
      UxAS.Get(u, args[1], vec);
      IF (signal = OS.SigCont) AND (vec.handler.h = UxTypes.SignalIgnore) THEN
        RAISE(OS.Error, OS.BadStateForSignalEC);
      END;
      (* ONLY FOR OLD OS: *)
      IF vec.handler.h = UxTypes.SignalAlert THEN UxAS.SetHandler(u); END;
      vec.mask := vec.mask - PowerfulSignals;
      LOCK gx DO
        p^.sigVec[signal] := vec;
        IF vec.handler.h = UxTypes.SignalIgnore THEN
          EXCL(p^.posted, signal);
        END;
      END;
    END;
    RETURN 0;
  END USigVec;

PROCEDURE UStartNewSpace(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* ONLY FOR OLD OS *)
  VAR name: Text.T; uNew: Ux.T; file: OS.File; success: BOOLEAN;
  BEGIN
    name := UxAS.GetText(u, args[0]);
    file := UxFile.OpenForLoading(u, NIL, name);
    New(u, uNew);
    TRY
      success := FALSE;
      IF NOT DoExec(uNew, file, NIL, NIL, name, NIL, NIL, u, args[1], args[2])
      THEN
        RaiseError(UxTypes.EIO);
      END;
      UxFile.Exec2(uNew);
      success := TRUE; (* no more errors raised after this point *)
      LOCK gx DO DoForkCommon(u, uNew); END; (* LOCK gx *)
    FINALLY
      IF NOT success THEN DisposeResources(uNew); END;
    END;
    RETURN uNew^.pid.i;
  END UStartNewSpace;

PROCEDURE USTrace(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  TYPE STrace = OSFriends.STrace;
  BEGIN
    IF (args[0] > ORD(LAST(STrace))) OR (args[1] > ORD(LAST(STrace))) THEN
      RAISE(OS.Error, OS.InvalidArgumentEC);
    END;
    LOCK gx DO
      RemoteOSAS.SetSTrace(u, LOOPHOLE(args[0], STrace),
        LOOPHOLE(args[1], STrace));
    END;
    RETURN 0;
  END USTrace;

PROCEDURE UTerminate(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR signal: OS.Signal; dump: BOOLEAN;
  BEGIN
    IF (LOOPHOLE(FIRST(OS.Signal), UNSIGNED) <= args[0])
       AND (args[0] <= LOOPHOLE(LAST(OS.Signal), UNSIGNED)) THEN
      signal := LOOPHOLE(args[0], OS.Signal);
    ELSE
      signal := OS.SigIll;
    END;
    dump := args[1] # 0;
    LOCK gx DO PostTerminate(u, signal, dump); END;
    RETURN 0;
  END UTerminate;

PROCEDURE UUnimplemented(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    Signal(u, OS.SigSys);
    RaiseError(UxTypes.EINVAL);
    RETURN 0;
  END UUnimplemented;

PROCEDURE UVFork(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN DoFork(u, (*vfork:*) TRUE); END UVFork;

PROCEDURE UWait(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* This procedure implements both wait and wait3. wait(s) is equivalent to
     wait3(s, 0, 0). Since 'args' has missing arguments filled in with zeros,
     the right thing happens automatically. *)
  VAR
    uu: Ux.T;
    p, pp: T;
    noChildren, noHang, untraced: BOOLEAN;
    rUsage: UxTypes.RUsage;
  PROCEDURE R0AndR1Results(): INTEGER;
    VAR ws32: BITS 32 FOR UxTypes.WStatus;
    BEGIN
      ws32 := pp^.ws; (* 0's high order bytes *)
      UxAS.R1Result(u, LOOPHOLE(ws32, INTEGER));
      RETURN LOOPHOLE(uu^.pid, INTEGER)
    END R0AndR1Results;
  BEGIN
    noHang := (UxTypes.WNOHANG IN LOOPHOLE(args[1], UxTypes.WOptions));
    untraced := (UxTypes.WUNTRACED IN LOOPHOLE(args[1], UxTypes.WOptions));
    p := u^.p;
    LOCK gx DO
      LOOP
        uu := uHead; (* uHead can't be NIL *) (* or child ! *)
        noChildren := TRUE;
        LOOP
          pp := uu^.p;
          IF pp^.uParent = u THEN
            noChildren := FALSE;
            IF pp^.state = Terminated THEN
```

```
            IF args[2] # 0 THEN
              System.Zero(System.Adr(rUsage), System.ByteSize(rUsage));
              rUsage.uTime := pp^.descCTime;
              rUsage.sTime := pp^.descWTime;
              UxAS.Put(u, args[2], rUsage, 0, UxTypes.U11RUsageLen);
            END;
            Remove(uu);
            RETURN R0AndR1Results();
          END;
          IF (pp^.state = Stopped) AND pp^.untraced AND untraced THEN
            pp^.untraced := FALSE;
            RETURN R0AndR1Results();
          END;
        END;
        IF pp^.uNext = NIL THEN EXIT; END;
        uu := pp^.uNext;
      END;
      IF noChildren THEN UxAS.SetError(u); RETURN ORD(UxTypes.ECHILD); END;
      IF noHang THEN RETURN 0; END;
      Thread.AlertWait(gx, p^.childStateChanged); (* alert => signal *)
    END;
  END;
END UWait;


(*************************************************************)
(* Procedures exported thru OS via RemoteOSPro and RemoteOS *)
(*************************************************************)

PROCEDURE CloseTemplate(u: Ux.T; t: OS.ProcessTemplate) RAISES {OS.Error};
  VAR remove: BOOLEAN;
  BEGIN
    IF t = NIL THEN RAISE(OS.Error, OS.InvalidArgumentEC); END;
    LOCK gx DO
      remove := t^.p^.state = Nascent;
      IF remove THEN t^.p^.state := Terminated; RemoveTemplate(u, t); END;
    END;
    IF remove THEN DisposeResources(t, TRUE); END;
  END CloseTemplate;

PROCEDURE GetProcessInfo(
    u: Ux.T;
    pid: OS.PID;
    VAR processInfoOUT: OS.ProcessInfo;
    getUser: BOOLEAN)
  RAISES {OS.Error};
  VAR
    uu: Ux.T;
    pp: T;
    info: OS.ProcessInfo;
    targetState: TDState.TargetState;
    signal: OS.Signal;
  BEGIN
    IF pid.i = Ux.NullPID THEN pid := u^.pid; END; (* default *)
    LOCK gx DO
      uu := ProcessFromPID(pid, (*esrch:*) TRUE, (*terminatedOK:*) TRUE);
      pp := uu^.p;
      (* The compiler says it is too dumb to allow WITH to open an RC VAR
         parameter, so we use a temporary. *)
      System.Zero(System.Adr(info), System.ByteSize(info));
      info.pid := pid;
      WITH info DO
        IF pp^.uParent # NIL THEN ppid := pp^.uParent^.pid; END;
        pgrp := uu^.pgrp;
        IF getUser THEN
          effUser := User.GetUserName(uu^.euser);
          realUser := User.GetUserName(uu^.ruser);
        END;
        CASE pp^.state OF
        | Running: processState := OS.PSRunning;
        | Stopped: processState := OS.PSStopped;
        | Terminated: processState := OS.PSTerminating;
        END;
        IF processState = OS.PSTerminating THEN (* unanswerable questions *)
          (* ctlTty := NIL; *)
          (* windowSystem := NIL; *)
          (* space := NubTypes.NullSpace; *)
          (* numThreads := 0; *)
          (* debugState := DSRunning; *)
          (* debuggerConnected := FALSE; *)
          (* signalStates := default; *)
          (* residentBytes := 0; *)
          (* mappedBytes := 0; *)
          (* rUsage[OS.Self] := 0; *)
        ELSE
          ctlTty := UxFile.GetControlTtyName(uu);
          (* ***** windowSystem := ???; *)
          space := uu^.space;
          IF space # NubTypes.NullSpace THEN
            numThreads := TPFriends.NHandles(space);
          END;

          (* If TDState.NewTarget has been called... *)
          IF (space # NubTypes.NullSpace) AND (space > Ux.uTaos^.space)
             AND NOT pp^.vForkChild THEN
            TDState.GetState(pid.i, space, debuggerConnected, targetState);
            CASE targetState OF
            | TDState.Running: debugState := OS.DSRunning;
            | TDState.Stopped: debugState := OS.DSStopped;
            | TDState.Trapped: debugState := OS.DSTrapped;
            END;
          ELSE
            debugState := OS.DSRunning;
          END;
          FOR signal := FIRST(OS.Signal) TO LAST(OS.Signal) DO
            signalStates[signal] :=
              GetSignalState(pp^.sigVec[signal].handler, signal IN pp^.mask);
          END;
          IF (uu # uInit) THEN
            UxAS.GetInfo(uu, mappedBytes, residentBytes);
            IF (uu # Ux.uTaos) AND (uu # uNub) THEN
              rUsage[OS.Self].userTime := UxAS.GetClientTime(uu);
            END;
          END;
          IF pp^.watcherThread # NIL THEN
            rUsage[OS.Self].systemTime := GetWTime(uu);
          END;
        END;
        rUsage[OS.Children].userTime := pp^.descCTime;
        rUsage[OS.Children].systemTime := pp^.descWTime;
        command := pp^.name;
      END; (* WITH *)
      processInfoOUT := info;
    END; (* LOCK gx *)
  END GetProcessInfo;

PROCEDURE NewProcessTemplate(u: Ux.T): OS.ProcessTemplate RAISES {};
  VAR t: Ux.T;
```

```
  BEGIN
    New(u, t, (*vFork:*) FALSE, (*topaz:*) TRUE);
    InsertTemplate(u, t);
    RETURN t;
  END NewProcessTemplate;

PROCEDURE NextProcess(u: Ux.T; pid: OS.PID): OS.PID RAISES {};
  VAR pidNext: OS.PID;
  BEGIN
    pidNext.i := Ux.NullPID;
    LOCK gx DO
      IF pid.i = Ux.NullPID THEN
        IF uHead # NIL THEN pidNext := uHead^.pid; END;
      ELSIF (uHead # NIL) AND (pid.i < uTail^.pid.i) THEN
        IF (uRover = NIL) OR (pid.i < uRover^.pid.i) THEN
          uRover := uHead;
        END;
        (* Now we know that uRover # NIL and uRover^.pid.i <= pid.i <
           uTail^.pid.i *)
        WHILE uRover^.pid.i <= pid.i DO uRover := uRover^.p^.uNext; END;
        (* Now we know that uRover^.pid is the least PID larger than pid. *)
        pidNext := uRover^.pid;
      ELSE (* pid greater than any in list *)
      END;
    END; (* LOCK *)
    RETURN pidNext;
  END NextProcess;

PROCEDURE SendSignal(
    u: Ux.T;
    pid: OS.PID;
    signal: OS.SignalOrNone;
    euser: User.T)
  RAISES {OS.Error};
  VAR user: User.T;
  BEGIN
    LOCK gx DO
      user := u^.euser;
      IF (euser # NIL) THEN
        IF NOT User.IsSuper(user) THEN
          RAISE(OS.Error, OS.NotSuperUserEC);
        END;
        user := euser;
      END;
      IF pid.i = Ux.NullPID THEN pid := u^.pid; END; (* default *)
      IF NOT DoKill(u, ProcessFromPID(pid), signal, user) THEN
        RAISE(OS.Error, OS.NotOwnerEC);
      END;
    END; (* LOCK gx *)
  END SendSignal;

PROCEDURE SendSignalToGroup(
    u: Ux.T;
    pgrp: OS.PGRP;
    signal: OS.SignalOrNone;
    euser: User.T)
  RAISES {OS.Error};
  VAR user: User.T; uu: Ux.T; killed: BOOLEAN;
  BEGIN
    killed := FALSE;
    LOCK gx DO
      user := u^.euser;
      IF (euser # NIL) THEN
        IF NOT User.IsSuper(user) THEN
          RAISE(OS.Error, OS.NotSuperUserEC);
        END;
        user := euser;
      END;
      IF pgrp.i = Ux.NullPID THEN pgrp := u^.pgrp; END; (* default *)
      uu := uHead;
      WHILE uu # NIL DO
        IF (uu^.pgrp.i = pgrp.i) AND (uu^.p^.state # Terminated)
           AND DoKill(u, uu, signal, user) THEN
          killed := TRUE;
        END;
        uu := uu^.p^.uNext;
      END;
    END;
    IF NOT killed THEN RAISE(OS.Error, OS.BadPIDEC); END;
  END SendSignalToGroup;

PROCEDURE SetMySignalState(
    u: Ux.T;
    signal: OS.Signal;
    state: OS.SignalState)
    : OS.SignalState
  RAISES {OS.Error};
  VAR p: T; e, eOld: UxTypes.SignalVectorEntry; mask, maskOld: BOOLEAN;
  BEGIN
    p := u^.p;
    CASE state OF
    | OS.SignalDefault: e.handler.h := UxTypes.SignalDefault; mask := FALSE;
    | OS.SignalIgnore:
        e.handler.h := UxTypes.SignalIgnore;
        mask := FALSE;
        IF signal IN PowerfulSignals THEN
          RAISE(OS.Error, OS.BadStateForSignalEC);
        END;
    | OS.SignalHandle:
        e.handler.h := UxTypes.SignalDefault;
        mask := TRUE;
        IF (signal IN (PowerfulSignals - OS.Signals{OS.SigCont})) THEN
          RAISE(OS.Error, OS.BadStateForSignalEC);
        END;
    END;
    e.mask := OS.Signals{};
    e.onStack := FALSE;
    LOCK gx DO
      eOld := p^.sigVec[signal];
      p^.sigVec[signal] := e;
      IF state = OS.SignalIgnore THEN EXCL(p^.posted, signal); END;
      maskOld := signal IN p^.mask;
      IF mask THEN
        INCL(p^.mask, signal);
      ELSE
        EXCL(p^.mask, signal);
        (* Restore SIGPOSTED: *)
        IF signal IN p^.posted THEN PostWatcherEvent(p, WESignal); END;
      END;
      Thread.Broadcast(p^.signaled); (* ensure SIGWAIT *)
      RETURN GetSignalState(eOld.handler, maskOld);
    END;
  END SetMySignalState;

PROCEDURE SetPGRP(u: Ux.T; t: OS.ProcessTemplate) RAISES {OS.Error};
  BEGIN
    IF t = NIL THEN RAISE(OS.Error, OS.InvalidArgumentEC); END;
    LOCK gx DO
```

```
      IF t^.p^.state # Nascent THEN
        RAISE(OS.Error, OS.InvalidArgumentEC);
      END;
      t^.pgrp.i := Ux.NullPID;
    END;
  END SetPGRP;

PROCEDURE SetSignalState(
    u: Ux.T;
    t: OS.ProcessTemplate;
    signal: OS.Signal;
    state: OS.SignalState)
  RAISES {OS.Error};
  VAR handler: UxTypes.SignalHandler; mask: BOOLEAN;
  BEGIN
    IF t = NIL THEN RAISE(OS.Error, OS.InvalidArgumentEC); END;
    LOCK gx DO
      IF t^.p^.state # Nascent THEN
        RAISE(OS.Error, OS.InvalidArgumentEC);
      END;
      CASE state OF
      | OS.SignalDefault: handler.h := UxTypes.SignalDefault; mask := FALSE;
      | OS.SignalIgnore:
          handler.h := UxTypes.SignalIgnore;
          mask := FALSE;
          IF signal IN PowerfulSignals THEN
            RAISE(OS.Error, OS.BadStateForSignalEC);
          END;
      | OS.SignalHandle:
          handler.h := UxTypes.SignalDefault;
          mask := TRUE;
          IF (signal IN (PowerfulSignals - OS.Signals{OS.SigCont})) THEN
            RAISE(OS.Error, OS.BadStateForSignalEC);
          END;
      END;
      t^.p^.sigVec[signal].handler := handler;
      IF mask THEN
        INCL(t^.p^.mask, signal);
      ELSE
        EXCL(t^.p^.mask, signal);
      END;
    END;
  END SetSignalState;

PROCEDURE SetUser(
    u: Ux.T;
    t: OS.ProcessTemplate;
    effective, real: User.T;
    pswd: Text.T)
  RAISES {OS.Error};
  VAR set: BOOLEAN;
  BEGIN
    IF (t = NIL) OR (effective = NIL) THEN
      RAISE(OS.Error, OS.InvalidArgumentEC);
    END;
    IF NOT ((real # NIL) AND User.Equal(effective, real)
            OR User.Equal(effective, u^.ruser)
            OR User.Equal(effective, u^.euser) OR User.IsSuper(u^.euser)) THEN
      RAISE(OS.Error, OS.NotSuperUserEC);
    END;
    LOCK gx DO
      IF t^.p^.state # Nascent THEN
        RAISE(OS.Error, OS.InvalidArgumentEC);
      END;
      IF real # NIL THEN
        set := TRUE;
        IF (pswd = NIL) AND User.Equal(real, u^.ruser) THEN
          set := FALSE;
        ELSIF (pswd = NIL) AND User.IsSuper(real) THEN
          IF NOT User.IsSuper(u^.euser) THEN
            RAISE(OS.Error, OS.NotSuperUserEC);
          END
        ELSE
          (* Specifying allowPasswordHack = TRUE to User.CheckPassword is to
             ease conversion from OLD OS ONLY ... it will go away before too
             long. *)
          IF NOT User.CheckPassword(real, pswd, TRUE) THEN
            RAISE(OS.Error, OS.InvalidCredentialsEC);
          END;
        END;
        IF set THEN
          t^.ruser := real;
          NEW(t^.password);
          t^.password^ := pswd;
        END;
      END;
      t^.euser := effective;
    END; (* LOCK *)
  END SetUser;

PROCEDURE StartProcess(
    u: Ux.T;
    file: OS.File;
    dir: OS.Dir;
    euser: User.T;
    name: Text.T;
    argv: Text.RefArray;
    t: OS.ProcessTemplate;
    relationship: OS.Relationship;
    env: Text.RefArray)
    : OS.PID
  RAISES {OS.Error};
  BEGIN
    IF t = NIL THEN
      New(u, t, (*vFork:*) FALSE, (*topaz:*) TRUE);
    ELSE
      LOCK gx DO
        IF t^.p^.state # Nascent THEN
          RAISE(OS.Error, OS.InvalidArgumentEC);
        END;
        (* Make template invalid for others before releasing gx. *)
        t^.p^.state := Running;
        RemoveTemplate(u, t);
      END;
    END;
    IF NOT DoExec(t, file, dir, euser, name, argv, env) THEN
      DisposeResources(t, TRUE);
      RAISE(OS.Error, OS.IOErrorEC);
    END;
    UxAS.SetClientPriority(t, t^.p^.priority);
    LOCK gx DO
      IF relationship = OS.Orphan THEN t^.p^.uParent := uInit; END;
      DoForkCommon(u, t);
      IF t^.pgrp.i = Ux.NullPID THEN t^.pgrp := t^.pid; END;
      UxFile.Exec2(t);
    END;
    RETURN t^.pid;
  END StartProcess;
```

```
PROCEDURE WaitForChild(
    u: Ux.T;
    pid: OS.PID;
    VAR (* OUT *) rUsage: OS.RUsage)
    : OS.WStatus
  RAISES {OS.Error, Thread.Alerted};
  VAR uu: Ux.T; pp: T; ws: OS.WStatus;
  BEGIN
    LOCK gx DO
      LOOP
        uu := ProcessFromPID(pid, (*esrch:*) TRUE, (*terminatedOK:*) TRUE);
        pp := uu^.p;
        IF pp^.state = Terminated THEN
          rUsage.userTime := pp^.descCTime;
          rUsage.systemTime := pp^.descWTime;
          IF pp^.ws.wTermSig = OS.SigNone THEN
            ws.reason := OS.Exited;
            ws.wRetCode := pp^.ws.wRetCode;
          ELSE
            ws.reason := OS.Killed;
            ws.wTermSig := pp^.ws.wTermSig;
            ws.wCoreDump := pp^.ws.wCoreDump;
          END;
          Remove(uu);
          EXIT;
        ELSIF (pp^.state = Stopped) AND pp^.untraced THEN
          ws.reason := OS.Stopped;
          ws.wStopSig := pp^.ws.wStopSig;
          pp^.untraced := FALSE;
          EXIT;
        ELSE
          Thread.AlertWait(gx, pp^.stateChanged);
        END;
      END; (* LOOP *)
    END; (* LOCK gx *)
    RETURN ws;
  END WaitForChild;

PROCEDURE WaitForSignal(
    u: Ux.T;
    allowed: OS.Signals)
    : OS.Signal
  RAISES {OS.Error, Thread.Alerted};
  VAR p: T; signal: OS.Signal; signalFound: OS.SignalOrNone;
  BEGIN
    p := u^.p;
    LOCK gx DO
      LOOP
        signalFound := OS.SigNone;
        (* Allowed signals must be defaulted and masked. *)
        FOR signal := FIRST(OS.Signal) TO LAST(OS.Signal) DO
          IF (signal IN allowed) THEN
            IF (p^.sigVec[signal].handler.h # UxTypes.SignalDefault)
               OR NOT (signal IN p^.mask) THEN
              RAISE(OS.Error, OS.BadStateForSignalEC);
            END;
            IF signal IN p^.posted THEN signalFound := signal; END;
          END;
        END;
        (* Is there an allowed, posted signal? *)
        IF signalFound # OS.SigNone THEN
          EXCL(p^.posted, signalFound);
          RETURN signalFound;
        END;
        (* Wait for posted*masked to be nonempty. *)
        Thread.AlertWait(gx, p^.signaled);
      END; (* LOOP *)
    END; (* LOCK gx *)
  END WaitForSignal;


(*********************************************************************)
(* Procedures exported thru OSFriends via RemoteOSPro and RemoteOS *)
(*********************************************************************)

PROCEDURE GetPriority(
    u: Ux.T;
    pid: OS.PID)
    : ThreadFriends.Priority
  RAISES {OS.Error};
  VAR uu: Ux.T;
  BEGIN
    LOCK gx DO
      IF pid.i = Ux.NullPID THEN pid := u^.pid; END; (* default *)
      uu := ProcessFromPID(pid);
      RETURN uu^.p^.priority;
    END;
  END GetPriority;

PROCEDURE SetPriority(
    u: Ux.T;
    t: OS.ProcessTemplate;
    priority: ThreadFriends.Priority)
  RAISES {OS.Error};
  BEGIN
    IF t = NIL THEN RAISE(OS.Error, OS.InvalidArgumentEC); END;
    LOCK gx DO
      IF t^.p^.state # Nascent THEN
        RAISE(OS.Error, OS.InvalidArgumentEC);
      END;
      t^.p^.priority := priority;
    END;
  END SetPriority;

PROCEDURE SetPriorityPID(
    u: Ux.T;
    pid: OS.PID;
    priority: ThreadFriends.Priority)
  RAISES {OS.Error};
  VAR p: T;
  BEGIN
    IF pid.i = Ux.NullPID THEN pid := u^.pid; END; (* default *)
    LOCK gx DO PostSetPriority(ProcessFromPID(pid), priority); END;
  END SetPriorityPID;

PROCEDURE SetPriorityPGRP(
    u: Ux.T;
    pgrp: OS.PGRP;
    priority: ThreadFriends.Priority)
  RAISES {OS.Error};
  BEGIN
    IF pgrp.i = Ux.NullPID THEN pgrp := u^.pgrp; END; (* default *)
    DoSetPriorityPGRP(pgrp, priority);
  END SetPriorityPGRP;

PROCEDURE SetPriorityUser(
    u: Ux.T;
```

```
    userName: Text.T;
    priority: ThreadFriends.Priority)
  RAISES {OS.Error};
  VAR user: User.T;
  BEGIN
    LOCK gx DO
      user := User.LookupUserByName(userName);
      IF user = NIL THEN RAISE(OS.Error, OS.BadPIDEC); END;
      DoSetPriorityUser(user, priority);
    END;
  END SetPriorityUser;

PROCEDURE SetMyUser(
    u: Ux.T;
    effective, real: User.T;
    pswd: Text.T)
  RAISES {OS.Error};
  VAR set: BOOLEAN;
  BEGIN
    IF effective = NIL THEN RAISE(OS.Error, OS.InvalidArgumentEC); END;
    IF NOT ((real # NIL) AND User.Equal(effective, real)
            OR User.Equal(effective, u^.ruser)
            OR User.Equal(effective, u^.euser) OR User.IsSuper(u^.euser)) THEN
      RAISE(OS.Error, OS.NotSuperUserEC);
    END;
    LOCK gx DO
      IF real # NIL THEN
        set := TRUE;
        IF (pswd = NIL) AND User.Equal(real, u^.ruser) THEN
          set := FALSE;
        ELSIF (pswd = NIL) AND User.IsSuper(real) THEN
          IF NOT User.IsSuper(u^.euser) THEN
            RAISE(OS.Error, OS.NotSuperUserEC);
          END
        ELSE
          (* Specifying allowPasswordHack = TRUE to User.CheckPassword is for
             compatibility with the OLD OS ONLY ... it will go away before too
             long. *)
          IF NOT User.CheckPassword(real, pswd, TRUE) THEN
            RAISE(OS.Error, OS.InvalidCredentialsEC);
          END;
        END;
        IF set THEN
          u^.ruser := real;
          NEW(u^.password);
          u^.password^ := pswd;
        END;
      END;
      u^.euser := effective;
    END; (* LOCK *)
  END SetMyUser;


(*********************************************************************)
(* Procedures exported thru OSSpecial via RemoteOSPro and RemoteOS *)
(*********************************************************************)

PROCEDURE Exit(u: Ux.T; status: OS.ExitStatus) RAISES {};
  BEGIN
    ASSERT(u # Ux.uTaos, "OS.Exit called from Taos");
    LOCK gx DO
      u^.p^.ws.wRetCode := status;
      PostTerminate(u, OS.SigNone, FALSE);
      (* We must not return until we are sure the client thread that called us
         has been stopped. However the watcher thread must wait for all the
         RPC server threads servicing this address space, including ours, to
         finish. *)
      WHILE NOT u^.p^.terminating DO Thread.Wait(gx, terminating); END;
    END;
  END Exit;

BEGIN initialized := FALSE; (* see exported procedure Init *) END UxPro.
```

M. D. Schroeder -- started June, 1985

UxPro.mod is the main control program of the Ultrix kernel simulation on the Firefly. It implements all transfers of control between the application and os address spaces: delivering signals to a process, resuming execution after a signal handler returns, Ultrix kernel calls, and the subsequent return. UxPro.mod also contains the implementing procedures for some Ultrix kernel calls. It dispatches the remaining Ultrix kernel calls to implementing procedures in other modules.

UxPro is synchronized with a global lock. To prevent unnecessary contention, this lock is never held when kernel entry procedures are called. Some of the kernel entry procedures inside UxPro.mod reacquire the global lock.

The Ultrix kernel simulator recognizes two kinds of address spaces -- Ultrix and Topaz. An Ultrix address space contains one thread and no RPC machinery. If either of these constraints are violated, then an address space is Topaz. Kernel calls are accepted from both sorts of address space. For a Topaz address space all kernel calls are serialized -- even when the current call involves a long-term wait. For a Topaz address space, any attempt to deliver a signal to a client-defined signal handler, or to use the kernel entries "fork" or "execve", will result in termination of the process (with a core image).

```
(* DIGITAL EQUIPMENT CORPORATION  CONFIDENTIAL AND PROPRIETARY *)
(* Last modified on Thu Jun 25 10:05:39 1987 by mcjones *)
(*      modified on Mon Nov 11 11:48:23 1985 by price *)

SAFE IMPLEMENTATION MODULE UxTime;

(* This module implements the interval timer kernel calls GetITimer and
   SetITimer and the time-of-day calls GetTimeOfDay and SetTimeOfDay. *)

IMPORT Base, OS, System, Time, Thread, User, Ux, UxAS, UxPro, UxTypes;
FROM UxTypes IMPORT ITimerVal, ITimer;

TYPE
  Timer = RECORD valid: BOOLEAN; target, interval: Time.T; END;
  T = Ux.UxTimeT;
  T =
    REF RECORD
      mutex: Thread.Mutex;
      timer: ARRAY ITimer OF Timer;
      timerThread: Thread.T; (* not created until first SetITimer *)
      exiting: BOOLEAN;
    END;

VAR
  zero, infinity: Time.T;
  epsilon: ARRAY ITimer OF Time.T; (* error tolerance *)
  signal: ARRAY ITimer OF OS.Signal;


(***********************)
(* Internal procedures *)
(***********************)

PROCEDURE GetTimerVal(u: Ux.T; t: ITimer; VAR (*out*) tv: ITimerVal);
  (* Sets 'tv' to current value of timer 't' for process 'u'. *)
  (* Called with u^.i^.mutex LOCKed. *)
  VAR timer: Timer; cur: Time.T;
  BEGIN
    timer := u^.i^.timer[t];
    tv.interval := timer.interval;
    tv.value.seconds := 0;
    tv.value.microseconds := 0;
    IF timer.valid THEN
      CASE t OF
      | iTimerReal: cur := Time.Now();
      | iTimerVirtual: cur := UxAS.GetClientTime(u);
      | iTimerProf: cur := Time.Add(UxAS.GetClientTime(u), UxPro.GetWTime(u));
      END;
      IF Time.Compare(timer.target, cur) = Base.Gt THEN
        tv.value := Time.Subtract(timer.target, cur);
      END;
    END;
  END GetTimerVal;

PROCEDURE IntervalTimer(ref: REFANY): REFANY;
  (* There is a separate thread executing this procedure for each address
     space that has invoked the USetITimer kernel call. *)
  VAR u: Ux.T; i: T; t: ITimer; cur, delta, minDelta: Time.T; tv: ITimerVal;
  BEGIN
    u := NARROW(ref, Ux.T);
    i := u^.i;
    LOOP
      LOCK i^.mutex DO
        minDelta := infinity; (* in case no timers set *)
        FOR t := FIRST(ITimer) TO LAST(ITimer) DO
          WITH i^.timer[t] DO
            IF valid THEN
              CASE t OF
              | iTimerReal: cur := Time.Now();
              | iTimerVirtual: cur := UxAS.GetClientTime(u);
              | iTimerProf:
                  cur := Time.Add(UxAS.GetClientTime(u), UxPro.GetWTime(u));
              END;
              IF NOT (Time.Compare(Time.Add(cur, epsilon[t]), target) =
                      Base.Lt) THEN
                UxPro.Signal(u, signal[t]);
                tv.interval := interval;
                tv.value := interval;
                SetTimer(u, t, tv);
              END;
            END; (* IF valid *)
            IF valid THEN (* SetTimer can set valid := FALSE *)
              delta := Time.Subtract(target, cur);
              IF Time.Compare(delta, minDelta) = Base.Lt THEN
                minDelta := delta;
              END;
            END; (* IF valid *)
          END; (* WITH *)
        END; (* FOR *)
      END; (* LOCK *)
      TRY
        Time.Pause(MIN(minDelta.seconds, 1000) * 1000000 +
          minDelta.microseconds);
      EXCEPT
        Thread.Alerted:
      END;
      LOCK i^.mutex DO IF i^.exiting THEN RETURN NIL; END; END;
    END; (* LOOP *)
  END IntervalTimer;

PROCEDURE New(u: Ux.T);
  (* Allocates and initializes u^.i^. *)
  VAR i: T; t: ITimer;
  BEGIN
    NEW(u^.i);
    i := u^.i;
    FOR t := FIRST(ITimer) TO LAST(ITimer) DO
      WITH i^.timer[t] DO
        valid := FALSE;
        interval := zero;
        target := infinity;
      END;
    END;
    i^.exiting := FALSE;
    Thread.InitMutex(i^.mutex);
  END New;

PROCEDURE SetTimer(u: Ux.T; t: ITimer; tv: ITimerVal);
  (* Sets timer 't' of process 'u' using the value 'tv'. *)
  (* Called with u^.i^.mutex LOCKed. *)
  VAR i: T; base: Time.T;
  BEGIN
    i := u^.i;
    IF (tv.value.seconds # 0) OR (tv.value.microseconds # 0) THEN
      i^.timer[t].valid := TRUE;
      i^.timer[t].interval := tv.interval;
      CASE t OF
      | iTimerReal: base := Time.Now();
```

```
      | iTimerVirtual: base := UxAS.GetClientTime(u);
      | iTimerProf:
          base := Time.Add(UxAS.GetClientTime(u), UxPro.GetWTime(u));
      END;
      i^.timer[t].target := Time.Add(base, tv.value);
    ELSE
      i^.timer[t].valid := FALSE;
    END;
  END SetTimer;


(***********************)
(* Exported procedures *)
(***********************)

PROCEDURE Exit(u: Ux.T) RAISES {};
  VAR i: T;
  BEGIN
    i := u^.i;
    IF (i = NIL) OR (i^.timerThread = NIL) THEN RETURN END;
    LOCK i^.mutex DO i^.exiting := TRUE; END;
    Thread.Alert(i^.timerThread);
  END Exit;

PROCEDURE Init() RAISES {};
  BEGIN
    zero.seconds := 0;
    zero.microseconds := 0;

    infinity.seconds := LAST(CARDINAL);
    infinity.microseconds := 999999;

    epsilon[iTimerReal] := zero;
    epsilon[iTimerVirtual].seconds := 0;
    epsilon[iTimerVirtual].microseconds := 250000; (* one quarter second *)
    epsilon[iTimerProf] := epsilon[iTimerVirtual];

    signal[iTimerReal] := OS.SigAlrm;
    signal[iTimerVirtual] := OS.SigVTAlrm;
    signal[iTimerProf] := OS.SigProf;

    UxPro.Register(UxTypes.SYSgetitimer, UGetITimer);
    UxPro.Register(UxTypes.SYSsetitimer, USetITimer);
    UxPro.Register(UxTypes.SYSgettimeofday, UGetTimeOfDay);
    UxPro.Register(UxTypes.SYSsettimeofday, USetTimeOfDay);
  END Init;


(**************************)
(* Kernel entry procedures *)
(**************************)

PROCEDURE UGetITimer(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR t: ITimer; tv: ITimerVal;
  BEGIN
    IF u^.i = NIL THEN New(u); END;
    IF args[0] > ORD(LAST(ITimer)) THEN UxPro.RaiseError(UxTypes.EINVAL); END;
    t := LOOPHOLE(args[0], ITimer);
    LOCK u^.i^.mutex DO GetTimerVal(u, t, tv); UxAS.Put(u, args[1], tv); END;
    RETURN (0);
  END UGetITimer;

PROCEDURE UGetTimeOfDay(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR time: Time.T; zone: UxTypes.TimeZone;
  BEGIN
    time := Time.Now();
    UxAS.Put(p, args[0], time);
    IF args[1] # 0 THEN
      Time.GetTimeZone(zone);
      UxAS.Put(p, args[1], zone);
    END;
    RETURN 0;
  END UGetTimeOfDay;

PROCEDURE USetITimer(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR t: ITimer; tvNew, tvOld: ITimerVal;
  BEGIN
    IF u^.i = NIL THEN New(u); END;
    IF args[0] > ORD(LAST(ITimer)) THEN UxPro.RaiseError(UxTypes.EINVAL); END;
    t := LOOPHOLE(args[0], ITimer);
    UxAS.Get(u, args[1], tvNew);
    IF (LOOPHOLE(tvNew.value.seconds, UNSIGNED) > System.MaxCard)
       OR (LOOPHOLE(tvNew.value.microseconds, UNSIGNED) > 999999)
       OR (LOOPHOLE(tvNew.interval.seconds, UNSIGNED) > System.MaxCard)
       OR (LOOPHOLE(tvNew.interval.microseconds, UNSIGNED) > 999999) THEN
      UxPro.RaiseError(UxTypes.EINVAL);
    END;
    LOCK u^.i^.mutex DO
      IF args[2] # 0 THEN
        GetTimerVal(u, t, tvOld);
        UxAS.Put(u, args[2], tvOld);
      END;
      SetTimer(u, t, tvNew);
      IF u^.i^.timerThread = NIL THEN
        u^.i^.timerThread := Thread.Fork(IntervalTimer, u);
      ELSE
        Thread.Alert(u^.i^.timerThread);
      END;
    END;
    RETURN (0);
  END USetITimer;

PROCEDURE USetTimeOfDay(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR time: Time.T; zone: UxTypes.TimeZone;
  BEGIN
    (*  IF NOT User.IsSuper(p^.euser) THEN *)
    UxPro.RaiseError(UxTypes.EPERM);
    (* ELSE UxAS.Get(p, args[0], time); UxAS.Get(p, args[1], zone);
       (* ***** what if args[1] = 0 ? *) Time.SetNow(time.seconds,
       time.microseconds); (* ***** Time.SetTimeZone(zone); *) END; *)
    RETURN 0;
  END USetTimeOfDay;

BEGIN END UxTime.
```