```
(* Last modified on Fri Nov 22 15:34:12 1985 by mcjones *)

(* Base definition module for the Firefly Topaz Ultrix kernel emulation. *)

SAFE DEFINITION MODULE Ux;

IMPORT Text, Unix42;

CONST
  SuperUID = 0; (* user id of 'super user' *)
  InitPID = 1; (* id of the 'init' process *)

TYPE
  ProcessID = INTEGER;
  UserID = INTEGER;

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

  T = REF TObject;

  TObject = RECORD
      pT: UxProT;
      aT: UxAST;
      fT: UxFileT;
      tT: UxTimeT; (* NIL until first call to getitimer/setitimer *)
    END;

TYPE ErrorArg = RECORD e: Unix42.SysError; s: Unix42.Signal; END;
EXCEPTION Error(ErrorArg);

TYPE RefArrayOfText = REF ARRAY OF Text.T; (* move to Ref someday? *)

TYPE
  ArgList = ARRAY [0..6] OF UNSIGNED;
  EntryProc = PROCEDURE(T, VAR ArgList): INTEGER;
    (* Such procedures are expected to raise 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 Mon Sep  8 11:19:23 1986 by mcjones *)

SAFE DEFINITION MODULE UxAS;

IMPORT NubTypes, System, Text, Thread, Time, TPFriends, Ux, Unix42;

TYPE
  T = Ux.UxAST;
  ClientThread; (* handle for individual client thread *)

  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: Unix42.KernelEntry; argList: Ux.ArgList;
    | Exception: sig: Unix42.Signal; code: INTEGER;
    | Alert:
    | HandlerReturn: mask: Unix42.SigMask; onStack: BOOLEAN;
    END;
  END;

CONST AlertPriority = TPFriends.PriIOLow;

PROCEDURE AlertHandler(aT: T) RAISES {};
(* Sends an alert to the handler thread specified by a prior call
   'SetHandler(aT)'. *)

PROCEDURE BackupKernelCall(
    aT: T;
    entry: Unix42.KernelEntry)
  RAISES {Ux.Error};
(* Backs up the program counter of the current thread of 'aT' so it will
   reexecute the current kernel call, 'entry', when restarted. *)

PROCEDURE CopySpace(aT, aTNew: T; vFork: BOOLEAN) RAISES {Ux.Error};
(* Assigns an actual VM.Space to 'aTNew'.  (If 'vFork=TRUE', this is just
   the space of 'aT'; 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 'aT' to 'aTNew'. Assumes no
   threads are running in the address space 'aT'. *)

PROCEDURE Get(
    aT: T;
    a: UNSIGNED;
    VAR v: ARRAY OF System.Byte;
    offset: CARDINAL := 0;
    count: CARDINAL := System.MaxCard)
  RAISES {Ux.Error};
(* 'a' is interpreted as an address in the address space 'aT'. 'v' is filled,
   starting at 'v[offset]', with 'MIN(count, NUMBER(v)-offset)' bytes read
   starting at 'a'. Calls RaiseError(Unix42.EFAULT) if an address fault is
   encountered. *)

PROCEDURE GetArrayOfText(
    aT: T;
    a: UNSIGNED)
    : Ux.RefArrayOfText
  RAISES {Ux.Error};
(* 'a' is interpreted as an address in the address space 'aT'. 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.
   Calls UxPro.RaiseError(Unix42.EFAULT) if there is an address fault. *)

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

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

PROCEDURE GetRLimit(
    aT: T;
    resource: Unix42.Resource;
    VAR (*out*) rLimit: Unix42.RLimit)
  RAISES {};
(* Returns the current resource limit values for address space 'aT' when
   'resource' is one of Unix42.rLimitData or Unix42.rLimitStack. No-ops for
   other values of 'resource'. *)

PROCEDURE GetSpace(aT: T): NubTypes.Space RAISES {};
(* Returns the address space number of 'aT'. This should NOT be depended on to
   remain constant across kernel calls (e.g. VFork). *)

PROCEDURE GetText(aT: T; a: UNSIGNED): Text.T RAISES {Ux.Error};
(* Returns a Text with the contents of the null-terminated character string
   starting at the given location 'a' within the address space 'aT'. Calls
   RaiseError(Unix42.EFAULT) if an address fault is encountered. *)

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

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

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

PROCEDURE Load(
    p: Ux.T;
    name: Text.T;
    argv, envp: Ux.RefArrayOfText;
    aTArgs: T;
    argva, envpa: UNSIGNED;
    vFork: BOOLEAN) (* if true, must create new space *)
    : BOOLEAN
  RAISES {Ux.Error, Thread.Alerted};
(* Arranges for process 'p' to execute the file 'name' (either an a.out file
   or a script file whose first line names the a.out file of an interpreter).
   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
   'aTArgs' (possibly 'p^.aT') using addresses 'argva' and 'envpa'
   respectively. Assumes the current thread of 'p' 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, Ux.Error is raised with the
   appropriate Unix42.SysError. After that point, any errors are reflected by
   returning FALSE, or raising Thread.Alerted. Success is reflected by
   returning TRUE. *)

PROCEDURE MakeCoreImage(
    p: Ux.T;
    sig: Unix42.Signal)
  RAISES {Ux.Error, Thread.Alerted};
(* Writes out a file named 'core' in the working directory of process 'p'
   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 'p'. *)

PROCEDURE New(aT: T): T RAISES {Ux.Error};
(* Returns a new address space descriptor and allocates an initial client
   thread (no VM.Space is actually assigned).  If 'aT' is nonnull, the
   relevant fields of its ThreadsPort.State (except registers, copied by
   CopySpace) are copied to the new thread, otherwise those fields are set
   to appropriate initial values. The new thread becomes the current thread,
   so it will be started when Start is called. Before that, CopySpace or
   Load must be called to assign and fill in the address space. *)

PROCEDURE PushHandlerCall(
    a: T;
    sig: Unix42.Signal;
    code: INTEGER;
    handler: Unix42.SigHandler;
    mask: Unix42.SigMask;
    onStack: Unix42.SigOnStack;
    switchStacks: BOOLEAN;
    sp: UNSIGNED)
  RAISES {Ux.Error};
(* Pushes context onto stack of current thread of address space 'aT' so that
   it is ready to execute 'handler' with arguments 'sig', 'code', and 'scp',
   where 'scp' is a pointer to a Unix42.SigContext record fabricated by
   PushHandlerCall. 'mask' and 'onStack' are saved on the stack of 'aT' 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 Ux.Error with SigSegV if pushing context would
   extend stack beyond rlimit. *)

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

PROCEDURE PutText(
    aT: T;
    a: UNSIGNED;
    t: Text.T;
    count: CARDINAL := System.MaxCard)
    : CARDINAL
  RAISES {Ux.Error};
(* 'a' is interpreted as an address in the address space 'aT. 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)'. Calls UxPro.RaiseError(Unix42.EFAULT) if there is an
   address fault. *)
```

```
PROCEDURE R0Result(aT: T; result: INTEGER; error: BOOLEAN := FALSE) RAISES {};
(* Sets register R0 in the current thread of address space 'aT' 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(aT: T; result: INTEGER) RAISES {};
(* 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 Release(aT: T; vFork: BOOLEAN) RAISES {};
(* Destroys all resources, including threads and virtual memory, in the
   address space 'aT'. It assumes all threads are already stopped. *)

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

PROCEDURE SetClientPriority(aT: T; priority: TPFriends.Priority) RAISES {};
  (* Sets priority of all threads in address space 'aT'. *)

PROCEDURE SetError(aT: T) RAISES {};
(* Sets error flag of 'aT', 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(aT: T) RAISES {Ux.Error};
(* If handler thread of 'aT' is currently null, sets it to current thread,
   else if handler thread is different than current thread calls
   RaiseError(EINVAL). Should only be called directly or indirectly from a
   kernel entry procedure. See AlertHandler. *)

PROCEDURE SetRLimit(
    aT: T;
    resource: Unix42.Resource;
    rLimit: Unix42.RLimit)
  RAISES {};
(* Sets the limit for 'resource' within address space 'aT'. No-ops unless
   'resource' is Unix42.rlimitData or Unix42.rLimitStack. Assumes checking
   ('cur' <= 'max', etc.) done by caller, but silently enforces that limits
   are less than 'maxSegSize'. *)

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

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

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

PROCEDURE WaitEvent(aT: T; VAR (*out*) event: Event) RAISES {Ux.Error};
(* Starts the current thread of 'aT', 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 'aT'.
   Returns in 'event' the description of what happened.  Raises Ux.Error with
   SigSegV if stack of 'aT' grows to rlimit. *)

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 Mon Sep  8 11:38:05 1986 by mcjones *)
(*      modified on Tue Sep 17  9:31:00 1985 by mds *)

SAFE DEFINITION MODULE UxPro;

IMPORT Text, Thread, Time, Unix42, User, Ux;

TYPE
  T = Ux.UxProT;

PROCEDURE EUser(pT: T): User.T RAISES {};
(* Returns the effective user of the process 'pT'. *)

PROCEDURE ForkExecve(
    ppid: Ux.ProcessID;
    name: Text.T;
    argv, envp: Ux.RefArrayOfText;
    fT: Ux.UxFileT := NIL)
    : Ux.ProcessID
  RAISES {Ux.Error, Thread.Alerted};
(* About like doing a fork from the process with id 'ppid' immediately
   followed by execve(name, argv, envp) in the new process. The id of the new
   process is returned. When 'ppid' is 0, can be used to create the first
   process with real effective user = root and pid=InitPID. When 'fT' is
   non-NIL, can be used to supply custom set of descriptors (otherwise relies
   on UxFile.Fork). No signals or errors are produced in ppid, rather all
   errors result in UxPro.Error or Thread.Alerted being raised on this call
   (in which case, if 'fT' is non-null then UxFile.Exit(fT) is performed). *)

PROCEDURE GetHost(): Text.T RAISES {};
(* Returns the current value of the host name for this machine, possibly
NIL. *)

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

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

PROCEDURE PGRP(pT: T): Ux.ProcessID RAISES {};
(* Returns the process group id of the process 'pT'. *)

PROCEDURE PID(pT: T): Ux.ProcessID RAISES {};
(* Returns the process id of the process 'pT'. *)

PROCEDURE PIDsPGRP(pid: Ux.ProcessID): Ux.ProcessID RAISES {Ux.Error};
(* Returns the process group id of the process identified by 'pid'. Calls
   RaiseError(Unix42.ESRCH) if process 'pid' does not exist. *)

PROCEDURE RaiseError(
    e: Unix42.SysError;
    s: Unix42.Signal := Unix42.SigNone)
  RAISES {Ux.Error};
(* Raises Error with 'e' and 's' in the ErrorArg *)

PROCEDURE Register(entry: Unix42.KernelEntry; proc: Ux.EntryProc) RAISES {};
(* Registers the procedure 'proc' as the implementation of the ultrix kernel
   entry 'entry'. See below. *)

PROCEDURE RUser(pT: T): User.T RAISES {};
(* Returns the real user of the process 'pT'. *)

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

PROCEDURE SetHost(name: Text.T; override: BOOLEAN := FALSE);
(* Sets the host name for this machine to 'name' if the host name is currently
   NIL or 'override' is TRUE. *)

PROCEDURE SigEffective(pT: T; signo: Unix42.Signal): BOOLEAN RAISES {};
(* Returns TRUE iff for process 'pT' the signal 'signo' is not ignored nor
   masked, and if the id of the parent of process 'pT' 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(pT: T; signo: Unix42.Signal) RAISES {};
(* Posts signal 'signo' for delivery to process 'pT'. Intended for use by
   interval timer on SigAlrm, SigVTAlrm, and SigProf. *)

PROCEDURE SignalPG(
    pgrp: Ux.ProcessID;
    signo: Unix42.Signal;
    pT: T := NIL)
  RAISES {};
(* Posts signal 'signo' for delivery in all processes in group 'pgrp', except
   process 'pT' if it is nonnil. *)

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 Unix42.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 Unix42.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 Unix42.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 (Unix42.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 Fri Nov 22 15:36:00 1985 by mcjones *)
(*      modified on Fri Oct 11 14:09:54 1985 by price *)
SAFE DEFINITION MODULE UxTime;

(* File UxTime.def, last modified by:

    C. Price Fri Oct 11 14:09:54 1985

   This module provides interval timer support to Ux. It implements
   the kernel calls GetITimer and SetITimer.

*)

IMPORT Ux;

TYPE
  T = Ux.UxTimeT;

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

PROCEDURE Exit(tT: T) RAISES {};
(* This procedure is called when a Ux process is terminated. This procedure
   cancels any pending timers on the T.
*)

END UxTime.
```

```
(* 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.
```

```
(* Last modified on Wed Sep 10 16:29:28 1986 by mcjones *)

IMPLEMENTATION MODULE UxAS;

IMPORT
  Hardware, RPCLocal, RdV, SBuf, System, TDTraps, Text, Thread, ThreadFriends,
  ThreadsPort, TPFriends, TPSpecial, Unix42, UnsafeOS, User, UxCoreFile,
  UxFile, UxPro, VM, VMFriends, 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 in page at HoleAddr *)

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

CONST
  initRLimDataCur = 5f8000H;
  initRLimDataMax = 5f8000H;
  initRLimStackCur = 80000H;
  initRLimStackMax = 5f8000H;

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

VAR
  initialized: BOOLEAN;

VAR (* but logically constant *)
  VMBytesPerPage: CARDINAL;
  phonyBytesPerPage: CARDINAL; (* ***** See SBreak *)
  initialStackAddr: Addr;
  nullTHandle: ThreadsPort.Handle;
  osSpace: VMFriends.Space; (* operating system address space *)
  handlerLinkage: ARRAY [0 .. 3] OF INTEGER;
  nilTrapEnabled: BOOLEAN;

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

(* Representations of opaque types: *)
TYPE
  T = REF TObj;
  ClientThread = ThreadsPort.Handle; (* which is still opaque to us *)

(* Private state for UxAS: *)
(* Fields below marked (* ! *) are inherited by forked process. In case of
   tState, only some subfields are inherited. *)
TYPE
  ThreadState = (RealKernelCall, Other);
  TObj = RECORD
    space: NubTypes.Space; (* address space for this process *)
    firstWritableAddr: Addr; (* lowest not write protected *)
    traps: TPSpecial.TrapKinds; (* ones we are catching *)
    handlerThread: ThreadsPort.Handle; (* thread for AlertHandler *)
    strace: BOOLEAN; (* whether to pass traps to debugger *) (*!*)
    rLimData: Unix42.RLimit; (* <= BytesPerRegion *) (*!*)
    rLimStack: Unix42.RLimit; (* <= BytesPerRegion *) (*!*)
    currentThread: ThreadsPort.Handle; (* thread for current action *)
    CASE threadState: ThreadState OF
    | RealKernelCall: sState: TPSpecial.SState; resultSet: BOOLEAN;
    | Other: state: TPSpecial.State;
    END;
  END;

CONST
  MaxIdleSpaces = 7;

(* Global variables under protection of mutex. *)
(* ***** Make this into nested monitor. *)
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 = VMGetBoundary(s, VM.Low) *)
    END;
  maxIdleSpaces: CARDINAL; (* maxIdleSpaces <= MaxIdleSpaces *)
  maxIdleBreakAddr: Addr; (* zero disables space caching *)


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

PROCEDURE AlertHandler(aT: T) RAISES {};
  (* The argument to this procedure is in general not a thread currently
     executing a kernel call. *)
  VAR state: TPSpecial.State;
  BEGIN
    ASSERT(NOT ThreadsPort.Equal(aT^.handlerThread, nullTHandle));

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

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

PROCEDURE BackupKernelCall(
    aT: T;
    entry: Unix42.KernelEntry)
  RAISES {Ux.Error};
  CONST
    opChmk = 0bcH;
    immMode = 08fH;
    maxLit = 03fH;
    regR0 = 050H;
    regR15 = 05fH;
  TYPE Byte = BITS 8 FOR CARDINAL;
  VAR pc: Addr; bytes: ARRAY [0 .. 3] OF Byte;
  BEGIN
    pc := Reg(aT, Hardware.PCReg);
```

```
    Get(aT, 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 (Reg(aT, bytes[3] - regR0) = ORD(entry)))
    THEN
      pc := pc - 2;
    ELSE
      ASSERT(FALSE);
    END;
    SetReg(aT, Hardware.PCReg, pc);
  END BackupKernelCall;

PROCEDURE CopySpace(aT, aTNew: T; vFork: BOOLEAN) RAISES {Ux.Error};
  VAR breakAddr, stackAddr: Addr; aTState: TPSpecial.State;
  BEGIN

    (* New space gets copy of registers of old. *)
    TPSpecial.GetState(aT^.currentThread, aTState);
    aTNew^.state.machineState.regs := aTState.machineState.regs;

    (* Allocate appropriate space. *)
    ASSERT(aTNew^.space = VMFriends.NullSpace);
    aTNew^.firstWritableAddr := aT^.firstWritableAddr;
    IF vFork THEN
      aTNew^.space := aT^.space;
      RestoreThread(aTNew);
    ELSE
      breakAddr := VMGetBoundary(aT^.space, VM.Low); (* lowest free *)
      ASSERT(breakAddr > 0);
      stackAddr := VMGetBoundary(aT^.space, VM.High); (* lowest allocated *)
      ASSERT(stackAddr < HoleAddr);
      AllocSpace(aTNew, breakAddr, breakAddr, stackAddr);

      (* Copy low and high regions. *)
      IF NOT VMFriends.CopyBlock(aT^.space, TextAddr, aTNew^.space, TextAddr,
               breakAddr - TextAddr)
         OR NOT VMFriends.CopyBlock(aT^.space, stackAddr, aTNew^.space,
                  stackAddr, HoleAddr - stackAddr) THEN
        UxPro.RaiseError(Unix42.EFAULT);
      END;

      (* Write-protect prefix of low region. *)
      VMFriends.MakePagesReadOnly(aTNew^.space, TextPage,
        (MIN(aTNew^.firstWritableAddr, breakAddr) - TextAddr) DIV
          VMBytesPerPage); (* round size down *)
    END;
  END CopySpace;

PROCEDURE Get(
    aT: T;
    a: Addr;
    VAR v: ARRAY OF System.Byte;
    offset: CARDINAL := 0;
    count: CARDINAL := System.MaxCard)
  RAISES {Ux.Error};
  VAR actualCount: INTEGER;
  BEGIN
    actualCount := MIN(count, NUMBER(v) - offset);
    IF actualCount > 0 THEN
      IF NOT VMFriends.ReadBlock(aT^.space, a, System.Adr(v[offset]),
               actualCount) THEN
        UxPro.RaiseError(Unix42.EFAULT)
      END;
    END;
  END Get;

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

PROCEDURE GetClientPriority(aT: T): TPFriends.Priority RAISES {};
  VAR
    space: CARDINAL;
    maxPriority: TPFriends.Priority;
    handle: ThreadsPort.Handle;
    state: TPSpecial.State;
  BEGIN
    space := aT^.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(
    aT: T;
```

```
    alreadyStopped: BOOLEAN := FALSE)
    : Time.T
  RAISES {};
  VAR
    space: CARDINAL;
    handle: ThreadsPort.Handle;
    time, totalTime: Time.T;
    us: CARDINAL;
  BEGIN
    space := aT^.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);
      (* TimeINC(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 GetRLimit(
    aT: T;
    resource: Unix42.Resource;
    VAR (*out*) rLimit: Unix42.RLimit)
  RAISES {};
  BEGIN
    CASE resource OF
    | Unix42.rLimitData: rLimit := aT^.rLimData;
    | Unix42.rLimitStack: rLimit := aT^.rLimStack;
    END;
  END GetRLimit;

PROCEDURE GetSpace(aT: T): NubTypes.Space RAISES {};
  BEGIN RETURN aT^.space; END GetSpace;

PROCEDURE GetText(aT: T; a: Addr): Text.T RAISES {Ux.Error};
  VAR
    wr: WrV.T;
    bufPos, bufCount: INTEGER;
    addr: Addr;
    buf: ARRAY [0 .. 127] OF CHAR;
  PROCEDURE NextBufPos();
    BEGIN
      INC(bufPos);
      IF bufPos >= bufCount THEN
        bufPos := 0;
        INC(addr, bufCount);
        TRY
          Get(aT, addr, buf);
          bufCount := NUMBER(buf);
        EXCEPT
          Ux.Error: Get(aT, addr, buf[0]); bufCount := 1;
        END;
      END;
    END NextBufPos;
  BEGIN
    IF a = 0 THEN
      RETURN NIL;
    ELSE
      addr := a;
      bufCount := 0;
      bufPos := -1;
      WrV.New(wr);
      LOOP
        NextBufPos();
        IF buf[bufPos] = 0C THEN EXIT END;
        WrV.PutChar(wr, buf[bufPos]);
      END;
      RETURN WrV.ToText(wr);
    END;
  END GetText;

PROCEDURE HideThread(aT: T) RAISES {};
  BEGIN
    GetState(aT);
    aT^.state.space := osSpace;
    TPSpecial.SetState(aT^.currentThread, aT^.state);
  END HideThread;

PROCEDURE Init() RAISES {};
  VAR i: CARDINAL;
  BEGIN
    ASSERT(NOT initialized);
    initialized := TRUE;

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

    phonyBytesPerPage := UltrixBytesPerPage; (* ***** *)

    ASSERT(TextAddr MOD VMBytesPerPage = 0); (* if not TextAddr = 0 *)
    ASSERT(
      System.TByteSize(UxCoreFile.Header) = TPSpecial.PreferredStackHole);
    initialStackAddr :=
      HoleAddr - TPSpecial.PreferredStackSize - VMBytesPerPage;
    (* extra page will be unmapped by Modula-2+ runtime *)

    nullTHandle := ThreadsPort.Null();

    osSpace := TPFriends.GetSpace();
    IF osSpace = 2 THEN (* no user TTD server *)
      IF NIL = Thread.Fork(SelfWatcher, NIL) THEN END;
    END;

    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                     *)

    nilTrapEnabled := FALSE;

    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(UnsafeOS.SYSspare72, VAdvise); (* Ultrix too *)
    UxPro.Register(UnsafeOS.SYSreservetraps, ReserveTraps); (* Ux only *)
    UxPro.Register(Unix42.SYSgetpagesize, GetPageSize);
    UxPro.Register(UnsafeOS.SYSsbreak, SBreak); (* Ultrix too *)
    UxPro.Register(UnsafeOS.SYSstrace, STrace); (* Ux only *)
  END Init;

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

PROCEDURE Load(
    p: Ux.T;
    name: Text.T;
    argv, envp: Ux.RefArrayOfText;
    aTArgs: T;
    argva, envpa: Addr;
    vFork: BOOLEAN)
    : BOOLEAN
  RAISES {Ux.Error, Thread.Alerted};

  (* Assumes no holes (unmapped pages) in 'p^.aT^.space' other than the one
     set up by Load itself: the hole beyond the stack. If/when we unmap page
     zero (to trap dereferences of NIL pointers), a matching call to
     VMFriends.MapPages will have to be added. *)

  (* Format of executable object file (see /usr/include/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;

  VAR
    aT: T; (* address space of 'p' *)
    argsFetched: BOOLEAN;
    f: UxFile.File; (* reads 'name' or its interpreter *)
    buf: ARRAY [0 .. 511] OF CHAR;
    bufCount, bufIndex: CARDINAL;
    e: Exec; (* first few bytes from 'f' *)
    extraCount: INTEGER; (* > 0 if named interpreter is found *)
    extras: Ux.RefArrayOfText; (* [name [, arg]] *)

  PROCEDURE Open(
      name: Text.T;
      VAR mode: Unix42.FileMode;
      VAR user: User.T)
      : BOOLEAN;
    (* Opens 'name' on behalf of 'p', setting 'f', 'e', 'mode', and 'user'. If
       file appears to contain an executable object program, returns TRUE and
       leaves 'f' positioned to the beginning of the text segment. *)

    (* Assumes 'f' does not reference an open file. *)
    VAR sbuf: SBuf.T; headerSize: CARDINAL;
    BEGIN
      UxFile.OpenForLoading(UxPro.EUser(p^.pT), UxPro.RUser(p^.pT), p^.fT,
        name, f, mode, user);
      SBuf.New(sbuf, buf);
      bufCount := UxFile.ReadF(f, sbuf);
      bufIndex := 0;
      System.Copy(System.Adr(buf), System.Adr(e), System.ByteSize(e));
      IF (bufCount < System.ByteSize(e))
        OR (e.magic # ZMagic) AND (e.magic # NMagic) AND (e.magic # OMagic)
      THEN
        RETURN FALSE;
      END;

      headerSize := System.ByteSize(e);
      IF e.magic = ZMagic THEN
        headerSize := UltrixRoundUp(headerSize);
        e.text := UltrixRoundUp(e.text);
        e.data := UltrixRoundUp(e.data);
      END;
      UxFile.SeekF(f, headerSize);

      (* Check that lengths in header are consistent with length of file. *)
      IF UxFile.LengthF(f) < headerSize + e.text + e.data THEN
        UxFile.CloseF(f);
        UxPro.RaiseError(Unix42.EFAULT);
      END;

      RETURN TRUE;
    END Open;

  PROCEDURE FetchArgs;
    BEGIN
      IF NOT argsFetched THEN
        IF argv = NIL THEN
          argv := GetArrayOfText(aTArgs, argva);
          IF NUMBER(argv^) = 0 THEN NEW(argv, 1); argv^[0] := name; END;
          envp := GetArrayOfText(aTArgs, envpa);
        END;
        argsFetched := TRUE;
      END;
    END FetchArgs;

  PROCEDURE OpenInterpreter;
    (* File 'f' 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: Unix42.FileMode;
      user: User.T;
    PROCEDURE GetChar(): CHAR;
      VAR sbuf: SBuf.T;
      BEGIN
        IF bufIndex = bufCount THEN
          SBuf.New(sbuf, buf);
          bufCount := UxFile.ReadF(f, sbuf);
          bufIndex := 0;
        END;
        IF bufCount = 0 THEN UxPro.RaiseError(Unix42.ENOEXEC); END;
        INC(bufIndex);
```

```
      RETURN (buf[bufIndex - 1]);
    END GetChar;
  BEGIN
    TRY
      IF (GetChar() # "#") OR (GetChar() # "!") THEN
        UxPro.RaiseError(Unix42.ENOEXEC);
      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
      UxFile.CloseF(f);
    END;
    iarg := WrV.ToText(w); (* arg *)
    IF Open(iname, mode, user) THEN (* ignore this mode, user *)
      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
      UxFile.CloseF(f);
      UxPro.RaiseError(Unix42.ENOEXEC);
    END;
  END OpenInterpreter;

PROCEDURE LoadChunk(addr: Addr; length: CARDINAL);
  (* Copies 'length' bytes from 'f' to address 'addr' in address space 'aT'.
  *)
  VAR sbuf: SBuf.T; n: CARDINAL;
  BEGIN
    SBuf.SNew(sbuf, aT^.space, addr, length);
    n := UxFile.ReadF(f, sbuf);
    IF n # length THEN UxPro.RaiseError(Unix42.EFAULT); END;
  END LoadChunk;

VAR
  zero: ARRAY [0 .. 6] OF CHAR;
  mode: Unix42.FileMode;
  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 's' *)
  sbuf: SBuf.T;
BEGIN

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

  aT := p^.aT;

  argsFetched := FALSE;

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

```
  (* Set 'f' (leaving it positioned ready to read the text portion), 'e',
     'extraCount', 'extras', 'mode', and 'user'. *)
  IF NOT Open(name, mode, user) THEN OpenInterpreter; END;

  TRY (* FINALLY UxFile.CloseF(f) *)

    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 = NMagic THEN dataAddr := UltrixRoundUp(dataAddr); END;

    bssAddr := dataAddr + e.data;
    breakAddr := UltrixRoundUp(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(aT^.rLimData.cur) < breakAddr)
       OR (stackAddr < Addr(HoleAddr - TPSpecial.PreferredStackSize)) THEN
      UxPro.RaiseError(Unix42.ENOMEM);
    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
       Ux.Error. *)

    TRY (* EXCEPT SBuf.Fault, Ux.Error *)

      IF vFork OR (aT^.space = VMFriends.NullSpace) (* ForkExec *) THEN
        AllocSpace(aT, bssAddr, breakAddr, initialStackAddr);
      ELSE (* normal Exec not following VFork *)
        minBreakAddr := MIN(VMGetBoundary(aT^.space, VM.Low), breakAddr);
        VMSetBoundary(aT^.space, VM.Low, breakAddr);
        VMFriends.MakePagesReadWrite(aT^.space, TextPage,
          (MIN(aT^.firstWritableAddr, minBreakAddr) - TextAddr) DIV
            VMBytesPerPage); (* round size down *)
        IF bssAddr < minBreakAddr THEN
          (* We only need to zero from bssAddr to minBreakAddr; any new
             pages will be zeroed by VM. *)
          SBuf.SNew(sbuf, aT^.space, bssAddr, minBreakAddr - bssAddr);
```

```
      SBuf.Zero(sbuf);
    END;
    VMSetBoundary(aT^.space, VM.High, initialStackAddr);
  END;

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

  (* Copy text to 's'. *)
  LoadChunk(TextAddr, e.text);

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

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

  (* Set up the stack. *)

  addr := stackAddr;

  (* Copy argument count longword. *)
  Put(aT, 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(aT, addr, stringAddr, 0, 4);
    INC(addr, 4);
    INC(stringAddr, Text.Length(extras^[i]) + 1);
  END;
  FOR i := 0 TO NUMBER(argv^) - 1 DO
    Put(aT, addr, stringAddr, 0, 4);
    INC(addr, 4);
    INC(stringAddr, Text.Length(argv^[i]) + 1);
  END;
  Put(aT, addr, zero, 0, 4);
  INC(addr, 4);

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

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

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

  (* Copy padding (to longword boundary) and final zero longword. *)
  Put(aT, addr, zero, 0, padSize + 4);
  ASSERT(addr + padSize + 4 = HoleAddr);

  (* Set up thread state. *)
  SetReg(aT, Hardware.PCReg, e.entry + 2); (* skip register save mask *)
  SetReg(aT, Hardware.SPReg, stackAddr);

  (* If file being executed has the appropriate mode, its owner becomes
     the effective user. *)
  IF (Unix42.SetUIDonExec IN mode) AND (user # NIL) THEN
    UxPro.SetEUser(p^.pT, user);
  END;

 EXCEPT
   SBuf.Fault, Ux.Error: RETURN FALSE;
 END;
FINALLY
 UxFile.CloseF(f);
END;

 RETURN TRUE; (* success *)
END Load;

PROCEDURE MakeCoreImage(
  p: Ux.T;
  sig: Unix42.Signal)
RAISES {Ux.Error, Thread.Alerted};
VAR space: VMFriends.Space; f: UxFile.File;
PROCEDURE W(p: CARDINAL; v: UNSIGNED; l: CARDINAL := 4);
  (* Write 'l' bytes of 'v' at position 'p' in 'f'. *)
  VAR s: SBuf.T;
  BEGIN
    SBuf.New(s, v, 0, l);
    UxFile.SeekF(f, p);
    UxFile.WriteF(f, s);
  END W;
VAR pos: CARDINAL; (* position in 'f', for use with SeekF *)
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; sbuf: SBuf.T;
  BEGIN
    WHILE addr < limitAddr DO
      pageState :=
        VMFriends.StatePage(space, addr DIV Hardware.BytesPerPage);
      IF pageState.contents IN Unreal THEN
        (* Seek forward one page, leaving zeros. *)
        INC(addr, Hardware.BytesPerPage);
        INC(pos, Hardware.BytesPerPage);
        UxFile.SeekF(f, pos);
      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;
        SBuf.SNew(sbuf, space, runAddr, count);
        UxFile.WriteF(f, sbuf);
        INC(pos, count);
      END;
    END;
  END CarefullyWrite;
VAR
  aT: T;
  count: CARDINAL;
  i: CARDINAL;
  dataAddr, breakAddr, stackAddr: Addr;
  dataPages, stackPages: CARDINAL; (* of Hardware.BytesPerPage *)
  sbuf: SBuf.T;
  tME: UxCoreFile.ThreadMapEntry;

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

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

  f :=
    UxFile.OpenCoreFile(UxPro.EUser(p^.pT), UxPro.RUser(p^.pT), p^.fT,
      System.TByteSize(UxCoreFile.Header) +
        Hardware.BytesPerPage * (dataPages + stackPages) +
        System.ByteSize(tME) * TPFriends.NHandles(space));

  TRY

    (* Write header. *)
    W(UxCoreFile.oPcbKsp, 7fffffffH); (* kernel stack pointer *)
    W(UxCoreFile.oPcbUsp, Reg(aT, Hardware.SPReg)); (* user stack pointer *)
    W(UxCoreFile.oStatus, ORD(sig), UxCoreFile.lStatus);
    W(UxCoreFile.oTextPages,
      aT^.firstWritableAddr DIV Hardware.BytesPerPage);
    W(UxCoreFile.oDataPages, dataPages);
    W(UxCoreFile.oStackPages, stackPages);
    W(UxCoreFile.oAp, Reg(aT, Hardware.APReg));
    W(UxCoreFile.oFp, Reg(aT, Hardware.FPReg));
    FOR i := 0 TO 11 DO W(UxCoreFile.oRegs + 4 * i, Reg(aT, i)); END;
    W(UxCoreFile.oSp, Reg(aT, Hardware.SPReg));
    W(UxCoreFile.oPc, Reg(aT, Hardware.PCReg));
    W(UxCoreFile.oPsw, LOOPHOLE(aT^.state.machineState.status, UNSIGNED));
    pos := System.TByteSize(UxCoreFile.Header);
    UxFile.SeekF(f, pos);

    (* Write data pages. *)
    count := breakAddr - dataAddr;
    SBuf.SNew(sbuf, space, dataAddr, count);
    TRY
      UxFile.WriteF(f, sbuf);
      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 aT^.currentThread. *)
    tME.handle := aT^.currentThread;
    GetState(aT);
    tME.state := aT^.state;
    SBuf.New(sbuf, tME);
    UxFile.WriteF(f, sbuf);
    tME.handle := nullTHandle;
    LOOP
      tME.handle := TPFriends.NextHandleInSpace(space, tME.handle);
      IF ThreadsPort.Equal(tME.handle, nullTHandle) THEN EXIT; END;
      IF ThreadsPort.Equal(tME.handle, aT^.currentThread) THEN
        TPSpecial.GetState(tME.handle, tME.state);
        SBuf.New(sbuf, tME);
        UxFile.WriteF(f, sbuf);
      END;
    END;
  FINALLY
    UxFile.CloseF(f);
  END;
END MakeCoreImage;

PROCEDURE New(aT: T): T RAISES {Ux.Error};
  VAR aTNew: T; aTState: TPSpecial.State;
  BEGIN
    NEW(aTNew);
    WITH aTNew^ DO

      (* The space itself is not allocated until Copy or Load is called. *)
      space := VMFriends.NullSpace;

      traps := MyTraps;

      handlerThread := nullTHandle;

      (* Create the (initial) thread to run in the new space. *)
      currentThread := TPSpecial.Create(NIL, NIL, 0);
      threadState := Other;
      TPSpecial.GetState(currentThread, state);
      state.stackHigh := HoleAddr; (* for the garbage collector *)

      IF aT = NIL THEN (* first process *)
        strace := FALSE;
        rLimData.cur := initRLimDataCur;
        rLimData.max := initRLimDataMax;
        rLimStack.cur := initRLimStackCur;
        rLimStack.max := initRLimStackMax;
      ELSE (* forked process *)
        strace := aT^.strace;
        rLimData := aT^.rLimData;
        rLimStack := aT^.rLimStack;
        (* We would really like to use "aT^.initialThread" in the following: *)
        TPSpecial.GetState(aT^.currentThread, aTState);
        state.priority := aTState.priority;
        state.processor := aTState.processor;
      END;
```

```
    END;
    RETURN aTNew;
  END New;

PROCEDURE PushHandlerCall(
    aT: T;
    sig: Unix42.Signal;
    code: INTEGER;
    handler: Unix42.SigHandler;
    mask: Unix42.SigMask;
    onStack: Unix42.SigOnStack;
    switchStacks: BOOLEAN;
    sp: Addr)
  RAISES {Ux.Error};
  PROCEDURE Push(w: System.Word); (* onto stack of 'aT' *)
    VAR sp: Addr;
    BEGIN
      sp := Reg(aT, Hardware.SPReg) - 4;
      TRY
        Put(aT, sp, w);
      EXCEPT
        Ux.Error:
          IF GrowStack(aT, sp) THEN
            Put(aT, sp, w);
          ELSE
            UxPro.RaiseError(Unix42.ok, Unix42.SigSegV);
          END;
      END;
      SetReg(aT, Hardware.SPReg, sp);
    END Push;
  VAR status: Hardware.ProcessorStatus; scp: Addr; (* SigContext pointer *)
  BEGIN
    GetState(aT);

    (* Push SigContext on current stack, and save pointer in 'scp'. *)
    status := aT^.state.machineState.status; (* PSL *)
    Push(status);
    Push(Reg(aT, Hardware.PCReg));
    Push(Reg(aT, Hardware.SPReg)); (* = .+4 *)
    Push(mask);
    Push(onStack);
    scp := Reg(aT, Hardware.SPReg);

    (* If appropriate, switch to signal stack. *)
    IF switchStacks THEN SetReg(aT, Hardware.SPReg, sp); END;

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

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

    (* Push handler arguments (in reverse order). *)
    Push(scp);
    Push(code); (* in case SIGFPE or SIGILL *)
    Push(sig);

    (* Set program counter to handler linkage routine. *)
    SetReg(aT, Hardware.PCReg, HandlerEntry);

    (* Clean out PSL. *)
    status.cc.c := FALSE;
    status.cc.v := FALSE;
    status.cc.z := FALSE;
    status.cc.n := FALSE;
    (* t *)
    (* iv, fu, dv *)
    status.fpd := FALSE; (* important! *)
    aT^.state.machineState.status := status;

  END PushHandlerCall;

PROCEDURE Put(
    aT: T;
    a: Addr;
    VAR v: ARRAY OF System.Byte;
    offset: CARDINAL := 0;
    count: CARDINAL := System.MaxCard)
  RAISES {Ux.Error};
  VAR actualCount: INTEGER;
  BEGIN
    actualCount := MIN(count, NUMBER(v) - offset);
    IF actualCount > 0 THEN
      ASSERT((a # 0) OR NOT nilTrapEnabled, "Put through NIL");
      IF NOT VMFriends.WriteBlock(aT^.space, a, System.Adr(v[offset]),
              actualCount) THEN
        UxPro.RaiseError(Unix42.EFAULT)
      END;
    END;
  END Put;

PROCEDURE PutText(
    aT: T;
    a: Addr;
    t: Text.T;
    count: CARDINAL := System.MaxCard)
    : CARDINAL
  RAISES {Ux.Error};
  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);
      Put(aT, aNow, buf, 0, n);
      INC(aNow, n);
    END;
    RETURN count;
  END PutText;

PROCEDURE R0Result(aT: T; result: INTEGER; error: BOOLEAN := FALSE) RAISES {};
  BEGIN
    IF aT^.threadState = RealKernelCall THEN
      aT^.sState.r0Val.val := result;
      aT^.resultSet := TRUE;
    ELSE (* aT^.threadState = Other *)
      SetReg(aT, 0, result);
    END;
    IF error THEN SetError(aT); END;
  END R0Result;

PROCEDURE R1Result(aT: T; result: INTEGER) RAISES {};
  BEGIN
    IF aT^.threadState = RealKernelCall THEN
      aT^.sState.r1Val.val := result;
      aT^.sState.setR1 := TRUE;
    ELSE (* aT^.threadState = Other *)
      SetReg(aT, 1, result);
```

```
    END;
  END R1Result;

PROCEDURE Release(aT: T; vFork: BOOLEAN) RAISES {};
  VAR idle, ok: BOOLEAN; nHandles, i: CARDINAL; breakAddr: Addr;
  BEGIN
    IF aT^.space = VMFriends.NullSpace THEN
      TPSpecial.Destroy(aT^.currentThread);
    ELSE
      nHandles := TPFriends.NHandles(aT^.space);
      TPSpecial.DestroyAll(aT^.space);
      IF NOT vFork THEN
        breakAddr := VMGetBoundary(aT^.space, 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(aT^.space, TextPage,
              (MIN(aT^.firstWritableAddr, breakAddr) - TextAddr) DIV
                VMBytesPerPage); (* round size down *)
            WITH idleSpaces[i] DO s := aT^.space; b := breakAddr; END;
          END;
        END;
        IF NOT idle THEN
          ok := VMFriends.FreeSpace(aT^.space);
          ASSERT(ok);
        END;
      END;
    END;
  END Release;

PROCEDURE RestoreThread(aT: T) RAISES {};
  BEGIN
    GetState(aT);
    aT^.state.space := aT^.space;
    TPSpecial.SetState(aT^.currentThread, aT^.state);
  END RestoreThread;

PROCEDURE SetClientPriority(aT: T; priority: TPFriends.Priority) RAISES {};
  VAR space: CARDINAL; handle: ThreadsPort.Handle; state: TPSpecial.State;
  BEGIN
    space := aT^.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(aT: T) RAISES {};
  BEGIN
    IF aT^.threadState = RealKernelCall THEN
      aT^.sState.error := TRUE;
    ELSE (* aT^.threadState = Other *)
      aT^.state.machineState.status.cc.c := TRUE;
    END;
  END SetError;

PROCEDURE SetHandler(aT: T) RAISES {Ux.Error};
  BEGIN
    IF ThreadsPort.Equal(aT^.handlerThread, nullTHandle) THEN
      aT^.handlerThread := aT^.currentThread;
    ELSIF NOT ThreadsPort.Equal(aT^.handlerThread, aT^.currentThread) THEN
      UxPro.RaiseError(Unix42.EINVAL);
    END;
  END SetHandler;

PROCEDURE SetRLimit(
    aT: T;
    resource: Unix42.Resource;
    rLimit: Unix42.RLimit)
  RAISES {};
  BEGIN
    rLimit.max := MIN(rLimit.max, BytesPerRegion);
    rLimit.cur := MIN(rLimit.cur, BytesPerRegion);
    CASE resource OF
    | Unix42.rLimitData: aT^.rLimData := rLimit;
    | Unix42.rLimitStack: aT^.rLimStack := 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);
      TPSpecial.GetState(handle, state);
      state.priority := priority;
      TPSpecial.SetState(handle, state);
      TPSpecial.Start(handle);
    END;
  END SetThreadPriority;

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

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

PROCEDURE WaitEvent(aT: T; VAR (*out*) event: Event) RAISES {Ux.Error};
  VAR handledHere, delegated: BOOLEAN;
  PROCEDURE ClassifyTrap(
      VAR trapInfo: TPSpecial.TrapInfo;
      passedBack: BOOLEAN := FALSE);
    VAR
      scp: Addr; (* pointer to SigContext in client address space *)
      sigContext: Unix42.SigContext;
    BEGIN
```

```
    event.kind := Exception; (* default *)
    CASE trapInfo.kind OF
    | TPSpecial.TKVmFailure: event.sig := Unix42.SigSegV;
    | TPSpecial.TKIllegalNubCall: (* = TKCHMKInstruction *)
        ASSERT(trapInfo.arg = ORD(UnsafeOS.SYShandlerreturn));
        event.kind := HandlerReturn;

        (* Pop pointer to SigContext record and fetch record. *)
        Get(aT, Reg(aT, Hardware.SPReg), scp);
        Get(aT, scp, sigContext);

        (* Restore process state from SigContext record. *)
        event.mask := sigContext.mask;
        event.onStack := sigContext.onStack;
        SetReg(aT, Hardware.SPReg, sigContext.sp);
    | TPSpecial.TKFromDebugger:
        trapInfo := TDTraps.FromDebugger(aT^.currentThread);
        ClassifyTrap(trapInfo, TRUE);
    | TPSpecial.TKIllegalAddress:
        handledHere := GrowStack(aT, trapInfo.addr);
        IF NOT handledHere THEN event.sig := Unix42.SigSegV; END;
    | TPSpecial.TKIllegalAccess: event.sig := Unix42.SigBus;
    | TPSpecial.TKArithmetic:
        event.sig := Unix42.SigFPE;
        event.code := ORD(trapInfo.code);
    | TPSpecial.TKReservedInstruction, TPSpecial.TKCompatibilityMode:
        event.sig := Unix42.SigIll;
        event.code := Unix42.IllPrivinFault;
    | TPSpecial.TKReservedOperand:
        event.sig := Unix42.SigIll;
        event.code := Unix42.IllResopFault;
    | TPSpecial.TKReservedAddressingMode:
        event.sig := Unix42.SigIll;
        event.code := Unix42.IllResadFault;
        (* TKBreakpoint, TKTrace handled by debugger *)
    | TPSpecial.TKXFCInstruction: event.sig := Unix42.SigEMT;
    | TPSpecial.TKCHMEInstruction, TPSpecial.TKCHMSInstruction,
        TPSpecial.TKCHMUInstruction:
        event.sig := Unix42.SigSegV;
    |
    ELSE (* TKBadParameter or ? *)
        event.sig := Unix42.SigIll;
        event.code := Unix42.IllPrivinFault;
    END; (* CASE trapKind *)

    (* In strace mode (our substitute for ptrace), show trap to debugger
       before converting it to Unix signal. Note that this could be extended
       to other event types. *)
    IF (event.kind = Exception) AND aT^.strace AND NOT passedBack THEN
      TDTraps.ToDebugger(aT^.currentThread);
      delegated := TRUE;
    END;
  END ClassifyTrap;
VAR tState: TPSpecial.TState; ap: UNSIGNED;
BEGIN

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

    (* Get the thread ready for a StartKernelCaller. *)
    IF aT^.threadState = Other THEN GetKernelCallerState(aT); END;

    REPEAT (* UNTIL NOT delegated *)
      delegated := FALSE;
```

```
      (* Start the current thread (unless 'delegated=TRUE'), and wait for
         some thread in 'aT' 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 delegated THEN
          aT^.currentThread :=
            TPSpecial.GetKernelCaller(aT^.space, aT^.traps, tState);
        ELSE
          (* ... ResumeKernelCaller ... *)
          TPSpecial.StartKernelCaller(aT^.currentThread, aT^.sState);
          aT^.currentThread :=
            TPSpecial.GetKernelCaller(aT^.space, aT^.traps, tState);
        END;
        IF (tState.trapInfo.kind = TPSpecial.TKIllegalNubCall)
           AND (tState.trapInfo.arg # ORD(UnsafeOS.SYShandlerreturn)) THEN
          aT^.threadState := RealKernelCall;
          aT^.resultSet := FALSE;
          aT^.sState.setR1 := FALSE;
          aT^.sState.error := FALSE;
        ELSE
          aT^.threadState := Other;
          TPSpecial.GetState(aT^.currentThread, aT^.state);
        END;
      EXCEPT
      | Thread.Alerted: (* asynchronous signal to handle *)
          RPCLocal.Exclude(aT^.space, ExcludedStop, aT);
          aT^.threadState := Other;
          TPSpecial.GetState(aT^.currentThread, aT^.state);
          tState.trapInfo := aT^.state.trapInfo;
          IF NOT tState.trapInfo.reported
             AND (tState.trapInfo.kind IN aT^.traps) THEN
            TPSpecial.Start(aT^.currentThread);
            Thread.Alert(Thread.Self()); (* pretend trap came first *)
            IF (tState.trapInfo.kind = TPSpecial.TKIllegalNubCall)
               AND (tState.trapInfo.arg # ORD(UnsafeOS.SYShandlerreturn))
            THEN
              ap := Reg(aT, Hardware.APReg);
              TRY
                Get(aT, ap, tState.arguments);
              EXCEPT Ux.Error:
                Get(aT, ap, tState.arguments.n);
                Get(aT, ap + 4, tState.arguments.a, 0, tState.arguments.n);
              END;
              tState.validArguments := TRUE;
              aT^.state.trapInfo.reported := TRUE;
              GetKernelCallerState(aT);
            END;
          ELSE
            (* The thread does not have a trap waiting for us to handle. *)
            tState.trapInfo.kind := TPSpecial.TKNoTrap;
          END;
      END;

      IF aT^.threadState = RealKernelCall THEN
```

```
        IF NOT tState.validArguments THEN
          UxPro.RaiseError(Unix42.EFAULT);
        END;
        event.kind := KernelCall;
        IF (ORD(FIRST(Unix42.KernelEntry)) <= tState.trapInfo.arg)
           AND (tState.trapInfo.arg <= ORD(LAST(Unix42.KernelEntry))) THEN
          event.entry := VAL(Unix42.KernelEntry, tState.trapInfo.arg);
        ELSE
          event.entry := Unix42.SYSspare0; (* illegal, in range *)
        END;
        System.Zero(System.Adr(event.argList),
          System.ByteSize(event.argList));
        IF tState.validArguments THEN
          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]));
        END;
        (* ***** Kludge "OS.Dummy()" for timing purposes: *)
        IF tState.trapInfo.arg = ORD(UnsafeOS.SYSdummy) THEN
          handledHere := TRUE;
        END;
        (* ***** End "OS.Dummy() kludge. *)
      ELSIF tState.trapInfo.kind IN aT^.traps THEN
        (* Exception or HandlerReturn *)
        ClassifyTrap(tState.trapInfo);
      ELSE
        event.kind := Alert;
      END;
    UNTIL NOT delegated;
  UNTIL NOT handledHere;
END WaitEvent;


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

PROCEDURE AllocSpace(aT: T; bssAddr, breakAddr, stackAddr: Addr);
  (* Assign VM.Space to 'aT' with low and high segment boundaries equal to
     'breakAddr' and 'stackAddr' respectively; ensure bytes from 'bssAddr' to
     'breakAddr - 1' are zero. *)
  VAR
    space: VMFriends.Space;
    i, j: CARDINAL;
    delta, bestDelta: INTEGER;
    minBreakAddr: Addr;
    sbuf: SBuf.T;
  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;
          aT^.space := space;
          RestoreThread(aT);
          minBreakAddr := MIN(VMGetBoundary(aT^.space, VM.Low), breakAddr);
          VMSetBoundary(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.SNew(sbuf, space, bssAddr, minBreakAddr - bssAddr);
            SBuf.Zero(sbuf);
          END;
          VMSetBoundary(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 UxPro.RaiseError(Unix42.EAGAIN) END;

      aT^.space := space;
      RestoreThread(aT);
      VMSetBoundary(space, VM.Low, breakAddr);
      VMSetBoundary(space, VM.High, stackAddr);

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

      (* Make the hole above the stack read-only to trap erroneous stores. *)
      VMFriends.MakePagesReadOnly(space,
        (HoleAddr + VMBytesPerPage - 1) DIV VMBytesPerPage,
        TPSpecial.PreferredStackHole DIV VMBytesPerPage);
      (* round size down *)
    END;
    aT^.firstWritableAddr := 0;
  END AllocSpace;

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

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

PROCEDURE GetState(aT: T);
  (* Ensures that full thread state is cached. *)
  VAR sState: TPSpecial.SState; resultSet: BOOLEAN;
  BEGIN
    IF aT^.threadState = RealKernelCall THEN
```

```
      sState := aT^.sState;
      resultSet := aT^.resultSet;
      aT^.threadState := Other;
      TPSpecial.GetState(aT^.currentThread, aT^.state);
      IF resultSet THEN SetReg(aT, 0, sState.r0Val.val); END;
      IF sState.setR1 THEN SetReg(aT, 1, sState.r1Val.val); END;
      aT^.state.machineState.status.cc.c := sState.error;
    END;
  END GetState;

PROCEDURE GetKernelCallerState(aT: T);
  (* Ensures that only state needed for StartKernelCaller is cached. *)
  VAR sState: TPSpecial.SState;
  BEGIN
    IF aT^.threadState = Other THEN
      TPSpecial.SetState(aT^.currentThread, aT^.state);
      sState.r0Val.val := aT^.state.machineState.regs[0].val;
      sState.setR1 := FALSE;
      sState.error := aT^.state.machineState.status.cc.c;
      aT^.threadState := RealKernelCall;
      aT^.sState := sState;
      aT^.resultSet := TRUE;
    END;
  END GetKernelCallerState;

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

PROCEDURE Reg(aT: T; reg: Hardware.RegNum): UNSIGNED;
  BEGIN GetState(aT); RETURN aT^.state.machineState.regs[reg].val; END Reg;

PROCEDURE SelfWatcher(a: Thread.ForkeeArg): Thread.ForkeeReturn;
  (* ***** Remove this if/when through with Frankenstein mode. *)
  VAR thread: ThreadsPort.Handle; state: TPSpecial.State;
  BEGIN
    WHILE TRUE DO
      thread := TPSpecial.GetTrappedHandle(osSpace, MyTraps);
      TPSpecial.GetState(thread, state);
      ASSERT(FALSE, "Trap in OS address space");
    END;
    RETURN NIL;
  END SelfWatcher;

PROCEDURE SetReg(aT: T; reg: Hardware.RegNum; val: UNSIGNED);
  BEGIN GetState(aT); aT^.state.machineState.regs[reg].val := val; END SetReg;

PROCEDURE UltrixRoundUp(addr: Addr): Addr;
  (* Rounds an address to the smallest multiple of UltrixBytesPerPage not less
     than the original address *)
  BEGIN
    RETURN
      (addr + UltrixBytesPerPage - 1) DIV UltrixBytesPerPage *
        UltrixBytesPerPage;
  END UltrixRoundUp;
```

```
PROCEDURE VMGetBoundary(space: VMFriends.Space; seg: VM.Segment): Addr;
  (* Returns current "boundary address" of 'seg' in 'space': if 'seg=VM.Low',
     then returns lowest address not in 'seg'; if 'seg=VM.High' then returns
     one plus highest address not in 'seg'. *)
  VAR page: VM.PageNumber; ok: BOOLEAN;
  BEGIN
    ok := VMFriends.AllocPages(space, seg, 0, page);
    RETURN page * VMBytesPerPage;
  END VMGetBoundary;

PROCEDURE VMSetBoundary(space: VMFriends.Space; seg: VM.Segment; addr: Addr);
  (* Changes length of 'seg' in 'space' to smallest multiple of VMBytesPerPage
     not less than that implied by new "boundary address" 'addr' (see
     VMGetBoundary for definition of "boundary address"). Assumes caller has
     done appropriate checking. *)
  VAR
    tempAddr: Addr;
    delta: INTEGER; (* alloc if > 0; free if < 0 *)
    ok: BOOLEAN;
    junk: VM.PageNumber;
  BEGIN

    tempAddr := VMGetBoundary(space, seg);

    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, junk);
      ASSERT(ok);
    ELSIF delta <= -VMBytesPerPage THEN (* round down *)
      VMFriends.FreePages(space, seg, (-delta) DIV VMBytesPerPage);
    END;
  END VMSetBoundary;


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

PROCEDURE ReserveTraps(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  TYPE Traps = BITS 32 FOR TPSpecial.TrapKinds;
  VAR aT: T; traps: Traps;
  BEGIN
    aT := p^.aT;
    traps := LOOPHOLE(args[0], Traps);
    IF (traps * UnsafeOS.AlwaysHandledTraps # TPSpecial.TrapKinds{})
       OR NOT (traps <= aT^.traps) THEN
      UxPro.RaiseError(Unix42.EINVAL);
    END;
    aT^.traps := aT^.traps - traps;
    RETURN 0;
  END ReserveTraps;

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

PROCEDURE SBreak(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR aT: T; breakAddr: Addr;
```

```
  BEGIN
    aT := p^.aT;
    IF Addr(aT^.rLimData.cur) < args[0] THEN
      UxPro.RaiseError(Unix42.ENOMEM);
    END;
    breakAddr := UltrixRoundUp(args[0]);

    (* ***** Experiment with other page sizes: *)
    breakAddr :=
      (breakAddr + phonyBytesPerPage - 1) DIV phonyBytesPerPage *
        phonyBytesPerPage;

    VMSetBoundary(aT^.space, VM.Low, breakAddr);
    RETURN 0;
  END SBreak;

PROCEDURE STrace(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN p^.aT^.strace := LOOPHOLE(args[0], INTEGER) # 0; RETURN 0; END STrace;

PROCEDURE VAdvise(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN RETURN 0; END VAdvise;

BEGIN initialized := FALSE; END UxAS.

(* Notes:

   1. At some point we would like to leave the first page of each space
   unmapped to help trap dereferences of NIL pointers. The simplest approach
   seems to be for the Modula-2+ runtime to unmap that page after making sure
   control has permanently left that page.

   2. The emphasis in Load has been on correctness rather than performance.
   Measurements may very well indicate the need for some speed-ups. E.g. a)
   Batching the stack initialization into a single Put call. b)
   Demand-loading, as in Ultrix.

   3. No support for sharing of read-only text currently exists. In the Topaz
   environment, there are not expected to be many copies of the same program
   in different address spaces. (How about shells?)

*)
```

```
(* Last modified on Wed Sep 10 16:21:42 1986 by mcjones *)
(*      modified on Sun Sep 17 15:45:00 1985 by mds *)

(* SAFE if RPCLocal is *) IMPLEMENTATION MODULE UxPro;

IMPORT
  NubTypes, OS, RPCLocal, RPCRuntime, System, TDAdvise, TDAdviseClient,
  Time, Thread, ThreadFriends, UnsafeOS, UxAS, UxFile, UxTime;

CONST
  maxNameLen = 255; (* for Ultrix hostname *)
  UltrixPriorityIncrement = 10;
  MaxPID = 32767; (* 2^15 - 1 *)

TYPE (* private state for UxPro *)
  OpState = (Running, Stopped, Terminated);
  SignalVector = ARRAY [Unix42.FirstSig..Unix42.LastSig] OF Unix42.SigVecRec;
  T = REF TObj;
  TObj = RECORD  (* fields marked (*!*) are inherited by forked process *)
    next: Ux.T; (* chain of process states *)
    c: Thread.Condition; (* notification *)
    wThread: Thread.T; (* watcher thread for this Ultrix process *)
    wPriority: ThreadFriends.Priority; (* priority of watcher thread *)
  (* state of process *)
    opState: OpState;
    untraced: BOOLEAN; (* opState=Stopped & seen by wait3(,WUNTRACED,) *)
    vForkParent, vForkChild: BOOLEAN;
    ws: Unix42.WStatus;
  (* signal stuff *)
    mask: Unix42.SigMask; (* current signal mask for process *)          (*!*)
    posted: Unix42.SigMask; (* signals waiting to be delivered *)
    alertPending: BOOLEAN; (* TRUE iff Thread.TestAlert() *)
    alerted: Unix42.SigMask; (* signals waiting for GetSignal call *)
    illCode: INTEGER; (* code for last SigIll *)
    fpeCode: INTEGER; (* code for last SigFPE *)
    sigStack: Unix42.SigStackRec;                                        (*!*)
    sigVec: SignalVector;                                                (*!*)
  (* identification *)
    pid: Ux.ProcessID; (* process id of self -- IMMUTABLE *)
    ppid: Ux.ProcessID; (* process id of parent *)
    pgrp: Ux.ProcessID; (* process id of process group leader *)          (*!*)
    rUser: User.T; (* real user identity of responsible principal *)     (*!*)
    eUser: User.T; (* user identity for access control *)                (*!*)
    name: Text.T; argv, envp: Ux.RefArrayOfText; (* args to execve *)    (*!*)
    initWTime: Time.T; (* not necessarily 0 *)
  (* rusage of descendants (after termination, includes self) *)
    descCTime: Time.T; (* client cpu time *)
    descWTime: Time.T; (* watcher cpu time *)
  END;


(* Internal TYPEs and CONSTs *)

CONST
  AllSigs = Unix42.SigMask{Unix42.FirstSig .. Unix42.LastSig};

  (* Classification of signals according to default action: *)
  NoActionSigs = Unix42.SigMask{
    Unix42.SigUrg, Unix42.SigCont, Unix42.SigChld, Unix42.SigIO};
    (* default action is ignore *)
  StopSigs = Unix42.SigMask{
    Unix42.SigStop, Unix42.SigTStp, Unix42.SigTTIn, Unix42.SigTTOu};
    (* default action is stop *)
  CoreImageSigs = Unix42.SigMask{
    Unix42.SigQuit, Unix42.SigIll, Unix42.SigTrap, Unix42.SigIOT,
    Unix42.SigEMT, Unix42.SigFPE, Unix42.SigBus, Unix42.SigSegV,
    Unix42.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. *)
  NoAlertSigs = Unix42.SigMask{
    Unix42.SigUrg, Unix42.SigChld, Unix42.SigIO};

  (* Signals that can't be masked; must have default handler (except
  SigCont, which can also be caught. *)
  PowerfulSigs = Unix42.SigMask{
    Unix42.SigKill, Unix42.SigStop, Unix42.SigCont};

  NoSigs = Unix42.SigMask{};

EXCEPTION Terminate; (* RAISEd to exit the watcher loop *)


(* Global variables w/ initialization *)

VAR
  initialized: BOOLEAN;
  tDAdvise: BOOLEAN; (* import succeeded *)
  hostName: Text.T;
  hostID: INTEGER;
  SigDfl: Unix42.SigHandler; (* should be CONST *)
  SigIgn: Unix42.SigHandler; (* should be CONST *)
  SigAlert: Unix42.SigHandler; (* should be CONST *)
  backgroundReleaseEnabled: BOOLEAN;
  gx: Thread.Mutex; (* module lock *)
  lastPID: Ux.ProcessID; (* last pid assigned *)
  lastPIDOverflow: BOOLEAN; (* if true, not the first time through *)
  pHead: Ux.T; (* head of list of processes (for searches) *)
  pTail: Ux.T; (* last of list of processes (for additions) *)
  kep: ARRAY Unix42.KernelEntry OF Ux.EntryProc; (* entry procs *)
  kec: ARRAY Unix42.KernelEntry OF CARDINAL; (* counters *)
  ket: ARRAY Unix42.KernelEntry OF Time.T; (* cumulative CPU time *)
  timingEnabled: BOOLEAN; (* enables ket *)


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

PROCEDURE EUser(pT: T): User.T RAISES {};
  BEGIN
    LOCK gx DO RETURN pT^.eUser END;
  END EUser;

PROCEDURE ForkExecve(
    ppid: Ux.ProcessID;
    name: Text.T;
    argv, envp: Ux.RefArrayOfText;
    fT: UxFile.T := NIL)
    : Ux.ProcessID
  RAISES {Ux.Error, Thread.Alerted};
  BEGIN
    ASSERT(argv # NIL);
    RETURN DoForkExec(ppid, name, argv, envp, NIL, 0, 0, fT);
  END ForkExecve;

PROCEDURE GetHost(): Text.T RAISES {};
  BEGIN
    LOCK gx DO RETURN hostName; END;
  END GetHost;
```

```
PROCEDURE GetWTime(pT: T): Time.T RAISES {};
  BEGIN
    RETURN TimeSub(ThreadFriends.GetCPUTime(pT^.wThread), pT^.initWTime);
  END GetWTime;

PROCEDURE Init() RAISES {};
  VAR e: Unix42.KernelEntry;
  BEGIN
    ASSERT(NOT initialized, "UxPro.Init: Called twice");
    initialized := TRUE;
    tDAdvise := FALSE;  (* TDAdviseClientImporter will set TRUE later *)
    IF NIL = Thread.Fork(TDAdviseClientImporter, NIL) THEN END;
    hostName := NIL;
    hostID := 0;
    SigDfl := LOOPHOLE(0, Unix42.SigHandler);
    SigIgn := LOOPHOLE(1, Unix42.SigHandler);
    SigAlert := LOOPHOLE(2, Unix42.SigHandler);
    backgroundReleaseEnabled := FALSE;
    Thread.InitMutex(gx);
    lastPID := Ux.InitPID - 1; (* i.e. 0 *)
    lastPIDOverflow := FALSE;
    pHead := NIL;
    pTail := NIL;
    FOR e := FIRST(Unix42.KernelEntry) TO LAST(Unix42.KernelEntry) DO
      kep[e] := Unimplemented;
      kec[e] := 0;
      ket[e].seconds := 0; ket[e].microseconds := 0;
    END;
    timingEnabled := FALSE;

    kep[Unix42.SYSexecve] := ExecVE;
    kep[Unix42.SYSexit] := Exit;
    kep[Unix42.SYSfork] := Fork;
    kep[Unix42.SYSgetgid] := GetGID;
    kep[Unix42.SYSgethostid] := GetHostID;
    kep[Unix42.SYSgethostname] := GetHostName;
    kep[Unix42.SYSgetpgrp] := GetPGRP;
    kep[Unix42.SYSgetpid] := GetPID;
    kep[Unix42.SYSgetpriority] := GetPriority;
    kep[Unix42.SYSgetrlimit] := GetRLimit;
    kep[Unix42.SYSgetrusage] := GetRUsage;
    kep[UnsafeOS.SYSgetsignal] := GetSignal;
    kep[UnsafeOS.SYSgetspaceinfo] := GetSpaceInfo; (* Ux, but not Ultrix *)
    kep[Unix42.SYSgetuid] := GetUID;
    kep[Unix42.SYSkill] := Kill;
    kep[Unix42.SYSkillpg] := KillPG;
    kep[Unix42.SYSquota] := Quota;
    kep[Unix42.SYSsetgroups] := SetGroups;
    kep[Unix42.SYSsethostid] := SetHostID;
    kep[Unix42.SYSsethostname] := SetHostName;
    kep[Unix42.SYSsetpgrp] := SetPGRP;
    kep[Unix42.SYSsetpriority] := SetPriority;
    kep[Unix42.SYSsetregid] := SetREGID;
    kep[Unix42.SYSsetreuid] := SetREUID;
    kep[Unix42.SYSsetrlimit] := SetRLimit;
    kep[Unix42.SYSsigblock] := SigBlock;
    kep[Unix42.SYSsigpause] := SigPause;
    kep[Unix42.SYSsigsetmask] := SigSetMask;
    kep[Unix42.SYSsigstack] := SigStack;
    kep[Unix42.SYSsigvec] := SigVec;
    kep[UnsafeOS.SYSstartnewspace] := StartNewSpace; (* Ux, but not Ultrix *)
    kep[UnsafeOS.SYSvfork] := VFork; (* Ultrix too *)
    kep[Unix42.SYSwait] := Wait;
  END Init;
```

```
PROCEDURE PGRP(pT: T): Ux.ProcessID RAISES {};
  BEGIN
    LOCK gx DO RETURN pT^.pgrp END;
  END PGRP;

PROCEDURE PID(pT: T): Ux.ProcessID RAISES {};
  BEGIN RETURN pT^.pid END PID; (* pid is immutable *)

PROCEDURE PIDsPGRP(pid: Ux.ProcessID): Ux.ProcessID RAISES {Ux.Error};
  VAR pT: T;
  BEGIN
    LOCK gx DO pT := PTFromPID(pid); RETURN pT^.pgrp END;
  END PIDsPGRP;

PROCEDURE RaiseError(e: Unix42.SysError; s: Unix42.Signal := Unix42.SigNone)
    RAISES {Ux.Error};
  VAR a: Ux.ErrorArg;
  BEGIN
    ASSERT((e # Unix42.ok) OR (s # Unix42.SigNone)); (* unimpl. in fs *)
    a.e := e; a.s := s;
    RAISE(Ux.Error, a);
  END RaiseError;

PROCEDURE Register(entry: Unix42.KernelEntry; proc: Ux.EntryProc) RAISES {};
  BEGIN
    ASSERT(kep[entry] = Unimplemented, "UxPro.Register: Overwrite");
    kep[entry] := proc;
  END Register;

PROCEDURE RUser(pT: T): User.T RAISES {};
  BEGIN
    LOCK gx DO RETURN pT^.rUser END;
  END RUser;

PROCEDURE SetEUser(pT: T; eUser: User.T) RAISES{};
  BEGIN
    LOCK gx DO pT^.eUser := eUser END;
  END SetEUser;

PROCEDURE SetHost(name: Text.T; override: BOOLEAN := FALSE);
  BEGIN
    LOCK gx DO
      IF (hostName = NIL) OR override THEN hostName := name; END;
    END;
  END SetHost;

PROCEDURE SigEffective(pT: T; signo: Unix42.Signal): BOOLEAN RAISES {};
  BEGIN
    ASSERT((signo = Unix42.SigTTIn) OR (signo = Unix42.SigTTOu));
    LOCK gx DO
      RETURN NOT (
        (pT^.sigVec[signo].handler = SigIgn) OR (signo IN pT^.mask)
        OR (pT^.ppid = Ux.InitPID) OR pT^.vForkChild);
    END;
  END SigEffective;

PROCEDURE Signal(pT: T; signo: Unix42.Signal) RAISES {};
  BEGIN
    LOCK gx DO
      IF pT^.opState # Terminated THEN PostSignal(pT, signo); END;
    END;
  END Signal;

PROCEDURE SignalPG(
```

```
  pgrp: Ux.ProcessID; signo: Unix42.Signal; pT: T := NIL)
  RAISES {};
  VAR
    pp: Ux.T;
    ppT: T;
  BEGIN
    LOCK gx DO
      pp := pHead;
      WHILE pp # NIL DO
        ppT := pp^.pT;
        IF (ppT^.pgrp = pgrp) AND (ppT # pT)
          AND (ppT^.opState # Terminated) THEN PostSignal(ppT, signo);
        END;
        pp := ppT^.next;
      END;
    END;
  END SignalPG;


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

PROCEDURE ActOnSignals(p: Ux.T; s: Unix42.Signal; oldMask: Unix42.SigMask);
  (* Called with gx LOCKed. *)
  (* Delivers all posted unmasked signals, starting with 's' if it is a
  candidate.  Arranges that when execution finally resumes from the current
  kernel call (after any handlers), the mask is set to 'oldMask'.  *)
  VAR
    pT: T;
    sigs: Unix42.SigMask;
    h: Unix42.SigHandler;
    code: INTEGER;
    switchStacks: BOOLEAN;
    ea: Ux.ErrorArg;
  BEGIN
    pT := p^.pT;
    LOOP
      sigs := pT^.posted - pT^.mask;
      IF pT^.vForkChild THEN EXCL(sigs, Unix42.SigTStp); END;
      IF sigs = NoSigs THEN EXIT; END;
      IF (s = Unix42.SigNone) OR NOT (s IN sigs) THEN
        s := Unix42.FirstSig;
        WHILE NOT (s IN sigs) DO INC(s) END;
      END;
      EXCL(pT^.posted, s);
      h := pT^.sigVec[s].handler;
      IF h = SigDfl THEN
        IF s IN NoActionSigs THEN (* nothing *) s := Unix42.SigNone;
        ELSIF s IN StopSigs THEN DoStop(p, s); s := Unix42.SigCont;
        ELSE RaiseTerminate(pT, s, s IN CoreImageSigs);
        END;
      ELSIF h = SigIgn THEN (* nothing *) s := Unix42.SigNone;
      ELSIF h = SigAlert THEN (* Topaz-style handler *)
        INCL(pT^.alerted, s);
        UxAS.AlertHandler(p^.aT);
        s := Unix42.SigNone;
      ELSE (* Ultrix-style handler *)
        IF UxAS.IsTopaz(p^.aT) THEN RaiseTerminate(pT, s, TRUE); END;
        IF s = Unix42.SigFPE THEN code := pT^.fpeCode;
        ELSIF s = Unix42.SigIll THEN code := pT^.illCode;
        ELSE code := 0;
        END;
        switchStacks := pT^.sigVec[s].onStack AND NOT pT^.sigStack.onStack;
        TRY
          UxAS.PushHandlerCall(
            p^.aT,
            s, code,
            pT^.sigVec[s].handler,
            oldMask, pT^.sigStack.onStack,
            switchStacks, pT^.sigStack.sp);
        EXCEPT Ux.Error(ea):
          ASSERT(ea.s # Unix42.SigNone); (* ***** remove later *)
          RaiseTerminate(pT, ea.s, ea.s IN CoreImageSigs);
        END;
        pT^.mask := pT^.mask + pT^.sigVec[s].mask;
        INCL(pT^.mask, s);
        oldMask := pT^.mask;
        IF switchStacks THEN pT^.sigStack.onStack := TRUE; END;
        s := Unix42.SigNone;
      END; (* real handler *)
    END;
    pT^.mask := oldMask;
  END ActOnSignals;

PROCEDURE DoExec(
    p: Ux.T;
    name: Text.T;
    argv, envp: Ux.RefArrayOfText;
    aTArgs: UxAS.T;
    argva, envpa: UNSIGNED)
    : BOOLEAN
  RAISES {Ux.Error, Thread.Alerted};
  (* If DoExec raises Ux.Error, 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. *)
  VAR s: Unix42.Signal; pT, ppT: T; ok: BOOLEAN;
  BEGIN
    pT := p^.pT;
    ok := UxAS.Load(p, name, argv, envp, aTArgs, argva, envpa, pT^.vForkChild);
    IF ok AND tDAdvise THEN
      TDAdvise.SpaceLoaded(UxAS.GetSpace(p^.aT), name);
    END;
    LOCK gx DO
      IF pT^.vForkChild THEN (* give address space back to parent *)
        pT^.vForkChild := FALSE;
        ppT := PTFromPID(pT^.ppid, FALSE);
        Thread.Signal(ppT^.c);
      END;
      FOR s := Unix42.FirstSig TO Unix42.LastSig DO
        WITH pT^.sigVec[s] DO
          IF handler # SigIgn THEN handler := SigDfl; END;
          onStack := FALSE;
          mask := NoSigs;
        END;
        pT^.sigStack.sp := 0;
        pT^.sigStack.onStack := FALSE;
      END;
      pT^.name := name;
      pT^.argv := argv;
      pT^.envp := envp;
    END;
    UxFile.Exec(p^.fT);
    RETURN ok;
  END DoExec;

PROCEDURE DoFork(
    p: Ux.T;
    VAR args: Ux.ArgList;
```

```
    vFork: BOOLEAN := FALSE)
    : INTEGER;
  VAR pT, pTNew: T; pNew: Ux.T; ea: Ux.ErrorArg;
  BEGIN
    pT := p^.pT;
    IF UxAS.IsTopaz(p^.aT) THEN RaiseError(Unix42.EINVAL); END;
    pNew := New(p, NIL, vFork);
    pTNew := pNew^.pT;
    TRY
      UxAS.CopySpace(p^.aT, pNew^.aT, vFork);
    EXCEPT
      Ux.Error(ea): Release(pNew); RaiseError(ea.e);
    END;
    IF NOT vFork AND tDAdvise THEN
      TDAdvise.SpaceLoaded(UxAS.GetSpace(pNew^.aT), pT^.name);
    END;
    LOCK gx DO
      pTNew^.pid := NewPID();
      Insert(pNew);

      UxAS.R0Result(pNew^.aT, pT^.pid);
      UxAS.R1Result(pNew^.aT, 1); (* non-zero means we're in the child *)
      pTNew^.wThread := Thread.Fork(Watcher, pNew);
      UxAS.SetThreadPriority(pTNew^.wThread, pTNew^.wPriority);
      pTNew^.initWTime := ThreadFriends.GetCPUTime(pTNew^.wThread);
      IF vFork THEN
        UxAS.HideThread(p^.aT);
        pT^.vForkParent := TRUE;
        WHILE pTNew^.vForkChild DO Thread.Wait(gx, pT^.c); END;
        pT^.vForkParent := FALSE;
        UxAS.RestoreThread(p^.aT);
      END;
    END;
    UxAS.R1Result(p^.aT, 0); (* 0 means we're in the parent *)
    RETURN pTNew^.pid;
  END DoFork;

PROCEDURE DoForkExec(
    ppid: Ux.ProcessID; (* parent process id, or 0 for first process *)
    name: Text.T;
    argv, envp: Ux.RefArrayOfText;
    aTArgs: UxAS.T;
    argva, envpa: UNSIGNED;
    fT: UxFile.T := NIL)
    : Ux.ProcessID;
  VAR p, pNew: Ux.T; pTNew: T; success: BOOLEAN;
  BEGIN
    IF ppid = 0 THEN p := NIL; ELSE LOCK gx DO p := TFromPID(ppid); END; END;
    pNew := New(p, fT, FALSE);
    pTNew := pNew^.pT;
    TRY
      success := FALSE;
      IF NOT DoExec(pNew, name, argv, envp, aTArgs, argva, envpa) THEN
        RaiseError(Unix42.EIO);
      END;
      LOCK gx DO
        IF p = NIL THEN (* only Init process may be parentless *)
          IF pHead # NIL THEN RaiseError(Unix42.EINVAL); END;
        ELSE
          IF p^.pT^.opState = Terminated THEN RaiseError(Unix42.ESRCH) END;
        END;

        success := TRUE; (* no more errors raised after this point *)

        pTNew^.pid := NewPID();
        Insert(pNew);

        pTNew^.wThread := Thread.Fork(Watcher, pNew);
        UxAS.SetThreadPriority(pTNew^.wThread, pTNew^.wPriority);
        pTNew^.initWTime := ThreadFriends.GetCPUTime(pTNew^.wThread);
      END;
    FINALLY
      IF NOT success THEN Release(pNew); END;
    END;
    RETURN pTNew^.pid;
  END DoForkExec;

PROCEDURE DoKernelCall(
  p: Ux.T; entry: Unix42.KernelEntry; VAR (*in*) args: Ux.ArgList)
  : Unix42.Signal;
  VAR ea: Ux.ErrorArg;
  BEGIN
    ea.s := Unix42.SigNone;
    TRY UxAS.R0Result(p^.aT, kep[entry](p, args));
    EXCEPT
    | Ux.Error(ea):
        IF ea.e = Unix42.ok THEN
          ASSERT(ea.s # Unix42.SigNone);
          IF ((ea.s = Unix42.SigTTIn) OR (ea.s = Unix42.SigTTOu))
            AND (p^.pT^.sigVec[ea.s].handler = SigAlert) THEN
            (* Kernel call returns with error, not signal. *)
            UxAS.R0Result(p^.aT, ORD(Unix42.EWOULDBLOCK), TRUE);
            ea.s := Unix42.SigNone;   must also remove from posted ?
          ELSE (* kernel call will be retried if handler for ea.s returns. *)
            UxAS.BackupKernelCall(p^.aT, entry);
          END;
        ELSE (* kernel call returns with error *)
          UxAS.R0Result(p^.aT, ORD(ea.e), TRUE); (* set error flag *)
        END;
    | Thread.Alerted:
        (* Kernel call will be retried if handler for async. signal returns. *)
        UxAS.BackupKernelCall(p^.aT, entry);
    END;
    RETURN ea.s;
  END DoKernelCall;

PROCEDURE DoKill(from, to: T; s: Unix42.Signal): BOOLEAN;
  (* Called with gx LOCKed. *)
  PROCEDURE NotSigContOK(): BOOLEAN;
    VAR ppT: T;
    BEGIN
      IF s # Unix42.SigCont THEN RETURN TRUE END;
      ppT := to;
      LOOP
        IF ppT^.ppid = from^.pid THEN RETURN FALSE END;
        IF ppT^.ppid = Ux.InitPID THEN RETURN TRUE END;
        ppT := PTFromPID(ppT^.ppid, FALSE);
      END;
    END NotSigContOK;
  BEGIN
    IF NOT User.IsSuper(from^.eUser) AND
      NOT User.Equal(from^.eUser, to^.eUser) AND
      NotSigContOK() THEN RETURN FALSE END;
    PostSignal(to, s);
    RETURN TRUE;
  END DoKill;

PROCEDURE DoNothing(p: Ux.T): INTEGER;
  BEGIN
    UxAS.R1Result(p^.aT, 0);
```

```
    RETURN 0;
  END DoNothing;

PROCEDURE DoStop(p: Ux.T; s: Unix42.Signal);
  (* Called with gx LOCKed. *)
  VAR pT, pTParent: T;
  BEGIN
    pT := p^.pT;
    UxAS.StopAll(p^.aT);
    pT^.opState := Stopped; pT^.untraced := TRUE;
    pT^.ws.wStopVal := Unix42.WSTOPPED;
    pT^.ws.wStopSig := s;
    pTParent := PTFromPID(pT^.ppid, FALSE);
    PostSignal(pTParent, Unix42.SigChld);
    Thread.Signal(pTParent^.c);
    LOOP
      IF Unix42.SigKill IN pT^.posted THEN
        RaiseTerminate(pT, Unix42.SigKill, FALSE);
      END;
      IF Unix42.SigCont IN pT^.posted THEN EXIT END;
      TRY Thread.AlertWait(gx, pT^.c);
      EXCEPT Thread.Alerted:
      END;
    END;
    FOR s := Unix42.FirstSig TO Unix42.LastSig DO
      IF (s IN StopSigs) AND (s IN pT^.posted)
        AND (pT^.sigVec[s].handler = SigDfl) THEN EXCL(pT^.posted, s);
      END;
    END;
    UxAS.StartAll(p^.aT);
    pT^.opState := Running; pT^.untraced := FALSE;
  END DoStop;

PROCEDURE DoTermination(p: Ux.T);
  VAR
    pp: Ux.T;
    pT, ppT: T;
    cTime, wTime: Time.T;
    backgroundRelease, terminatedChild: BOOLEAN;
  BEGIN
    pT := p^.pT;
    UxAS.StopAll(p^.aT);
    cTime := UxAS.GetClientTime(p^.aT, TRUE);
    wTime := GetWTime(pT);

    (* Since the client thread(s) will never run again, and will be
    destroyed soon, we must not deliver any alerts to them.  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
      InitializeSigVec(pT^.sigVec);
      pT^.mask := AllSigs - Unix42.SigMask{Unix42.SigInt};
    END;

    IF pT^.ws.wCoreDump THEN
      TRY UxAS.MakeCoreImage(p, pT^.ws.wTermSig);
      EXCEPT Ux.Error, Thread.Alerted: pT^.ws.wCoreDump := FALSE;
      END;
    END;

    backgroundRelease := backgroundReleaseEnabled AND NOT pT^.vForkChild;
    IF NOT backgroundRelease THEN Release(p); END;
    LOCK gx DO
      pT^.vForkChild := FALSE;
      pT^.opState := Terminated; pT^.untraced := FALSE;

      (* Add our rusage in with that of our descendants. *)
      TimeINC(pT^.descCTime, cTime);
      TimeINC(pT^.descWTime, wTime);

      (* Transfer our children to init process and inform our parent of
      our termination. *)
      pp := pHead;
      terminatedChild := FALSE;
      (* (pT^.ppid = 0) => Init process is terminating *)
      WHILE pp # NIL DO
        ppT := pp^.pT;
        IF ppT^.ppid = pT^.pid THEN (* one of our children *)
          ppT^.ppid := Ux.InitPID;
          terminatedChild := terminatedChild OR (ppT^.opState = Terminated);
        ELSIF ppT^.pid = pT^.ppid THEN (* our parent *)

          (* Add our rusage to that of our parent's descendants. *)
          TimeINC(ppT^.descCTime, pT^.descCTime);
          TimeINC(ppT^.descWTime, pT^.descWTime);

          PostSignal(ppT, Unix42.SigChld);
          Thread.Signal(ppT^.c);
        END;
        pp := ppT^.next;
      END;
      IF terminatedChild THEN
        PostSignal(PTFromPID(Ux.InitPID, FALSE), Unix42.SigChld);
      END;
    END; (* LOCK gx *)
    IF backgroundRelease THEN Release(p); END;
  END DoTermination;

PROCEDURE InitializeSigVec(VAR sigVec: SignalVector);
  VAR s: Unix42.Signal;
  BEGIN
    FOR s := Unix42.FirstSig TO Unix42.LastSig DO
      sigVec[s].handler := SigDfl;
      sigVec[s].mask := NoSigs;
      sigVec[s].onStack := FALSE;
    END;
  END InitializeSigVec;

PROCEDURE Insert(pNew: Ux.T);
  (* Called with gx LOCKed. *)
  (* Inserts pNew into list of processes, preserving ascending order. *)
  VAR p: Ux.T; pT, pTPrev: T; pid: Ux.ProcessID;
  BEGIN
    IF lastPIDOverflow THEN
      pid := pNew^.pT^.pid;
      pTPrev := NIL;
      p := pHead; pT := p^.pT;
      WHILE (pT^.pid < pid) AND (pT^.next # NIL) DO
        pTPrev := pT;
        p := pT^.next;
        pT := p^.pT;
      END;
      IF pT^.pid > pid THEN (* pPrev -> pNew -> p *)
        pTPrev^.next := pNew;
        pNew^.pT^.next := p;
      ELSE (* p, pTail -> pNew *)
        pT^.next := pNew;
        pTail := pNew;
      END;
    ELSE (* can just append *)
```

```
      IF pTail # NIL THEN pTail^.pT^.next := pNew; END;
      pTail := pNew;
      IF pHead = NIL THEN pHead := pNew; END;
    END;
  END Insert;

PROCEDURE New(p: Ux.T; fT: UxFile.T := NIL; vFork: BOOLEAN := FALSE): Ux.T;
  (* Creates new Ux.T based on parent 'p' (possibly NIL).  If 'fT' is
  nonNIL it will be used, otherwise a new file system state will be
  created (based on 'p^.fT' if 'p#NIL'.) *)
  VAR
    pNew: Ux.T;
    pAT: UxAS.T; (* parent's address space or NIL *)
    pFT: UxFile.T; (* parent's descriptor state or NIL *)
  BEGIN
    NEW(pNew);
    NEW(pNew^.pT);
    WITH pNew^.pT^ DO (* private state of new process *)
      IF p = NIL THEN (* first process *)
        pAT := NIL;
        pFT := NIL;
        wPriority := ThreadFriends.NormalPriority;
        mask := NoSigs;
        sigStack.sp := 0;
        sigStack.onStack := FALSE;
        InitializeSigVec(pNew^.pT^.sigVec);
        ppid := 0; pgrp := Ux.InitPID;
        rUser := User.SuperUser(); ASSERT(rUser # NIL); eUser := rUser;
      ELSE (* child process *)
        pAT := p^.aT;
        pFT := p^.fT;
        LOCK gx DO pNew^.pT^ := p^.pT^ END; (* copy parent's state *)
        ppid := p^.pT^.pid; (* change parent id *)
      END;
      next := NIL;
      Thread.InitCondition(c);
      opState := Running; untraced := FALSE;
      vForkParent := FALSE;
      vForkChild := vFork;
      posted := NoSigs; (* posted sigs never inherited *)
      alertPending := FALSE; (* ditto *)
      alerted := NoSigs; (* ditto *)
      pid := 0; (* correct value filled in later *)
      initWTime.seconds := LAST(Time.Seconds);
      initWTime.microseconds := LAST(Time.Microseconds);
      descCTime.seconds := 0; descCTime.microseconds := 0;
      descWTime := descCTime;
    END; (* WITH pNew^.pT^ *)
    pNew^.aT := UxAS.New(pAT);
    IF fT = NIL THEN pNew^.fT := UxFile.Fork(pFT);
    ELSE pNew^.fT := fT;
    END;
    pNew^.tT := NIL;
    RETURN pNew;
  END New;

PROCEDURE NewPID(): Ux.ProcessID;
  (* Called with gx LOCKed. *)
  (* Returns new process identifier, guaranteed to be different than that
     of any other in use on this machine. *)
  VAR p: Ux.T;
  BEGIN
    INC(lastPID);
    IF MaxPID < lastPID THEN
      lastPID := Ux.InitPID + 1;
      lastPIDOverflow := TRUE;
    END;
    IF lastPIDOverflow THEN
      LOOP (* until unique pid found *)
        p := pHead;
        WHILE (p # NIL) AND (p^.pT^.pid < lastPID) DO p := p^.pT^.next; END;
        IF (p = NIL) OR (p^.pT^.pid > lastPID) THEN EXIT; END;
        (* p^.pT^.pid = lastPID *)
        INC(lastPID);
        IF MaxPID < lastPID THEN lastPID := Ux.InitPID + 1; END;
      END;
    END;
    RETURN lastPID;
  END NewPID;

PROCEDURE PostSignal(pT: T; s: Unix42.Signal);
  (* Called with gx LOCKed. *)
  VAR
    h: Unix42.SigHandler;
    p: Ux.T;
  BEGIN
    ASSERT(pT^.opState # Terminated);
    IF s # Unix42.SigNone THEN
      h := pT^.sigVec[s].handler;
      IF h # SigIgn THEN
        INCL(pT^.posted, s);
        IF NOT (s IN pT^.mask) THEN
          IF h = SigAlert THEN (* Topaz-style handler *)
            EXCL(pT^.posted, s);
            INCL(pT^.alerted, s);
            p := TFromPID(pT^.pid, FALSE); (* ugh *)
            UxAS.AlertHandler(p^.aT);
          ELSIF ((h # SigDfl) OR NOT (s IN NoAlertSigs))
            AND ((s # Unix42.SigTStp) OR NOT pT^.vForkChild) THEN
            pT^.alertPending := TRUE;
            UxAS.SetThreadPriority(pT^.wThread, UxAS.AlertPriority);
            Thread.Alert(pT^.wThread);
          END;
        END;
      END;
    END;
  END PostSignal;

PROCEDURE PTFromPID(pid: Ux.ProcessID; esrch: BOOLEAN := TRUE): T;
  (* Called with gx LOCKed. *)
  VAR p: Ux.T;
  BEGIN
    p := TFromPID(pid, esrch);
    RETURN p^.pT;
  END PTFromPID;

PROCEDURE RaiseTerminate(pT: T; s: Unix42.Signal; dump: BOOLEAN)
    RAISES{Terminate};
  BEGIN
    pT^.ws.wTermSig := s;
    pT^.ws.wCoreDump := dump;
    RAISE(Terminate);
  END RaiseTerminate;

PROCEDURE Release(p: Ux.T);
  VAR space: NubTypes.Space; vForkChild: BOOLEAN;
  BEGIN
    space := UxAS.GetSpace(p^.aT);
    vForkChild := p^.pT^.vForkChild;
    UxTime.Exit(p^.tT);
```

```
    RPCLocal.Moribund(space);
    UxFile.Exit(p^.fT);
    IF NOT vForkChild AND tDAdvise THEN TDAdvise.SpaceGone(space); END;
    UxAS.Release(p^.aT, vForkChild);
  END Release;

PROCEDURE RLimit(p: Ux.T; resource: Unix42.Resource): Unix42.RLimit;
  CONST rLimitInfinity = 07fffffffH;
  VAR rLimit: Unix42.RLimit;
  BEGIN
    CASE resource OF
    | Unix42.rLimitData, Unix42.rLimitStack:
        UxAS.GetRLimit(p^.aT, resource, rLimit);
    | ELSE rLimit.cur := rLimitInfinity; rLimit.max := rLimitInfinity;
    END;
    RETURN rLimit;
  END RLimit;

PROCEDURE TDAdviseClientImporter(arg: REFANY): REFANY;
  (* Dally until userttd exports TDAdvise. *)
  VAR ie: RPCRuntime.ImportErrors;
  BEGIN
    REPEAT
      TRY TDAdviseClient.Import("", (*localOnly:*)TRUE); tDAdvise := TRUE;
      EXCEPT RPCRuntime.ImportFailed(ie): Time.LongPause(5); (* 5 seconds *)
      END;
    UNTIL tDAdvise;
    RETURN NIL;
  END TDAdviseClientImporter;

PROCEDURE TFromPID(pid: Ux.ProcessID; esrch: BOOLEAN := TRUE): Ux.T;
  (* Called with gx LOCKed. *)
  VAR p: Ux.T;
  BEGIN
    p := pHead;
    WHILE (p # NIL) AND (p^.pT^.pid < pid) DO p := p^.pT^.next; END;
    IF (p = NIL) OR (p^.pT^.pid > pid) OR (p^.pT^.opState = Terminated) THEN
      IF esrch THEN RaiseError(Unix42.ESRCH); END;
      ASSERT (FALSE); (* pid must exist! *)
    END;
    RETURN p;
  END TFromPID;

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 TimeINC(VAR (*in out*) a: Time.T; b: Time.T);
  VAR us: CARDINAL;
  BEGIN
    us := a.microseconds + b.microseconds;
    INC(a.seconds, b.seconds);
    IF us >= 1000000 THEN
      a.microseconds := us - 1000000;
      INC(a.seconds, 1);
    ELSE
      a.microseconds := us;
    END;
  END TimeINC;

PROCEDURE TimeSub(a, b: Time.T): Time.T;
  (* Returns 'a' - 'b'. *)
  VAR t: Time.T;
  BEGIN
    IF a.microseconds >= b.microseconds THEN
      t.microseconds := a.microseconds - b.microseconds;
      t.seconds := a.seconds - b.seconds;
    ELSE
      t.microseconds := 1000000 + a.microseconds - b.microseconds;
      t.seconds := a.seconds - 1 - b.seconds;
    END;
    RETURN t;
  END TimeSub;

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;
  VAR
    p: Ux.T;
    pT: T;
    s: Unix42.Signal;
    event: UxAS.Event;
    ea: Ux.ErrorArg;
    time: Time.T;
  BEGIN
    p := NARROW(a, Ux.T);
    time.seconds := 0; time.microseconds := 0;
    LOCK gx DO (* wait for parent to complete p^ *)
      pT := p^.pT;
    END;
    TRY
      LOOP (* main watcher loop *)

        (* Run client until it traps or we are alerted. *)
        TRY UxAS.WaitEvent(p^.aT, event);
        EXCEPT Ux.Error(ea): (* e.g. Unix42.SigSegV *)
          ASSERT(ea.s # Unix42.SigNone); (* ***** remove later *)
          RaiseTerminate(pT, ea.s, ea.s IN CoreImageSigs);
        END;

        (* If kernel call, execute without holding gx. *)
        CASE event.kind OF
        | UxAS.KernelCall:
            IF timingEnabled THEN
              time := ThreadFriends.GetCPUTime(pT^.wThread);
            END;
            s := DoKernelCall(p, event.entry, event.argList);
              (* Exit raises Terminate *)
            IF s # Unix42.SigNone THEN
              event.kind := UxAS.Exception; event.sig := s; event.code := 0;
            END;
        | ELSE ;
        END;
```

```
      LOCK gx DO
        s := Unix42.SigNone;
        CASE event.kind OF
        | UxAS.KernelCall:
            INC(kec[event.entry]);
            IF timingEnabled THEN
              TimeINC(ket[event.entry],
                TimeSub(ThreadFriends.GetCPUTime(pT^.wThread), time));
            END;
        | UxAS.Exception:
            s := event.sig; (* first signal to look for *)
            INCL(pT^.posted, s);
            IF s = Unix42.SigFPE THEN pT^.fpeCode := event.code;
            ELSIF s = Unix42.SigIll THEN pT^.illCode := event.code;
            END;
        | UxAS.Alert:
        | UxAS.HandlerReturn:
            pT^.mask := event.mask - PowerfulSigs;
            pT^.sigStack.onStack := event.onStack;
        END;

        IF pT^.alertPending THEN
          pT^.alertPending := FALSE;
          IF Thread.TestAlert() THEN END; (* drain pending alert *)
          IF 0 = ThreadFriends.SetPriority(pT^.wPriority) THEN END;
        END;

        IF (pT^.posted - pT^.mask) # NoSigs THEN
          (* ***** Is pT^.alertPending an adequate guard? *)
          ActOnSignals(p, s, pT^.mask);
        END;

      END; (* LOCK gx *)

    END; (* main watcher loop *)
  EXCEPT Terminate: DoTermination(p); (* only exit from main loop *)
  END;
  RETURN NIL;
END Watcher;


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

PROCEDURE ExecVE(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    aT: UxAS.T;
    name: Text.T;
    ok, dump: BOOLEAN;
  BEGIN
    aT := p^.aT;
    IF UxAS.IsTopaz(aT) THEN RaiseError(Unix42.EINVAL); END;
    name := UxAS.GetText(aT, args[0]);
    TRY
      ok := DoExec(p, name, NIL, NIL, aT, args[1], args[2]);
      dump := TRUE;
    EXCEPT
      Thread.Alerted: ok := FALSE; dump := FALSE;
    END;
    IF NOT ok THEN RaiseTerminate(p^.pT, Unix42.SigKill, dump); END;
    RETURN 0;
  END ExecVE;
```

```
PROCEDURE Exit(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* Sometimes known as _exit or underbarexit. *)
  BEGIN
    p^.pT^.ws.wRetCode := args[0] MOD 256;
    RaiseTerminate(p^.pT, Unix42.SigNone, FALSE);
    ASSERT(FALSE);
    RETURN 0; (* to make compiler happy *)
  END Exit;

PROCEDURE Fork(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    RETURN DoFork(p, args);
  END Fork;

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

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

PROCEDURE GetHostName(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR count: CARDINAL; zero: CHAR;
  BEGIN
    LOCK gx DO
      count :=
        UxAS.PutText(p^.aT, args[0], hostName, LOOPHOLE(args[1], CARDINAL));
      IF count < LOOPHOLE(args[1], CARDINAL) THEN
        zero := 0C;
        UxAS.Put(p^.aT, args[0] + count, zero);
      END;
    END;
    RETURN 0;
  END GetHostName;

PROCEDURE GetPGRP(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pid: Ux.ProcessID;
    pT: T;
  BEGIN
    pid := LOOPHOLE(args[0], Ux.ProcessID);
    LOCK gx DO
      IF pid = 0 THEN pT := p^.pT ELSE pT := PTFromPID(pid) END;
      RETURN LOOPHOLE(pT^.pgrp, INTEGER);
    END;
  END GetPGRP;

PROCEDURE GetPID(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    LOCK gx DO
      UxAS.R1Result(p^.aT, LOOPHOLE(p^.pT^.ppid, INTEGER));
      RETURN LOOPHOLE(p^.pT^.pid, INTEGER)
    END;
  END GetPID;

PROCEDURE GetPriority(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    which: Unix42.WhichPrio;
    who, ultrixPriority: INTEGER;
    found, match: BOOLEAN;
    pp: Ux.T;
  BEGIN
```

```
    IF ORD(LAST(Unix42.WhichPrio)) < args[0] THEN
      RaiseError(Unix42.EINVAL);
    END;
    which := LOOPHOLE(args[0], Unix42.WhichPrio);
    who := LOOPHOLE(args[1], INTEGER);
    ultrixPriority := LAST(INTEGER); (* lowest Ultrix priority *)
    found := FALSE;
    LOCK gx DO
      pp := pHead;
      WHILE (pp # NIL) AND ((which # Unix42.prioProcess) OR NOT found) DO
        IF pp^.pT^.opState # Terminated THEN
          CASE which OF
          | Unix42.prioProcess:
              match := (who = 0) AND (pp = p) OR (who = pp^.pT^.pid);
              (* ***** EACCES unless eUser or rUser matches *)
          | Unix42.prioPgrp:
              match := who = pp^.pT^.pgrp;
              (* ***** EACCES unless eUser or rUser matches *)
          | Unix42.prioUser: match := who = User.GetUserID(pp^.pT^.rUser);
              (* ***** or eUser? *)
          END;
          IF match THEN
            ultrixPriority :=
              MIN(ultrixPriority,
                ThreadToUltrixPriority(UxAS.GetClientPriority(pp^.aT)));
            found := TRUE;
          END;
        END;
        pp := pp^.pT^.next;
      END;
    END;
    IF NOT found THEN RaiseError(Unix42.ESRCH); END;
    RETURN ultrixPriority;
  END GetPriority;

PROCEDURE GetRLimit(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR rLimit: Unix42.RLimit;
  BEGIN
    rLimit := RLimit(p, LOOPHOLE(args[0], Unix42.Resource));
    UxAS.Put(p^.aT, args[1], rLimit);
    RETURN 0;
  END GetRLimit;

PROCEDURE GetRUsage(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pT: T;
    rUsage: Unix42.RUsage;
  BEGIN
    pT := p^.pT;
    System.Zero(System.Adr(rUsage), System.ByteSize(rUsage));
    (* args[0] tells 'who': self or children. *)
    IF LOOPHOLE(args[0], INTEGER) = Unix42.rUsageSelf THEN
      rUsage.uTime := UxAS.GetClientTime(p^.aT);
      rUsage.sTime := GetWTime(pT);
    ELSIF LOOPHOLE(args[0], INTEGER) = Unix42.rUsageChildren THEN
      rUsage.uTime := pT^.descCTime;
      rUsage.sTime := pT^.descWTime;
    ELSE
      RaiseError(Unix42.EINVAL);
    END;
    UxAS.Put(p^.aT, args[1], rUsage);
    RETURN 0;
  END GetRUsage;

PROCEDURE GetSignal(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pT: T;
    s: Unix42.Signal;
  BEGIN
    pT := p^.pT;
    UxAS.SetHandler(p^.aT);
    LOCK gx DO
      FOR s := Unix42.FirstSig TO Unix42.LastSig DO
        IF s IN pT^.alerted THEN
          EXCL(pT^.alerted, s);
          RETURN LOOPHOLE(s, INTEGER);
        END;
      END;
      RETURN LOOPHOLE(Unix42.SigNone, INTEGER);
    END;
  END GetSignal;

PROCEDURE GetSpaceInfo(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pid: Ux.ProcessID;
    addr: UNSIGNED;
    nElts, nEltsSoFar: INTEGER;
    info: UnsafeOS.TrapSpaceInfo;
    pp: Ux.T;
    ppT: T;
    i, j, n: CARDINAL;
  BEGIN
    pid := args[0];
    addr := args[1];
    nElts := args[2];
    IF nElts < 0 THEN RaiseError(Unix42.EINVAL); END;
    nEltsSoFar := 0;
    LOCK gx DO
      pp := pHead;
      LOOP (* report next process *)
        IF nEltsSoFar = nElts THEN RETURN nEltsSoFar; END;
        LOOP (* find next unreported process *)
          IF pp = NIL THEN RETURN nEltsSoFar; END;
          ppT := pp^.pT;
          IF ppT^.pid > pid THEN EXIT; END;
          pp := ppT^.next;
        END;
        WITH info DO
          pid := ppT^.pid; ppid := ppT^.ppid; pgrp := ppT^.pgrp;
          uid := User.GetUserID(ppT^.rUser);
          space := UxAS.GetSpace(pp^.aT);
          ctlTty := "??";
          runState := LOOPHOLE(ppT^.opState, OS.SpaceState);
          IF runState = OS.SSTerminating THEN (* unanswerable questions *)
            topaz := FALSE;
            cTime.seconds := LAST(Time.Seconds);
            cTime.microseconds := LAST(Time.Microseconds);
            wTime := cTime;
          ELSE
            topaz := UxAS.IsTopaz(pp^.aT);
            cTime := UxAS.GetClientTime(pp^.aT);
            wTime := GetWTime(ppT);
          END;
          ws := ppT^.ws;
          FOR j := 0 TO HIGH(commandString) DO commandString[j] := ' '; END;
          n := Text.GetSub(ppT^.name, 0, commandString);
(*
          i := 0; j := 0;
          LOOP (* append i'th argument to command string *)
            IF i = NUMBER(ppT^.argv^) THEN EXIT; END;
```

```
            n := Text.GetSub(ppT^.argv^[i], 0, commandString, j);
            INC(j, n + 1); (* blank terminated if room *)
            IF j >= NUMBER(commandString) THEN EXIT; END;
            INC(i, 1);
          END;
*)
        END; (* WITH info *)
        UxAS.Put(p^.aT, addr, info);
        INC(addr, System.ByteSize(info));
        INC(nEltsSoFar, 1);
        pid := info.pid;
      END; (* report next process loop *)
    END; (* LOCK gx *)
    (* Can't get here. *)
  END GetSpaceInfo;

PROCEDURE GetUID(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    LOCK gx DO
      UxAS.R1Result(p^.aT, User.GetUserID(p^.pT^.eUser));
      RETURN User.GetUserID(p^.pT^.rUser);
    END;
  END GetUID;

PROCEDURE Kill(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pp: Ux.T;
    pT, ppT: T;
    pid: Ux.ProcessID;
    sig: Unix42.Signal;
    dummy: BOOLEAN;
  BEGIN
    pT := p^.pT;
    IF args[1] > ORD(Unix42.LastSig) THEN RaiseError(Unix42.EINVAL) END;
    sig := LOOPHOLE(args[1], Unix42.Signal);
    pid := LOOPHOLE(args[0], Ux.ProcessID);
    LOCK gx DO
      IF pid = -1 THEN (* broadcast *)
        IF NOT User.IsSuper(pT^.eUser) THEN RaiseError(Unix42.EPERM) END;
        pp := pHead;
        WHILE pp # NIL DO
          ppT := pp^.pT;
          IF (ppT # pT) AND (ppT^.pid # Ux.InitPID)
              AND (ppT^.opState # Terminated) THEN
            PostSignal(ppT, sig);
          END;
          pp := ppT^.next;
        END;
      ELSIF pid = 0 THEN (* process group *)
        pp := pHead;
        WHILE pp # NIL DO
          ppT := pp^.pT;
          IF (ppT^.pgrp = pT^.pgrp) AND (ppT^.opState # Terminated) THEN
            dummy := DoKill(pT, ppT, sig);
          END;
          pp := ppT^.next;
        END;
      ELSE (* specific process *)
        ppT := PTFromPID(pid);
        IF NOT DoKill(pT, ppT, sig) THEN RaiseError(Unix42.EPERM) END;
      END;
    END; (* LOCK gx *)
    RETURN 0;
  END Kill;
```

```
PROCEDURE KillPG(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pgrp: Ux.ProcessID;
    pp: Ux.T;
    ppT: T;
    sig: Unix42.Signal;
    killed: BOOLEAN;
  BEGIN
    IF args[1] > ORD(Unix42.LastSig) THEN RaiseError(Unix42.EINVAL); END;
    sig := LOOPHOLE(args[1], Unix42.Signal);
    pgrp := LOOPHOLE(args[0], Ux.ProcessID);
    IF pgrp = 0 THEN pgrp := p^.pT^.pgrp; END;
    killed := FALSE;
    LOCK gx DO
      pp := pHead;
      WHILE pp # NIL DO
        ppT := pp^.pT;
        IF (ppT^.pgrp = pgrp) AND (ppT^.opState # Terminated)
          AND DoKill(p^.pT, ppT, sig) THEN killed := TRUE END;
        pp := ppT^.next;
      END;
    END;
    IF NOT killed THEN RaiseError(Unix42.ESRCH); END;
    RETURN 0;
  END KillPG;

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

PROCEDURE SetGroups(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    IF NOT User.IsSuper(p^.pT^.eUser) THEN RaiseError(Unix42.EPERM); END;
    RETURN DoNothing(p);
  END SetGroups;

PROCEDURE SetHostID(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    IF NOT User.IsSuper(p^.pT^.eUser) THEN RaiseError(Unix42.EPERM); END;
    hostID := LOOPHOLE(args[0], INTEGER);
    RETURN 0;
  END SetHostID;

PROCEDURE SetHostName(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR buffer: ARRAY [0..maxNameLen-1] OF CHAR; count: CARDINAL;
  BEGIN
    LOCK gx DO
      IF NOT User.IsSuper(p^.pT^.eUser) THEN RaiseError(Unix42.EPERM); END;
      count := MIN(LOOPHOLE(args[1], CARDINAL), maxNameLen);
      UxAS.Get(p^.aT, args[0], buffer, 0, count);
      hostName := Text.FromSub(buffer, 0, count);
    END;
    RETURN 0;
  END SetHostName;

PROCEDURE SetPGRP(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pid, pgrp: Ux.ProcessID;
    pT, ppT: T;
  PROCEDURE NotDescendent(): BOOLEAN;
    VAR pppT: T;
    BEGIN
      pppT := ppT;
      LOOP
```

```
        IF pppT^.ppid = pT^.pid THEN RETURN FALSE END;
        IF pppT^.ppid = Ux.InitPID THEN RETURN TRUE END;
        pppT := PTFromPID(pppT^.ppid, FALSE);
      END;
    END NotDescendent;
  BEGIN
    pT := p^.pT;
    pid := LOOPHOLE(args[0], Ux.ProcessID);
    pgrp := LOOPHOLE(args[1], Ux.ProcessID);
    LOCK gx DO
      IF pid = 0 THEN ppT := pT ELSE ppT := PTFromPID(pid) END;
      IF NOT User.IsSuper(pT^.eUser) AND
          NOT User.Equal(pT^.eUser, ppT^.eUser) AND
          NotDescendent() THEN RaiseError(Unix42.EPERM) END;
      ppT^.pgrp := pgrp;
    END;
    RETURN 0;
  END SetPGRP;

PROCEDURE SetPriority(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    which: Unix42.WhichPrio;
    who: INTEGER;
    found, match: BOOLEAN;
    priority: ThreadFriends.Priority;
    pp: Ux.T;
  BEGIN
    IF ORD(LAST(Unix42.WhichPrio)) < LOOPHOLE(args[0], UNSIGNED) THEN
      RaiseError(Unix42.EINVAL);
    END;
    which := LOOPHOLE(args[0], Unix42.WhichPrio);
    who := LOOPHOLE(args[1], INTEGER);
    priority := UltrixToThreadPriority(LOOPHOLE(args[2], INTEGER));
    (* ***** EACCES if Ultrix priority < 0 *)
    found := FALSE;
    LOCK gx DO
      pp := pHead;
      WHILE (pp # NIL) AND ((which # Unix42.prioProcess) OR NOT found) DO
        IF pp^.pT^.opState # Terminated THEN
          CASE which OF
          | Unix42.prioProcess:
              match := (who = 0) AND (p = pp) OR (who = pp^.pT^.pid);
              (* ***** EACCES unless eUser or rUser matches *)
          | Unix42.prioPgrp:
              match := who = pp^.pT^.pgrp;
              (* ***** EACCES unless eUser or rUser matches *)
          | Unix42.prioUser:
              match := who = User.GetUserID(pp^.pT^.rUser); (* or eUser? *)
          END;
          IF match THEN
            UxAS.SetClientPriority(pp^.aT, priority);
            UxAS.SetThreadPriority(pp^.pT^.wThread, priority);
            found := TRUE;
          END;
        END;
        pp := pp^.pT^.next;
      END;
    END;
    IF NOT found THEN RaiseError(Unix42.ESRCH); END;
    RETURN 0;
  END SetPriority;

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

PROCEDURE SetREUID(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pT: T;
    rOld, eOld, rNew, eNew: Ux.UserID;
    u: User.T;
  BEGIN
    pT := p^.pT;
    rOld := User.GetUserID(pT^.rUser);
    eOld := User.GetUserID(pT^.eUser);
    rNew := LOOPHOLE(args[0], Ux.UserID);
    eNew := LOOPHOLE(args[1], Ux.UserID);
    LOCK gx DO
      IF rNew = -1 THEN rNew := rOld; END;
      IF eNew = -1 THEN eNew := eOld; END;
      IF (eOld # Ux.SuperUID)
        AND ((rNew # rOld) AND (rNew # eOld)
          OR (eNew # rOld) AND (eNew # eOld)) THEN RaiseError(Unix42.EPERM);
      END;
      IF rNew # rOld THEN
        u := User.LookupUserByID(rNew);
        IF u # NIL THEN
          pT^.rUser := u;
        ELSE
          RaiseError(Unix42.EINVAL);
        END;
      END;
      IF eNew # eOld THEN
        u := User.LookupUserByID(eNew);
        IF u # NIL THEN
          pT^.eUser := u;
        ELSE
          RaiseError(Unix42.EINVAL);
        END;
      END;
    END;
    RETURN 0;
  END SetREUID;

PROCEDURE SetRLimit(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    resource: Unix42.Resource;
    rLimit, oldRLimit: Unix42.RLimit;
  BEGIN
    resource := LOOPHOLE(args[0], Unix42.Resource);
    UxAS.Get(p^.aT, args[1], rLimit);
    oldRLimit := RLimit(p, resource);
    IF NOT User.IsSuper(p^.pT^.eUser)
      AND ((oldRLimit.max < rLimit.max)
        OR (rLimit.cur < 0) OR (rLimit.max < rLimit.cur)) THEN
      RaiseError(Unix42.EPERM);
    END;
    CASE resource OF
    | Unix42.rLimitData, Unix42.rLimitStack:
        UxAS.SetRLimit(p^.aT, resource, rLimit);
    | ELSE (* nothing *);
    END;
    RETURN 0;
  END SetRLimit;

PROCEDURE SigBlock(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pT: T;
    oldMask: Unix42.SigMask;
```

```
  BEGIN
    pT := p^.pT;
    LOCK gx DO
      oldMask := pT^.mask;
      pT^.mask := pT^.mask
        + (LOOPHOLE(args[0], Unix42.SigMask) - PowerfulSigs);
    END;
    RETURN LOOPHOLE(oldMask, INTEGER)
  END SigBlock;

PROCEDURE SigPause(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pT: T;
    oldMask: Unix42.SigMask;
    s: Unix42.Signal;
  BEGIN
    pT := p^.pT;
    oldMask := pT^.mask;
    pT^.mask := LOOPHOLE(args[0], Unix42.SigMask) - PowerfulSigs;

    LOCK gx DO
      WHILE (pT^.posted - pT^.mask) = NoSigs DO
        TRY Thread.AlertWait(gx, pT^.c); 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
      result (in this case EINTR) must (also) be set in the normal
      fashion, because DoKernelCall always sets R0 on return, and the
      error code will not have been saved yet. *)

      UxAS.R0Result(p^.aT, ORD(Unix42.EINTR), TRUE);

      ActOnSignals(p, Unix42.SigNone, oldMask);
    END; (* LOCK gx *)

    (* Ensure R0 contains EINTR when the client is started. *)
    RETURN ORD(Unix42.EINTR);
  END SigPause;

PROCEDURE SigSetMask(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pT: T;
    oldMask: Unix42.SigMask;
  BEGIN
    pT := p^.pT;
    LOCK gx DO
      oldMask := pT^.mask;
      pT^.mask := LOOPHOLE(args[0], Unix42.SigMask) - PowerfulSigs;
    END;
    RETURN LOOPHOLE(oldMask, INTEGER);
  END SigSetMask;

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

PROCEDURE SigVec(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* Also known as SetSignalState. *)
  VAR
    pT: T;
    sig: Unix42.Signal;
    vec: Unix42.SigVecRec;
  BEGIN
    pT := p^.pT;
    IF (args[0] < ORD(Unix42.FirstSig)) OR (args[0] > ORD(Unix42.LastSig)) THEN
      RaiseError(Unix42.EINVAL);
    END;
    sig := LOOPHOLE(args[0], Unix42.Signal);
    IF (sig IN PowerfulSigs) AND (sig # Unix42.SigCont) THEN
      RaiseError(Unix42.EINVAL); (* even if args[1] = 0 *)
    END;
    IF args[2] # 0 THEN UxAS.Put(p^.aT, args[2], pT^.sigVec[sig]) END;
    IF args[1] # 0 THEN
      UxAS.Get(p^.aT, args[1], vec);
      IF (sig = Unix42.SigCont) AND (vec.handler = SigIgn) THEN
        RaiseError(Unix42.EINVAL);
      END;
      IF vec.handler = SigAlert THEN UxAS.SetHandler(p^.aT); END;
      vec.mask := vec.mask - PowerfulSigs;
      LOCK gx DO
        pT^.sigVec[sig] := vec;
        IF vec.handler = SigIgn THEN EXCL(pT^.posted, sig); END;
      END;
    END;
    RETURN 0;
  END SigVec;

PROCEDURE StartNewSpace(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR aT: UxAS.T; name: Text.T;
  BEGIN
    aT := p^.aT;
    name := UxAS.GetText(aT, args[0]);
    RETURN DoForkExec(p^.pT^.pid, name, NIL, NIL, aT, args[1], args[2]);
  END StartNewSpace;

PROCEDURE Unimplemented(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    RaiseError(Unix42.EINVAL, Unix42.SigSys);
    ASSERT(FALSE);
    RETURN 0; (* make compiler happy *)
  END Unimplemented;

PROCEDURE VFork(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    RETURN DoFork(p, args, TRUE);
  END VFork;

PROCEDURE Wait(p: 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
    pp, ppPrev: Ux.T;
    pT, ppT: T;
    noChildren, noHang, untraced: BOOLEAN;
  PROCEDURE WaitResults(rUsageOK: BOOLEAN): INTEGER;
    VAR
      ws32: BITS 32 FOR Unix42.WStatus;
      rUsage: Unix42.RUsage;
    BEGIN
      IF rUsageOK AND (args[2] # 0) THEN
        System.Zero(System.Adr(rUsage), System.ByteSize(rUsage));
        rUsage.uTime := ppT^.descCTime;
        rUsage.sTime := ppT^.descWTime;
        UxAS.Put(p^.aT, args[2], rUsage);
```

```
      END;
      ws32 := ppT^.ws; (* 0's high order bytes *)
      UxAS.R1Result(p^.aT, LOOPHOLE(ws32, INTEGER));
      RETURN LOOPHOLE(ppT^.pid, INTEGER)
    END WaitResults;
  BEGIN
    noHang := (Unix42.WNOHANG IN LOOPHOLE(args[1], Unix42.WOptions));
    untraced := (Unix42.WUNTRACED IN LOOPHOLE(args[1], Unix42.WOptions));
    pT := p^.pT;
    LOCK gx DO
      LOOP
        pp := pHead; (* pHead can't be NIL *) (* or child ! *)
        ppPrev := NIL;
        noChildren := TRUE;
        LOOP
          ppT := pp^.pT;
          IF ppT^.ppid = pT^.pid THEN
            noChildren := FALSE;
            IF ppT^.opState = Terminated THEN
              IF pHead = pp THEN pHead := ppT^.next; (* ? *)
              ELSE ppPrev^.pT^.next := ppT^.next; (* ppPrev # NIL *) END;
              IF pTail = pp THEN pTail := ppPrev; (* ppPrev # NIL *) END;
              RETURN WaitResults(TRUE);
            END;
            IF (ppT^.opState = Stopped) AND ppT^.untraced AND untraced THEN
              ppT^.untraced := FALSE;
              RETURN WaitResults(FALSE);
            END;
          END;
          IF ppT^.next = NIL THEN EXIT END;
          ppPrev := pp;
          pp := ppT^.next;
        END;
        IF noChildren THEN
          UxAS.SetError(p^.aT); RETURN ORD(Unix42.ECHILD);
        END;
        IF noHang THEN RETURN 0; END;
        Thread.AlertWait(gx, pT^.c); (* alert handled by DoKernelCall *)
      END;
    END;
  END Wait;

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 the following Ultrix kernel calls: execve, exit, fork, getprgp, getpid, getuid, kill, killpg, setpgrp, setreuid, sigblock, sigpause, setsigmask, sigstack, sigvec, vfork and wait. 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).

```
(* Last modified on Mon Sep  8 12:11:29 1986 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 System, Time, Thread, Unix42, User, Ux, UxAS, UxPro;

CONST
  MaxErrorUSec = 250000; (* error tolerance for virtual, profile timers *)
  MaxErrorSec = 0;

(* These are temporary until we can use static storage ***** *)
  MinutesWest = 480;
  DSTFlag = 1;

TYPE
  TimerRec = RECORD interval, value: Time.T; END;
  Timer = RECORD valid: BOOLEAN; tRec: TimerRec; target: Time.T; END;
  TimerTypes = (Real, Virtual, Profile);
  CompOp = (LSS, LEQ, GTR, GEQ, EQL);
  T =
    REF RECORD
      mutex: Thread.Mutex;
      curPauseVal: Time.T;
      timer: ARRAY TimerTypes OF Timer;
      tThread: Thread.T; (* not created until first SetITimer *)
      aT: UxAS.T;
      exiting: BOOLEAN;
    END;
  TimeZone = RECORD minutesWest, dstFlag: INTEGER; END;

VAR
  (* This is temporary until we can use safe storage ***** *)
  timeZone: TimeZone;

PROCEDURE Init() RAISES {};
  (* This procedure is called only once at Ux startup. *)
  BEGIN
    UxPro.Register(Unix42.SYSgetitimer, GetITimer);
    UxPro.Register(Unix42.SYSsetitimer, SetITimer);
    UxPro.Register(Unix42.SYSgettimeofday, GetTimeOfDay);
    UxPro.Register(Unix42.SYSsettimeofday, SetTimeOfDay);

    (* This is temporary. In long term, use safe storage *)
    timeZone.minutesWest := MinutesWest;
    timeZone.dstFlag := DSTFlag;
  END Init;

PROCEDURE Exit(tT: T) RAISES {};
  (* This procedure is called when a Ux process is terminated. Its job is to
     cancel a timer thread if one exists. *)
  VAR dontCare: REFANY;
  BEGIN
    IF (tT = NIL) OR (tT^.tThread = NIL) THEN RETURN END;
    LOCK tT^.mutex DO tT^.exiting := TRUE; END;
    Thread.Alert(tT^.tThread);
    dontCare := Thread.Join(tT^.tThread);
  END Exit;

PROCEDURE CreateT(uxT: Ux.T): T;
  (* Initializes a new T*)
  VAR tT: T; iter: TimerTypes;
  BEGIN
    NEW(tT);
    FOR iter := FIRST(TimerTypes) TO LAST(TimerTypes) DO
      WITH tT^.timer[iter] DO
        tRec.interval.seconds := 0;
        tRec.interval.microseconds := 0;
        tRec.value.seconds := 0;
        tRec.value.microseconds := 0;
        valid := FALSE;
      END;
    END;
    tT^.curPauseVal.seconds := LAST(CARDINAL);
    tT^.curPauseVal.microseconds := 999999;
    tT^.exiting := FALSE;
    tT^.aT := uxT^.aT;
    Thread.InitMutex(tT^.mutex);
    RETURN tT;
  END CreateT;

PROCEDURE GetTimeOfDay(p: Ux.T; VAR args: Ux.ArgList) : INTEGER;
  VAR time: Time.T;
  BEGIN
    time := Time.Now();
    UxAS.Put(p^.aT, args[0], time);
    IF args[1] # 0 THEN
      UxAS.Put(p^.aT, args[1], timeZone);
    END;
    RETURN 0;
  END GetTimeOfDay;

PROCEDURE SetTimeOfDay(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR time: Time.T;
  BEGIN
    IF NOT User.IsSuper(UxPro.EUser(p^.pT)) THEN
      UxPro.RaiseError(Unix42.EPERM);
    ELSE
      UxAS.Get(p^.aT, args[0], time);
      UxAS.Get(p^.aT, args[1], timeZone); (* temporary ***** *)
      (* load stable storage with new timezone *)
      Time.SetNow(time.seconds, time.microseconds);
      RETURN 0;
    END;
  END SetTimeOfDay;

PROCEDURE GetUpdatedTRec(tT: T; pT: UxPro.T; t: TimerTypes): TimerRec;
  (* returns current timer record (current value) of the requested timer.
     Assumes tT^.mutex is locked. *)
  VAR tRec: TimerRec;
  BEGIN
    tRec := tT^.timer[t].tRec;
    IF tT^.timer[t].valid THEN
      CASE t OF
      | Real: tRec.value := Subtract(tT^.timer[t].target, Time.Now());
      | Virtual:
          tRec.value :=
            Subtract(tT^.timer[t].target, UxAS.GetClientTime(tT^.aT));
      | Profile:
          tRec.value :=
            Subtract(tT^.timer[t].target,
              Add(UxAS.GetClientTime(tT^.aT), UxPro.GetWTime(pT)));
      END;
    END;
    RETURN tRec;
  END GetUpdatedTRec;
```

```
PROCEDURE GetITimer(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR t: TimerTypes; tRec: TimerRec;
  BEGIN
    IF p^.tT = NIL THEN
      p^.tT := CreateT(p);
    END;
    IF args[0] > ORD(LAST(TimerTypes)) THEN
      UxPro.RaiseError(Unix42.EINVAL); (* does not return *)
    END;
    t := LOOPHOLE(args[0], TimerTypes);
    LOCK p^.tT^.mutex DO
      tRec := GetUpdatedTRec(p^.tT, p^.pT, t);
      UxAS.Put(p^.aT, args[1], tRec);
    END;
    RETURN (0);
  END GetITimer;

PROCEDURE SetITimer(p: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR aT: UxAS.T; tT: T; t: TimerTypes; timerHold: TimerRec; tRec: TimerRec;
  BEGIN
    aT := p^.aT;
    IF p^.tT = NIL THEN
      p^.tT := CreateT(p);
    END;
    tT := p^.tT;

    IF args[0] > ORD(LAST(TimerTypes)) THEN
      UxPro.RaiseError(Unix42.EINVAL);
    END;

    UxAS.Get(aT, args[1], timerHold);

    IF (LOOPHOLE(timerHold.value.seconds, UNSIGNED) > System.MaxCard)
       OR (LOOPHOLE(timerHold.interval.seconds, UNSIGNED) > System.MaxCard)
    THEN
      UxPro.RaiseError(Unix42.EINVAL);
    END;

    t := VAL(TimerTypes, args[0]);

    LOCK tT^.mutex DO

      (* optionally return old value of timer *)
      IF args[2] # 0 THEN
        tRec := GetUpdatedTRec(tT, p^.pT, t);
        UxAS.Put(aT, args[2], tRec);
      END;

      TimerSet(t, p, timerHold);

    END;
    RETURN (0);
  END SetITimer;

VAR
  zeroTime: Time.T; (* Record constant *)
  (* IF we had record constructors we wouldn't need this*)

PROCEDURE TimerSet(t: TimerTypes; uxT: Ux.T; tRec: TimerRec);
  (* Performs details of individual timer startup. Must be called with
     uxT^.tT^.mutex LOCKED! *)
  VAR tT: T;
  BEGIN
    tT := uxT^.tT;
    IF Compare(tRec.value, zeroTime) > 0 THEN
      tT^.timer[t].tRec := tRec;
      tT^.timer[t].valid := TRUE;
      (* Compute target time *)
      CASE t OF
      | Real: tT^.timer[t].target := Add(tRec.value, Time.Now());
      | Virtual:
          tT^.timer[t].target := Add(tRec.value,
            UxAS.GetClientTime(tT^.aT));
      | Profile:
          tT^.timer[t].target :=
            Add(tRec.value,
              Add(UxAS.GetClientTime(tT^.aT), UxPro.GetWTime(uxT^.pT)));
      END; (*CASE*)

      (* Now, if the interval timer thread isn't running, start it. If it is
         running, then alert it only if it would not wake up anyway before
         this timer expires. *)

      IF tT^.tThread = NIL THEN
        tT^.tThread := Thread.Fork(IntervalTimer, uxT);
      ELSIF Compare(tT^.timer[t].tRec.value, tT^.curPauseVal) < 0 THEN
        Thread.Alert(tT^.tThread);
      END;
    ELSE
      tT^.timer[t].valid := FALSE;
    END; (* IF *)
  END TimerSet;

PROCEDURE IntervalTimer(ref: REFANY): REFANY;
  (* Performs interval timing based upon the timer record passed it within the
     Ux.T. Timers can be changed at any time by calling TimerSet():

     Also note: Currently, once this thread starts, it will only terminate
     when the client process exits. If no timers are active, it goes to sleep
     for "maxTime", or until it is ALERTed.

     This can be easily changed if needed by having it sleep for a shorter
     period of time, and then if there are still no timers, terminate. *)
  VAR
    pauseStart, pauseTime: Time.T;
    doPause: BOOLEAN;
    errorTime: Time.T;
    maxTime: Time.T;
    uxT: Ux.T;
    tT: T;
    t: TimerTypes;
  BEGIN
    uxT := NARROW(ref, Ux.T);
    tT := uxT^.tT;

    maxTime.seconds := 300;
    (* only wait 5 mins, due to Time.Pause storage leak. *)
    maxTime.microseconds := 0;

    errorTime.seconds := MaxErrorSec;
    errorTime.microseconds := MaxErrorUSec;

    LOOP
      tT^.curPauseVal := maxTime; (* set minPause so any pausetime is < it *)
      doPause := FALSE; (* only do timing if timers are set *)
      LOCK tT^.mutex DO
        FOR t := FIRST(TimerTypes) TO LAST(TimerTypes) DO
          WITH tT^.timer[t] DO
            CASE t OF
            | Real:
```

```
            IF valid THEN
              tRec.value := Subtract(target, Time.Now());
              IF Compare(tRec.value, zeroTime) <= 0 THEN
                UxPro.Signal(uxT^.pT, Unix42.SigAlrm);
                tRec.value := tRec.interval;
                TimerSet(t, uxT, tRec);
              END;
            END;
          | Virtual:
            IF valid THEN
              tRec.value := Subtract(target,
                UxAS.GetClientTime(tT^.aT));
              IF Compare(tRec.value, errorTime) <= 0 THEN
                UxPro.Signal(uxT^.pT, Unix42.SigVTAlrm);
                tRec.value := tRec.interval;
                TimerSet(t, uxT, tRec);
              END;
            END;
          | Profile:
            IF valid THEN
              tRec.value :=
                Subtract(target,
                  Add(UxAS.GetClientTime(tT^.aT),
                    UxPro.GetWTime(uxT^.pT)));
              IF Compare(tRec.value, errorTime) <= 0 THEN
                UxPro.Signal(uxT^.pT, Unix42.SigProf);
                tRec.value := tRec.interval;
                TimerSet(t, uxT, tRec);
              END;
            END;
          END; (* case *)
          IF valid THEN
            doPause := TRUE;
            IF Compare(tRec.value, tT^.curPauseVal) < 0 THEN
              tT^.curPauseVal := tRec.value;
            END;
          END;
        END;
      END; (* FOR *)
    END; (* LOCK *)

    (*********************************************************************)
    (* At this point, if any timers were set, tT^.curPauseVal is set to  *)
    (* the MIN of those timers and doPause is TRUE.                      *)
    (* If no timers were set, doPause is FALSE, and curPauseVal = maxTime.*)
    (*********************************************************************)

    IF NOT doPause THEN (* pause until alerted or a *long* time *)
      TRY
        pauseTime := Add(tT^.curPauseVal, Time.Now());
        Time.PauseUntil(pauseTime.seconds, pauseTime.microseconds);
      EXCEPT
      | Thread.Alerted:
      END;
    ELSE
      pauseStart := Time.Now();
      pauseTime := Add(tT^.curPauseVal, pauseStart);
      TRY
        Time.PauseUntil(pauseTime.seconds, pauseTime.microseconds);
      EXCEPT
      | Thread.Alerted:
      END;
    END;
    LOCK tT^.mutex DO IF tT^.exiting THEN RETURN NIL; END; END;
  END; (* LOOP *)
```

```
  END IntervalTimer;

PROCEDURE Add(t1, t2: Time.T): Time.T;
  VAR t: Time.T; us: CARDINAL;
  BEGIN
    us := t1.microseconds + t2.microseconds;
    IF us >= 1000000 THEN
      t.microseconds := us - 1000000;
      t.seconds := t1.seconds + t2.seconds + 1;
    ELSE
      t.microseconds := us;
      t.seconds := t1.seconds + t2.seconds;
    END;
    RETURN t;
  END Add;

PROCEDURE Subtract(t1, t2: Time.T): Time.T;
  (* Note that this procedure NEVER returns negative time! Negative time
     becomes zero time. *)

  VAR t: Time.T;
  BEGIN
    IF Compare(t2, t1) >= 0 THEN
      t.seconds := 0;
      t.microseconds := 0;
    ELSE
      IF t1.microseconds >= t2.microseconds THEN
        t.microseconds := t1.microseconds - t2.microseconds;
        t.seconds := t1.seconds - t2.seconds;
      ELSE
        t.microseconds := 1000000 + t1.microseconds - t2.microseconds;
        t.seconds := t1.seconds - 1 - t2.seconds;
      END;
    END;
    RETURN t;
  END Subtract;

PROCEDURE Compare(t1, t2: Time.T): INTEGER;
  (* Returns some integer less than, equal to, or greater than zero as 't1' is
     less than, equal to, or greater than 't2', respectively. *)
  BEGIN
    IF t1.seconds < t2.seconds THEN
      RETURN -1;
    ELSIF t1.seconds = t2.seconds THEN
      RETURN t1.microseconds - t2.microseconds;
    ELSE (* t1.seconds > t2.seconds *)
      RETURN 1;
    END;
  END Compare;

BEGIN zeroTime.seconds := 0; zeroTime.microseconds := 0; END UxTime.
```