TDState.NewTarget

```
(* Copyright 1988 Digital Equipment Corporation.                    *)
(* Distributed only by permission.                                  *)
(* Last modified on Tue Jan 22 12:43:31 PST 1991 by mann *)
(*      modified on Fri Jan 12 16:51:33 1990 by hisgen *)
(*      modified on Tue Jul 11 16:58:49 PDT 1989 by mcjones *)
(*      modified on Fri Jun 17 13:23:53 1988 by owicki *)
(*      modified on Tue Mar 29 16:14:21 PST 1988 by wobber *)
(*      modified on Mon Jan 11 11:08:16 PST 1988 by swart *)
(*      modified on Sun Sep 17 15:45:00 1985 by mds *)

MODULE UxPro IMPLEMENTS UxPro, RemoteOSPro;

IMPORT ACL, Auth, AuthSpecial, DebuggerFriends, NS, NSUtil, NubTypes,
  OS1, OSFriends1, PortCommon, RPC, RPCLocal, RemoteOSAS, SBuf,
  TDState, TPFriends, Thread, ThreadFriends, ThreadsPort, Time, Trace,
  UIDExtras, UxAS, UxAuth, UxError, UxFile, UxSocket, UxTime, VM,
  VMFriends;

CONST
  UltrixPriorityIncrement = 10;
  MinPID = Ux.InitPID;
  MaxPID = 32767;
(* Classification of signals according to default action: *)
  NoActionSignals =
    OS1.Signals{OS1.SigUrg, OS1.SigChld, OS1.SigIO, OS1.SigWinCh};
    (* default action is ignore *)
  StopSignals =
    OS1.Signals{OS1.SigStop, OS1.SigTStp, OS1.SigTTIn, OS1.SigTTOu};
    (* default action is stop *)
  CoreImageSignals =
    OS1.Signals{OS1.SigQuit, OS1.SigIll, OS1.SigTrap, OS1.SigIOT, OS1.SigEMT,
                OS1.SigFPE, OS1.SigBus, OS1.SigSegV, OS1.SigSys};
    (* default action is terminate with core image *)
    (* For others, the default action is termination. *)

(* Signals that can't be masked; must have default handler (except SigCont,
   which can also be caught). *)
  PowerfulSignals = OS1.Signals{OS1.SigKill, OS1.SigStop, OS1.SigCont};

(* Signals that can interrupt a core dump. *)
  CoreDumpInterruptSignals = OS1.Signals{OS1.SigKill};

  NoSignals = OS1.Signals{};

TYPE
  ProcessTemplate = OS1.ProcessTemplate;
  ProcessTemplate = Ux.T;
  State           = (Nascent, Running, Stopped, Terminated);
  WatcherEvent    = (WESignal, WESetPriority, WETerminate);
  WatcherEvents   = SET OF WatcherEvent;

  T = REF TObj; (* private state for UxPro *)
  TObj = RECORD
    uNext: Ux.T; (* process with next higher pid *)

      (* identification: *)
    uParent: Ux.T;
      (* pid, pgrp, euser, ruser, password are in Ux.T *)
      (* ppid is uParent^.pid *)
    name: Text.T;

      (* state of process: *)
    state:       State;
    xTemplate:   Thread.Mutex; (* serialize when state=Nascent *)
    untraced:    BOOLEAN;      (* state=Stopped & seen by parent *)
    terminating: BOOLEAN;      (* client threads will never run again *)
    vForkChild:  BOOLEAN;      (* running in parent's address space *)
    ws: UxTypes.WStatus; (* valid if state=Stopped or state=Terminated *)
    stateChanged: Thread.Condition; (* stopped, started, or terminated *)
    priority:     OSFriends1.ProcessPriority;
    watcherThread: Thread.T;
    initWTime:    Time.T;                (* not necessarily 0 *)

      (* signal stuff: *)
    mask:      OS1.Signals; (* current signal mask for process *)
    pending:   OS1.Signals; (* signals waiting to be delivered *)
    interrupt: BOOLEAN;    (* blocking kernel call should be interrupted *)
    signaled:      Thread.Condition;    (* signal state or pending changed *)
    watcherEvents: WatcherEvents;
    illCode:       INTEGER;             (* code for last SigIll *)
    fpeCode:       INTEGER;             (* code for last SigFPE *)
    sigStack:      UxTypes.SignalStack;
    sigVec: UxTypes.SignalVector;

      (* state of descendants: *)
    childStateChanged: Thread.Condition;
      (* stopped, terminated, or no longer vforked *)
    uTemplates: Ux.T;   (* list of templates linked through uNext *)
    descCTime:  Time.T; (* descendants' client cpu time *)
    descWTime:  Time.T; (* descendants' watcher cpu time *)
  END;


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


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


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

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

PROCEDURE ActOnSignals(u: Ux.T; s: OS1.SignalOrNone; oldMask: OS1.Signals);
  (* Deliver all pending unmasked signals, starting with 's' if it is a
     candidate. Arrange that when execution finally resumes from the current
     kernel call (after any handlers), the mask is set to 'oldMask'.
     Maintaining SIGPENDING is caller's responsibility (e.g. by ensuring
     oldMask contains current mask. *)
  (* Call with gx LOCKed. *)
  VAR
    p:           T;
    signals:     OS1.Signals;
    h:           UxTypes.SignalHandler;
    code:        INTEGER;
    switchStacks: BOOLEAN;
  BEGIN
    p := u^.p;
    LOOP
      signals := (p^.pending - p^.mask);
      IF p^.vForkChild THEN EXCL(signals, OS1.SigTStp) END;
      IF signals = NoSignals THEN EXIT END;
      IF (s = OS1.SigNone) OR NOT (s IN signals) THEN
        s := FIRST(OS1.Signal);
        WHILE NOT (s IN signals) DO INC(s) END
      END;
      EXCL(p^.pending, s);
      h := p^.sigVec[s].handler;
      IF h.h = UxTypes.SignalDefault THEN
        IF (s IN NoActionSignals) OR (s = OS1.SigCont) THEN (* nothing *)
        ELSIF (s IN StopSignals) AND
              ((p^.uParent # uInit) OR (s = OS1.SigStop))
        THEN
            (* The qualification regarding orphans is not documented in
               section 2 of the manual, but was described by Dave Lennert of
               HP in comp.unix.wizards. *)
          DoStop(u, s);
          s := OS1.SigCont
        ELSE
          PostTerminate(u, s, s IN CoreImageSignals)
        END
      ELSIF h.h = UxTypes.SignalIgnore THEN (* nothing *)
      ELSE (* Ultrix-style handler *)
        IF s = OS1.SigFPE THEN
          code := p^.fpeCode
        ELSIF s = OS1.SigIll THEN
          code := p^.illCode
        ELSE
          code := 0
        END;
        switchStacks := (UxTypes.SVOnStack IN p^.sigVec[s].flags) AND
                        NOT p^.sigStack.onStack;
        IF UxAS.IsTopaz(u) THEN
          PostTerminate(u, s, TRUE)
        ELSE
          TRY
            UxAS.PushHandlerCall(
              u, s, code, p^.sigVec[s].handler, oldMask, p^.sigStack.onStack,
              switchStacks, p^.sigStack.sp)
          EXCEPT SBuf.Fault:
            PostTerminate(u, OS1.SigSegV, TRUE)
          END;
          IF switchStacks THEN p^.sigStack.onStack := TRUE END
        END;
        p^.mask := p^.mask + p^.sigVec[s].mask;
        INCL(p^.mask, s);
        oldMask := p^.mask
      END
    END; (* LOOP *)
    p^.mask := oldMask
  END ActOnSignals;

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

PROCEDURE DisposeResources(u: Ux.T; template: BOOLEAN := FALSE);
  (* Release limited resources (virtual address space, threads) and perform
     exit-time file system actions (e.g. closes) required by Ultrix
     semantics; leave behind state used by UWait and WaitForChild. *)
  VAR
    t:        OS1.ProcessTemplate;
    ownSpace: BOOLEAN;
  BEGIN
    ownSpace := NOT u^.p^.vForkChild;
    t        := u^.p^.uTemplates;
    WHILE t # NIL DO
      DisposeResources(t, TRUE);
      t := t^.p^.uNext
    END;
    UxTime.Exit(u);
    UxFile.Exit(u);
    IF NOT template THEN UxAS.Dispose(u, ownSpace) END
  END DisposeResources;

PROCEDURE DoExec(
  u:            Ux.T;
  file:         OS1.File;
  dir:          OS1.Dir;
  eAuth:        Auth.T;
  name:         Text.T;
  argv, envp:   Text.RefArray;
  uArgs:        Ux.T := NIL;
  argva, envpa: UNSIGNED := 0
  ): BOOLEAN RAISES {OS1.Error, SBuf.Fault, Thread.Alerted};
  (* If DoExec raises OS1.Error or SBuf.Fault, the address space hasn't been
     modified and the error may be returned to the user process in the normal
     fashion. If DoExec returns FALSE or raises Thread.Alerted, the address
     space may have been modified so the user process should be terminated.
     Caller must eventually call UxFile.Exec2(u). *)
  VAR
    s:  OS1.Signal;
    p:  T;
    ok: BOOLEAN;
  VAR
    rt, innerRt: Trace.TimeRec;
  BEGIN
    IF Trace.enabled THEN
```

```
      Trace.Register(u, rt);
      Trace.Register(u, innerRt);
      Trace.Enter(
        u, Trace.OpenForLoad, innerRt, name, NIL, 0, 0, 0, 0, 0, 0, file)
    ELSE
      rt.realTime := 0
    END;
    TRY
      p := u^.p;
      ok := UxAS.Load(u, file, dir, eAuth, name, argv, envp, uArgs,
                      argva, envpa, p^.vForkChild);
      LOCK gx DO
        IF p^.vForkChild THEN (* give address space back to parent *)
          p^.vForkChild := FALSE;
          Thread.Broadcast(p^.uParent^.p^.childStateChanged)
        END;
        FOR s := FIRST(OS1.Signal) TO LAST(OS1.Signal) DO
          WITH p^.sigVec[s] DO
            IF handler.h # UxTypes.SignalIgnore THEN
              handler.h := UxTypes.SignalDefault
            END;
            EXCL(flags, UxTypes.SVOnStack);
              (* Handle all signals on standard stack, but keep resume/error
                 flag. *)
            mask := NoSignals
          END;
          p^.sigStack.sp      := 0;
          p^.sigStack.onStack := FALSE
        END;
        p^.name := name
      END;
      UxFile.Execl(u);
      RETURN ok
    FINALLY
      IF Trace.enabled THEN
        Trace.Enter(u, Trace.Execve, rt, name, NIL, 0, 0)
      END
    END
  END DoExec;

PROCEDURE DoFork(u: Ux.T; vFork: BOOLEAN): INTEGER
  RAISES {OS1.Error, SBuf.Fault};
  VAR
    p, pNew: T;
    uNew:    Ux.T;
    ec:      OS1.EC;
  BEGIN
    p := u^.p;
    IF UxAS.IsTopaz(u) THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
    New(u, uNew, vFork);
    pNew := uNew^.p;
    TRY UxAS.Copy(u, uNew, vFork) EXCEPT
    | OS1.Error(ec):
      DisposeResources(uNew);
      RAISE(OS1.Error, ec)
    | SBuf.Fault:
      DisposeResources(uNew);
      RAISE(SBuf.Fault)
    END;
    LOCK gx DO
      UxAS.R0Result(uNew, u^.pid.i);
      UxAS.R1Result(uNew, 1); (* non-zero means we're in the child *)

      DoForkCommon(u, uNew, vFork);
      IF vFork THEN
        IF NOT p^.vForkChild THEN TDState.PreVFork(u^.pid.i, u^.space) END;
        UxAS.HideThread(u);
        WHILE (pNew^.state # Terminated) AND pNew^.vForkChild DO
          Thread.Wait(gx, p^.childStateChanged)
        END;
        UxAS.RestoreThread(u);
        IF NOT p^.vForkChild THEN TDState.PostVFork(u^.pid.i, u^.space) END
      END
    END; (* LOCK gx *)
    UxAS.R1Result(u, 0); (* 0 means we're in the parent *)
    RETURN uNew^.pid.i
  END DoFork;

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

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

PROCEDURE DoKill(
  uFrom, uTo: Ux.T;
  signal:     OS1.SignalOrNone;
  eAuth:  Auth.T
  ): BOOLEAN;
  (* Call with gx LOCKed. *)
  VAR
    ok: BOOLEAN;
  BEGIN
    TRY
      ok := UxAuth.ProcessAclCheck(uTo, eAuth) OR
            ((signal = OS1.SigCont) AND Ancestor(uFrom, uTo));
    EXCEPT
    | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
    END;
    IF ok THEN PostSignal(uTo, signal) END;
    RETURN ok
  END DoKill;

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

PROCEDURE DoSetPriorityPGRP(pgrp: OS1.PGRP; priority: OSFriends1.ProcessPriority
  VAR
    found: BOOLEAN;
    u:     Ux.T;
    p:     T;
  BEGIN
    found := FALSE;
    LOCK gx DO
      u := uHead;
      WHILE u # NIL DO
        p := u^.p;
        IF (p^.state # Terminated) AND (u^.pgrp.i = pgrp.i) THEN
```

```
      (* Moribund doesn't return until all RPC calls from the given space
         have returned to the server stubs. *)
  END;
  cTime := UxAS.GetClientTime(u, TRUE);
  wTime := GetWTime(u);

  IF p^.ws.wCoreDump THEN
    TRY
        (* Since the client thread(s) will never run again, and will be
           destroyed soon, we must not deliver any alerts to it, with the
           exception of a SIGKILL. *)
      LOCK gx DO
        p^.mask := OS1.AllSignals - CoreDumpInterruptSignals;
        IF Thread.TestAlert() THEN END (* drain pending alert *)
      END;
      UxAS.MakeCoreImage(u, p^.ws.wTermSig)
    EXCEPT OS1.Error, Thread.Alerted:
      p^.ws.wCoreDump := FALSE
    END
  END;

  LOCK gx DO
    p^.state := Terminated;

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

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

      (* Our children become orphans. *)
    uu := uHead;
    WHILE uu # NIL DO
      pp := uu^.p;
      IF pp^.uParent = u THEN (* one of our children *)
        pp^.uParent := uInit;
        IF pp^.state = Stopped THEN
            (* Although it isn't documented in section 2 of the manual,
               it appears newly orphaned stopped processes are given a
               chance to "clean up their act": *)
          PostSignal(uu, OS1.SigHUp);
          PostSignal(uu, OS1.SigCont)
        ELSIF pp^.state = Terminated THEN
            (* The child is an orphan zombie; simulate Init. *)
          Remove(uu)
        END
      END;
      uu := pp^.uNext
    END; (* WHILE *)

      (* Update our parent's descendant-rusage. *)
    uu             := p^.uParent;
    pp             := uu^.p;
    pp^.descCTime := Time.Add(pp^.descCTime, p^.descCTime);
    pp^.descWTime := Time.Add(pp^.descWTime, p^.descWTime);

      (* Inform our parent of our termination. *)
    IF uu = uInit THEN
        (* We are an orphan zombie; simulate Init. *)
      Remove(u)
    ELSE
      Thread.Broadcast(p^.stateChanged);
      PostSignal(uu, OS1.SigChld);
      Thread.Signal(pp^.childStateChanged)
    END
  END; (* LOCK gx *)

    (* Now that the address space is marked as terminated, no operations
       executed by other address spaces (e.g. UGetPriority,
       GetProcessInfo) will try to look at it, so it is safe to delete
       the threads and space. *)
  IF NOT p^.vForkChild THEN DisposeResources(u) END
  FINALLY
    IF Trace.enabled THEN
      Trace.Enter(u, Trace.DoTermination, rt, NIL, NIL, 0, 0)
    END
  END
END DoTermination;

PROCEDURE GetSignalState(
  h:    UxTypes.SignalHandler;
  mask: BOOLEAN
  ): OS1.SignalState;
  (* Convert internal to external representation. *)
  BEGIN
    IF h.h = UxTypes.SignalIgnore THEN
      RETURN OS1.SignalIgnore
    ELSIF (h.h = UxTypes.SignalDefault) AND NOT mask THEN
      RETURN OS1.SignalDefault
    ELSE
      RETURN OS1.SignalHandle
    END
  END GetSignalState;

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

```
        uu     := pp^.uNext;
        IF uu = NIL THEN uu := uHead END;
        pp := uu^.p
      END;
      IF pidLast < uu^.pid.i THEN (* ppPrev -> uNew -> uu *)
        ppPrev^.uNext  := uNew;
        uNew^.p^.uNext := uu;
        EXIT
      END;
        (* pidLast = uu^.pid.i, so go around again *)
      uStart := uu
    END (* IF uTail^.pid.i < pidLast *)
  END; (* LOOP *)
  uNew^.pid.i := pidLast
END Insert;

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

PROCEDURE New(
      u:     Ux.T;
  VAR uNew:  Ux.T;
      vFork: BOOLEAN := FALSE;
      topaz: BOOLEAN := FALSE
  );
  (* Set uNew to new Ux.T based on parent u (possibly NIL). If topaz=FALSE,
     the new system state will have the same signal vector and signal mask as
     its parent and will include copies of the file descriptors from the file
     system state of u (which should be non-NIL). If topaz=TRUE, the new
     state will have no signals masked and all vector entries defaulted
     except it will ignore those its parent igonores; the file system state
     will have no open descriptors. *)
  VAR
    p: T;          (* parent's process management state *)
    s: OS1.Signal;
  BEGIN
    NEW(uNew);
    UIDExtras.Get(uNew^.uid);
    NEW(uNew^.p);
    WITH uNew^.p^ DO (* private state of new process *)
      uParent := u;
      state   := Nascent;
      Thread.InitMutex(xTemplate);
      untraced    := FALSE;
      terminating := FALSE;
      vForkChild  := vFork;
      Thread.InitCondition(stateChanged);
      initWTime.seconds      := LAST(Time.Seconds); (* bug catcher *)
      initWTime.microseconds := LAST(Time.Microseconds); (* bug catcher *)
      mask      := NoSignals;
      pending   := NoSignals;
      interrupt := FALSE;
      Thread.InitCondition(signaled);
      watcherEvents    := WatcherEvents{};
      sigStack.sp      := 0;
      sigStack.onStack := FALSE;
      FOR s := FIRST(OS1.Signal) TO LAST(OS1.Signal) DO
        WITH sigVec[s] DO
          handler.h := UxTypes.SignalDefault;
          mask      := NoSignals;
          flags     := UxTypes.SVFlags{}
        END
      END;

      Thread.InitCondition(childStateChanged);
      uTemplates             := NIL;
      descCTime.seconds      := 0;
      descCTime.microseconds := 0;
      descWTime              := descCTime;

      IF u = NIL THEN (* first process *)
        WITH uNew^ DO
          defaultAuth := NIL; (*temporary; reset later by Init2*)
          extraAuths := NIL;  (*temporary; reset later by Init2*)
          compatibilityMode := UxTypes.BSD
        END;
        priority := ThreadFriends.NormalPriority
      ELSE (* subsequent process *)
        p := u^.p;
        LOCK gx DO (* copy relevant portions of parent's state *)
          uNew^.pgrp              := u^.pgrp;
          uNew^.defaultAuth       := u^.defaultAuth;
          uNew^.extraAuths        := u^.extraAuths;
          uNew^.acl               := u^.acl;
          uNew^.compatibilityMode := u^.compatibilityMode;
          name      := p^.name;
          priority  := ThreadFriends.GetPriority();
          IF NOT topaz THEN
            mask    := p^.mask;
            sigStack := p^.sigStack;
            sigVec   := p^.sigVec
          END
        END
      END
    END; (* WITH uNew^.p^ *)
    UxAS.New(u, uNew); (* sets uNew^.a, uNew^.space *)
    uNew^.f := UxFile.Fork(u, (*desc:*) NOT topaz);
    uNew^.i := NIL
  END New;

PROCEDURE PostSetPriority(u: Ux.T; priority: OSFriends1.ProcessPriority);
  (* Only the watcher thread for a given address space can do the SetStates
     to change the client thread priorities, so we do something almost like
     sending a signal. *)
  (* Call with gx LOCKed. *)
  VAR
    p: T;
  BEGIN
    p           := u^.p;
    p^.priority := priority;
    PostWatcherEvent(p, WESetPriority)
  END PostSetPriority;

PROCEDURE PostSignal(u: Ux.T; s: OS1.SignalOrNone);
  (* Call with gx LOCKed. *)
  VAR
    h: UxTypes.SignalHandler;
    p: T;
  BEGIN
    p := u^.p;
    ASSERT(p^.state # Terminated);
    IF s = OS1.SigNone THEN RETURN END;
    h := p^.sigVec[s].handler;
```

```
      (* Don't deliver a signal that the recipient would ignore. *)
    IF (h.h = UxTypes.SignalIgnore) OR
       (h.h = UxTypes.SignalDefault) AND NOT (s IN p^.mask) AND
       ((s IN NoActionSignals) OR (s = OS1.SigCont) AND (p^.state # Stopped))
    THEN
      RETURN
    END;

    INCL(p^.pending, s);
    IF s IN p^.mask THEN (* maybe an OS1.WaitForSignal client *)
      Thread.Broadcast(p^.signaled)
    ELSIF ((s = OS1.SigTStp) AND p^.vForkChild) THEN
        (* Act as if signal is masked. *)
    ELSE
        (* Recipient must take action now (stop, continue, terminate, or push
           handler). *)
      IF UxTypes.SVInterrupt IN p^.sigVec[s].flags THEN
        p^.interrupt := TRUE
      END;
      PostWatcherEvent(p, WESignal); (* restore SIGPENDING *)
    END
  END PostSignal;

PROCEDURE PostTerminate(u: Ux.T; s: OS1.SignalOrNone; dump: BOOLEAN);
  (* Call with gx LOCKed. *)
  VAR
    p: T;
  BEGIN
    p := u^.p;
    u^.p^.ws.wTermSig  := s;
    u^.p^.ws.wCoreDump := dump;
    PostWatcherEvent(p, WETerminate)
  END PostTerminate;

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

PROCEDURE ProcessFromPID(
  pid:          OS1.PID;
  esrch:        BOOLEAN := TRUE;
  terminatedOK: BOOLEAN := FALSE
  ): Ux.T;
  (* Call with gx LOCKed. *)
  VAR
    uu: Ux.T;
  BEGIN
    uu := uHead;
    WHILE (uu # NIL) AND (uu^.pid.i < pid.i) DO uu := uu^.p^.uNext END;
    IF (uu = NIL) OR (uu^.pid.i > pid.i) OR
       ((uu^.p^.state = Terminated) AND NOT terminatedOK)
    THEN
      IF esrch THEN RAISE(OS1.Error, OS1.BadPIDEC) END;
      ASSERT(FALSE); (* pid must exist! *)
    END;
    RETURN uu
  END ProcessFromPID;

PROCEDURE Remove(u: Ux.T);
```

```
  (* Remove the specified process from the list of processes. *)
  (* Call with gx LOCKed. *)
  VAR
    uu, uuPrev: Ux.T;
  BEGIN
    IF uRover = u THEN uRover := NIL END; (* maintain ROVER *)
    u^.p^.uParent := NIL; (* avoid circular link *)
    uu            := uHead;
    uuPrev        := NIL;
    WHILE uu # u DO
      uuPrev := uu;
      uu     := uu^.p^.uNext
    END;
    IF uu = uHead THEN
      uHead := uu^.p^.uNext
    ELSE
      uuPrev^.p^.uNext := uu^.p^.uNext
    END;
    IF uu = uTail THEN uTail := uuPrev END
  END Remove;

PROCEDURE RemoveTemplate(u: Ux.T; t: OS1.ProcessTemplate);
  (* Remove 't' from the list of as-yet unstarted templates associated with
     'u'. *)
  (* Call with gx LOCKed. *)
  VAR
    tt: OS1.ProcessTemplate;
  BEGIN
    tt := u^.p^.uTemplates;
    IF tt = t THEN
      u^.p^.uTemplates := t^.p^.uNext
    ELSE
      WHILE tt^.p^.uNext # t DO tt := tt^.p^.uNext END;
      tt^.p^.uNext := t^.p^.uNext
    END
  END RemoveTemplate;

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

PROCEDURE SetRLimit(
  u:        Ux.T;
  resource: UxTypes.Resource;
  rLimit:   UxTypes.RLimit;
  auth:     Auth.T
  );
  VAR
    oldRLimit: UxTypes.RLimit;
  BEGIN
    oldRLimit := GetRLimit(u, resource);
    TRY
      IF ((oldRLimit.max < rLimit.max) OR (rLimit.cur < 0) OR
```

```
          (rLimit.max < rLimit.cur)) AND
        NOT UxAuth.SystemAclCheck(auth)
      THEN
        RAISE(OS1.Error, OS1.NotSuperUserEC)
      END;
    EXCEPT
    | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
    END;
    CASE resource OF
    | UxTypes.rLimitData, UxTypes.rLimitStack, UxTypes.rLimitCore:
      UxAS.SetRLimit(u, resource, rLimit)
    ELSE (* nothing *)
    END
  END SetRLimit;

PROCEDURE SigTT(
  u:      Ux.T;
  ec:     OS1.EC;
  signal: OS1.Signal;
  entry:  UxTypes.KernelEntry
  );
  (* Convert NotTtyOwner{Read,Write}EC into appropriate synchronous signal.
     *)
  BEGIN
    IF NOT (UxTypes.SVInterrupt IN u^.p^.sigVec[signal].flags) THEN
        (* Retry kernel call if signal handler returns. *)
      UxAS.BackupKernelCall(u, entry)
    ELSE
        (* Kernel call returns with EINTR, not signal. *)
      UxAS.R0Result(u, ORD(UxTypes.EINTR), TRUE)
    END;
    LOCK gx DO DoSignalPG(u^.pgrp, signal) END
  END SigTT;

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

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

PROCEDURE Watcher(a: Thread.ForkeeArg): Thread.ForkeeReturn;
  (* The watcher thread for a process executes this procedure until the
     process is terminated. *)
  VAR
    u:         Ux.T;
    p:         T;
    s:         OS1.SignalOrNone;
    event:     UxAS.Event;
    time:      Time.T;
    ec:        OS1.EC;
    pe:        PortCommon.ErrorCode;
    e:         UxTypes.SysError;
    interrupt: BOOLEAN;

  PROCEDURE WatcherAlerted();
    BEGIN
      (* Was the signal that alerted us resumable or interrupting? *)
      LOCK gx DO
        interrupt    := p^.interrupt;
        p^.interrupt := FALSE
      END;
      IF interrupt THEN
        UxAS.R0Result(u, ORD(UxTypes.EINTR), TRUE)
      ELSE
        UxAS.BackupKernelCall(u, event.entry)
      END
    END WatcherAlerted;

  BEGIN (*Watcher*)
    u                := NARROW(a, Ux.T);
    time.seconds     := 0;
    time.microseconds := 0;
    LOCK gx DO (* wait for parent to complete u^ *) p := u^.p END;

    LOOP (* main watcher loop *)

      s := OS1.SigNone;
      TRY
          (* Run client until it traps or we are alerted. *)
        UxAS.WaitEvent(u, event)
      EXCEPT SBuf.Fault:
        event.kind := UxAS.Alert;
        LOCK gx DO PostTerminate(u, OS1.SigSegV, (*dump:*) TRUE) END
      END;

      CASE event.kind OF
      | UxAS.KernelCall:
        IF timingEnabled THEN
          time := ThreadFriends.GetCPUTime(p^.watcherThread)
        END;
        IF traceKernelCalls THEN DebuggerFriends.Pause("Kernel call") END;
        TRY UxAS.R0Result(u, kep[event.entry](u, event.argList)) EXCEPT
        | OS1.Error(ec):
          IF ec = OS1.NotTtyOwnerReadEC THEN
            s := OS1.SigTTIn;
            SigTT(u, ec, s, event.entry)
          ELSIF ec = OS1.NotTtyOwnerWriteEC THEN
            s := OS1.SigTTOu;
            SigTT(u, ec, s, event.entry)
          ELSIF ec = OS1.PipeHasNoReaderEC THEN
            s := OS1.SigPipe;
            Signal(u, s)
          ELSIF ec = OS1.NotImplementedEC THEN
            s := OS1.SigSys;
            Signal(u, s)
          ELSIF (ec = OS1.WouldBlockEC) AND
                ((u^.compatibilityMode = UxTypes.POSIX) OR
                 (u^.compatibilityMode = UxTypes.SYSV))
          THEN
            UxAS.R0Result(u, ORD(UxTypes.EAGAIN), TRUE)

(****************** Experimental code; disabled for now -- tpm **********
```

```
        ELSIF ec = OS1.LostUpdatesEC THEN
          s := OS1.Sig29; (*SigLost; see sigvec(2)*)
          LOCK gx DO DoSignalPG(u^.p^.pgrp, s); END;
        ELSIF (ec = OS1.ServerNotAvailableEC) AND Thread.TestAlert() THEN
          (* A normally non-alertable system call became alertable
             because it was waiting inside the Echo clerk for a server
             to come back to life.  The clerk reports this by raising
             ServerNotAvailableEC with the alert still pending, because
             Alerted is not in all the raises clauses between here and
             the place where the call was waiting.
          *)
          WatcherAlerted();
*************************************************************************)

        ELSE
          UxAS.R0Result(u, ORD(UxError.errTran[ec]), TRUE)
        END
      | SBuf.Fault:
        UxAS.R0Result(u, ORD(UxTypes.EFAULT), TRUE)
      | PortCommon.Error(pe):
        UxAS.R0Result(u, ORD(UxSocket.convert[pe]), TRUE)
      | Error(e):
        UxAS.R0Result(u, ORD(e), TRUE)
      | Thread.Alerted:
        WatcherAlerted();
      END;
      INC(kec[event.entry]);
      IF timingEnabled THEN
        ket[event.entry] :=
          Time.Add(ket[event.entry],
                   Time.Subtract(
                     ThreadFriends.GetCPUTime(p^.watcherThread), time))
      END
    | UxAS.Exception:
      s := event.sig; (* first signal to look for *)
      LOCK gx DO
        PostSignal(u, s);
        IF s = OS1.SigFPE THEN
          p^.fpeCode := event.code
        ELSIF s = OS1.SigIll THEN
          p^.illCode := event.code
        END
      END
    | UxAS.Alert:
    | UxAS.HandlerReturn:
      LOCK gx DO
        p^.mask             := event.mask - PowerfulSignals;
        p^.sigStack.onStack := event.onStack;
        IF (p^.pending - p^.mask) # NoSignals THEN
            (* restore SIGPENDING *)
          INCL(p^.watcherEvents, WESignal)
        END
      END
    END; (* CASE *)

    IF p^.watcherEvents # WatcherEvents{} THEN
      LOCK gx DO
        p^.interrupt := FALSE;
        REPEAT
          IF WETerminate IN p^.watcherEvents THEN
            EXIT (* only way out of main watcher loop *)
          END;
          IF WESignal IN p^.watcherEvents THEN
            ActOnSignals(u, s, p^.mask);
            EXCL(p^.watcherEvents, WESignal)
          END;
          IF WESetPriority IN p^.watcherEvents THEN
            UxAS.SetClientPriority(u, p^.priority);
            EXCL(p^.watcherEvents, WESetPriority)
          END;
          IF 0 = ThreadFriends.SetPriority(p^.priority) THEN END;
          IF Thread.TestAlert() THEN END (* drain pending alert *)
        UNTIL p^.watcherEvents = WatcherEvents{}
      END (* LOCK *)
    END (* IF p^.watcherEvents # WatcherEvents {} *)

  END; (* LOOP *)
  DoTermination(u);
  RETURN NIL
END Watcher;


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

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

PROCEDURE AcquireProcessTemplate(t: OS1.ProcessTemplate) RAISES {OS1.Error};
  BEGIN
    ThreadsPort.Acquire(t^.p^.xTemplate);
    IF (t = NIL) OR (t^.p^.state # Nascent) THEN
      ThreadsPort.Release(t^.p^.xTemplate);
      RAISE(OS1.Error, OS1.InvalidArgumentEC)
    END
  END AcquireProcessTemplate;

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

PROCEDURE GetParentPID(u: Ux.T): INTEGER;
  BEGIN
    LOCK gx DO RETURN u^.p^.uParent^.pid.i; END;
  END GetParentPID;

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

PROCEDURE HandleFromPID(pid: OS1.PID): Ux.T RAISES {OS1.Error};
  BEGIN
    LOCK gx DO RETURN ProcessFromPID(pid) END
  END HandleFromPID;

PROCEDURE Import() RAISES {};
  BEGIN
  END Import;
```

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

      (* uInit *must* be created first, so it gets a PID of 1 *)
    ForkSystemProcess(NIL, uInit, NubTypes.NullSpace, "Init");
    ForkSystemProcess(uInit, uNub, NubTypes.NubSpace, "Nub");
      (* Note uTaos is not an orphan--its parent is not uInit: *)
    ForkSystemProcess(uNub, Ux.uTaos, TPFriends.GetSpace(), "Taos");
    ASSERT(Ux.uTaos^.pid.i = TDState.TaosPID);
    Ux.uTaos^.p^.mask := OS1.AllSignals; (* i.e. all handled *)

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

    kep[SYSexecve]                := UExecVE;
    kep[SYSexit]                  := UExit;
    kep[SYSfork]                  := UFork;
    kep[SYSgetgid]                := UGetGID;
    kep[SYSgethostid]             := UGetHostID;
    kep[SYSgethostname]           := UGetHostName;
    kep[SYSgetpgrp]               := UGetPGRP;
    kep[SYSgetpid]                := UGetPID;
    kep[SYSgetpriority]           := UGetPriority;
    kep[SYSgetsysinfo]            := UGetSysInfo;
    kep[UxTypes.SYSgetpw]         := UGetPW; (* Taos only *)
    kep[SYSgetrlimit]             := UGetRLimit;
    kep[SYSgetrusage]             := UGetRUsage;
    kep[UxTypes.SYSgetspaceinfo] := UGetProcessInfo; (* Taos only *)
    kep[SYSgetuid]                := UGetUID;
    kep[SYSkill]                 := UKill;
    kep[SYSkillpg]               := UKillPG;
    kep[SYSquota]                := UQuota;
```

```
    kep[SYSsetgroups]          := USetGroups;
    kep[SYSsethostid]          := USetHostID;
    kep[SYSsethostname]        := USetHostName;
    kep[SYSsetpgrp]            := USetPGRP;
    kep[SYSsetpriority]        := USetPriority;
    kep[SYSsetregid]           := USetREGID;
    kep[SYSsetreuid]           := USetREUID;
    kep[SYSsetrlimit]          := USetRLimit;
    kep[SYSsigblock]           := USigBlock;
    kep[UxTypes.SYSspare115]   := USigGet; (* Taos only *)
    kep[SYSsigpause]           := USigPause;
    kep[SYSsigpending]         := USigPending;
    kep[SYSsigsetmask]         := USigSetMask;
    kep[SYSsigstack]           := USigStack;
    kep[SYSsigvec]             := USigVec;
    kep[UxTypes.SYSstrace]     := UTrace; (* Taos only *)
    kep[UxTypes.SYSterminate] := UTerminate; (* Taos only *)
    kep[SYSvfork]              := UVFork;
    kep[SYSwait]               := UWait;
    kep[SYSwaitpid]            := UWaitPid;
    traceKernelCalls           := FALSE;
    FOR op := FIRST(UxTypes.GSIOp) TO LAST(UxTypes.GSIOp) DO
      gsiProc[op] := NIL
    END;
    RegisterGSIProc(UxTypes.GSIProgEnv, GSIProgEnvProc);
    RegisterGSIProc(UxTypes.GSIMaxUProcs, GSIMaxUProcsProc)
  END Init;

PROCEDURE Init2() RAISES {};
  VAR auth: Auth.T;
  BEGIN
    auth := AuthSpecial.Root();
    SetMyAuths(uInit, auth);
    SetMyAuths(uNub, auth);
    SetMyAuths(Ux.uTaos, auth);
  END Init2;

PROCEDURE IsVForkChild(u: Ux.T): BOOLEAN RAISES {};
  BEGIN
    RETURN u^.p^.vForkChild
  END IsVForkChild;

PROCEDURE PIDsPGRP(pid: OS1.PID): OS1.PID RAISES {OS1.Error};
  VAR
    u: Ux.T;
  BEGIN
    LOCK gx DO
      u := ProcessFromPID(pid);
      RETURN u^.pgrp
    END
  END PIDsPGRP;

PROCEDURE PipeHasNoReader(u: Ux.T) RAISES {OS1.Error, Thread.Alerted};
  BEGIN
    LOCK gx DO
      IF u^.p^.sigVec[OS1.SigPipe].handler.h = UxTypes.SignalIgnore THEN
        RAISE(OS1.Error, OS1.PipeHasNoReaderEC)
      ELSE
        PostSignal(u, OS1.SigPipe);
        WHILE NOT u^.p^.terminating DO Thread.AlertWait(gx, terminating) END
      END
    END
  END PipeHasNoReader;

PROCEDURE RaiseError(e: UxTypes.SysError) RAISES {Error};
```

```
  BEGIN
    ASSERT(e # UxTypes.ok); (* unimpl. in fs *)
    RAISE(Error, e)
  END RaiseError;

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

PROCEDURE RegisterGSIProc(op: UxTypes.GSIOp; proc: GSIProc);
  BEGIN
    ASSERT(gsiProc[op] = NIL);
    gsiProc[op] := proc
  END RegisterGSIProc;

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

PROCEDURE ReleaseProcessTemplate(t: OS1.ProcessTemplate) RAISES {};
  BEGIN
    ThreadsPort.Release(t^.p^.xTemplate)
  END ReleaseProcessTemplate;

PROCEDURE SetDefaultAuth(u: Ux.T; defaultAuth: Auth.T) RAISES {};
  BEGIN
    LOCK gx DO u^.defaultAuth := defaultAuth END;
  END SetDefaultAuth;

PROCEDURE SendTTSignal(u: Ux.T; signal: OS1.Signal): TTResult
  RAISES {OS1.Error};
  VAR
    ec: OS1.EC;
  BEGIN
    IF signal = OS1.SigTTIn THEN
      ec := OS1.NotTtyOwnerReadEC
    ELSE
      ASSERT(signal = OS1.SigTTOu, "UxPro.SendTTSignal given wrong signal");
      ec := OS1.NotTtyOwnerWriteEC
    END;
    LOCK gx DO
      IF u^.p^.watcherThread = Thread.Self() THEN
        IF DoSigEffective(u, signal) THEN
          RAISE(OS1.Error, ec)
        ELSE (* not effective *)
          RETURN TTNotEffective
        END (* IF effective *)
      END; (* IF watcher *)
      IF u^.p^.state # Stopped THEN
        IF DoSigEffective(u, signal) THEN
          DoSignalPG(u^.pgrp, signal);
          RETURN TTSentSignal
        ELSE
          RAISE(OS1.Error, ec)
        END (* IF effective *)
      ELSE (* not running *)
        RETURN TTNotRunning
      END (* IF running *)
    END (* LOCK gx *)
  END SendTTSignal;

PROCEDURE Signal(u: Ux.T; signal: OS1.Signal) RAISES {};
  BEGIN
    LOCK gx DO IF u^.p^.state # Terminated THEN PostSignal(u, signal) END END
  END Signal;


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

PROCEDURE UExecVE(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    name: Text.T;
    file: OS1.File;
    ok:   BOOLEAN;
  BEGIN
    IF UxAS.IsTopaz(u) THEN RaiseError(UxTypes.EINVAL) END;
    name := UxAS.GetText(u, args[0]);
    file := UxFile.OpenForLoading(u, NIL, name);
    TRY
      ok := DoExec(u, file, NIL, NIL, name, NIL, NIL, u, args[1], args[2]);
      UxFile.Exec2(u)
    EXCEPT Thread.Alerted:
      ok := FALSE
    END;
    IF NOT ok THEN LOCK gx DO PostTerminate(u, OS1.SigKill, FALSE) END END;
    RETURN 0
  END UExecVE;

PROCEDURE UExit(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    rt: Trace.TimeRec;
  BEGIN
    IF Trace.enabled THEN Trace.Register(u, rt) ELSE rt.realTime := 0 END;
    TRY
      LOCK gx DO
        u^.p^.ws.wRetCode := args[0] MOD 256;
        PostTerminate(u, OS1.SigNone, FALSE)
      END;
      RETURN 0
    FINALLY
      IF Trace.enabled THEN
        Trace.Enter(u, Trace.Exit, rt, NIL, NIL, 0, 0)
      END
    END
  END UExit;

PROCEDURE UFork(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    i: INTEGER;
  VAR
    rt: Trace.TimeRec;
  BEGIN
    IF Trace.enabled THEN
      Trace.Register(u, rt);
      i := 0
    ELSE
      rt.realTime := 0
    END;
    TRY
      i := DoFork(u, (*vfork:*) FALSE);
      RETURN i
    FINALLY
      IF Trace.enabled THEN
        Trace.Enter(u, Trace.Fork, rt, NIL, NIL, i, 0)
      END
```

```
    END
  END UFork;

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

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

PROCEDURE UGetHostName(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    name:  Text.T;
    count: CARDINAL;
    zero:  CHAR;
  BEGIN
    TRY name := NSUtil.GetMyHost() EXCEPT
    | RPC.CallFailed:
      RaiseError(UxTypes.ETIMEDOUT)
    | NS.LabelNotFound:
      RaiseError(UxTypes.ENOENT)
    END;
    LOCK gx DO
      count := UxAS.PutText(u, args[0], name, LOOPHOLE(args[1], CARDINAL));
      IF count < LOOPHOLE(args[1], CARDINAL) THEN
        zero := '\000';
        UxAS.Put(u, args[0] + count, zero)
      END
    END;
    RETURN 0
  END UGetHostName;

PROCEDURE UGetPGRP(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pid: OS1.PID;
    uu:  Ux.T;
  BEGIN
    pid := LOOPHOLE(args[0], OS1.PID);
    LOCK gx DO
      IF pid.i = 0 THEN uu := u ELSE uu := ProcessFromPID(pid) END;
      RETURN uu^.pgrp.i
    END
  END UGetPGRP;

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

PROCEDURE UGetPriority(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    which:                UxTypes.WhichPrio;
    who, ultrixPriority: INTEGER;
    found, match:         BOOLEAN;
    uu: Ux.T;
  BEGIN
    IF ORD(LAST(UxTypes.WhichPrio)) < args[0] THEN
      RaiseError(UxTypes.EINVAL)
    END;
```

```
    which          := LOOPHOLE(args[0], UxTypes.WhichPrio);
    who            := LOOPHOLE(args[1], INTEGER);
    ultrixPriority := LAST(INTEGER); (* lowest Ultrix priority *)
    found          := FALSE;
    LOCK gx DO
      uu := uHead;
      LOOP
        IF uu = NIL THEN EXIT END;
        IF uu^.p^.state # Terminated THEN
          CASE which OF
          | UxTypes.prioProcess:
            match := (who = 0) AND (uu = u) OR (who = uu^.pid.i)
          | UxTypes.prioPgrp:
            match := who = uu^.pgrp.i
          | UxTypes.prioUser:
            RaiseError(UxTypes.EINVAL); (*!!?*)
(*****!!?
*            match := who = User.GetUserID(uu^.ruser)
*******)
          END;
          IF match THEN
            ultrixPriority :=
              MIN(ultrixPriority,
                  ThreadToUltrixPriority(UxAS.GetClientPriority(uu)));
            found := TRUE;
            IF which = UxTypes.prioProcess THEN EXIT END
          END
        END;
        uu := uu^.p^.uNext
      END
    END;
    IF NOT found THEN RaiseError(UxTypes.ESRCH) END;
    RETURN ultrixPriority
  END UGetPriority;

PROCEDURE UGetPW(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* Taos only *)
  BEGIN
    IF LOOPHOLE(args[1], INTEGER) < 0 THEN RaiseError(UxTypes.EINVAL) END;
    (*!!interim kludge:*)
    LOCK gx DO
      TRY
        RETURN UxAS.PutText(u, args[0],
                  UxAuth.ExtractPassword(u^.extraAuths^[0]), args[1]);
      EXCEPT
      | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
      END;
    END;
  END UGetPW;

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

PROCEDURE UGetRUsage(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    p:      T;
    rUsage: UxTypes.RUsage;
  BEGIN
    p := u^.p;
```

```
    ZERO(ADDR(rUsage), BYTESIZE(rUsage));
      (* args[0] tells 'who': self or children. *)
    IF LOOPHOLE(args[0], INTEGER) = UxTypes.rUsageSelf THEN
      rUsage.uTime := UxAS.GetClientTime(u);
      rUsage.sTime := GetWTime(u)
    ELSIF LOOPHOLE(args[0], INTEGER) = UxTypes.rUsageChildren THEN
      rUsage.uTime := p^.descCTime;
      rUsage.sTime := p^.descWTime
    ELSE
      RaiseError(UxTypes.EINVAL)
    END;
    UxAS.Put(u, args[1], rUsage, 0, UxTypes.U11RUsageLen);
    RETURN 0
  END UGetRUsage;

PROCEDURE UGetProcessInfo(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* Taos only *)
  VAR
    pid:               OS1.PID;
    addr:              UNSIGNED;
    nElts, nEltsSoFar: INTEGER;
    info:              UxTypes.TrapSpaceInfo;
    uu: Ux.T;
    pp: T;
    j:  CARDINAL;
  BEGIN
    pid.i := args[0];
    addr  := args[1];
    nElts := args[2];
    IF nElts < 0 THEN RaiseError(UxTypes.EINVAL) END;
    nEltsSoFar := 0;
    LOCK gx DO
      uu := uHead;
      LOOP (* report next process *)
        IF nEltsSoFar = nElts THEN RETURN nEltsSoFar END;
        LOOP (* find next unreported process *)
          IF uu = NIL THEN RETURN nEltsSoFar END;
          pp := uu^.p;
          IF uu^.pid.i > pid.i THEN EXIT END;
          uu := pp^.uNext
        END;
        WITH info DO
          pid := uu^.pid.i;
          IF pp^.uParent # NIL THEN
            ppid := pp^.uParent^.pid.i
          ELSE
            ppid := 0
          END;
          pgrp   := uu^.pgrp.i;
          TRY
            uid := UxAuth.AuthToUserID(uu^.extraAuths^[0]);
          EXCEPT
          | Auth.Error: uid := UxAuth.UnknownUserID;
          END;
          space  := uu^.space;
          ctlTty := "  ";
          IF Text.GetSub(UxFile.GetControlTtyName(u), 0, ctlTty) = 0 THEN END;
          CASE pp^.state OF
          | Running:
            runState := UxTypes.SSRunning
          | Stopped:
            runState := UxTypes.SSStopped
          | Terminated:
            runState := UxTypes.SSTerminating
          END;
          cTime.seconds      := 0;
          cTime.microseconds := 0;
          wTime              := cTime;
          topaz              := FALSE;
          IF runState # UxTypes.SSTerminating THEN
            IF uu^.space # NubTypes.NullSpace THEN
              topaz := UxAS.IsTopaz(uu)
            END;
            IF (uu # uNub) AND (uu # uInit) AND (uu # Ux.uTaos) THEN
              cTime := UxAS.GetClientTime(uu)
            END;
            IF pp^.watcherThread # NIL THEN wTime := GetWTime(uu) END
          END;
          ws := pp^.ws;
          FOR j := 0 TO HIGH(commandString) DO commandString[j] := ' ' END;
          IF Text.GetSub(pp^.name, 0, commandString) = 0 THEN END
        END; (* WITH info *)
        UxAS.Put(u, addr, info);
        INC(addr, BYTESIZE(info));
        INC(nEltsSoFar, 1);
        pid.i := info.pid
      END (* report next process loop *)
    END (* LOCK gx *)
    (* Can't get here. *)
  END UGetProcessInfo;

PROCEDURE UGetSysInfo(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    op:     UxTypes.GSIOp;
    proc:   GSIProc;
    result: SBuf.T;
    start:  INTEGER;
    nItems: CARDINAL;
  BEGIN
    IF (args[0] < ORD(FIRST(UxTypes.GSIOp))) OR
       (ORD(LAST(UxTypes.GSIOp)) < args[0])
    THEN
      RaiseError(UxTypes.EINVAL)
    ELSE
      result := SBuf.SNew(u^.space, args[1], args[2]);
      proc   := gsiProc[LOOPHOLE(args[0], UxTypes.GSIOp)];
      IF proc = NIL THEN RaiseError(UxTypes.EINVAL) END;
      UxAS.Get(u, args[3], start);
      nItems := proc(u, result, start, args[4]);
      UxAS.Put(u, args[3], start);
      RETURN nItems
    END
  END UGetSysInfo;

PROCEDURE GSIProgEnvProc(
          u:      Ux.T;
  VAR IN result: SBuf.T;
  VAR    start:  INTEGER;
         arg:    UNSIGNED
  ): CARDINAL;
  BEGIN
    SBuf.Put(u^.compatibilityMode, 0, result);
    RETURN 1
  END GSIProgEnvProc;

PROCEDURE GSIMaxUProcsProc(
          u:      Ux.T;
  VAR IN result: SBuf.T;
  VAR    start:  INTEGER;
         arg:    UNSIGNED
```

```
  ): CARDINAL;
  VAR
    maxUProcs: INTEGER;
  BEGIN
    maxUProcs := LAST(INTEGER);
    SBuf.Put(maxUProcs, 0, result);
    RETURN 1
  END GSIMaxUProcsProc;

PROCEDURE UGetUID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    LOCK gx DO
      TRY
        UxAS.R1Result(u, UxAuth.AuthToUserID(u^.defaultAuth));
        RETURN UxAuth.AuthToUserID(u^.extraAuths^[0]);
      EXCEPT
      | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
      END;
    END
  END UGetUID;

PROCEDURE UKill(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    uu:     Ux.T;
    eAuth:  Auth.T;
    pid:    OS1.PID;
    signal: OS1.SignalOrNone;
  BEGIN
    eAuth := u^.defaultAuth;
    IF args[1] > ORD(LAST(OS1.Signal)) THEN RaiseError(UxTypes.EINVAL) END;
    signal := LOOPHOLE(args[1], OS1.SignalOrNone);
    pid    := LOOPHOLE(args[0], OS1.PID);
    LOCK gx DO
      IF pid.i = -1 THEN (* broadcast *)
        TRY
          IF NOT UxAuth.SystemAclCheck(eAuth) THEN
            RaiseError(UxTypes.EPERM);
          END;
        EXCEPT
        | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
        END;
        uu := uHead;
        WHILE uu # NIL DO
          IF (uu # u) AND (uu^.p^.state # Terminated) THEN
            PostSignal(uu, signal)
          END;
          uu := uu^.p^.uNext
        END
      ELSIF pid.i = 0 THEN (* process group *)
        DoSignalPG(u^.pgrp, signal)
      ELSE (* specific process *)
        IF NOT DoKill(u, ProcessFromPID(pid), signal, eAuth) THEN
          RaiseError(UxTypes.EPERM)
        END
      END
    END; (* LOCK gx *)
    RETURN 0
  END UKill;

PROCEDURE UKillPG(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pgrp:   OS1.PID;
    uu:     Ux.T;
    signal: OS1.SignalOrNone;
    killed: BOOLEAN;
  BEGIN
    IF args[1] > ORD(LAST(OS1.Signal)) THEN RaiseError(UxTypes.EINVAL) END;
    signal := LOOPHOLE(args[1], OS1.SignalOrNone);
    pgrp   := LOOPHOLE(args[0], OS1.PID);
    IF pgrp.i = 0 THEN pgrp := u^.pgrp END; (* default *)
    killed := FALSE;
    LOCK gx DO
      uu := uHead;
      WHILE uu # NIL DO
        IF (uu^.pgrp.i = pgrp.i) AND (uu^.p^.state # Terminated) AND
           DoKill(u, uu, signal, u^.defaultAuth)
        THEN
          killed := TRUE
        END;
        uu := uu^.p^.uNext
      END
    END;
    IF NOT killed THEN RaiseError(UxTypes.ESRCH) END;
    RETURN 0
  END UKillPG;

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

PROCEDURE USetGroups(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    TRY
      IF NOT UxAuth.SystemAclCheck(u^.defaultAuth) THEN
        RaiseError(UxTypes.EPERM);
      END;
    EXCEPT
    | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
    END;
    RETURN DoNothing(u)
  END USetGroups;

PROCEDURE USetHostID(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    TRY
      IF NOT UxAuth.SystemAclCheck(u^.defaultAuth) THEN
        RaiseError(UxTypes.EPERM);
      END;
    EXCEPT
    | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
    END;
    hostID := LOOPHOLE(args[0], INTEGER);
    RETURN 0
  END USetHostID;

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

PROCEDURE USetPGRP(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pid, pgrp: OS1.PID;
    p:         T;
    uu:        Ux.T;
  BEGIN
    p    := u^.p;
    pid  := LOOPHOLE(args[0], OS1.PID);
```

```
  END USetRLimit;

PROCEDURE USigBlock(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    p:       T;
    oldMask: OS1.Signals;
  BEGIN
    p := u^.p;
    LOCK gx DO
      oldMask := p^.mask;
      p^.mask := p^.mask + (LOOPHOLE(args[0], OS1.Signals) - PowerfulSignals)
    END;
    RETURN LOOPHOLE(oldMask, INTEGER)
  END USigBlock;

PROCEDURE USigGet(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    p:       T;
    signals: OS1.Signals;
  BEGIN
    p := u^.p;
    LOCK gx DO
      IF LOOPHOLE(args[0], OS1.Signals) - p^.mask # OS1.Signals{} THEN
        RAISE(OS1.Error, OS1.InvalidArgumentEC)
      END;
      signals    := p^.pending * LOOPHOLE(args[0], OS1.Signals);
      p^.pending := p^.pending - signals
    END;
    RETURN LOOPHOLE(signals, INTEGER)
  END USigGet;

PROCEDURE USigPause(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    p:       T;
    oldMask: OS1.Signals;
  BEGIN
    p       := u^.p;
    oldMask := p^.mask;
    p^.mask := LOOPHOLE(args[0], OS1.Signals) - PowerfulSignals;
    LOCK gx DO
      WHILE (p^.pending - p^.mask) = NoSignals DO
          (* Wait for alert. *)
        TRY Thread.AlertWait(gx, never) EXCEPT Thread.Alerted: END
      END;

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

      ActOnSignals(u, OS1.SigNone, oldMask);

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

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

```
PROCEDURE USigPending(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    s: OS1.Signals;
  BEGIN
    LOCK gx DO
      s := u^.p^.pending * u^.p^.mask;
      RETURN LOOPHOLE(s, INTEGER)
    END
  END USigPending;

PROCEDURE USigSetMask(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    p:       T;
    oldMask: OS1.Signals;
  BEGIN
    p := u^.p;
    LOCK gx DO
      oldMask := p^.mask;
      p^.mask := LOOPHOLE(args[0], OS1.Signals) - PowerfulSignals;
      IF (p^.pending - p^.mask) # NoSignals THEN (* restore SIGPENDING *)
        INCL(p^.watcherEvents, WESignal)
      END
    END;
    RETURN LOOPHOLE(oldMask, INTEGER)
  END USigSetMask;

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

PROCEDURE USigVec(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    p:      T;
    signal: OS1.Signal;
    vec:    UxTypes.SignalVectorEntry;
  BEGIN
    p := u^.p;
    IF (args[0] < ORD(FIRST(OS1.Signal))) OR (args[0] > ORD(LAST(OS1.Signal)))
    THEN
      RaiseError(UxTypes.EINVAL)
    END;
    signal := LOOPHOLE(args[0], OS1.Signal);
    IF (signal IN (PowerfulSignals - OS1.Signals{OS1.SigCont})) THEN
      RAISE(OS1.Error, OS1.BadStateForSignalEC)
    END;
    IF args[2] # 0 THEN UxAS.Put(u, args[2], p^.sigVec[signal]) END;
    IF args[1] # 0 THEN
      UxAS.Get(u, args[1], vec);
      IF (signal = OS1.SigCont) AND (vec.handler.h = UxTypes.SignalIgnore)
      THEN
        RAISE(OS1.Error, OS1.BadStateForSignalEC)
      END;
      vec.mask := vec.mask - PowerfulSignals;
      LOCK gx DO
        p^.sigVec[signal] := vec;
        IF vec.handler.h = UxTypes.SignalIgnore THEN
          EXCL(p^.pending, signal)
        END
      END
    END;
    RETURN 0
```

```
  END USigVec;

PROCEDURE USTrace(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* Taos only *)
  TYPE
    STrace = OSFriends1.STrace;
  BEGIN
    IF (args[0] > ORD(LAST(STrace))) OR (args[1] > ORD(LAST(STrace))) THEN
      RAISE(OS1.Error, OS1.InvalidArgumentEC)
    END;
    LOCK gx DO
      RemoteOSAS.SetSTrace(
        u, LOOPHOLE(args[0], STrace), LOOPHOLE(args[1], STrace))
    END;
    RETURN 0
  END USTrace;

PROCEDURE UTerminate(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  (* Taos only *)
  VAR
    signal: OS1.Signal;
    dump:   BOOLEAN;
  BEGIN
    IF (LOOPHOLE(FIRST(OS1.Signal), UNSIGNED) <= args[0]) AND
       (args[0] <= LOOPHOLE(LAST(OS1.Signal), UNSIGNED))
    THEN
      signal := LOOPHOLE(args[0], OS1.Signal)
    ELSE
      signal := OS1.SigIll
    END;
    dump := args[1] # 0;
    LOCK gx DO PostTerminate(u, signal, dump) END;
    RETURN 0
  END UTerminate;

PROCEDURE UUnimplemented(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  BEGIN
    Signal(u, OS1.SigSys);
    RaiseError(UxTypes.EINVAL);
    RETURN 0
  END UUnimplemented;

PROCEDURE UVFork(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    i: INTEGER;
  VAR
    rt: Trace.TimeRec;
  BEGIN
    IF Trace.enabled THEN
      Trace.Register(u, rt);
      i := 0
    ELSE
      rt.realTime := 0
    END;
    TRY
      i := DoFork(u, (*vfork:*) TRUE);
      RETURN i
    FINALLY
      IF Trace.enabled THEN
        Trace.Enter(u, Trace.VFork, rt, NIL, NIL, i, 0)
      END
    END
  END UVFork;

PROCEDURE UWait(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
```

```
  (* This procedure implements both wait and wait3. wait(s) is equivalent to
     wait3(s, 0, 0). Since 'args' has missing arguments filled in with zeros,
     the right thing happens automatically. *)
  VAR
    uu:    Ux.T;
    p, pp: T;
    noChildren, noHang, untraced: BOOLEAN;
    rUsage: UxTypes.RUsage;
  PROCEDURE R0AndR1Results(): INTEGER;
    VAR
      ws32: BITS 32 FOR UxTypes.WStatus;
    BEGIN
      ws32 := pp^.ws; (* 0's high order bytes *)
      UxAS.R1Result(u, LOOPHOLE(ws32, INTEGER));
      RETURN LOOPHOLE(uu^.pid, INTEGER)
    END R0AndR1Results;
  BEGIN
    noHang   := (UxTypes.WNOHANG IN LOOPHOLE(args[1], UxTypes.WOptions));
    untraced := (UxTypes.WUNTRACED IN LOOPHOLE(args[1], UxTypes.WOptions));
    p        := u^.p;
    LOCK gx DO
      LOOP
        uu         := uHead; (* uHead can't be NIL *) (* or child ! *)
        noChildren := TRUE;
        LOOP
          pp := uu^.p;
          IF pp^.uParent = u THEN
            noChildren := FALSE;
            IF pp^.state = Terminated THEN
              IF args[2] # 0 THEN
                ZERO(ADDR(rUsage), BYTESIZE(rUsage));
                rUsage.uTime := pp^.descCTime;
                rUsage.sTime := pp^.descWTime;
                UxAS.Put(u, args[2], rUsage, 0, UxTypes.U11RUsageLen)
              END;
              Remove(uu);
              RETURN R0AndR1Results()
            END;
            IF (pp^.state = Stopped) AND pp^.untraced AND untraced THEN
              pp^.untraced := FALSE;
              RETURN R0AndR1Results()
            END
          END;
          IF pp^.uNext = NIL THEN EXIT END;
          uu := pp^.uNext
        END;
        IF noChildren THEN
          UxAS.SetError(u);
          RETURN ORD(UxTypes.ECHILD)
        END;
        IF noHang THEN RETURN 0 END;
        Thread.AlertWait(gx, p^.childStateChanged); (* alert => signal *)
      END
    END
  END UWait;

PROCEDURE UWaitPid(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    pid:   OS1.PID;
    uu:    Ux.T;
    p, pp: T;
    noChildren, noHang, untraced: BOOLEAN;
  PROCEDURE R0AndR1Results(): INTEGER;
    VAR
      ws32: BITS 32 FOR UxTypes.WStatus;
```

```
    BEGIN
      ws32 := pp^.ws; (* 0's high order bytes *)
      UxAS.R1Result(u, LOOPHOLE(ws32, INTEGER));
      RETURN LOOPHOLE(uu^.pid, INTEGER)
    END R0AndR1Results;
  BEGIN
    pid := LOOPHOLE(args[0], OS1.PID);
      (* args[1] is status, which we return via R1. *)
    noHang   := (UxTypes.WNOHANG IN LOOPHOLE(args[2], UxTypes.WOptions));
    untraced := (UxTypes.WUNTRACED IN LOOPHOLE(args[2], UxTypes.WOptions));
    p        := u^.p;
    LOCK gx DO
      LOOP
        uu         := uHead; (* uHead can't be NIL *) (* or child ! *)
        noChildren := TRUE;
        LOOP
          pp := uu^.p;
          IF (pp^.uParent = u) AND ((pid.i < -1) AND (uu^.pgrp.i = -pid.i))
             OR (pid.i = -1) OR ((pid.i = 0) AND (uu^.pgrp.i = u^.pgrp.i)) OR
             ((pid.i > 0) AND (uu^.pid.i = pid.i))
          THEN
            noChildren := FALSE;
            IF pp^.state = Terminated THEN
              Remove(uu);
              RETURN R0AndR1Results()
            END;
            IF (pp^.state = Stopped) AND pp^.untraced AND untraced THEN
              pp^.untraced := FALSE;
              RETURN R0AndR1Results()
            END
          END;
          IF pp^.uNext = NIL THEN EXIT END;
          uu := pp^.uNext
        END;
        IF noChildren THEN
          UxAS.SetError(u);
          RETURN ORD(UxTypes.ECHILD)
        END;
        IF noHang THEN RETURN 0 END;
        Thread.AlertWait(gx, p^.childStateChanged); (* alert => signal *)
      END
    END
  END UWaitPid;


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

PROCEDURE CloseTemplate(u: Ux.T; t: OS1.ProcessTemplate) RAISES {OS1.Error};
  VAR
    remove: BOOLEAN;
  BEGIN
    IF t = NIL THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
    LOCK t^.p^.xTemplate DO
      remove := t^.p^.state = Nascent;
      IF remove THEN t^.p^.state := Terminated END
    END;
    IF remove THEN
      LOCK gx DO
        RemoveTemplate(u, t);
        DisposeResources(t, TRUE)
      END
    END
  END CloseTemplate;
```

```
PROCEDURE GetProcessInfo(
      u:              Ux.T;
      pid:            OS1.PID;
  VAR processInfoOUT: OS1.ProcessInfo;
      getUser:        BOOLEAN
  ) RAISES {OS1.Error};
  VAR
    uu:          Ux.T;
    pp:          T;
    info:        OS1.ProcessInfo;
    targetState: TDState.TargetState;
    signal:      OS1.Signal;
  BEGIN
    IF pid.i = Ux.NullPID THEN pid := u^.pid END; (* default *)
    LOCK gx DO
      uu := ProcessFromPID(pid, (*esrch:*) TRUE, (*terminatedOK:*) TRUE);
      pp := uu^.p;
        (* The compiler says it is too dumb to allow WITH to open an RC VAR
           parameter, so we use a temporary. *)
      ZERO(ADDR(info), BYTESIZE(info));
      info.pid := pid;
      WITH info DO
        IF pp^.uParent # NIL THEN ppid := pp^.uParent^.pid END;
        pgrp := uu^.pgrp;
        IF getUser THEN
          TRY
            effUser  := UxAuth.NameFromAuth(uu^.defaultAuth);
          EXCEPT
          | Auth.Error: effUser := NIL;
          END;
          TRY
            realUser := UxAuth.NameFromAuth(uu^.extraAuths^[0]);
          EXCEPT
          | Auth.Error: realUser := NIL;
          END;
        END;
        IF pp^.terminating THEN
          processState := OS1.PSTerminating
        ELSE
          CASE pp^.state OF
          | Running:
            processState := OS1.PSRunning
          | Stopped:
            processState := OS1.PSStopped
          | Terminated:
            processState := OS1.PSTerminating
          END
        END;
        IF processState = OS1.PSTerminating THEN (* unanswerable questions *)
            (* ctlTty := NIL; *)
            (* windowSystem := NIL; *)
            (* space := NubTypes.NullSpace; *)
            (* numThreads := 0; *)
            (* debugState := DSRunning; *)
            (* debuggerConnected := FALSE; *)
            (* signalStates := default; *)
            (* residentBytes := 0; *)
            (* mappedBytes := 0; *)
            (* rUsage[OS1.Self] := 0; *)
        ELSE
          ctlTty := UxFile.GetControlTtyName(uu);
            (* ***** windowSystem := ???; *)
          space := uu^.space;
          IF space # NubTypes.NullSpace THEN
```

```
        numThreads := TPFriends.NHandles(space)
      END;

        (* If TDState.NewTarget has been called... *)
      IF (space # NubTypes.NullSpace) AND (space > Ux.uTaos^.space) AND
         NOT pp^.vForkChild
      THEN
        TDState.GetState(pid.i, space, debuggerConnected, targetState);
        CASE targetState OF
        | TDState.Running:
          debugState := OS1.DSRunning
        | TDState.Stopped:
          debugState := OS1.DSStopped
        | TDState.Trapped:
          debugState := OS1.DSTrapped
        END
      ELSE
        debugState := OS1.DSRunning
      END;
      FOR signal := FIRST(OS1.Signal) TO LAST(OS1.Signal) DO
        signalStates[signal] :=
          GetSignalState(pp^.sigVec[signal].handler, signal IN pp^.mask)
      END;
      IF (uu # uInit) THEN
        UxAS.GetInfo(uu, mappedBytes, residentBytes);
        IF (uu # Ux.uTaos) AND (uu # uNub) THEN
          rUsage[OS1.Self].userTime := UxAS.GetClientTime(uu)
        END
      END;
      IF pp^.watcherThread # NIL THEN
        rUsage[OS1.Self].systemTime := GetWTime(uu)
      END
    END;
    rUsage[OS1.Children].userTime   := pp^.descCTime;
    rUsage[OS1.Children].systemTime := pp^.descWTime;
    command := pp^.name
  END; (* WITH *)
  processInfoOUT := info
 END (* LOCK gx *)
END GetProcessInfo;

PROCEDURE NewProcessTemplate(u: Ux.T): OS1.ProcessTemplate RAISES {};
  VAR
    t: Ux.T;
  BEGIN
    New(u, t, (*vFork:*) FALSE, (*topaz:*) TRUE);
    InsertTemplate(u, t);
    RETURN t
  END NewProcessTemplate;

PROCEDURE NextProcess(u: Ux.T; pid: OS1.PID): OS1.PID RAISES {};
  VAR
    pidNext: OS1.PID;
  BEGIN
    pidNext.i := Ux.NullPID;
    LOCK gx DO
      IF pid.i = Ux.NullPID THEN
        IF uHead # NIL THEN pidNext := uHead^.pid END
      ELSIF (uHead # NIL) AND (pid.i < uTail^.pid.i) THEN
        IF (uRover = NIL) OR (pid.i < uRover^.pid.i) THEN uRover := uHead END;
          (* Now we know that uRover # NIL and uRover^.pid.i <= pid.i <
             uTail^.pid.i *)
        WHILE uRover^.pid.i <= pid.i DO uRover := uRover^.p^.uNext END;
          (* Now we know that uRover^.pid is the least PID larger than pid.
             *)
        pidNext := uRover^.pid
      ELSE (* pid greater than any in list *)
      END
    END; (* LOCK *)
    RETURN pidNext
  END NextProcess;

PROCEDURE SendSignal(
  u:      Ux.T;
  pid:    OS1.PID;
  signal: OS1.SignalOrNone;
  eAuth:  Auth.T
  ) RAISES {OS1.Error};
  BEGIN
    LOCK gx DO
      IF pid.i = Ux.NullPID THEN pid := u^.pid END; (* default *)
      IF NOT DoKill(u, ProcessFromPID(pid), signal, eAuth) THEN
        RAISE(OS1.Error, OS1.NotOwnerEC)
      END
    END (* LOCK gx *)
  END SendSignal;

PROCEDURE SendSignalToGroup(
  u:      Ux.T;
  pgrp:   OS1.PGRP;
  signal: OS1.SignalOrNone;
  eAuth:  Auth.T
  ) RAISES {OS1.Error};
  VAR
    uu:     Ux.T;
    killed: BOOLEAN;
  BEGIN
    killed := FALSE;
    LOCK gx DO
      IF pgrp.i = Ux.NullPID THEN pgrp := u^.pgrp END; (* default *)
      uu := uHead;
      WHILE uu # NIL DO
        IF (uu^.pgrp.i = pgrp.i) AND (uu^.p^.state # Terminated) AND
           DoKill(u, uu, signal, eAuth)
        THEN
          killed := TRUE
        END;
        uu := uu^.p^.uNext
      END
    END;
    IF NOT killed THEN RAISE(OS1.Error, OS1.BadPIDEC) END
  END SendSignalToGroup;

PROCEDURE SetMySignalState(
  u:      Ux.T;
  signal: OS1.Signal;
  state:  OS1.SignalState
  ): OS1.SignalState RAISES {OS1.Error};
  VAR
    p:             T;
    e, eOld:       UxTypes.SignalVectorEntry;
    mask, maskOld: BOOLEAN;
  BEGIN
    p := u^.p;
    CASE state OF
    | OS1.SignalDefault:
      e.handler.h := UxTypes.SignalDefault;
      mask        := FALSE
    | OS1.SignalIgnore:
      e.handler.h := UxTypes.SignalIgnore;
```

```
      mask        := FALSE;
      IF signal IN PowerfulSignals THEN
        RAISE(OS1.Error, OS1.BadStateForSignalEC)
      END
    | OS1.SignalHandle:
      e.handler.h := UxTypes.SignalDefault;
      mask        := TRUE;
      IF (signal IN (PowerfulSignals - OS1.Signals{OS1.SigCont})) THEN
        RAISE(OS1.Error, OS1.BadStateForSignalEC)
      END
    END;
    e.mask  := OS1.Signals{};
    e.flags := UxTypes.SVFlags{};
    LOCK gx DO
      eOld              := p^.sigVec[signal];
      p^.sigVec[signal] := e;
      IF state = OS1.SignalIgnore THEN EXCL(p^.pending, signal) END;
      maskOld := signal IN p^.mask;
      IF mask THEN
        INCL(p^.mask, signal)
      ELSE
        EXCL(p^.mask, signal);
          (* Restore SIGPENDING: *)
        IF signal IN p^.pending THEN PostWatcherEvent(p, WESignal) END
      END;
      Thread.Broadcast(p^.signaled); (* ensure SIGWAIT *)
      RETURN GetSignalState(eOld.handler, maskOld)
    END
  END SetMySignalState;

PROCEDURE SetPGRP(u: Ux.T; t: OS1.ProcessTemplate) RAISES {OS1.Error};
  BEGIN
    IF t = NIL THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
    LOCK t^.p^.xTemplate DO
      IF t^.p^.state # Nascent THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
      t^.pgrp.i := Ux.NullPID
    END
  END SetPGRP;

PROCEDURE SetSignalState(
  u:      Ux.T;
  t:      OS1.ProcessTemplate;
  signal: OS1.Signal;
  state:  OS1.SignalState
  ) RAISES {OS1.Error};
  VAR
    handler: UxTypes.SignalHandler;
    mask:    BOOLEAN;
  BEGIN
    IF t = NIL THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
    LOCK t^.p^.xTemplate DO
      IF t^.p^.state # Nascent THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
      CASE state OF
      | OS1.SignalDefault:
        handler.h := UxTypes.SignalDefault;
        mask      := FALSE
      | OS1.SignalIgnore:
        handler.h := UxTypes.SignalIgnore;
        mask      := FALSE;
        IF signal IN PowerfulSignals THEN
          RAISE(OS1.Error, OS1.BadStateForSignalEC)
        END
      | OS1.SignalHandle:
        handler.h := UxTypes.SignalDefault;
        mask      := TRUE;
        IF (signal IN (PowerfulSignals - OS1.Signals{OS1.SigCont})) THEN
          RAISE(OS1.Error, OS1.BadStateForSignalEC)
        END
      END;
      t^.p^.sigVec[signal].handler := handler;
      IF mask THEN INCL(t^.p^.mask, signal) ELSE EXCL(t^.p^.mask, signal) END
    END
  END SetSignalState;

PROCEDURE GetProcessAcl(
    u: Ux.T;
    pid: OS1.PID;
    VAR (*OUT*) acl: Text.RefArray;
    VAR (*OUT*) uid: UIDExtras.T)
  RAISES {OS1.Error};
  VAR uu: Ux.T;
  BEGIN
    IF pid.i = Ux.NullPID THEN pid := u^.pid END; (* default *)
    LOCK gx DO
      uu := ProcessFromPID(pid);
      acl := uu^.acl;
      uid := uu^.uid;
    END (* LOCK *)
  END GetProcessAcl;

PROCEDURE SetProcessAcl(
    u: Ux.T;
    t: OS1.ProcessTemplate;
    acl: Text.RefArray)
  RAISES {OS1.Error};
  BEGIN
    LOCK t^.p^.xTemplate DO
      IF t^.p^.state # Nascent THEN
        RAISE(OS1.Error, OS1.InvalidArgumentEC)
      END;
      t^.acl := acl;
    END (* LOCK *)
  END SetProcessAcl;

PROCEDURE SetAuths(
    u: Ux.T;
    t: OS1.ProcessTemplate;
    defaultAuth: Auth.T;
    extraAuths: OS1.AuthArray := NIL
  ) RAISES {OS1.Error};
  VAR
    set: BOOLEAN;
    i: CARDINAL;
    da: Auth.T;
    ea: OS1.AuthArray;
  BEGIN
    TRY
      LOCK t^.p^.xTemplate DO
        IF t^.p^.state # Nascent THEN
          RAISE(OS1.Error, OS1.InvalidArgumentEC);
        END;
        AuthSpecial.CheckEntitlement(defaultAuth);
        da := AuthSpecial.Localize(defaultAuth);
        IF extraAuths = NIL THEN
          NEW(ea, 1);
          ea^[0] := da;
        ELSE
          NEW(ea, NUMBER(extraAuths^));
          FOR i := 0 TO HIGH(extraAuths^) DO
            AuthSpecial.CheckEntitlement(extraAuths^[i]);
```

```
          ea^[i] := AuthSpecial.Localize(extraAuths^[i]);
        END;
      END;
      t^.defaultAuth := da;
      t^.extraAuths := ea;
    END (* LOCK *)
  EXCEPT
  | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
  END;
END SetAuths;

PROCEDURE StartProcess(
  u:            Ux.T;
  file:         OS1.File;
  dir:          OS1.Dir;
  eAuth:        Auth.T;
  name:         Text.T;
  argv:         Text.RefArray;
  t:            OS1.ProcessTemplate;
  relationship: OS1.Relationship;
  env:          Text.RefArray
  ): OS1.PID RAISES {OS1.Error};
  BEGIN
    IF t = NIL THEN
      New(u, t, (*vFork:*) FALSE, (*topaz:*) TRUE)
    ELSE
      LOCK t^.p^.xTemplate DO
        IF t^.p^.state # Nascent THEN
          RAISE(OS1.Error, OS1.InvalidArgumentEC)
        END;
          (* Make template is invalid for others before releasing mutex. *)
        t^.p^.state := Running
      END;
      LOCK gx DO RemoveTemplate(u, t) END
    END;
    IF NOT DoExec(t, file, dir, eAuth, name, argv, env) THEN
      DisposeResources(t, TRUE);
      RAISE(OS1.Error, OS1.TooManyProcessesEC)
    END;
    LOCK gx DO
        (* We need to set the initial client thread priority before that
           thread has a chance to fork others. For efficiency, we do use a
           special routine that avoids the need for stopping and enumerating
           all threads in the address space. *)
      UxAS.SetInitialClientPriority(t, t^.p^.priority);
      IF relationship = OS1.Orphan THEN t^.p^.uParent := uInit END;
      DoForkCommon(u, t);
      IF t^.pgrp.i = Ux.NullPID THEN t^.pgrp := t^.pid END;
      UxFile.Exec2(t)
    END;
    RETURN t^.pid
  END StartProcess;

PROCEDURE WaitForChild(
                u:      Ux.T;
                pid:    OS1.PID;
  VAR (* OUT *) rUsage: OS1.RUsage
  ): OS1.WStatus RAISES {OS1.Error, Thread.Alerted};
  VAR
    uu: Ux.T;
    pp: T;
    ws: OS1.WStatus;
  BEGIN
    LOCK gx DO
      LOOP
```

```
        uu := ProcessFromPID(pid, (*esrch:*) TRUE, (*terminatedOK:*) TRUE);
        pp := uu^.p;
        IF pp^.state = Terminated THEN
          rUsage.userTime   := pp^.descCTime;
          rUsage.systemTime := pp^.descWTime;
          IF pp^.ws.wTermSig = OS1.SigNone THEN
            ws.reason   := OS1.Exited;
            ws.wRetCode := pp^.ws.wRetCode
          ELSE
            ws.reason    := OS1.Killed;
            ws.wTermSig  := pp^.ws.wTermSig;
            ws.wCoreDump := pp^.ws.wCoreDump
          END;
          Remove(uu);
          EXIT
        ELSIF (pp^.state = Stopped) AND pp^.untraced THEN
          ws.reason    := OS1.Stopped;
          ws.wStopSig  := pp^.ws.wStopSig;
          pp^.untraced := FALSE;
          EXIT
        ELSE
          Thread.AlertWait(gx, pp^.stateChanged)
        END
      END (* LOOP *)
    END; (* LOCK gx *)
    RETURN ws
  END WaitForChild;

PROCEDURE WaitForSignal(u: Ux.T; allowed: OS1.Signals): OS1.Signal
  RAISES {OS1.Error, Thread.Alerted};
  VAR
    p:           T;
    signal:      OS1.Signal;
    signalFound: OS1.SignalOrNone;
  BEGIN
    p := u^.p;
    LOCK gx DO
      LOOP
        signalFound := OS1.SigNone;
          (* Allowed signals must be defaulted and masked. *)
        FOR signal := FIRST(OS1.Signal) TO LAST(OS1.Signal) DO
          IF (signal IN allowed) THEN
            IF (p^.sigVec[signal].handler.h # UxTypes.SignalDefault) OR
              NOT (signal IN p^.mask)
            THEN
              RAISE(OS1.Error, OS1.BadStateForSignalEC)
            END;
            IF signal IN p^.pending THEN signalFound := signal END
          END
        END;
          (* Is there an allowed, pending signal? *)
        IF signalFound # OS1.SigNone THEN
          EXCL(p^.pending, signalFound);
          RETURN signalFound
        END;
          (* Wait for pending*masked to be nonempty. *)
        Thread.AlertWait(gx, p^.signaled)
      END (* LOOP *)
    END; (* LOCK gx *)
  END WaitForSignal;


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

```
PROCEDURE GetPriority(u: Ux.T; pid: OS1.PID): OSFriends1.ProcessPriority
  RAISES {OS1.Error};
  VAR
    uu: Ux.T;
  BEGIN
    LOCK gx DO
      IF pid.i = Ux.NullPID THEN pid := u^.pid END; (* default *)
      uu := ProcessFromPID(pid);
      RETURN uu^.p^.priority
    END
  END GetPriority;

PROCEDURE GetMyResourceLimit(
  u:        Ux.T;
  resource: OSFriends1.Resource
  ): OSFriends1.ResourceLimit RAISES {};
  BEGIN
    RETURN LOOPHOLE(GetRLimit(u, LOOPHOLE(resource, UxTypes.Resource)),
                    OSFriends1.ResourceLimit)
  END GetMyResourceLimit;

PROCEDURE SetMyResourceLimit(
  u:        Ux.T;
  resource: OSFriends1.Resource;
  limit:    OSFriends1.ResourceLimit;
  auth:     Auth.T
  ) RAISES {OS1.Error};
  BEGIN
    SetRLimit(u, LOOPHOLE(resource, UxTypes.Resource),
              LOOPHOLE(limit, UxTypes.RLimit), auth)
  END SetMyResourceLimit;

PROCEDURE SetMyProcessAcl(
    u: Ux.T;
    acl: Text.RefArray)
  RAISES {OS1.Error};
  BEGIN
    LOCK gx DO
      u^.acl := acl;
    END (*lock*);
  END SetMyProcessAcl;

PROCEDURE SetMyAuths(
    u: Ux.T;
    defaultAuth: Auth.T;
    extraAuths: OS1.AuthArray := NIL)
  RAISES {OS1.Error};
  VAR i: CARDINAL; da: Auth.T; ea: OS1.AuthArray;
  BEGIN
    TRY
      LOCK gx DO
        AuthSpecial.CheckEntitlement(defaultAuth);
        da := AuthSpecial.Localize(defaultAuth);
        IF extraAuths = NIL THEN
          NEW(ea, 1);
          ea^[0] := da;
        ELSE
          NEW(ea, NUMBER(extraAuths^));
          FOR i := 0 TO HIGH(extraAuths^) DO
            AuthSpecial.CheckEntitlement(extraAuths^[i]);
            ea^[i] := AuthSpecial.Localize(extraAuths^[i]);
          END;
        END;
        u^.defaultAuth := da;
        u^.extraAuths := ea;
      END (* LOCK *)
    EXCEPT
    | Auth.Error: RAISE(OS1.Error, OS1.InvalidCredentialsEC);
    END;
  END SetMyAuths;

PROCEDURE SetPriority(
  u:        Ux.T;
  t:        OS1.ProcessTemplate;
  priority: OSFriends1.ProcessPriority;
  auth:     Auth.T
  ) RAISES {OS1.Error};
  BEGIN
    IF t = NIL THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
    (*!! could check permission here using auth *)
    LOCK t^.p^.xTemplate DO
      IF t^.p^.state # Nascent THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
      t^.p^.priority := priority
    END
  END SetPriority;

PROCEDURE SetPriorityPID(
  u:        Ux.T;
  pid:      OS1.PID;
  priority: OSFriends1.ProcessPriority;
  auth:     Auth.T
  ) RAISES {OS1.Error};
  VAR
    p: T;
  BEGIN
    IF pid.i = Ux.NullPID THEN pid := u^.pid END; (* default *)
    (*!! could check permission here using auth *)
    LOCK gx DO PostSetPriority(ProcessFromPID(pid), priority) END
  END SetPriorityPID;

PROCEDURE SetPriorityPGRP(
  u:        Ux.T;
  pgrp:     OS1.PGRP;
  priority: OSFriends1.ProcessPriority;
  auth:     Auth.T
  ) RAISES {OS1.Error};
  BEGIN
    IF pgrp.i = Ux.NullPID THEN pgrp := u^.pgrp END; (* default *)
    (*!! could check permission here using auth *)
    DoSetPriorityPGRP(pgrp, priority)
  END SetPriorityPGRP;

PROCEDURE SetPriorityUser(
  u:        Ux.T;
  userName: Text.T;
  priority: OSFriends1.ProcessPriority;
  auth:     Auth.T
  ) RAISES {OS1.Error};
  BEGIN
    (*!! could check permission here using auth *)
    LOCK gx DO
      DoSetPriorityUser(userName, priority)
    END
  END SetPriorityUser;

PROCEDURE SetResourceLimit(
  u:        Ux.T;
  t:        OS1.ProcessTemplate;
  resource: OSFriends1.Resource;
```

```
  limit:    OSFriends1.ResourceLimit;
  auth:     Auth.T
  ) RAISES {OS1.Error};
  BEGIN
    IF t = NIL THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
    LOCK t^.p^.xTemplate DO
      IF t^.p^.state # Nascent THEN RAISE(OS1.Error, OS1.InvalidArgumentEC) END;
      SetRLimit(t, LOOPHOLE(resource, UxTypes.Resource),
                LOOPHOLE(limit, UxTypes.RLimit), auth)
    END
  END SetResourceLimit;


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

PROCEDURE Exit(u: Ux.T; status: OS1.ExitStatus) RAISES {};
  BEGIN
    ASSERT(u # Ux.uTaos, "OS1.Exit called from Taos");
    LOCK gx DO
      u^.p^.ws.wRetCode := status;
      PostTerminate(u, OS1.SigNone, FALSE);
        (* We must not return until we are sure the client thread that called
           us has been stopped. However the watcher thread must wait for all
           the RPC server threads servicing this address space, including
           ours, to finish. *)
      WHILE NOT u^.p^.terminating DO Thread.Wait(gx, terminating) END
    END
  END Exit;

BEGIN
  initialized := FALSE; (* see exported procedure Init *)
END UxPro.
(* M. D. Schroeder -- started June, 1985

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

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

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

```
(* Copyright 1988 Digital Equipment Corporation.                  *)
(* Distributed only by permission.                                *)
(* Last modified on Fri Feb  1 14:30:15 PST 1991 by mann *)
(*      modified on Thu Jun  7 12:30:47 PDT 1990 by jerian *)
(*      modified on Thu Jan  4 09:46:24 1990 by kalsow *)
(*      modified on Tue Jun  6  9:23:16 PDT 1989 by mcjones *)
(*      modified on Wed Aug  3 16:07:08 PDT 1988 by swart *)
(*      modified on Tue Mar 29 16:13:54 PST 1988 by wobber *)
(*      modified on Mon Feb 15  9:10:56 PST 1988 by owicki *)
(*      modified on Tue Oct 21 13:17:16 1986 by mds *)

MODULE UxAS IMPLEMENTS UxAS, RemoteOSAS;

IMPORT ACL, Auth, Hardware, NS, NSUtil, OSFriends1, RemoteOSPro, RPC,
  RPCLocal, RdV, TDState, TDTraps, TPSpecial, Thread, ThreadFriends,
  ThreadsPort, UserAreaVax, UxAuth, UxFile, UxPro, VM, VMFriends,
  VMFriends2, VuAS, 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 - UserAreaVax.ByteSize;
    (* Stack starts here and grows towards smaller addresses. There is one
       more page at this address, containing the handler linkage code. *)
  HandlerEntry     = HoleAddr + 6cH; (* start of handler linkage *)
  initRLimDataCur  = BytesPerRegion;
  initRLimDataMax  = BytesPerRegion;
  initRLimStackCur = BytesPerRegion;
  initRLimStackMax = BytesPerRegion;
  initRLimCoreCur  = 7fffffffH;
  initRLimCoreMax  = 7fffffffH;

(* Constants determined by VM implementation. *)
(* VMBytesPerPage = 1024; *) (* variable while experimenting *)

VAR
  initialized: BOOLEAN;

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

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

(* Private state for UxAS. *)
(* Fields below marked (* ! *) are inherited by forked process. In case of
   state, only some subfields are inherited. *)
TYPE
  T            = REF TObj;
  Continuation = (Resume, Start, Nothing);
  TObj = RECORD
    firstWritableAddr: Addr;               (* lowest not write protected *)
    traps:             TPSpecial.TrapKinds; (* ones we are catching *)
    straceExec:      OSFriends1.STrace; (* whether to trap after execve *)
    straceFork:      OSFriends1.STrace; (* whether to trap after fork *)
    rLimData:        UxTypes.RLimit;    (* <= BytesPerRegion *) (*!*)
    rLimStack:       UxTypes.RLimit;    (* <= BytesPerRegion *) (*!*)
    rLimCore:        UxTypes.RLimit;    (*!*)
    isVulcan:        BOOLEAN;           (* <=> a Vulcan style address space *)
    tDStateAcquired: CARDINAL;
      (* surplus of AcquireTDState calls over ReleaseTDState calls *)
    currentThread: ThreadsPort.Handle; (* thread for current action *)
    CASE continuation: Continuation OF
    | Resume:
      sState: TPSpecial.SState
    END
  END;

CONST
  MaxIdleSpaces = 7;

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


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

PROCEDURE AcquireTDState(u: Ux.T);
  VAR
    a: T;
  BEGIN
    a := u^.a;
      (* We can't call HighTTD unless it knows about this target. It doesn't
         know about the child of a vfork. *)
    IF (a^.tDStateAcquired = 0) AND NOT UxPro.IsVForkChild(u) THEN
      TDState.Acquire(u^.pid.i, u^.space)
    END;
    INC(a^.tDStateAcquired)
  END AcquireTDState;

PROCEDURE AllocSpace(u: Ux.T; bssAddr, breakAddr, stackAddr: Addr)
  RAISES {OS.Error (* NotEnoughVMEC, TooManyProcessesEC *)};
  (* Assign space to u, with low and high segment boundaries equal to
     breakAddr and stackAddr respectively; ensure bytes from bssAddr to
     breakAddr-1 are zero. AllocSpace assumes the initial thread for u has
     already been allocated. *)
  VAR
    space:            VMFriends.Space;
    i, j:             CARDINAL;
    delta, bestDelta: INTEGER;
    minBreakAddr:     Addr;
    pages:            CARDINAL;
  BEGIN
    space := VMFriends.NullSpace;

      (* Use an idle space if one exists and this request is not too large.
         *)
    LOCK x DO
```

```
    bestDelta := LAST(INTEGER);
    IF breakAddr <= maxIdleBreakAddr THEN
      FOR i := 1 TO maxIdleSpaces DO
        WITH idleSpaces[i] DO
          IF s # VMFriends.NullSpace THEN
              (* ASSERT((b <= MaxCard) AND (breakAddr <= MaxCard)); *)
            delta :=
              ABS(LOOPHOLE(b, CARDINAL) - LOOPHOLE(breakAddr, CARDINAL));
            IF delta < bestDelta THEN
              j         := i;
              bestDelta := delta
            END
          END
        END
      END;
      IF bestDelta # LAST(INTEGER) THEN
        WITH idleSpaces[j] DO
          space := s;
          s     := VMFriends.NullSpace
        END;
        u^.space := space;
        RestoreThread(u);
        minBreakAddr := MIN(GetBoundary(u, VM.Low), breakAddr);
        SetBoundary(space, VM.Low, breakAddr);
        IF bssAddr < minBreakAddr THEN
            (* We only need to zero from bssAddr to minBreakAddr; any new
               pages will be zeroed by VM. *)
          SBuf.Zero(SBuf.SNew(space, bssAddr, minBreakAddr - bssAddr))
        END;
        SetBoundary(space, VM.High, stackAddr)
      END
    END
  END;

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

    space    := VMFriends.AllocSpace();
    u^.space := space;

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

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

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

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

PROCEDURE AllocThread(u: Ux.T);
  (* Create the initial thread to run in a new space. *)
  VAR
    a:     T;
    state: TPSpecial.State;
  BEGIN
    a := u^.a;
    a^.currentThread := TPSpecial.Create(NIL, NIL, 0);
    a^.continuation  := Resume;
    WITH a^.sState DO
      setR1 := FALSE;
      error := FALSE
    END;
    TPSpecial.GetState(a^.currentThread, state);
    state.stackHigh := HoleAddr; (* for the garbage collector *)
    TPSpecial.SetState(a^.currentThread, state)
  END AllocThread;

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

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

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

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

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

```
      RETURN FALSE
    END
  END GrowStack;

PROCEDURE ReleaseTDState(u: Ux.T);
  VAR
    a: T;
  BEGIN
    a := u^.a;
    ASSERT(a^.tDStateAcquired > 0); (* ***** *)
    DEC(a^.tDStateAcquired);
      (* See note in AcquireTDState. *)
    IF (a^.tDStateAcquired = 0) AND NOT UxPro.IsVForkChild(u) THEN
      TDState.Release(u^.pid.i, u^.space)
    END
  END ReleaseTDState;

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

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

      (* Allocate or free pages, as appropriate. *)
    IF delta > 0 THEN (* round up *)
      IF NOT
         VMFriends.AllocPages(
           space, seg, (delta + VMBytesPerPage - 1) DIV VMBytesPerPage, page)
      THEN
        RAISE(OS.Error, OS.NotEnoughVMEC)
      END
    ELSIF delta <= -VMBytesPerPage THEN (* round down *)
      VMFriends.FreePages(space, seg, (-delta) DIV VMBytesPerPage)
    END
  END SetBoundary;

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

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


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

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

PROCEDURE CheckError(u: Ux.T): BOOLEAN RAISES {};
  VAR
    a: T;
  BEGIN
    a := u^.a;
    IF a^.continuation = Resume THEN
      RETURN a^.sState.error
    ELSE
      RETURN FALSE
    END
  END CheckError;

PROCEDURE Copy(u, uNew: Ux.T; vFork: BOOLEAN)
  RAISES {OS.Error (* NotEnoughVMEC, TooManyProcessesEC *), SBuf.Fault};
  VAR
    a, aNew: T;
    lowAddr, breakAddr, stackAddr: Addr;
```

```
    pageState:       VM.PageState;
    state, stateNew: TPSpecial.State;
  BEGIN
    AllocThread(uNew);
    a    := u^.a;
    aNew := uNew^.a;

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

      (* Allocate appropriate space. *)
    ASSERT(uNew^.space = VMFriends.NullSpace, "UxAS.Copy: old=null");
    aNew^.firstWritableAddr := a^.firstWritableAddr;
    IF vFork THEN
      uNew^.space := u^.space;
      RestoreThread(uNew)
    ELSE
      (* The first page may have been unmapped by the Topaz runtime. *)
      lowAddr   := TextAddr;
      pageState := VMFriends.StatePage(u^.space, 0);
      IF pageState.contents = VM.Nonexistent THEN
        lowAddr := MAX(lowAddr, LOOPHOLE(VMBytesPerPage, Addr))
      END;

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

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

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

        (* Write-protect prefix of low region. *)
      VMFriends.MakePagesReadOnly(
        uNew^.space, lowAddr DIV VMBytesPerPage,
        (MIN(aNew^.firstWritableAddr, breakAddr) - lowAddr) DIV
        VMBytesPerPage); (* round size down *)
    END
  END Copy;

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

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

PROCEDURE GetArrayOfText(u: Ux.T; addr: Addr): Text.RefArray
  RAISES {SBuf.Fault};
  VAR
```

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

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

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

PROCEDURE GetInfo(u: Ux.T; VAR (*out*) mappedBytes, residentBytes: CARDINAL)
  RAISES {};
  (* Caller must first call UxPro.AcquireProcess(u). *)
  VAR
    spc: VMFriends2.SegmentPageCounts;
  BEGIN
    IF u^.space # VMFriends.NullSpace THEN
      IF NOT VMFriends2.GetSegmentPageCounts(u^.space, VM.Low, spc) THEN
        ASSERT(FALSE, "UxAS.GetInfo: bad space")
      END;
      mappedBytes   := spc.created * VMBytesPerPage;
      residentBytes := spc.resident * VMBytesPerPage;
      IF NOT VMFriends2.GetSegmentPageCounts(u^.space, VM.High, spc) THEN
        ASSERT(FALSE, "UxAS.GetInfo: bad space")
      END;
      INC(mappedBytes, spc.created * VMBytesPerPage);
      INC(residentBytes, spc.resident * VMBytesPerPage)
    END
  END GetInfo;

PROCEDURE GetRLimit(
              u:        Ux.T;
              resource: UxTypes.Resource;
  VAR (*out*) rLimit:   UxTypes.RLimit
  ) RAISES {};
  VAR
    a: T;
  BEGIN
    a := u^.a;
    CASE resource OF
```

```
    | UxTypes.rLimitData:
      rLimit := a^.rLimData
    | UxTypes.rLimitStack:
      rLimit := a^.rLimStack
    | UxTypes.rLimitCore:
      rLimit := a^.rLimCore
    END
  END GetRLimit;

PROCEDURE GetText(u: Ux.T; addr: Addr): Text.T RAISES {SBuf.Fault};
  VAR
    bufPos, bufCount: INTEGER;
    buf: ARRAY [0..127] OF CHAR; (* ASSERT(NUMBER(buf) <= VMPageSize *)
    t:   Text.T;
  BEGIN
    IF addr = 0 THEN
      RETURN NIL
    ELSE
      t := NIL;
      LOOP
        bufCount := NUMBER(buf);
        IF NOT VMFriends.ReadBlock(u^.space, addr, ADDR(buf), bufCount) THEN
          IF addr MOD VMBytesPerPage + NUMBER(buf) <= VMBytesPerPage THEN
            RAISE(SBuf.Fault)
          END;
          bufCount := VMBytesPerPage - (addr MOD VMBytesPerPage);
          IF NOT VMFriends.ReadBlock(u^.space, addr, ADDR(buf), bufCount)
          THEN
            RAISE(SBuf.Fault)
          END
        END;
        bufPos := 0;
        WHILE (bufPos < bufCount) AND (buf[bufPos] # '\000') DO
          INC(bufPos)
        END;
        IF t = NIL THEN (* the first buffer *)
          t := Text.FromSub(buf, 0, bufPos)
        ELSE (* text comes from multiple buffers *)
          t := Text.Cat(t, Text.FromSub(buf, 0, bufPos))
        END;
        IF (bufPos < bufCount) THEN EXIT END;
        INC(addr, bufCount)
      END;
      RETURN t
    END
  END GetText;

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

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

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

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

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

    nullTHandle := ThreadsPort.Null();

    TaosSpace := TPFriends.GetSpace();

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

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

    UxPro.Register(UxTypes.SYSgetpagesize, UGetPageSize);
    UxPro.Register(UxTypes.SYSreservetraps, UReserveTraps); (* Ux only *)
    UxPro.Register(UxTypes.SYSsbreak, USBreak);
    UxPro.Register(UxTypes.SYSvadvise, UVAdvise);
    UxPro.Register(UxTypes.SYSspare50, UVulcan);
  END Init;

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

PROCEDURE Load(
  u:           Ux.T;
  file:        OS.File;
  dir:         OS.Dir;
  euserAuth:   Auth.T;
  name:        Text.T;
  argv, envp:  Text.RefArray;
  uArgs:       Ux.T;
  argva, envpa: Addr;
  vFork:       BOOLEAN
  ): BOOLEAN RAISES {OS.Error, SBuf.Fault, Thread.Alerted};

  VAR
    a:                T;                    (* address space of u *)
    argsFetched:      BOOLEAN;
    buf:              ARRAY [0..127] OF CHAR;
    bufCount, bufIndex: CARDINAL;
    e:                UxTypes.Exec;  (* first few bytes from file *)
    extraCount: INTEGER;       (* > 0 if named interpreter is found *)
    extras:     Text.RefArray; (* [name [, arg]] *)

  PROCEDURE IsADotOut(VAR mode: OS.FileMode; VAR owner: Text.T): BOOLEAN;
```

```
(* Reads first few bytes of file and sets, mode, owner, and e. If file
   appears to contain an executable object program (i.e., an a.out file),
   returns TRUE and leaves file positioned at the beginning of the text
   segment. *)
VAR
  headerSize:    CARDINAL;
  fileInfo:      OS.FileInfo;
  instanceOwner: Text.T;
BEGIN
  OS.GetInfo(file, fileInfo, (*returnNames:*) TRUE);
  mode := fileInfo.mode;
  IF (OS.SetUIDonExec IN mode.fileState) AND
     NOT Text.IsEmpty(fileInfo.instance)
  THEN
    TRY
      IF NOT
         NSUtil.TestValue("TrustedHosts", "members", fileInfo.instance)
      THEN
        instanceOwner := NSUtil.GetValue(fileInfo.instance, "owner");
        IF NOT Text.Equal(instanceOwner, fileInfo.owner) AND
           NOT Text.Equal(NSUtil.GetValue(NSUtil.GetMyHost(), "owner"),
                          instanceOwner)
        THEN
          EXCL(mode.fileState, OS.SetUIDonExec)
        END
      END
    EXCEPT
    | NS.LabelNotFound:
      EXCL(mode.fileState, OS.SetUIDonExec)
    | RPC.CallFailed:
      EXCL(mode.fileState, OS.SetUIDonExec)
    END
  END;
  owner    := fileInfo.owner;
  bufCount := UxFile.ReadF(file, SBuf.New(buf));
  bufIndex := 0;
  COPY(ADDR(buf), ADDR(e), BYTESIZE(e));
  IF (bufCount < BYTESIZE(e)) OR
     (e.magic # UxTypes.ZMagic) AND (e.magic # UxTypes.NMagic) AND
     (e.magic # UxTypes.OMagic)
  THEN
    RETURN FALSE
  END;
    (* The UxTypes.Exec size fields should have been declared UNSIGNED
       rather than CARDINAL... *)
  IF (LOOPHOLE(e.text, INTEGER) < 0) OR
     (LOOPHOLE(e.data, INTEGER) < 0) OR (LOOPHOLE(e.bss, INTEGER) < 0)
  THEN
    CloseQuietly(file);
    RAISE(OS.Error, OS.ShortExecutableEC)
  END;
  headerSize := BYTESIZE(e);
  IF e.magic = UxTypes.ZMagic THEN
    headerSize := UltrixRoundUp(headerSize);
    e.text     := UltrixRoundUp(e.text);
    e.data     := UltrixRoundUp(e.data)
  END;
  IF LAST(UxTypes.CompatibilityMode) < e.mode THEN
    CloseQuietly(file);
    RAISE(OS.Error, OS.BadExecutableEC)
  END;
  IF OS.Seek(file, headerSize) = 0 THEN END;

    (* Check that lengths in header are consistent with length of file.
       *)
    IF fileInfo.length < headerSize + e.text + e.data THEN
      CloseQuietly(file);
      RAISE(OS.Error, OS.ShortExecutableEC)
    END;
    RETURN TRUE
  END IsADotOut;

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

PROCEDURE OpenInterpreter();
  (* File is assumed to be open to an executable file other than an object
     program. If its first line is of the form:
     '#![blanks]interpreter[blanks[args]]' and the named interpreter
     appears to contain an executable object program, we use it. Otherwise
     we raise ENOENT, ENOEXEC, or EACCES as appropriate. *)
  VAR
    c:           CHAR;        (* look ahead *)
    w:           WrV.T;       (* gathers interpreter name, arg *)
    iname, iarg: Text.T;      (* interpreter name, arg *)
    mode:        OS.FileMode;
    owner:       Text.T;
  PROCEDURE GetChar(): CHAR;
    BEGIN
      IF bufIndex = bufCount THEN
        bufCount := UxFile.ReadF(file, SBuf.New(buf));
        bufIndex := 0
      END;
      IF bufCount = 0 THEN RAISE(OS.Error, OS.BadExecutableEC) END;
      INC(bufIndex);
      RETURN (buf[bufIndex - 1])
    END GetChar;
  BEGIN
    TRY
      IF (GetChar() # "#") OR (GetChar() # "!") THEN
        RAISE(OS.Error, OS.BadExecutableEC)
      END;
      c := GetChar(); (* prime the look ahead *)
      WHILE c = " " DO c := GetChar() END;
      WrV.New(w);
      WHILE (c # "\n") AND (c # " ") DO
        WrV.PutChar(w, c);
        c := GetChar()
      END;
      iname := WrV.ToText(w); (* interpreter *)
      WHILE c = " " DO c := GetChar() END;
      WHILE c # "\n" DO
        WrV.PutChar(w, c);
        c := GetChar()
      END;
        (* Ultrix seems to include blanks in and after arg *)
    FINALLY
      CloseQuietly(file)
```

```
    END;
    iarg := WrV.ToText(w); (* arg *)
    file := UxFile.OpenForLoading(u, dir, iname, euserAuth);
    IF IsADotOut(mode, owner) THEN (* ignore this mode, owner *)
      IF Text.IsEmpty(iarg) THEN extraCount := 1 ELSE extraCount := 2 END;
      NEW(extras, extraCount);
      FetchArgs();
      extras^[0] := argv^[0];
      argv^[0]   := name;
      IF extraCount = 2 THEN extras^[1] := iarg END
    ELSE
      CloseQuietly(file);
      RAISE(OS.Error, OS.BadExecutableEC)
    END
  END OpenInterpreter;

PROCEDURE LoadChunk(addr: Addr; length: CARDINAL);
  (* Copies length bytes from file to address addr in address space of u.
     *)
  VAR
    n: CARDINAL;
  BEGIN
    n := UxFile.ReadF(file, SBuf.SNew(u^.space, addr, length));
    IF n # length THEN RAISE(OS.Error, OS.ShortExecutableEC) END
  END LoadChunk;

VAR
  zero:  ARRAY [0..6] OF CHAR;
  mode:  OS.FileMode;
  owner: Text.T;
  auth: Auth.T;
  argc: CARDINAL; (* number of argument strings (including extras) *)
  i:    INTEGER;  (* loop index *)
  sumArgSizes, sumEnvSizes, padSize: CARDINAL; (* in bytes *)
  dataAddr, bssAddr, minBreakAddr, breakAddr, addr, stringAddr, stackAddr:
    Addr;
      (* user-space addresses in u^.space *)
  execVE: BOOLEAN;            (* TRUE iff execve not following vfork *)
  state:  TPSpecial.State;
  handle: ThreadsPort.Handle; (* to strace *)

BEGIN (* Load *)

  ZERO(ADDR(zero), BYTESIZE(zero));

  a := u^.a;

  argsFetched := FALSE;

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

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

  TRY (* FINALLY *)

    FetchArgs();

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

    dataAddr := TextAddr + e.text;
```

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

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

    argc := extraCount + NUMBER(argv^);

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

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

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

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

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

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

    TRY (* EXCEPT *)

      u^.compatibilityMode := e.mode;
      execVE               := FALSE;
      IF u^.space = VMFriends.NullSpace THEN (* StartProcess *)
        AllocThread(u);
        AllocSpace(u, bssAddr, breakAddr, initialStackAddr)
      ELSE (* ExecVE *)
        IF vFork THEN (* following VFork *)
            (* Serialize w.r.t. GetProcessInfo from another process *)
          UxPro.AcquireProcess(u);
          TRY AllocSpace(u, bssAddr, breakAddr, initialStackAddr) FINALLY
            UxPro.ReleaseProcess(u)
          END
        ELSE (* ExecVE not following VFork *)
          execVE := TRUE;
          TDState.PreExec(u^.pid.i, u^.space);

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

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

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

```
        (* If the straceExec flag is set, clear it and cause the process to
           appear to have gotten a "help" trap right after the execve. *)
        IF a^.straceExec # OSFriends1.STNever THEN
          handle          := a^.currentThread;
          a^.continuation := Nothing;
          IF a^.straceExec = OSFriends1.STOnce THEN
            a^.straceExec := OSFriends1.STNever
          END
        ELSE
          handle := nullTHandle
        END;
        IF vFork THEN NOT execVE (* vfork a StartProcess *)
          TDState.NewTarget(u^.pid.i, u^.space, handle)
        ELSIF [illegible] THEN
          TDState.PostExec(u^.pid.i, u^.space, handle)
        END
      EXCEPT SBuf.Fault, OS.Error:
        IF vFork AND (u^.space # VMFriends.NullSpace) THEN NOT execVE (* vfork a StartProcess *)
          TDState.NewTarget(u^.pid.i, u^.space, nullTHandle)
        END;
        RETURN FALSE
      END
    FINALLY
      CloseQuietly(file)
    END;
    RETURN TRUE; (* success *)
  END Load;

PROCEDURE MakeCoreImage(u: Ux.T; sig: OS.Signal)
  RAISES {OS.Error, Thread.Alerted};
  BEGIN
    IF (u^.space = VMFriends.NullSpace) THEN
      (* ignore *)
    ELSIF (u^.a # NIL) AND (u^.a^.isVulcan) THEN
      VuAS.WriteCoreFile (u, sig, u^.a^.rLimCore.cur, u^.a^.currentThread);
    ELSE
      WriteUltrixCoreFile (u, sig);
    END;
  END MakeCoreImage;

PROCEDURE WriteUltrixCoreFile(u: Ux.T; sig: OS.Signal)
  RAISES {OS.Error, Thread.Alerted};

  VAR
    f: OS.File;

  PROCEDURE W(p: CARDINAL; v: UNSIGNED; l: CARDINAL := 4);
    (* Write l bytes of v at position p in f. *)
    BEGIN
      IF OS.Seek(f, p) = 0 THEN END;
      UxFile.WriteF(f, SBuf.New(v, 0, l))
    END W;

  VAR
    space: VMFriends.Space;
    pos:   CARDINAL;        (* position in f, for use with Seek *)

  CONST
    MaxChunkSize =
      (1000000 DIV Hardware.BytesPerPage * Hardware.BytesPerPage);
      (* check for alerts at least this often *)

  PROCEDURE CarefullyWrite(addr, limitAddr: Addr);
    (* Write bytes from addr through limitAddr-1 of space to f, substituting
       zeros for unmapped or undefined pages. Maintain pos. *)

    TYPE
      StateSet = SET OF VM.Contents;
    CONST
      Unreal = StateSet{VM.Nonexistent, VM.Undefined};
    VAR
      pageState: VM.PageState;
      runAddr:   Addr;
      count:     CARDINAL;

    BEGIN
      ASSERT(addr MOD Hardware.BytesPerPage = 0);
      WHILE addr < limitAddr DO
          (* Although the page counts in the core file format are defined in
             terms of hardware pages, we can only ask VM about its pages. To
             keep things simple, we ask about each hardware page in a VM
             page. *)
        pageState := VMFriends.StatePage(space, addr DIV VMBytesPerPage);
        IF pageState.contents IN Unreal THEN
            (* Seek forward one page, leaving zeros. *)
          INC(addr, Hardware.BytesPerPage);
          INC(pos, Hardware.BytesPerPage);
          IF OS.Seek(f, pos) = 0 THEN END
        ELSE
            (* Write run of mapped pages. *)
          runAddr := addr;
          count   := 0;
          LOOP
            INC(addr, Hardware.BytesPerPage);
            INC(count, Hardware.BytesPerPage);
            IF (addr = limitAddr) OR (MaxChunkSize <= count) THEN EXIT END;
            pageState := VMFriends.StatePage(space, addr DIV VMBytesPerPage);
            IF pageState.contents IN Unreal THEN EXIT END
          END;
          TRY
            UxFile.WriteF(f, SBuf.SNew(space, runAddr, count));
          EXCEPT
          | SBuf.Fault:
              (* VM problem: probably a read failure on backing store,
                 or running out of page table space due to fragmentation. *)
              RAISE(OS.Error, OS.IOErrorEC);
          END;
          INC(pos, count)
        END;
        IF Thread.TestAlert() THEN RAISE(Thread.Alerted) END
      END
    END CarefullyWrite;

VAR
  a:     T;
  count: CARDINAL;
  i:     CARDINAL;
  dataAddr, breakAddr, stackAddr: Addr;
  dataBytes, stackBytes:          CARDINAL;
  tME:     UxTypes.CoreFileThreadMapEntry;
  success: BOOLEAN;

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

```
dataBytes  := (breakAddr - dataAddr);
stackBytes := (HoleAddr - stackAddr);

count := UserAreaVax.ByteSize + dataBytes + stackBytes +
         BYTESIZE(tME) * TPFriends.NHandles(space);
IF a^.rLimCore.cur < count THEN
    (* Raising any error clears core dump bit in wait status. *)
  RAISE(OS.Error, OS.NotEnoughRoomEC)
END;

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

f       := UxFile.OpenCoreFile(u, count);
success := FALSE;

TRY (* FINALLY *)

    (* Write header. *)
    (* First the fields relative to the beginning of the user area: *)
  W(UserAreaVax.oPcbKsp, 7fffffffH); (* kernel stack pointer *)
  W(UserAreaVax.oPcbUsp, tME.state.machineState.regs[Hardware.SPReg].val);
    (* user stack pointer *)
  W(UserAreaVax.oRegOff, UserAreaVax.ByteSize);
  W(UserAreaVax.oStatus, ORD(sig), BYTESIZE(UxTypes.WStatus));
  W(UserAreaVax.oTextPages,
    a^.firstWritableAddr DIV Hardware.BytesPerPage);
  W(UserAreaVax.oDataPages, dataBytes DIV Hardware.BytesPerPage);
  W(UserAreaVax.oStackPages, stackBytes DIV Hardware.BytesPerPage);

    (* Now the fields relative to the end of the user area: *)
  W(UserAreaVax.ByteSize + UserAreaVax.oAp,
    tME.state.machineState.regs[Hardware.APReg].val);
  W(UserAreaVax.ByteSize + UserAreaVax.oFp,
    tME.state.machineState.regs[Hardware.FPReg].val);
  FOR i := 0 TO Hardware.APReg - 1 DO
    W(UserAreaVax.ByteSize + UserAreaVax.oRegs + 4 * i,
      tME.state.machineState.regs[i].val)
  END;
  W(UserAreaVax.ByteSize + UserAreaVax.oSp,
    tME.state.machineState.regs[Hardware.SPReg].val);
  W(UserAreaVax.ByteSize + UserAreaVax.oPc,
    tME.state.machineState.regs[Hardware.PCReg].val);
  W(UserAreaVax.ByteSize + UserAreaVax.oPsw,
    LOOPHOLE(tME.state.machineState.status, UNSIGNED));
  pos := UserAreaVax.ByteSize;
  IF OS.Seek(f, pos) = 0 THEN END;

    (* Write data pages. *)
  TRY
    WHILE dataAddr < breakAddr DO
      count := MIN(breakAddr - dataAddr, LOOPHOLE(MaxChunkSize, Addr));
      UxFile.WriteF(f, SBuf.SNew(space, dataAddr, count));
      INC(dataAddr, count);
      INC(pos, count);
      IF Thread.TestAlert() THEN RAISE(Thread.Alerted) END
    END
  EXCEPT SBuf.Fault:
    CarefullyWrite(dataAddr, breakAddr)
  END;

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

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

    success := TRUE
  FINALLY
    IF NOT success THEN UxFile.RemoveCoreFile(u) END;
    OS.Close(f)
  END
END WriteUltrixCoreFile;

PROCEDURE New(u, uNew: Ux.T) RAISES {};
  VAR
    a: T;
  BEGIN
    NEW(uNew^.a);
    uNew^.space := VMFriends.NullSpace; (* see Copy, Load *)
    WITH uNew^.a^ DO
      traps           := MyTraps;
      tDStateAcquired := 0;
      currentThread   := nullTHandle;
      isVulcan        := FALSE;
      IF u = NIL THEN (* first process *)
        straceExec    := OSFriends1.STNever;
        straceFork    := OSFriends1.STNever;
        rLimData.cur  := initRLimDataCur;
        rLimData.max  := initRLimDataMax;
        rLimStack.cur := initRLimStackCur;
        rLimStack.max := initRLimStackMax;
        rLimCore.cur  := initRLimCoreCur;
        rLimCore.max  := initRLimCoreMax
      ELSE (* subsequent processes *)
        a          := u^.a;
        straceExec := a^.straceExec;
        straceFork := a^.straceFork;
        rLimData   := a^.rLimData;
        rLimStack  := a^.rLimStack;
        rLimCore   := a^.rLimCore
      END
    END
  END New;

PROCEDURE NewTarget(u, uNew: Ux.T) RAISES {};
  VAR
    handle: ThreadsPort.Handle;
  BEGIN
    IF uNew^.a^.straceExec # OSFriends1.STNever THEN
      handle := uNew^.a^.currentThread
    ELSE
      handle := nullTHandle
    END;
    TDState.NewTarget(uNew^.pid.i, uNew^.space, handle);

    (* Something like this would be necessary for straceFork. But if the
       parent has done a StartProcess, continuation is irrelevant. |      IF
```

Move NewTarget to private section; remove from UxAS.def !!

```
      u^.a^.straceFork # OSFriends1.STNever THEN |        IF u^.a^.straceFork
      = OSFriends1.STOnce THEN |          u^.a^.straceFork =
      OSFriends1.STNever |        END |        TDState.PostFork(u^.pid.i,
      u^.space, u^.currentThread, uNew^.pid.i, |          uNew^.space) |
      u^.a^.continuation := Nothing END *)

  END NewTarget;

PROCEDURE PushHandlerCall(
  u:            Ux.T;
  signal:       OS.Signal;
  code:         INTEGER;
  handler:      UxTypes.SignalHandler;
  mask:         OS.Signals;
  onStack:      UxTypes.SignalOnStack;
  switchStacks: BOOLEAN;
  sp:           Addr
  ) RAISES {SBuf.Fault};
  VAR
    a:     T;
    state: TPSpecial.State;
  PROCEDURE Push(w: WORD); (* onto stack of u *)
    VAR
      sp: Addr;
    BEGIN
      sp := state.machineState.regs[Hardware.SPReg].val - 4;
      TRY Put(u, sp, w) EXCEPT SBuf.Fault:
        IF GrowStack(u, sp) THEN Put(u, sp, w) ELSE RAISE(SBuf.Fault) END
      END;
      state.machineState.regs[Hardware.SPReg].val := sp
    END Push;
  VAR
    scp: Addr; (* SigContext pointer *)

  BEGIN
    AcquireTDState(u);
    TRY
      a := u^.a;
      TPSpecial.GetState(a^.currentThread, state);
      IF a^.continuation = Resume THEN
          (* Make sure correct carry bit is pushed in SigContext. *)
        state.machineState.status.cc.c := a^.sState.error
      END;
      TRY
          (* Push SigContext on current stack, and save pointer in scp. *)
        Push(state.machineState.status); (* PSL *)
        Push(state.machineState.regs[Hardware.PCReg].val);
        Push(state.machineState.regs[Hardware.SPReg].val); (* = .+4 *)
        Push(mask);
        Push(onStack);
        scp := state.machineState.regs[Hardware.SPReg].val;

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

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

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

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

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

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

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

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

```
    END;
    RETURN count
  END PutText;

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

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

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

PROCEDURE SetClientPriority(u: Ux.T; priority: TPFriends.Priority) RAISES {};
  VAR
    space:  CARDINAL;
    handle: ThreadsPort.Handle;
    state:  TPSpecial.State;
  BEGIN
    AcquireTDState(u);
    space := u^.space;
    TPSpecial.StopAll(space);
    handle := nullTHandle;
    LOOP
      handle := TPFriends.NextHandleInSpace(space, handle);
      IF ThreadsPort.Equal(handle, nullTHandle) THEN EXIT END;
      TPSpecial.GetState(handle, state);
      state.priority := priority;
      TPSpecial.SetState(handle, state)
    END;
    TPSpecial.StartAll(space);
    ReleaseTDState(u)
  END SetClientPriority;

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

PROCEDURE SetInitialClientPriority(u: Ux.T; priority: TPFriends.Priority)
  RAISES {};
  VAR
    handle: ThreadsPort.Handle;
    state:  TPSpecial.State;
  BEGIN
    handle := u^.a^.currentThread;
    TPSpecial.GetState(handle, state);
    state.priority := priority;
    TPSpecial.SetState(handle, state)
  END SetInitialClientPriority;

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

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

PROCEDURE StartAll(u: Ux.T) RAISES {};
```

```
  BEGIN
    TPSpecial.StartAll(u^.space);
      (* at least one more start required to get u^.a^.currentThread going *)
    ReleaseTDState(u)
  END StartAll;

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

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

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

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

    a := u^.a;

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

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

           In any case, it is only single-threaded address spaces that are
           allowed to have Ultrix-style signal handlers; it is to prepare for
           such a handler that we must stop the address space. *)
      TRY
        IF a^.continuation = Resume THEN
          a^.currentThread :=
            TPSpecial.ResumeKernelCaller(
              a^.currentThread, a^.sState, u^.space, a^.traps, tState)
        ELSE
          IF a^.continuation = Nothing THEN
              (* It is highttd's responsibility to start this thread. *)
          ELSE (* a^.continuation = Start *)
            TPSpecial.Start(a^.currentThread);
            ReleaseTDState(u)
          END;
          a^.currentThread :=
            TPSpecial.GetKernelCaller(u^.space, a^.traps, tState)
        END;
        IF (tState.trapInfo.kind = TPSpecial.TKIllegalNubCall) AND
           (tState.trapInfo.arg # ORD(UxTypes.SYShandlerreturn))
        THEN
          a^.continuation := Resume
        ELSE
          a^.continuation := Start;
          AcquireTDState(u);
          TPSpecial.GetState(a^.currentThread, state)
        END
      EXCEPT Thread.Alerted: (* asynchronous signal to handle *)
        AcquireTDState(u);
        RPCLocal.Exclude(u^.space, ExcludedStop, a);
        a^.continuation := Start;
        TPSpecial.GetState(a^.currentThread, state);
        tState.trapInfo := state.trapInfo;
        IF NOT tState.trapInfo.reported AND
           (tState.trapInfo.kind IN a^.traps)
        THEN
          TPSpecial.Start(a^.currentThread);
```

```
        Thread.Alert(Thread.Self()); (* pretend trap came first *)
        IF (tState.trapInfo.kind = TPSpecial.TKIllegalNubCall) AND
           (tState.trapInfo.arg # ORD(UxTypes.SYShandlerreturn))
        THEN
          ap := state.machineState.regs[Hardware.APReg].val;
          TRY Get(u, ap, tState.arguments) EXCEPT SBuf.Fault:
            Get(u, ap, tState.arguments.n);
            Get(u, ap + 4, tState.arguments.a, 0, tState.arguments.n)
          END;
          tState.validArguments   := TRUE;
          state.trapInfo.reported := TRUE;
          TPSpecial.SetState(a^.currentThread, state);
          ReleaseTDState(u);
          a^.continuation := Resume
        END
      ELSE
          (* The thread does not have a trap waiting for us to handle. *)
        tState.trapInfo.kind := TPSpecial.TKNoTrap
      END
    END;

    IF a^.continuation = Resume THEN
      a^.sState.setR1 := FALSE;
      a^.sState.error := FALSE;
      IF NOT tState.validArguments THEN
          (* The C library routine signal(3) calls sigvec(2) with AP < SP,
             which worries ThreadsPort. Let's try to fetch the arguments
             ourselves. (In the case noted above, AP points to a zero
             argument count, so use the maximum. *)
        TPSpecial.GetState(a^.currentThread, state);
        Get(
          u, state.machineState.regs[Hardware.APReg].val, tState.arguments)
      END;
      event.kind := KernelCall;
      IF (ORD(FIRST(UxTypes.KernelEntry)) <= tState.trapInfo.arg) AND
         (tState.trapInfo.arg <= ORD(LAST(UxTypes.KernelEntry)))
      THEN
        event.entry := VAL(UxTypes.KernelEntry, tState.trapInfo.arg)
      ELSE
        event.entry := UxTypes.SYSspare0; (* illegal, in range *)
      END;
      ZERO(ADDR(event.argList), BYTESIZE(event.argList));
      COPY(ADDR(tState.arguments.a), ADDR(event.argList),
           MIN(tState.arguments.n, NUMBER(tState.arguments.a)) *
           BYTESIZE(tState.arguments.a[0]));
        (* ***** Kludge "Dummy()" for timing purposes: *)
      IF tState.trapInfo.arg = ORD(UxTypes.SYSspare29) THEN
        handledHere := TRUE
      END;
        (* ***** End "Dummy() kludge. *)
    ELSIF tState.trapInfo.kind IN a^.traps THEN
      ClassifyTrap(); (* Exception or HandlerReturn *)
    ELSE
      event.kind := Alert
    END;
  UNTIL NOT handledHere
END WaitEvent;


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

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

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

PROCEDURE USBreak(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR
    a:         T;
    breakAddr: Addr;
  BEGIN
    a := u^.a;
    IF Addr(a^.rLimData.cur) < args[0] THEN
      RAISE(OS.Error, OS.NotEnoughVMEC); (* mapped to ENOMEM *)
    END;
    breakAddr := UltrixRoundUp(args[0]);
    SetBoundary(u^.space, VM.Low, breakAddr);
    RETURN 0
  END USBreak;

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

PROCEDURE UVulcan(u: Ux.T; VAR args: Ux.ArgList): INTEGER;
  VAR a: T;  boot, link: ADDRESS;  status: INTEGER;
  BEGIN
    a := u^.a;
    IF (a # NIL) AND (NOT a^.isVulcan) THEN
      a^.isVulcan := TRUE;
      status := VuAS.Connect (u, args, boot, link);
      IF (status < 0) THEN
        SetError (u);
        RETURN status;
      ELSE
        R1Result (u, LOOPHOLE (link, INTEGER));
        RETURN LOOPHOLE (boot, INTEGER);
      END;
    END;
    SetError (u);
    RETURN -1;
  END UVulcan;

(*****************************************************************)
(* Procedures exported thru OSFriends1 via RemoteOSAS and RemoteOS *)
(*****************************************************************)

PROCEDURE SetSTrace(u: Ux.T; exec, fork: OSFriends1.STrace) RAISES {};
  BEGIN
```

```
    u^.a^.straceExec := exec;
    u^.a^.straceFork := fork
  END SetSTrace;

BEGIN
  initialized := FALSE
END UxAS.
```