```
(*

M. D. Schroeder -- started June, 1985

UnixProcess.mod is the main control program of the unix 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, unix kernel
calls, and the subsequent return.  UnixProcess.mod also contains the
implementing procedures for the following unix kernel calls: execve,
exit, fork, getprgp, getpid, getuid, kill, killpg, setpgrp, setreuid,
sigblock, sigpause, setsigmask, sigstack, sigvec, vfork and wait.  It
dispatches the remaining unix kernel calls to implementing procedures
in other modules.

UnixProcess 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 UnixProcess.mod
reacquire the global lock.

The unix kernel simulator recognizes two kinds of address spaces --
unix and topaz.  A unix 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).

*)

IMPLEMENTATION MODULE UnixProcess;

IMPORT RdV, ThreadsPort, UnixFile, UnixProcessFile, VM, WrV;

TYPE
  OpState  = (Running, Stopped, Terminated);
  PrivateT = REF Private;
  Private = RECORD (* private state for UnixProcess *)
  (* chain of process states *)
    next: T;
  (* notification *)
    c: Thread.Condition;
  (* resources *)
    wThread: Thread.T; (* watcher thread for this process *)
    aSpace: Hardware.Space; (* address space for this process *)
    cHandle: ThreadsPort.Handle; (* thread for current action *)
  (* state of process *)
    topaz: BOOLEAN; (* F => unix process; T => topaz process *)
    opState: OpState;
    ws: Unix42.WStatus;
  (* signal stuff *)
    mask: Unix42.SigMask; (* current signal mask for process *)
    posted: Unix42.SigMask; (* signals waiting to be delivered *)
    fpeCode: INTEGER; (* code for last SigFPE *)
    illCode: INTEGER; (* code for last SigIll *)
    sigStack: Unix42.SigStackRec;
    sigVec: ARRAY [Unix42.FirstSig .. Unix42.LastSig] OF Unix42.SigVecRec;
  (* identification *)
    pid: ProcessID; (* process id of self -- IMMUTABLE *)
    ppid: ProcessID; (* process id of parent *)
    pgrp: ProcessID; (* process id of process group leader *)
    uid: UserID; (* real user id of responsible principal *)
    euid: UserID; (* user id for access control *)
  END;


(* Internal TYPEs and CONSTs *)

TYPE
  TrapClass = (
    SignalReturn, (* signal handler is returning *)
    Exception, (* synchronous error within process *)
    KernelCall, (* kernel call from process *)
    NoTrap (* no trap occurred *)
    );
  TrapKind = RECORD
    CASE class: TrapClass OF
    | SignalReturn: mask: Unix42.SigMask;
    | Exception: sig: Unix42.Signal; code: INTEGER;
    | KernelCall: entry: Unix42.KernelEntry;
    | NoTrap:
    END;
  END;
  SigAct = (
    IgnoreSA, (* signal should be ignored *)
    DoSA, (* signal has handler *)
    StopSA, (* signal should stop process *)
    TerminateSA, (* signal should terminate process *)
    CoreImageSA (* signal should terminate with core image *)
    );

CONST
    (* for signals not in CoreImageSigs, IgnoreSigs, or StopSigs, the
    default action is termination *)
  CoreImageSigs = Unix42.SigMask {
      Unix42.SigQuit, Unix42.SigIll, Unix42.SigTrap, Unix42.SigIOT,
      Unix42.SigEMT, Unix42.SigFPE, Unix42.SigBus, Unix42.SigSegV,
      Unix42.SigSys };
    (* default action is terminate w/ core image *)
  IgnoreSigs = Unix42.SigMask {
      Unix42.SigUrg, Unix42.SigCont, Unix42.SigChld, Unix42.SigIO };
    (* default action is ignore *)
  StopSigs = Unix42.SigMask {
      Unix42.SigStop, Unix42.SigTStp, Unix42.SigTTIn, Unix42.SigTTOu };
    (* default action is stop *)
  PowerfulSigs = Unix42.SigMask {
      Unix42.SigKill, Unix42.SigStop, Unix42.SigCont };
    (* signals that can't be masked; must have default handler,
      except SigCont which also can be caught *)
  NoSigs = Unix42.SigMask{};


(* Global variables w/ initialization *)

VAR
  SigDfl: Unix42.SigHandler; (* should be CONST *)
  SigIgn: Unix42.SigHandler; (* should be CONST *)
  gx: Thread.Mutex; (* module lock *)
  nextID: ProcessID; (* next process id to assign *)
  pHead: T; (* head of list of processes *)
  kep: ARRAY Unix42.KernelEntry OF EntryProc; (* entry procs *)

PROCEDURE Init();
  VAR e: Unix42.KernelEntry;
  BEGIN
    SigDfl := LOOPHOLE(0, Unix42.SigHandler);
    SigIgn := LOOPHOLE(1, Unix42.SigHandler);
    Thread.InitMutex(gx);
    nextID := InitPID;
```

```
    pHead := NIL;
    FOR e := FIRST(Unix42.KernelEntry) TO LAST(Unix42.KernelEntry) DO
      kep[e] := NoProc;
    END;
    UnixFile.Init();
    kep[Unix42.SYSfork] := Fork;
    kep[Unix42.SYSgetpgrp] := GetPGRP;
    kep[Unix42.SYSgetpid] := GetPID;
    kep[Unix42.SYSgetuid] := GetUID;
    kep[Unix42.SYSkill] := Kill;
    kep[Unix42.SYSkillpg] := KillPG;
    kep[Unix42.SYSsetpgrp] := SetPGRP;
    kep[Unix42.SYSsetreuid] := SetREUID;
    kep[Unix42.SYSsigblock] := SigBlock;
    kep[Unix42.SYSsigpause] := SigPause;
    kep[Unix42.SYSsigsetmask] := SigSetMask;
    kep[Unix42.SYSsigstack] := SigStack;
    kep[Unix42.SYSsigvec] := SigVec;
    kep[Unix42.SYSspare66] := VFork;
    kep[Unix42.SYSwait] := Wait;
    RETURN;
  END Init;


(* EXPORTed procedures *)

PROCEDURE RaiseError(e: Unix42.SysError; s: Unix42.Signal := Unix42.SigNone)
    RAISES {Error};
  VAR a: ErrorArg;
  BEGIN
    a.e := e; a.s := s; RAISE(Error, a);
  END RaiseError;

PROCEDURE ForkExecve(ppid: ProcessID; name: Text.T;
    argv, envp: RefArrayOfText): ProcessID RAISES {Error};
  VAR
    p, pNew: T;
    pTNew: PrivateT;
    ea: ErrorArg;
  BEGIN
    IF ppid = 0 THEN
      p := NIL;
    ELSE
      LOCK gx DO p := TFromPID(ppid) END;
    END;
    pNew := NewT(p);
    pTNew := pNew^.pT;
    TRY
      IF NOT SetupForExec(pNew, name, argv, envp) THEN
        RaiseError(Unix42.EIO);
      END;
      LOCK gx DO
        IF (p=NIL) AND (pHead#NIL) THEN RaiseError(Unix42.EINVAL) END;
        IF p^.pT^.opState=Terminated THEN RaiseError(Unix42.ESRCH) END;
        pTNew^.pid := nextID; INC(nextID);
        pTNew^.next := pHead; pHead := pNew;
      END;
    EXCEPT | Error(ea):
      ReleaseProcessResources(pNew);
      RAISE(Error, ea);
    END;
    pTNew^.wThread := Thread.Fork(Watcher, pNew);
    RETURN pTNew^.pid;
  END ForkExecve;

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

PROCEDURE SigEffective(p: T; signo: Unix42.Signal): BOOLEAN RAISES {};
  VAR pT: PrivateT;
  BEGIN
    pT := p^.pT;
    LOCK gx DO
      RETURN (NOT (signo IN pT^.mask)) AND (pT^.ppid # InitPID)
          AND (SignalAction(pT, signo) # IgnoreSA);
    END;
  END SigEffective;

PROCEDURE SetEUID(p: T; euid: UserID) RAISES{};
  BEGIN
    LOCK gx DO p^.pT^.euid := euid END;
  END SetEUID;

PROCEDURE EUID(p: T): UserID RAISES {};
  BEGIN
    LOCK gx DO RETURN p^.pT^.euid END;
  END EUID;

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

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

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

PROCEDURE Register(entry: Unix42.KernelEntry; proc: EntryProc) RAISES {};
  BEGIN
    kep[entry] := proc;
  END Register;

PROCEDURE Get(p: T; a: UNSIGNED; VAR v: ARRAY OF System.Byte;
    offset: CARDINAL := 0; count: CARDINAL := System.MaxCard) RAISES {Error};
  BEGIN
  count := MIN(count, NUMBER(v)-offset);
  IF VM.ReadBlock(p^.pT^.aSpace, a, System.Adr(v[offset]), count) # count
      THEN RaiseError(Unix42.EFAULT) END;
  RETURN;
  END Get;

PROCEDURE Put(p: T; a: UNSIGNED; VAR v: ARRAY OF System.Byte;
```

```
    offset: CARDINAL := 0; count: CARDINAL := System.MaxCard) RAISES {Error};
  BEGIN
  count := MIN(count, NUMBER(v)-offset);
  IF VM.WriteBlock(p^.pT^.aSpace, a, System.Adr(v[offset]), count) # count
    THEN RaiseError(Unix42.EFAULT) END;
  RETURN;
  END Put;

PROCEDURE GetText(p: T; a: UNSIGNED): Text.T RAISES {Error};
  VAR
    wr: WrV.T;
    i, bufPos, bufCount: INTEGER;
    bufAdr: UNSIGNED;
    buf: ARRAY [0..127] OF CHAR;
  PROCEDURE NextBufPos();
    BEGIN
      INC(bufPos);
      IF bufPos >= bufCount THEN
        INC(bufAdr, bufCount);
        bufCount := VM.ReadBlock(p^.pT^.aSpace, bufAdr, System.Adr(buf[0]),
            System.TByteSize(CHAR) * NUMBER(buf));
        IF bufCount = 0 THEN RaiseError(Unix42.EFAULT) END;
        bufPos := 0;
      END;
    END NextBufPos;
  BEGIN
    bufAdr := a; bufCount := 0; bufPos := -1;
    WrV.New(wr);
    LOOP;
      NextBufPos();
      IF buf[bufPos] = 0C THEN EXIT END;
      WrV.PutChar(wr, buf[bufPos]);
    END;
    RETURN WrV.ToText(wr);
  END GetText;

PROCEDURE PutText(p: T; a: UNSIGNED; t: Text.T;
    count: CARDINAL := System.MaxCard): CARDINAL RAISES {Error};
  VAR
    buf: ARRAY [0..127] OF CHAR;
    rd: RdV.T;
    n: CARDINAL;
    aa: UNSIGNED;
    NullChar: CHAR;
  BEGIN
    NullChar := 0C;
    IF Text.Length(t)+1 <= count THEN count := Text.Length(t) + 1 END;
    RdV.FromText(rd, t);
    aa := a;
    WHILE a - aa < count DO
      n := RdV.GetSub(rd, buf);
      ASSERT(n # 0);
      Put(p, a, buf, 0, n);
      INC(a, n);
    END;
    IF a - aa < count THEN Put(p, a, NullChar) END;
    RETURN a - aa;
  END PutText;

PROCEDURE R1Result(p: T; result: INTEGER) RAISES {};
  BEGIN
    (* put result in R1 in the state *)
    RETURN;
  END R1Result;
```

But n will equal 0 when it is time to append null char.

Never true with loop as written.

```
(* Internal procedures *)

PROCEDURE R0Result(p: T; result: INTEGER) RAISES {};
  BEGIN
    (* put result in R0 in the state *)
    RETURN;
  END R0Result;

PROCEDURE GetArrayOfText(p: T; a: UNSIGNED): RefArrayOfText;
  VAR
    t: RefArrayOfText;
    i, bufPos, bufCount: INTEGER;
    bufAdr: UNSIGNED;
    buf: ARRAY [0 .. 31] OF UNSIGNED;
  PROCEDURE NextBufPos();
    BEGIN
      INC(bufPos);
      IF bufPos >= bufCount THEN
        INC(bufAdr, bufCount);
        bufCount := VM.ReadBlock(p^.pT^.aSpace, bufAdr, System.Adr(buf[0]),
            System.TByteSize(UNSIGNED) * NUMBER(buf));
        IF bufCount = 0 THEN RaiseError(Unix42.EFAULT) END;
        bufPos := 0;
      END;
    END NextBufPos;
  BEGIN
    bufAdr := a; bufCount := 0; bufPos := -1; i := 0;
    LOOP
      NextBufPos();
      IF buf[bufPos] = 0 THEN EXIT END;
      INC(i);
    END;
    NEW(t, i);
    IF bufAdr # a THEN bufAdr := a; bufCount := 0 END;
    bufPos := -1;
    FOR i := 0 TO HIGH(t^) DO
      NextBufPos();
      t^[i] := GetText(p, buf[bufPos]);
    END;
    RETURN t;
  END GetArrayOfText;

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

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

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

PROCEDURE PTFromPID(pid: ProcessID; esrch: BOOLEAN := TRUE): PrivateT;
  VAR p: T;
```

```
  BEGIN
    p := TFromPID(pid, esrch);
    RETURN p^.pT;
  END PTFromPID;

PROCEDURE PostSignal(pT: PrivateT; signo: Unix42.Signal);
  BEGIN
    ASSERT (pT^.opState # Terminated);
    IF (SignalAction(pT, signo)#IgnoreSA) OR
        ((signo=Unix42.SigCont) AND (pT^.opState=Stopped)) THEN
      INCL(pT^.posted, signo);
      IF NOT (signo IN pT^.mask) THEN Thread.Alert(pT^.wThread) END;
    END;
    RETURN;
  END PostSignal;

PROCEDURE SignalAction(pT: PrivateT; s: Unix42.Signal): SigAct;
    (* returns proper action for signal, ignoring signal mask *)
  VAR h: Unix42.SigHandler;
  BEGIN
    IF s = Unix42.SigNone THEN RETURN IgnoreSA END;
    h := pT^.sigVec[s].handler;
    IF h = SigDfl THEN
      IF s IN IgnoreSigs THEN RETURN IgnoreSA END;
      IF s IN StopSigs THEN RETURN StopSA END;
      IF s IN CoreImageSigs THEN RETURN CoreImageSA END;
      RETURN TerminateSA;
    END;
    IF h = SigIgn THEN RETURN IgnoreSA END;
    IF pT^.topaz THEN RETURN CoreImageSA END;
    RETURN DoSA;
  END SignalAction;

PROCEDURE GetTrapKind(h: ThreadsPort.Handle): TrapKind;
  BEGIN
  END GetTrapKind;

PROCEDURE Watcher(a: Thread.ForkeeArg): Thread.ForkeeReturn;
  VAR
    p: T;
    pT: PrivateT;
    sigs: Unix42.SigMask;
    s: Unix42.Signal;
    trap: TrapKind;
    wasAlerted, dummyB: BOOLEAN;
    ti: ThreadsPort.TrapInfo;
  BEGIN
    p := NARROW(a, T);
    pT := p^.pT;
    trap.class := NoTrap;
    TRY
      LOOP; (* main watcher loop *)
        LOCK gx DO
          s := Unix42.SigNone;
          CASE trap.class OF
          | SignalReturn:
              pT^.mask := trap.mask - PowerfulSigs;
          | KernelCall:
          | Exception:
              s := trap.sig; (* first signal to look for *)
              INCL(pT^.posted, s);
              IF s = Unix42.SigFPE THEN
                pT^.fpeCode := trap.code;
              ELSIF s = Unix42.SigIll THEN
                pT^.illCode := trap.code;
              END;
          | NoTrap:
          END;
          LOOP; (* deliver signals *)
            sigs := pT^.posted - pT^.mask;
            IF sigs = NoSigs THEN EXIT END;
            IF (s = Unix42.SigNone) OR NOT (s IN sigs) THEN
              s := Unix42.FirstSig;
              WHILE NOT (s IN sigs) DO INC(s) END;
            END;
            EXCL(pT^.posted, s);
            CASE SignalAction(pT, s) OF
            | IgnoreSA:
                s := Unix42.SigNone;
            | DoSA:
                ActOnSignal(pT, s); (* exceptions to catch? *)
                s := Unix42.SigNone;
            | StopSA:
                DoStop(pT, s);
                s := Unix42.SigCont;
            | TerminateSA:
                RaiseTerminate(pT, s, FALSE);
            | CoreImageSA:
                RaiseTerminate(pT, s, TRUE);
            END;
          END; (* deliver signals loop *)
          dummyB := Thread.TestAlert(); (* drain pending alerts *)
        END; (* LOCK gx *)
        ThreadsPort.Start(pT^.cHandle);
        wasAlerted := FALSE;
        TRY
          ThreadsPort.WaitForTrappedHandle(pT^.aSpace);
          IF NOT ThreadsPort.GetTrappedHandle(pT^.aSpace, ti) THEN
            ASSERT (FALSE);
          END;
          IF LOOPHOLE(pT^.cHandle, UNSIGNED) # LOOPHOLE(ti.h, UNSIGNED) THEN
            pT^.topaz := TRUE;
            pT^.cHandle := ti.h;
          END;
        EXCEPT | Thread.Alerted: (* asynchronous signal to handle *)
          ThreadsPort.Stop(pT^.cHandle);
          wasAlerted := TRUE;
        END;
        trap := GetTrapKind(pT^.cHandle);
        CASE trap.class OF
        | SignalReturn:
            (* restore state *)
        | KernelCall:
            IF wasAlerted THEN Thread.Alert(pT^.wThread) END;
            s := DoKernelCall(p, trap.entry); (*UExit raises Terminate*)
            IF s # Unix42.SigNone THEN
              trap.class := Exception;
              trap.sig := s;
            END;
        | Exception:
        | NoTrap:
        END;
      END; (* main watcher loop *)
    EXCEPT | Terminate: (* only exit from main loop *)
      DoTermination(p);
    END;
    RETURN NIL;
  END Watcher;

PROCEDURE DoKernelCall(p: T; entry: Unix42.KernelEntry): Unix42.Signal;
  VAR
    cnt: INTEGER;
```

```
    args: ArgList;
    ea: ErrorArg;
    result, i: INTEGER;
  BEGIN
    ea.s := Unix42.SigNone;
    (* copy argument count into cnt *)
    cnt := MIN(NUMBER(args), cnt MOD 256);
    (* copy cnt arguments into args *)
    FOR i := cnt TO HIGH(args) DO args[i] := 0 END;
    TRY
      result :=  kep[entry](p, args);
      (* put result in R0 in state*)
    EXCEPT
    | Error(ea):
        IF ea.e = Unix42.ok THEN
          ASSERT (ea.s # Unix42.SigNone);
          (* backup the pc is state for reexecution after signal ea.s *)
        ELSE
          (* put ea.e in R0 in state*)
          (* set the error flag in state *)
        END;
    | Thread.Alerted:
        (* backup the pc in state for reexecution after async. signals *)
    END;
    RETURN ea.s;
  END DoKernelCall;

PROCEDURE ActOnSignal(pT: PrivateT; s: Unix42.Signal);
  (* called with gx LOCKed *)
  BEGIN
    (* record current state on handler stack *)
    (* set state to do signal *)
    (* set up handler return on stack *)
    (* save pT^.mask on signal stack *)
    pT^.mask := pT^.mask + pT^.sigVec[s].mask;
    INCL(pT^.mask, s);
    RETURN;
  END ActOnSignal;

PROCEDURE DoStop(pT: PrivateT; s: Unix42.Signal);
  (* called with gx LOCKed *)
  VAR pTParent: PrivateT;
  BEGIN
    ThreadsPort.StopAll(pT^.aSpace);
      (* this is at least the 2nd stop for pT^.cHandle *)
    pT^.opState := Stopped;
    pT^.ws.wStopVal := Unix42.WSTOPPED;
    pT^.ws.wStopSig := s;
    pTParent := PTFromPID(pT^.ppid, FALSE);
    PostSignal(pTParent, Unix42.SigChld);
    Thread.Signal(pTParent^.c);
    LOOP;
      IF Unix42.SigKill IN pT^.posted THEN
        RaiseTerminate(pT, Unix42.SigKill, FALSE);
      END;
      IF Unix42.SigCont IN pT^.posted THEN EXIT END;
      TRY Thread.AlertWait(gx, pT^.c);
      EXCEPT | Thread.Alerted:
      END;
    END;
    s := Unix42.FirstSig;
    LOOP;
      IF (s IN StopSigs) AND (s IN pT^.posted) AND
          (SignalAction(pT, s) = StopSA) THEN
        EXCL(pT^.posted, s);
      END;
      IF s = Unix42.LastSig THEN EXIT END;
      INC(s);
    END;
(*
    ThreadsPort.StartAll(pT^.aSpace);
      (* at least one more start required to get pT^.cHandle going *)
*)
    pT^.opState := Running;
    RETURN;
  END DoStop;

PROCEDURE DoTermination(p: T);
  VAR
    pT: PrivateT;
    pp: T;
  BEGIN
    pT := p^.pT;
    IF pT^.ws.wCoreDump THEN (*UnixDesc.MakeCoreImage(p)*) END;
    LOCK gx DO
      IF Unix42.SigChld IN pT^.posted THEN
        PostSignal(PTFromPID(InitPID, FALSE), Unix42.SigChld);
      END;
      pp := pHead;
      (* (pT^.ppid = 0) => Init process is terminating *)
      WHILE pp # NIL DO
        IF pp^.pT^.ppid = pT^.pid THEN
            (* child of terminating process *)
          pp^.pT^.ppid := InitPID;
        ELSIF pp^.pT^.pid = pT^.ppid THEN
            (* parent of terminating process *)
          PostSignal(pp^.pT, Unix42.SigChld);
          Thread.Signal(pp^.pT^.c);
        END;
        pp := pp^.pT^.next;
      END;
      pT^.opState := Terminated;
    END;
    ReleaseProcessResources(p);
    RETURN;
  END DoTermination;

PROCEDURE ReleaseProcessResources(p: T);
  VAR opOK: BOOLEAN;
  BEGIN
    UnixFile.Exit(p^.dT);
    ThreadsPort.DestroyAll(p^.pT^.aSpace);
    opOK := VM.FreeSpace(p^.pT^.aSpace); ASSERT (opOK);
  END ReleaseProcessResources;

PROCEDURE DoKill(from, to: PrivateT; s: Unix42.Signal): BOOLEAN;
  PROCEDURE NotSigContOK(): BOOLEAN;
    VAR ppT: PrivateT;
    BEGIN
      IF s # Unix42.SigCont THEN RETURN TRUE END;
      ppT := to;
      LOOP;
        IF ppT^.ppid = from^.pid THEN RETURN FALSE END;
        IF ppT^.ppid = InitPID THEN RETURN TRUE END;
        ppT := PTFromPID(ppT^.ppid, FALSE);
      END;
    END NotSigContOK;
  BEGIN
    IF (from^.uid # SuperUID) AND (from^.euid # to^.euid) AND
      NotSigContOK() THEN RETURN FALSE END;
    PostSignal(to, s);
    RETURN TRUE;
```

```
  END DoKill;

PROCEDURE NewT(p: T): T;
  VAR
    pNew: T;
    s: Unix42.Signal;
  BEGIN
    NEW(pNew); NEW(pNew^.pT);
    WITH pNew^.pT^ DO (* private state of new process *)
      IF p = NIL THEN (* first process *)
        pNew^.dT := UnixFile.Fork(NIL);
        opState := Running; mask := NoSigs; sigStack.onStack := FALSE;
        FOR s := Unix42.FirstSig TO Unix42.LastSig DO
          sigVec[s].handler := SigDfl;
          sigVec[s].mask := NoSigs;
          sigVec[s].onStack := FALSE;
        END;
        ppid := 0; pgrp := InitPID;
        uid := SuperUID; euid := SuperUID;
      ELSE (* forked process *)
        pNew^.dT := UnixFile.Fork(p^.dT);
        LOCK gx DO pNew^.pT^ := p^.pT^ END; (* copy parent's state *)
        ppid := p^.pT^.pid; (* change parent id *)
      END;
      posted := NoSigs; (* posted sigs never inherited *)
      Thread.InitCondition(c);
      aSpace := VM.AllocSpace();
      IF aSpace = VM.NullSpace THEN RaiseError(Unix42.EAGAIN) END;
      cHandle := ThreadsPort.Create(NIL, 0, 0);
      topaz := FALSE;
      pid := 0;
    END;
    RETURN pNew;
  END NewT;

PROCEDURE IsTopaz(pT: PrivateT): BOOLEAN;
  BEGIN
    IF NOT pT^.topaz THEN
      (* IF multiple threads THEN pT^.topaz := TRUE
         ELSIF rpcInitialized THEN pT^.topaz := TRUE
         END;
      *)
    END;
    RETURN pT^.topaz;
  END IsTopaz;

PROCEDURE SetupForExec(p: T; name: Text.T; argv, envp: RefArrayOfText)
    :BOOLEAN;
  VAR
    s: Unix42.Signal;
    pT: PrivateT;
    opOK: BOOLEAN;
  BEGIN
    pT := p^.pT;
    opOK := UnixProcessFile.Load(p, pT^.aSpace, pT^.cHandle, name, argv, envp);
    LOCK gx DO (* set SigVec for the impending Exec *)
      FOR s := Unix42.FirstSig TO Unix42.LastSig DO
        WITH pT^.sigVec[s] DO
          IF handler # SigIgn THEN handler := SigDfl END;
          onStack := FALSE;
          mask := NoSigs;
        END;
      pT^.sigStack.onStack := FALSE;
      END;
    END;
    UnixFile.Exec(p^.dT);
    RETURN opOK;
  END SetupForExec;

PROCEDURE CopyAddressSpace(fromSpace, toSpace: VM.Space);
  BEGIN
  END CopyAddressSpace;

(* Kernel entry procedures *)

PROCEDURE NoProc(p: T; VAR args: ArgList): INTEGER;
  BEGIN
    RaiseError(Unix42.EINVAL, Unix42.SigSys);
  END NoProc;

PROCEDURE ExecVE(p: T; VAR args: ArgList): INTEGER;
  VAR
    pT: PrivateT;
    name: Text.T;
    argv, envp: RefArrayOfText;
  BEGIN
    pT := p^.pT;
    IF IsTopaz(pT) THEN RaiseError(Unix42.EINVAL) END;
    name := GetText(p, args[0]);
    argv := GetArrayOfText(p, args[1]);
    envp := GetArrayOfText(p, args[2]);
    IF NOT SetupForExec(p, name, argv, envp) THEN
      RaiseTerminate(pT, Unix42.SigKill, TRUE);
    END;
    RETURN 0;
  END ExecVE;

PROCEDURE UExit(p: T; VAR args: ArgList): INTEGER;
  BEGIN
    UnixFile.Exit(p^.dT);
    p^.pT^.ws.wRetCode := args[0] MOD 256;
    RaiseTerminate(p^.pT, Unix42.SigNone, FALSE)
  END UExit;

PROCEDURE Fork(p: T; VAR args: ArgList): INTEGER;
  VAR
    pT, pTNew: PrivateT;
    pNew: T;
    tState: ThreadsPort.State;
    opOK: BOOLEAN;
  BEGIN
    pT := p^.pT;
    IF IsTopaz(pT) THEN RaiseError(Unix42.EINVAL) END;
    pNew := NewT(p);
    pTNew := pNew^.pT;
    CopyAddressSpace(pT^.aSpace, pTNew^.aSpace);
    LOCK gx DO
      pTNew^.pid := nextID; INC(nextID);
      pTNew^.next := pHead; pHead := pNew;
    END;
    opOK := ThreadsPort.GetState(pT^.cHandle, tState); ASSERT (opOK);
    opOK := ThreadsPort.SetState(pTNew^.cHandle, tState); ASSERT (opOK);
    R0Result(pNew, p^.pT^.pid);
    R1Result(pNew, 1); (* non-zero means we're in the child *)
    pNew^.pT^.wThread := Thread.Fork(Watcher, pNew);
    R1Result(p, 0); (* 0 means we're in the parent *)
    RETURN pNew^.pT^.pid;
  END Fork;

PROCEDURE GetPGRP(p: T; VAR args: ArgList): INTEGER;
  VAR
```

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

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

PROCEDURE GetUID(p: T; VAR args: ArgList): INTEGER;
  BEGIN
    LOCK gx DO
      R1Result(p, LOOPHOLE(p^.pT^.euid, INTEGER));
      RETURN LOOPHOLE(p^.pT^.uid, INTEGER)
    END;
  END GetUID;

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

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

PROCEDURE SetPGRP(p: T; VAR args: ArgList): INTEGER;
  VAR
    pid, pgrp: ProcessID;
    pT, ppT: PrivateT;
  PROCEDURE NotDescendent(): BOOLEAN;
    VAR pppT: PrivateT;
    BEGIN
      pppT := ppT;
      LOOP;
        IF pppT^.ppid = pT^.pid THEN RETURN FALSE END;
        IF pppT^.ppid = InitPID THEN RETURN TRUE END;
        pppT := PTFromPID(pppT^.ppid, FALSE);
      END;
    END NotDescendent;
  BEGIN
    pT := p^.pT;
    pid := LOOPHOLE(args[0], ProcessID);
    pgrp := LOOPHOLE(args[1], ProcessID);
    LOCK gx DO
      IF pid = 0 THEN ppT := pT ELSE ppT := PTFromPID(pid) END;
      IF (pT^.uid # SuperUID) AND (pT^.euid # ppT^.euid) AND
          NotDescendent() THEN RaiseError(Unix42.EPERM) END;
      ppT^.pgrp := pgrp;
    END;
    RETURN 0;
  END SetPGRP;


PROCEDURE SetREUID(p: T; VAR args: ArgList): INTEGER;
  VAR
    uid, euid: UserID;
    pT: PrivateT;
  BEGIN
    pT := p^.pT;
    uid := LOOPHOLE(args[0], UserID);
    euid := LOOPHOLE(args[1], UserID);
    LOCK gx DO
      IF uid # -1 THEN
        IF pT^.uid = SuperUID THEN pT^.uid := uid
        ELSE RaiseError(Unix42.EPERM) END;
```

```
      END;
      IF euid # -1 THEN
        IF pT^.uid = SuperUID THEN pT^.euid := euid
        ELSE RaiseError(Unix42.EPERM) END;
      ELSE
        pT^.euid := pT^.uid
      END;
    END;
    RETURN 0;
  END SetREUID;

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

PROCEDURE SigPause(p: T; VAR args: ArgList): INTEGER;
  VAR
    pT: PrivateT;
    mask, sigs: Unix42.SigMask;
    s: Unix42.Signal;
  BEGIN
    pT := p^.pT;
    mask := LOOPHOLE(args[0], Unix42.SigMask) - PowerfulSigs;
    LOCK gx DO
      LOOP;
        sigs := pT^.posted - mask;
        IF sigs # NoSigs THEN EXIT END;
        TRY Thread.AlertWait(gx, pT^.c);
        EXCEPT | Thread.Alerted:
        END;
      END;
      FOR s := Unix42.FirstSig TO Unix42.LastSig DO
        IF s IN sigs THEN
          ActOnSignal(pT, s);
          RaiseError(Unix42.EINTR);
        END;
      END;
      ASSERT (FALSE);
    END;
  END SigPause;

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

PROCEDURE SigStack(p: T; VAR args: ArgList): INTEGER;
  VAR
    pT: PrivateT;
    ss: Unix42.SigStackRec;
  BEGIN
    pT := p^.pT;
    IF args[1] # 0 THEN Put(p, args[1], pT^.sigStack) END;
    IF args[0] # 0 THEN Get(p, args[0], ss) END;
    pT^.sigStack := ss;
    RETURN 0;
  END SigStack;

PROCEDURE SigVec(p: T; VAR args: ArgList): INTEGER;
  VAR
    pT: PrivateT;
    sig: Unix42.Signal;
    vec: Unix42.SigVecRec;
  BEGIN
    pT := p^.pT;
    IF (args[0] < ORD(Unix42.FirstSig)) OR
        (args[0] > ORD(Unix42.LastSig)) THEN
      RaiseError(Unix42.EINVAL);
    END;
    sig := LOOPHOLE(args[0], Unix42.Signal);
    IF args[2] # 0 THEN Put(p, args[2], pT^.sigVec[sig]) END;
    IF args[1] # 0 THEN
      Get(p, args[1], vec);
      IF (sig IN PowerfulSigs) AND
          ((sig # Unix42.SigCont) OR (vec.handler = SigIgn)) THEN
        RaiseError(Unix42.EINVAL);
      END;
      vec.mask := vec.mask - PowerfulSigs;
      LOCK gx DO pT^.sigVec[sig] := vec END;
    END;
    RETURN 0;
  END SigVec;

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

PROCEDURE Wait(p: T; VAR args: 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
    pT, ppT, ppTPrev: PrivateT;
    noChildren, noHang, unTraced: BOOLEAN;
  PROCEDURE WaitResults(): INTEGER;
    VAR ws32: BITS 32 FOR Unix42.WStatus;
    BEGIN
      IF args[2] # 0 THEN
        (* return rusage *)
      END;
      ws32 := ppT^.ws; (* 0's high order bytes *)
      RlResult(p, LOOPHOLE(ws32, INTEGER));
      RETURN LOOPHOLE(ppT^.pid, INTEGER)
    END WaitResults;
  BEGIN
    noHang := (Unix42.WNOHANG IN LOOPHOLE(args[1], Unix42.WOptions));
    unTraced := (Unix42.WUNTRACED IN LOOPHOLE(args[1], Unix42.WOptions));
    pT := p^.pT;
    LOCK gx DO
      LOOP;
        ppT := pHead^.pT; (* pHead can't be NIL *)
        ppTPrev := NIL;
        noChildren := TRUE;
```

```
      LOOP
        IF ppT^.ppid = pT^.pid THEN
          noChildren := FALSE;
          IF ppT^.opState = Terminated THEN
            IF ppTPrev = NIL THEN pHead := ppT^.next
            ELSE ppTPrev^.next := ppT^.next END;
            RETURN WaitResults();
          END;
          IF (ppT^.opState = Stopped) AND unTraced THEN
            RETURN WaitResults();
          END;
        END;
        IF ppT^.next = NIL THEN EXIT END;
        ppTPrev := ppT;
        ppT := ppT^.next^.pT;
      END;
      IF noChildren THEN RaiseError(Unix42.ECHILD) END;
      IF noHang THEN RETURN 0;
      Thread.AlertWait(gx, pT^.c); (* alert handled by DoKernelCall *)
      END;
    END;
  END;
END Wait;

(* Module initialization *)

BEGIN

  Init();

END UnixProcess.

Fri Aug 16 11:13:02 1985
```