```
-- Process.Mesa  Edited by Redell on July 31, 1978  3:39 PM

DIRECTORY
  ControlDefs: FROM "controldefs" USING [
    Frame, FrameHandle, Free, Lreg, NullFrame, SetReturnFrame, StateVector],
  InlineDefs: FROM "inlinedefs" USING [BITOR, BITSHIFT],
  NucleusDefs: FROM "nucleusdefs",
  ProcessDefs: FROM "processdefs" USING [
    ActiveWord, Broadcast, Clean, Condition, CurrentPSB, CurrentState,
    DefaultPriority, DefaultTimeout, DisableInterrupts, DIW, Empty,
    EnableAndRequeue, EnableInterrupts, Enter, Exit, MonitorLock, Notify,
    NullQueueHandle, Priority, ProcessHandle, PSB, ReadyList, ReEnter, SDC,
    Ticks, TimeoutLevel, TimerGrain, UnlockedEmpty, Wait],
  SDDefs: FROM "sddefs" USING [
    SD, sFirstProcess, sFirstStateVector, sFork, sJoin, sLastProcess,
    sProcessTrap];

Process: MONITOR LOCKS Processes
  EXPORTS NucleusDefs, ProcessDefs SHARES ProcessDefs =
  BEGIN OPEN ProcessDefs;

  PSBBase: CARDINAL = 0;

  Aborted: PUBLIC SIGNAL = CODE;
  TimedOut: PUBLIC SIGNAL = CODE;
  TooManyProcesses: PUBLIC ERROR = CODE;

  Processes: MONITORLOCK;
  frameReady, frameTaken, dead, rebirth: CONDITION;
  DyingFrameHandle: TYPE = POINTER TO dying ControlDefs.Frame;

  Fork: PUBLIC PROCEDURE [root: UNSPECIFIED] RETURNS [ProcessHandle] =
    BEGIN
    sv: ControlDefs.StateVector;
    self: ControlDefs.FrameHandle;
    Forker: PROCEDURE [ProcessHandle];
    newPSB: ProcessHandle;
    sv ← STATE;
    WHILE ~Enter[@Processes] DO NULL ENDLOOP;
    self ← REGISTER[ControlDefs.Lreg];
    Forker ← LOOPHOLE[self.returnlink];
    IF LOOPHOLE[rebirth, Condition].queue = Empty THEN
      BEGIN
      Exit[@Processes];
      ERROR TooManyProcesses;
      END;
    newPSB ← (PSBBase+LOOPHOLE[rebirth, Condition].queue).link;
    newPSB↑ ← PSB[
      link: newPSB.link,
      cleanup: Clean,
      timeout: 0,
      priority: CurrentPSB.priority,
      enterFailed: FALSE,
      detached: FALSE,
      fill:0,
      state: dead,      -- in case of timeout before Notify (below)
      timeoutAllowed: TRUE,
      abortPending: FALSE,
      timeoutPending: FALSE,
      waitingOnCV: TRUE,
      frame: self];
    Notify[@rebirth];   -- wake up newPSB, and set alive; DEPENDS
    newPSB.state ← alive;       -- on new process not preempting parent...
    ControlDefs.SetReturnFrame[ControlDefs.NullFrame];
    Forker[newPSB];      -- "returns" handle to site of FORK
    -- Note that the lines above are executed by the forking process, while
    -- the lines below are executed by the forked process. Note also that the
    -- monitor remains LOCKED during this fancy footwork...!
    sv.dest ← root;
    sv.source ← End;
    Exit[@Processes];
    RETURN WITH sv
    END;

  deadFrame: DyingFrameHandle ← NIL;  -- only for detached processes
  End: PROCEDURE =
    BEGIN OPEN p: CurrentPSB↑;
    sv: ControlDefs.StateVector;
    frame: DyingFrameHandle;
    sv ← STATE;
    WHILE ~Enter[@Processes] DO NULL ENDLOOP;
    frame ← REGISTER[ControlDefs.Lreg];
    frame.state ← alive;
    p.state ← frameReady;
    p.abortPending ← FALSE; -- too late for Aborts: they no-op
    Broadcast[@frameReady];
    UNTIL p.state = frameTaken OR p.detached DO
      Wait[@Processes, @frameTaken, LOOPHOLE[frameTaken, Condition].timeout];
      WHILE ~ReEnter[@Processes, @frameTaken] DO NULL ENDLOOP;
      ENDLOOP;
    IF deadFrame # NIL THEN
      BEGIN ControlDefs.Free[deadFrame]; deadFrame ← NIL END;
    IF p.detached THEN deadFrame ← frame; -- Leave our frame for freeing
    frame.state ← dead;
    p.state ← dead;
    Broadcast[@dead];
    Wait[@Processes, @rebirth, LOOPHOLE[rebirth, Condition].timeout];
    WHILE ~ReEnter[@Processes, @rebirth] DO NULL ENDLOOP; -- dying process exits here; JOINing process
**does below.
    sv.dest ← frame.returnlink; -- set to site of JOIN by Join
    sv.source ← 0;
    Exit[@Processes];
    RETURN WITH sv;
    END;

  Join: PUBLIC ENTRY PROCEDURE [process: UNSPECIFIED]
    RETURNS [ControlDefs.FrameHandle] =
    BEGIN
    p: ProcessHandle = process;
    frame: DyingFrameHandle;
    self: ControlDefs.FrameHandle = REGISTER[ControlDefs.Lreg];
    ValidateProcess[p];
    WHILE p.state # frameReady DO WAIT frameReady ENDLOOP;
    -- guaranteed to be a dying frame by the time we get here
    frame ← LOOPHOLE[p.frame];
    p.state ← frameTaken;
    BROADCAST frameTaken;
    WHILE frame.state # dead DO WAIT dead ENDLOOP;
    frame.returnlink ← self.returnlink; -- site of JOIN
    RETURN[frame]
    END;

  Detach: PUBLIC ENTRY PROCEDURE [process: UNSPECIFIED] =
    BEGIN
    p: ProcessHandle = process;
    ValidateProcess[p];
    p.detached ← TRUE;
    BROADCAST frameTaken;
    END;

  Abort: PUBLIC PROCEDURE [process: UNSPECIFIED] =
    BEGIN
    p: ProcessHandle = process;
    ValidateProcess[p];
    DisableInterrupts[];
    IF p.state = alive THEN
      BEGIN
      p.abortPending ← TRUE;
      IF p.waitingOnCV THEN
        BEGIN
        p.waitingOnCV ← FALSE;
        EnableAndRequeue[NullQueueHandle, ReadyList, p];
        RETURN;
        END;
      END;
    EnableInterrupts[];
    END;

  ProcessTrap: PROCEDURE RETURNS [BOOLEAN] =
    BEGIN
    abort, timeout: BOOLEAN;
    CurrentPSB.waitingOnCV ← FALSE;
    abort ← CurrentPSB.abortPending;
    CurrentPSB.abortPending ← FALSE;
    IF abort THEN ERROR Aborted;
    timeout ← CurrentPSB.timeoutPending;
    CurrentPSB.timeoutPending ← FALSE;
    IF timeout THEN SIGNAL TimedOut;
    RETURN[FALSE]
    END;

  DisableScheduling: PUBLIC PROCEDURE =
    BEGIN
    DisableInterrupts[];
    SDC↑ ← SDC↑ + 1;
    EnableInterrupts[];
    RETURN
    END;

  EnableScheduling: PUBLIC PROCEDURE =
    BEGIN
    DisableInterrupts[];
    SDC↑ ← SDC↑ - 1;
    EnableAndRequeue[ReadyList, ReadyList, CurrentPSB↑];
    RETURN
    END;

  Yield: PUBLIC PROCEDURE =
    BEGIN
    DisableInterrupts[];
    EnableAndRequeue[ReadyList, ReadyList, CurrentPSB↑];
    RETURN
    END;

  GetPriority: PUBLIC PROCEDURE RETURNS [p: Priority] =
    BEGIN
    DisableInterrupts[];
    p ← CurrentPSB.priority;
    EnableInterrupts[];
    RETURN
    END;

  SetPriority: PUBLIC PROCEDURE [p: Priority] =
    BEGIN
    DisableInterrupts[];
    CurrentPSB.priority ← p;
    EnableAndRequeue[ReadyList, ReadyList, CurrentPSB↑];
    END;

  SetTimeout: PUBLIC PROCEDURE [
    condition: POINTER TO CONDITION, ticks: CARDINAL] =
    BEGIN
    LOOPHOLE[condition, POINTER TO Condition].timeout ←
      IF ticks # 0 THEN ticks ELSE DefaultTimeout;
    RETURN
    END;

  DisableTimeout: PUBLIC PROCEDURE [condition: POINTER TO CONDITION] =
    BEGIN
    LOOPHOLE[condition, POINTER TO Condition].timeout ← 0;
    RETURN
    END;

  MsecToTicks: PUBLIC PROCEDURE [ms: CARDINAL] RETURNS [Ticks] =
    BEGIN
    RETURN[(ms+TimerGrain-1)/TimerGrain]
    END;

  TicksToMsec: PUBLIC PROCEDURE [ticks: Ticks] RETURNS [CARDINAL] =
    BEGIN
    RETURN[ticks*TimerGrain]
    END;

  InitializeMonitor: PUBLIC PROCEDURE [monitor: POINTER TO MONITORLOCK] =
    BEGIN
    LOOPHOLE[monitor, POINTER TO MonitorLock]↑ ← UnlockedEmpty;
    RETURN
    END;

  InitializeCondition: PUBLIC PROCEDURE [
    condition: POINTER TO CONDITION, ticks: CARDINAL] =
    BEGIN
    LOOPHOLE[condition, POINTER TO Condition]↑ ← Condition[no, Empty, ticks];
    RETURN
    END;

  ValidateProcess: PUBLIC PROCEDURE [p: ProcessHandle] =
    BEGIN OPEN SDDefs;
    sd: POINTER TO ARRAY [0..0) OF UNSPECIFIED ← SD;
    c: CARDINAL = LOOPHOLE[p];
    IF c < SD[sFirstProcess] OR c > sd[sLastProcess] OR
      (c - sd[sFirstProcess]) MOD SIZE[PSB] # 0 THEN SIGNAL InvalidProcess[p];
    RETURN
    END;

  InvalidProcess: PUBLIC SIGNAL [process: ProcessHandle] = CODE;

  InitializeProcesses: PROCEDURE =
    BEGIN OPEN SDDefs;
    sd: POINTER TO ARRAY [0..0) OF UNSPECIFIED ← SD;
    firstPSB: ProcessHandle ← sd[sFirstProcess];
    lastPSB: ProcessHandle ← sd[sLastProcess];
    psb: ProcessHandle;
    DisableTimeout[@dead];
    DisableTimeout[@frameReady];
    DisableTimeout[@frameTaken];
    DisableTimeout[@rebirth];
    SDC↑ ← 0;
    CurrentState↑ ← LOOPHOLE[sd[sFirstStateVector] +
      DefaultPriority*SIZE[ControlDefs.StateVector]];
    -- locate and initialize PSBs
    sd[sProcessTrap] ← ProcessTrap;
    sd[sFork] ← Fork;
    sd[sJoin] ← Join;
    -- fabricate PSB for self
    lastPSB↑ ← PSB[
      link: lastPSB,
      cleanup: Clean,
      timeout: 0,
      enterFailed: FALSE,
      detached: FALSE,
      fill: 0B,
      state: alive,
      timeoutAllowed:,
      abortPending: FALSE,
      timeoutPending: FALSE,
      waitingOnCV: FALSE,
      priority: DefaultPriority,
      frame: REGISTER[ControlDefs.Lreg]];
    CurrentPSB↑ ← ReadyList↑ ← lastPSB;
    -- set up free PSB pool ("rebirth" condition)
    FOR psb ← firstPSB, psb+SIZE[PSB]
    UNTIL psb = lastPSB DO
      psb↑ ← PSB[
        link: psb-SIZE[PSB],
        cleanup: Clean,
        timeout: 0,
        enterFailed: FALSE,
        detached: FALSE,
        fill: 0B,
        state: dead,
        timeoutAllowed:,
        abortPending: FALSE,
        timeoutPending: FALSE,
        waitingOnCV: TRUE,
        priority: DefaultPriority,
        frame: ControlDefs.NullFrame];
      ENDLOOP;
    LOOPHOLE[rebirth, Condition].queue ←
      (firstPSB.link ← lastPSB-SIZE[PSB])-PSBBase;
    -- CV↑ already set up by Nova code
    ActiveWord↑ ← 77777B;
    RETURN
    END;

  InitializeTimeouts: PROCEDURE =
    BEGIN OPEN InlineDefs;
    TimeoutMask: WORD = BITSHIFT[1,TimeoutLevel];
    DisableInterrupts[];
    -- Nova code has set up IntVec[TimeoutLevel]
    DIW↑ ← BITOR[DIW↑, TimeoutMask];
    EnableInterrupts[];
    RETURN
    END;


  InitializeProcesses[];
  InitializeTimeouts[];

  END.
```