4. The OS Kernel

4.1 Kernel Definitions and Objects

4.2 Queue Structures

4.3 Threads

4.4 Implementing Processes and Threads

- Process and Thread Descriptors
- Implementing the Operations

4.5 Implementing Synchronization and Communication Mechanisms

- Requesting and Releasing Resources
- Semaphores and Locks
- Building Monitor Primitives
- Clock and Time Management
- Communications Kernel

4.6 Interrupt Handling

Kernel Definitions and Objects

- Basic set of objects, primitives, data structures, processes
- Rest of OS is built on top of kernel
- Kernel defines/provides *mechanisms* to implement various *policies*
  - Process and thread management
  - Interrupt and trap handling
  - Resource management
  - Input/output

Queue Structures

- OS needs many different queues
- Single-level queues
  - Implemented as *array*
    - Fixed size
    - Efficient for simple FIFO operations
  - Implemented as *linked list*
    - Unbounded size
    - More overhead, but more flexible operations

Queues

- Multi-level queues (priority queues)
  - Support multiple priority levels
  - Implemented as *multiple single-level queues*
  - Implemented as *heap*

Priority Queues: Multiple queues

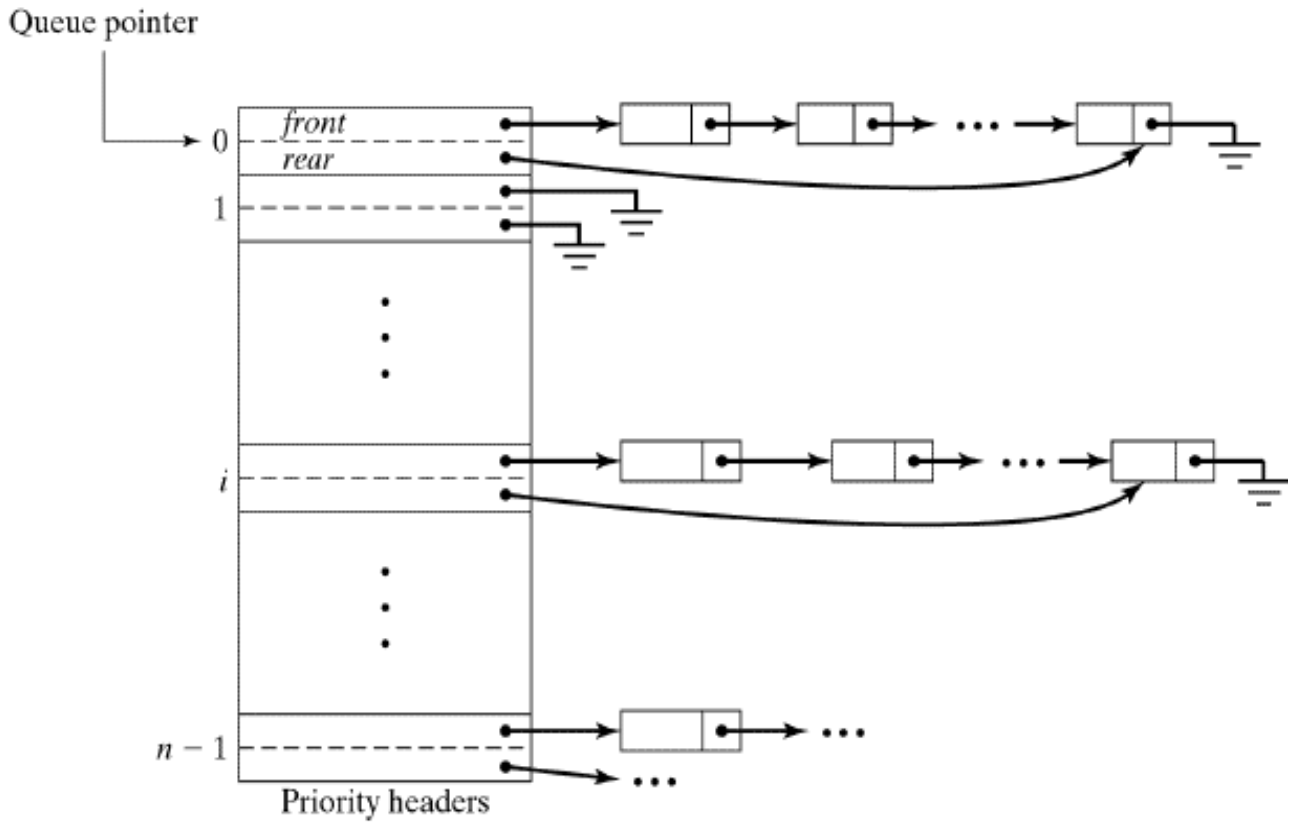

Figure 4-3(a)

Priority Queues: Binary Heap

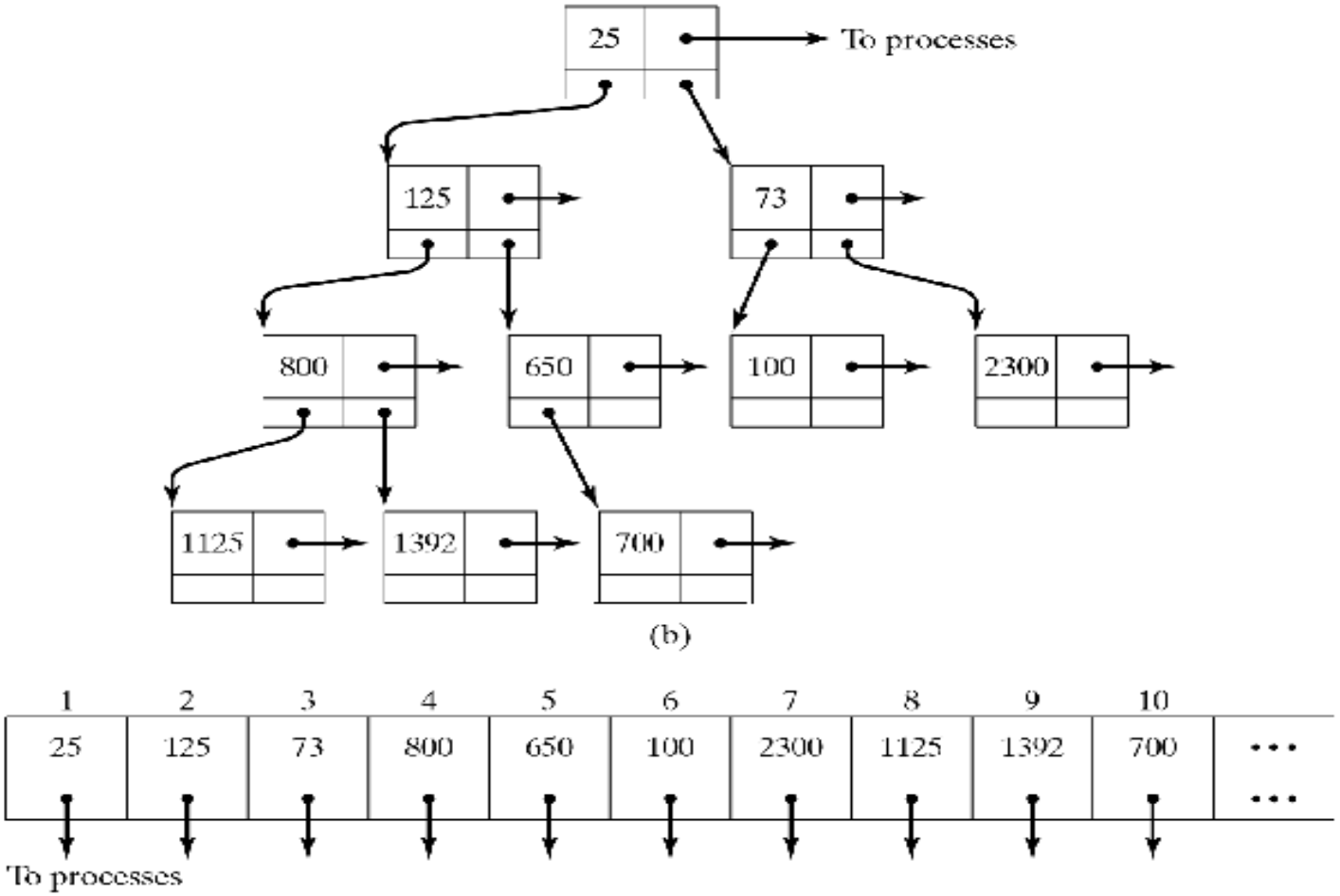

Figure 4-3(b)

Processes and threads

- Process has one or more threads
- All threads in a process share:
  - Memory space
  - Other resources
- Each thread has its own:
  - CPU state
    (registers, program counter)
  - Stack
- Implemented in user space or kernel space
- Threads are efficient, but lack protection from each other

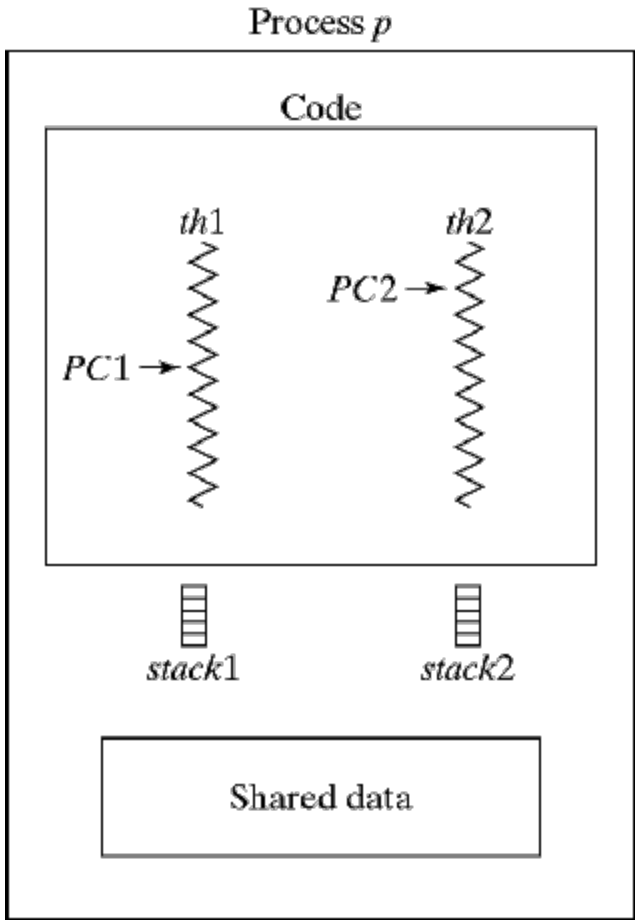

Figure 4-4

Process status types

Running / Ready / Blocked

- *Running*: the process is currently running on a processor
- *Ready*: the process is ready to run, waiting for a processor
- *Blocked*: the process cannot proceed until it is granted a particular resource (e.g., a lock, a file, a semaphore, a message, …)

Active / Suspended

- Internal process may *suspend* other processes to examine or modify their state (e.g., prevent deadlock, detect runaway process, swap the process out of memory…)

Implementing Processes and Threads

- Process Control Block (PCB)
  - State Vector = Information necessary to run process p
  - Status
    - Basic types: Running, Ready, Blocked
    - Additional types:
      - Ready_active, Ready_suspended
      - Blocked_active, Blocked_suspended

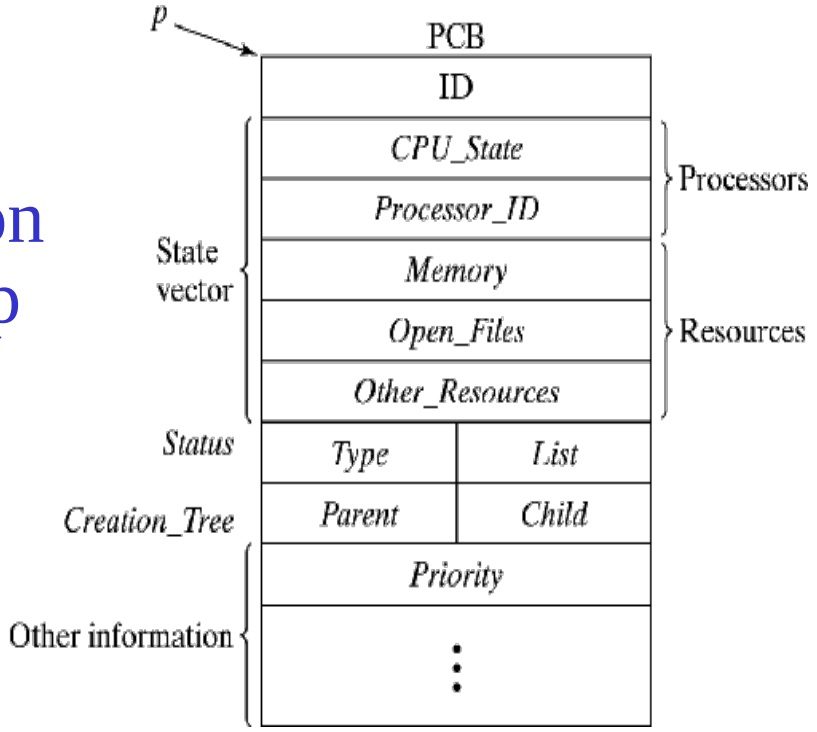

Figure 4-5

Implementing Processes and Threads

- State Transition Diagram

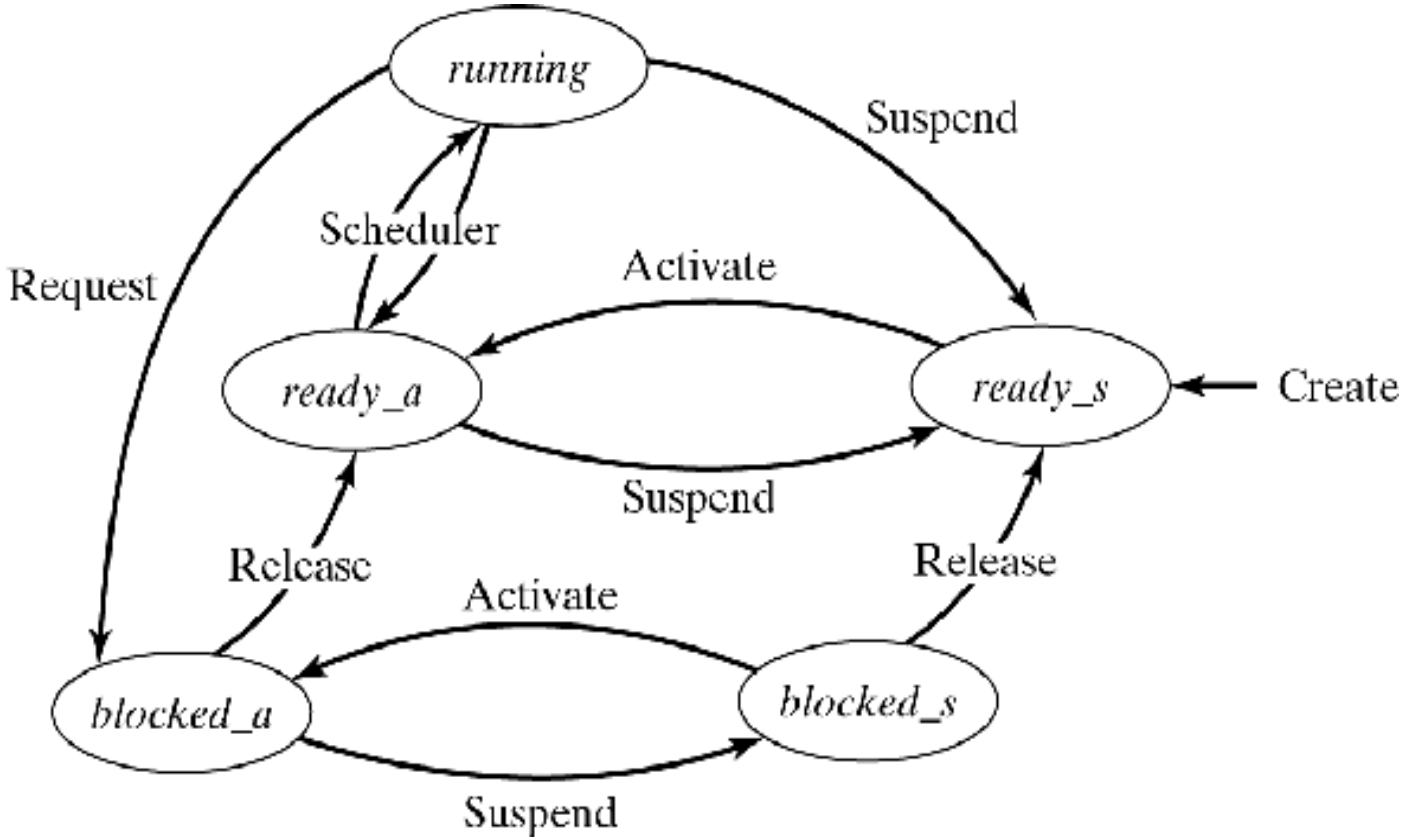

Figure 4-6

Process Operations: Create

```
Create(s0, m0, pi, pid)
{
    p = Get_New_PCB();  pid = Get_New_PID();
    p->ID = pid;        p->CPU_State = s0;
    p->Memory = m0;     p->Priority = pi;
    p->Status.Type = 'ready_s';
    p->Status.List = RL;
    p->Creation_Tree.Parent = self;
    p->Creation_Tree.Child = NULL;
    insert(self-> Creation_Tree.Child, p);
    insert(RL, p);
    Scheduler();
 }
```

Process Operations: Suspend

```
Suspend(pid)
{
   p = Get_PCB(pid);
   s = p->Status.Type;
   if ((s==’blocked_a’)||(s==’blocked_s’))
      p->Status.Type = ’blocked_s’;
   else p->Status.Type = ’ready_s’;
   if (s==’running’)
   {
      cpu = p->Processor_ID;
      p->CPU_State = Interrupt(cpu);
      Scheduler();
   }
}
```

Process Operations: Activate

```
Activate(pid)
{
   p = Get_PCB(pid);
   if (p->Status.Type == 'ready_s')
   {
      p->Status.Type = 'ready_a';
      Scheduler();
   }
   else p->Status.Type = 'blocked_a';
}
```

Process Operations: Destroy

```
Destroy(pid)
{  p = Get_PCB(pid); Kill_Tree(p); Scheduler();}

Kill_Tree(p)
{
   for (each q in p->Creation_Tree.Child)
      Kill_Tree(q);
   if (p->Status.Type == 'running')
   {
      cpu = p->Processor_ID; Interrupt(cpu);
   }
   Remove(p->Status.List, p);
   Release_all(p->Memory);
   Release_all(p->Other_Resources);
   Close_all(p->Open_Files);
   Delete_PCB(p);
}
```

Implementing Synchronization and Communication Mechanisms

- Semaphores, locks, monitors, messages, time, etc. are resources
- Generic code to request a resource:

```
Request(res)
{
  if (Free(res)) Allocate(res, self)
  else
  {
    Block(self, res);
    Scheduler();
  }
}
```

- Generic code to request a resource

```
Release(res)
{
  Deallocate(res, self);
  if (Process_Blocked_in(res,pr))
  {
     Allocate(res, pr);
     Unblock(pr, res);
     Scheduler();
  }
}
```

Specific Instantiations of Resource Request, Release

- P and V operations on semaphores
- Operations embedded in monitor procedures
- Calls to manage clocks, timers, delays, timeouts
- Send/receive operations

Implementing semaphores/locks

- Special hardware instruction: test_and_set
- Implementing binary semaphores
- Implementing general semaphores with busy waiting
- Avoiding the busy wait: Implementing general semaphores with blocking

Test_and_Set Instruction

- Special *test_and_set* instruction: TS(R,X)
- Operates on *variable* X, *register* R
- Behavior: R = X; X = 0;
  - Always set variable X = 0
  - Register R indicates whether variable X changed:
    - R=1 if X changed (1→0)
    - R=0 if X did not change (0→0)
- TS is indivisible (atomic) operation

Binary Semaphores

- Binary semaphore sb: only 0 or 1
- Also known as a *spin lock* or a *spinning lock* ("Spinning" = "Busy Waiting")
- Two atomic operations: Pb and Vb. Behavior is:

  Pb(sb): if (sb==1) sb=0;
  else wait until sb becomes 1

  Vb(sb): sb=1;
- Indivisible implementation of Pb and Vb using TS instruction:

  Pb(sb): do (TS(R,sb)) while (!R);/*wait loop*/

  Vb(sb): sb=1;
- Note: sb is shared, but each process has its own R

General Semaphores with busy wait

```
P(s) {
   Inhibit_Interrupts;
   Pb(mutex_s);
   s = s-1;
   if (s < 0)
   {
      Vb(mutex_s);
      Enable_Interrupts;
      Pb(delay_s);
   }
   Vb(mutex_s);
   Enable_Interrupts;
}

V(s) {
   Inhibit_Interrupts; Pb(mutex_s);
   s = s+1;
   if (s <= 0) Vb(delay_s);
   else Vb(mutex_s);
   Enable_Interrupts;
}
```

- Inhibit_interrupt prevents deadlock due to context switching
- Two binary semaphores used:
  - delay_s implements the actual wait, and may be held for a long time
  - mutex_s needed to implement critical section with multiple CPUs, only held for a few instructions
- Note than when V executes the call Pb(mutex_s), the corresponding Vb(mutex_s), may be executed by P

General Semaphores: avoiding busy wait

```
P(s) {
   Inhibit_Interrupts;
   Pb(mutex_s);  s = s-1;
   if (s < 0)
   {
      Block(self, Ls)
      Vb(mutex_s);
      Enable_Interrupts;
      Scheduler();
   }
   else
   {
      Vb(mutex_s);
      Enable_Interrupts;
   }
}
```

- Ls is a *blocked list* associated with the semaphore s.

```
V(s) {
   Inhibit_Interrupts;
   Pb(mutex_s);
   s = s+1;
   if (s <= 0)
   {
      Unblock(q,Ls)
      Vb(mutes_x);
      Enable_Interrupts;
      Scheduler();
   }
   else
   {
     Vb(mutex_s);
     Enable_Interrupts;
   }
}
```

Implementing Monitors

- Need to insert code to:
  - Guarantee mutual exclusion of procedures (entering/leaving)
  - Implement c.wait
  - Implement c.signal
- Implement 3 types of semaphores:
  1. mutex: for mutual exclusion
  2. condsem_c: for blocking on each condition c
  3. urgent: for blocking process after signal, to implement special high-priority queue

Implementing Monitor Primitives

- Code for each procedure:

```
P(mutex);
procedure_body;
if (urgentcnt > 0) V(urgent);
else V(mutex);
```

- Code for c.wait:

```
condcnt_c = condcnt_c + 1;
if (urgentcnt > 0) V(urgent);
else V(mutex);
P(condsem_c);
condcnt_c = condcnt_c - 1;
```

Code for c.signal:

```
if (condcnt_c)
{
  urgentcnt = urgentcnt + 1;
  V(condsem_c);
  P(urgent);
  urgentcnt = urgentcnt – 1;
}
```

Clock and Time Management

- Most systems provide hardware
  - *ticker:* issues periodic interrupt
  - *countdown timer:* issues interrupt after a set number of ticks
- Build higher-level services using this hardware
  - Wall clock timers
  - Countdown timers (how to implement *multiple logical timers* using a *single hardware countdown timer*)

Wall clock times

- Typical functions:

  Update_Clock: increment current time

  - typically number of clock ticks since some known time

  – Get_Time: return current time

  – Set_Time(tnew): set time to **`tnew`**

- Must maintain *monotonicity:* for two successive clock readings, the second time should always be ≥ the first time

  – So how do we set the clock back if we notice it is running fast?

Countdown Timer

- Main use: as alarm clocks
- Typical function:
  - Delay(tdel): block process for tdel time units
- Implementation using hardware countdown:

```
Delay(tdel)  {
   Set_Timer(tdel);  /*set hardware timer*/
   P(delsem);  /*wait for interrupt*/
}

Timeout()  {  /*called at interrupt*/
V(delsem);
}
```

Logical countdown timers

- Provides, at a minimum, the following functions:
  - tn = Create_LTimer() create new timer
  - Destroy_LTimer(tn)
  - Set_LTimer(tn,tdel) block process and call Timeout() at interrupt
- Each process will want one or more logical times of its own
- Implement multiple logical countdown timers using a single hardware timer
- Two approaches
  - Priority queue with absolute wakeup times
  - Priority queue with time differences

Priority queue with absolute wakeup times

- Store wakeup times of logical timers in a priority queue *TQ*
- Function of Set_LTimer(tn,tdel):
  - Compute absolute wakeup time using wall clock:

    wnew = tdel+tnow
  - Insert new request into *TQ* (ordered by absolute wakeup time)
  - If new request is earlier than previous head of queue, set hardware countdown to tdel

Clock and Time Management

Absolute Wakeup Times Example:

`Set_LTimer(tn,35)`

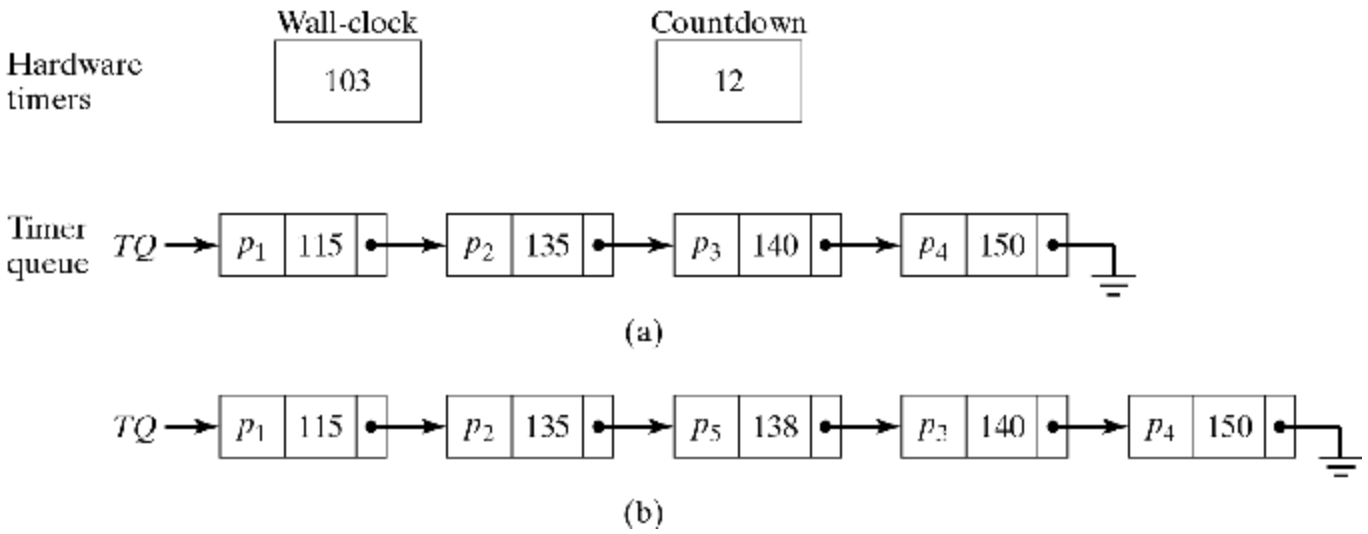

Figure 4-8

Priority queue with time differences

- Priority queue *TQ* records only time increments, no wall-clock is needed
- Function of Set_LTimer(tn,tdel)
  - Find the two elements *L* and *R* between which new request is to be inserted (add differences until tdel is reached)
  - split the current difference between *L* and *R* into difference between *L* and new element and difference between new element and *R*
  - If new request goes at front of *TQ*, reset the countdown time to tdel

Clock and Time Management

Time Differences Example:

`Set_LTimer(tn,35)`

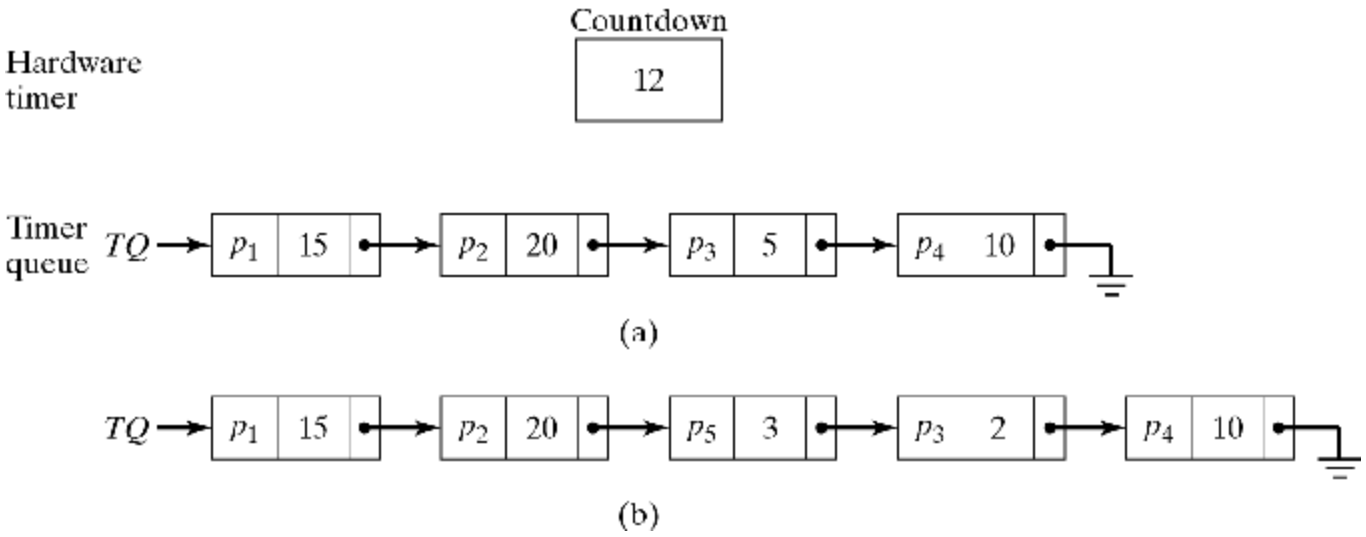

Figure 4-9

Communication Primitives

send and receive each use a buffer to hold message

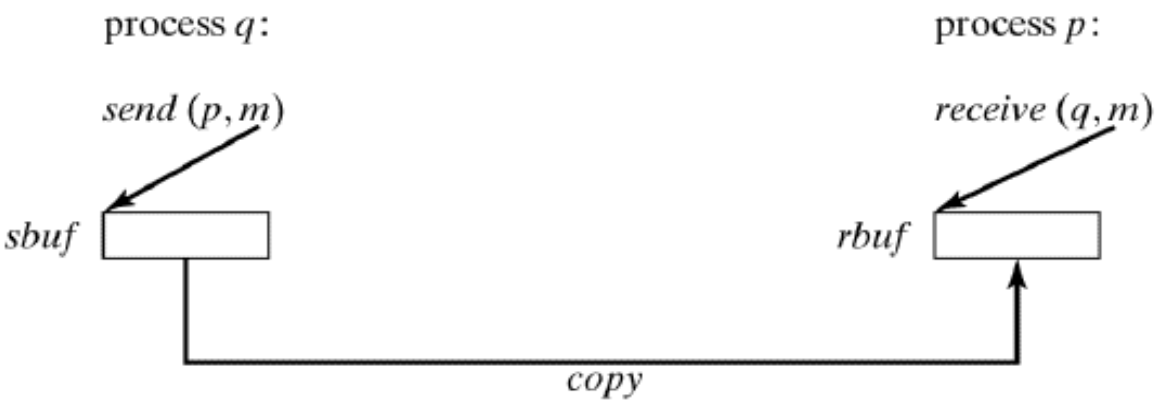

Figure 4-10a

1. How does sender process know that sbuf may be reused?
2. How does system know that rbuf may be reused overwritten?

Possible Solutions

- Reusing sbuf:
  - Use blocking send. Reuse when send returns
  - Provide a flag or interrupt for system to indicate release of sbuf
- Reusing rbuf:
  - Provide a flag for sender to indicate release of rbuf
- These solutions are awkward

Communication Primitives

Better solution: Use pool of *system buffers*

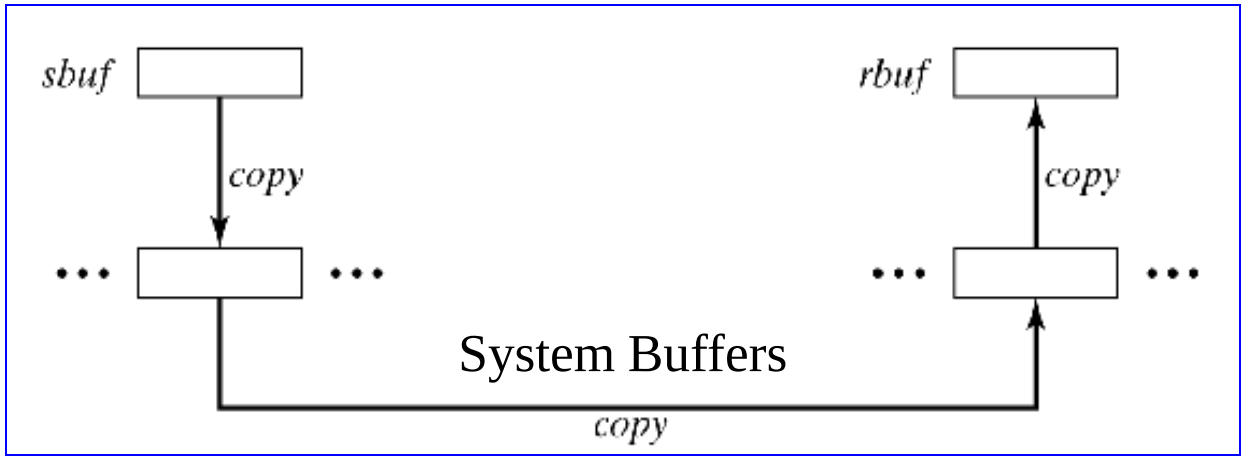

Figure 4-10b

1. send copies sbuf to a system buffer
2. send is free after copy is made
3. Sender may continue generating messages
4. System copies or reallocates full buffers to receiver
5. receive copies system buffer to rbuf
6. rbuf is overwritten with next message on nextcall to receive, which is controlled by the receiver.

Communications Kernel

- Copying of buffers is usually handled by a specialized communications kernel.
- Involves considerable additional processing
  - Breaking into transmission packets
  - Routing packets through network
  - Reassembling message from packets at the destination
  - Handling transmission errors

Interrupt Handling

Standard interrupt-handling sequence:

1. Save state of interrupted process/thread
2. Identify interrupt type and invoke appropriate interrupt handler *(IH)*
3. *IH* services interrupt
4. Restore state of interrupted process (or of another one)

Figure 4-11a

Typical Interrupt Handling Scenario

- User process p calls device interface procedure Fn
- Fn initiates device, then blocks.
- OS takes over, selects another process to run
- When device terminates, it generates an interrupt, which invokes IH
- IH services interrupt, unblocks P, and returns control to OS.

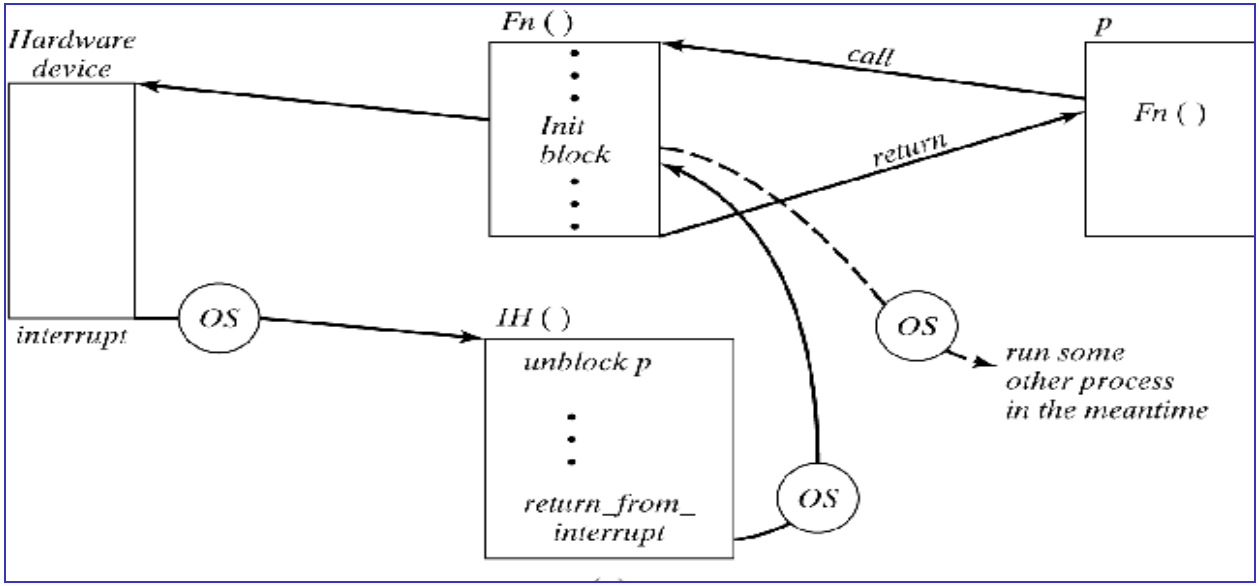

Figure 4-11a

Interrupt Handling

Main challenges:

- Fn must be able to block itself on a given event.
  - If Fn is written by user, requires knowledge of the OS kernel, possibly modification of the OS kernel.
- IH must be able to unblock p
- IH must be able to return from interrupt.

- Classical approach: specially designed kernel mechanisms
- Another approach: extend process model into the hardware (so IH is included) and use standard synchronization constructs, such as monitors.

Interrupt Handling Using a Monitor

- *Fn* waits on *c*
- *IH* invoked by hardware process
- *IH* signals *c*

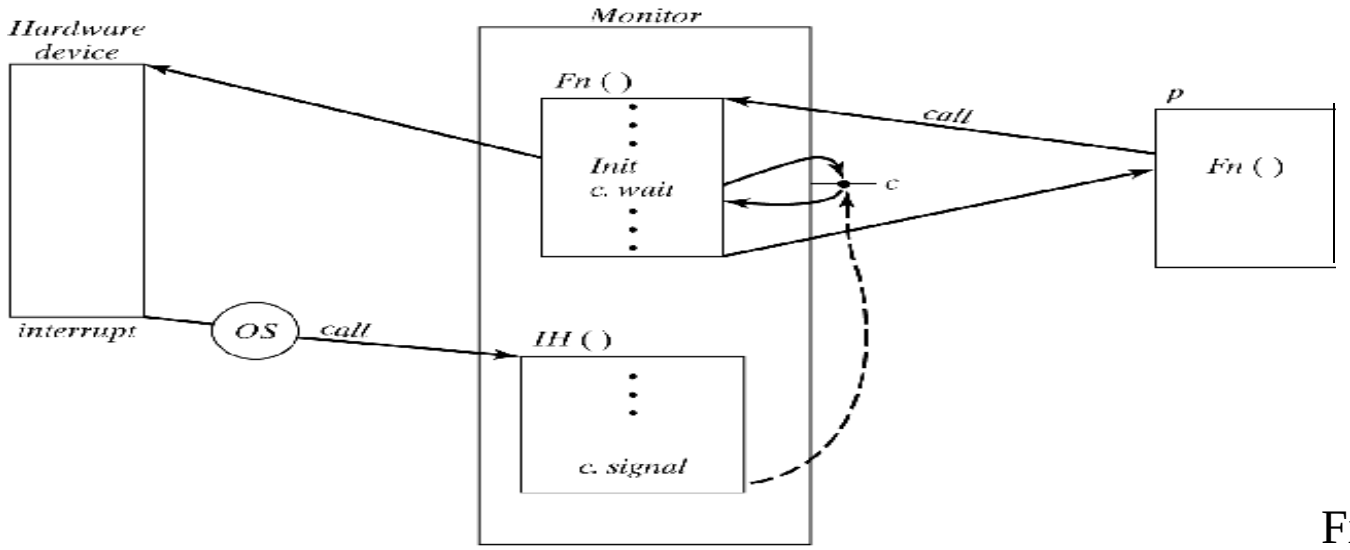

Figure 4-11b