

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
 - Semaphores and Locks
 - Building Monitor Primitives
 - Clock and Time Management
 - Communications Kernel
- 4.6 Interrupt Handling

ICS 143

1

The OS Kernel

- basic set of objects, primitives, data structures, processes
- rest of OS is built on top of kernel
- kernel provides *mechanism* to implement various *policies*
 - process/thread management
 - interrupt/trap handling
 - resource management
 - input/output

ICS 143

2

Queues

- 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
 - priority queues
- Figure 4-3

ICS 143

3

Processes and threads

- process may have one or more threads
- all threads within a process share:
 - memory space
 - other resources
- each thread has its own:
 - CPU state (registers, program counter)
 - stack

Figure 4-4

- implemented in user space or kernel space
- threads are efficient but lack protection from each other

ICS 143

4

Implementing processes/threads

- PCB

Figure 4-5

state vector: information necessary to run p

- status
 - basic types: running, ready, blocked
 - additional status types
 - ready active/suspended
 - blocked active/suspended
- state transition diagram

Figure 4-6

ICS 143

5

Operations on processes

```
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();  
}
```

ICS 143

6

Operations on processes

```
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();
  }
}
```

ICS 143

7

Operations on processes

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

ICS 143

8

Operations on processes

```
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);
}
```

ICS 143

9

Implementing synch/comm.

- semaphores, locks, monitors, etc. are resources

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

ICS 143

10

Implementing semaphores/locks

- need special test_and_set instruction:

TS(R,X)

- behavior:

```
R = X;
X = 0;
```

- always set variable X=0
- register R indicates change:
 - R=1 if X changed (1→0)
 - R=0 if X did not change (0→0)

- TS is indivisible (atomic) operation

ICS 143

11

Spinning locks

- binary semaphore sb (only 0 or 1)

- behavior of Pb/Vb:

- Pb(sb): if (s==1) s=0; else wait
- Vb(sb): sb=1;

- indivisible implementation of Pb/Vb:

- Pb(sb): do (TS(R,sb)) while (!R);
/* spin lock */
- Vb(sb): sb=1;

ICS 143

12

Gen'l semaphores w/ spin locks

```
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;
}
```

- inhibiting interrupts prevents deadlock due to context switching
- mutex_s needed to implement critical section with multiple CPUs

ICS 143

13

General semaphores w/ busy wait

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

ICS 143

14

Avoiding the busy wait

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

ICS 143

15

Avoiding the busy wait

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

ICS 143

16

Implementing monitors

- need to insert code to:
 - guarantee mutual exclusion of procedures (entering/leaving)
 - implement c.wait
 - implement s.signal
- implement 3 types of semaphores:
 - *mutex*: for mutual exclusion
 - *condsem_c*: for blocking on each condition c
 - *urgent*: for blocking process after signal

ICS 143

17

Implementing monitors

each procedure:
P(mutex);
procedure_body;
if (urgentcnt) V(urgent); else V(mutex);

c.wait:
condcnt_c = condcnt_c + 1;
if (urgentcnt) V(urgent); else V(mutex);
P(condsem_c);
condcnt_c = condcnt_c - 1;

c.signal:
if (condcnt_c) {
 urgentcnt = urgentcnt + 1;
 V(condsem_c); P(urgent);
 urgentcnt = urgentcnt - 1; }

ICS 143

18

Clock and time management

- most systems provide HW:
 - **ticker**: issues periodic interrupt
 - **countdown timer**: issues interrupt after set number of ticks
- build higher-level services using above HW
- wall clock timers
 - typical functions:
 - *Update_Clock*: increment *tnow*
 - *Get_Time*: return current time
 - *Set_Time(tnew)*: set time to *tnew*
 - must maintain **monotonicity**

ICS 143

19

Clock and time management

- countdown timers
 - main use: as alarm clocks
 - typical function:
 - *Delay(tdel)*: block process for *tdel* time units
 - implementation using HW countdown:

```
Delay(tdel) {
    Set_Timer(tdel); /*set HW timer*/
    P(delsem); /*wait for interrupt*/
}
Timeout() { /*called at interrupt*/
    V(delsem);
}
```

ICS 143

20

Clock and time management

- implement multiple logical countdown timers using a single HW timer
- functions:
 - *tn = Create_LTimer()*: create new timer
 - *Destroy_LTimer(tn)*
 - *Set_LTimer(tn,tdel)*: block process and call *Timeout()* at interrupt

ICS 143

21

Clock and time management

- implement *Set_LTimer()* using absolute wakeup times:
 - priority queue TQ records wakeup times
 - function of *Set_LTimer(tn,tdel)*:
 - compute absolute time of *tdel* (using wall-clock)
 - insert new request into TQ (according to time)
 - if new request is earlier than previous head of queue, set HW countdown to *tdel*
 - Example: *Set_LTimer(tn,35)*

Figure 4-8

ICS 143

22

Clock and time management

- implement *Set_LTimer()* using 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 *tdel* is to be inserted (add differences until *tdel* is reached)
 - split the current difference between L and R over new element and R
 - Example: *Set_LTimer(tn,35)*

Figure 4-9

ICS 143

23

Communication primitives

- send/receive operations each use a buffer to hold message

Figure 4-10a

1. how does sender know that *sbuf* may be reused (overwritten)?
2. how does system know that *rbuf* may be reused (overwritten)?

ICS 143

24

Communication primitives

- reusing *sbuf*:
 - use blocking send: reuse as soon as 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*
- solutions are awkward

ICS 143

25

Communication primitives

- better solution: use pool of **system buffers**
Figure 4-10b
 - *send* copies *sbuf* to a system buffer
 - *send* is free after copy is made
 - sender may continue generating messages
 - system copies or reallocates full buffers to receiver
 - *receive* copies system buffer to *rbuf*
 - receiver decides when to overwrite *rbuf* with next message

ICS 143

26

Interrupt handling

- standard interrupt-handling sequence:
 - save state of interrupted process/thread
 - identify interrupt type and invoke IH
 - IH services interrupt
 - restore state of interrupted process (or some other process)

Figure 4-11a

ICS 143

27

Interrupt handling

- main challenges:
 - Fn must be able to block; lowest-level Fn is generally provided by OS
 - IH must be able to unblock p and return from interrupt; IH is generally provided by OS
- better solution:
 - view HW device as a process
 - implement Fn and IH as monitor functions;
 - Fn waits on c
 - IH invoked by HW process
 - IH signals c

Figure 4-11b

ICS 143

28
