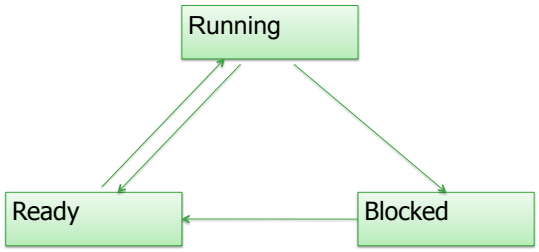
Operating Systems Design (CS 423)

Elsa L Gunter
2112 SC, UIUC
http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum

Thread States



Creating a new thread

- Overall: create state for thread and add it to the ready queue
- When saving a thread to its thread control block, we remembered its current state
- We can **construct** the state of new thread as if it had been running and got switched out

Creating a new thread

- Steps:
  - Allocate and initialize new thread control block
  - Allocate and init new stack
  - Add to ready queue
- From the scheduler's perspective, what is the different between a new and an old thread?
- What state does a new thread get init to?

Lock Implementation

- Concurrent programs use high-level synchronization operations
  - Used by multiple threads so they need to worry about atomicity (e.g., they use data structures)
  - Can't use the high-level synchronization operations themselves
- How to get off the ground?
  - How to assure atomicity?
  - When can a program be switched?

Interrupt enable/disable for atomicity

- On uniprocessor, operation is atomic as long as context switch does not occur
  - How does thread get context switched out?
  - Prevent context switches at wrong time by preventing these events

Interrupt enable/disable for atomicity

- With interrupt enable/disable, why do we need locks?
  - User program could call interrupt disable before entering critical section and call interrupt enable after leaving
  - What is wrong with this?

Lock impl. #1

- Disable interrupts with busy waiting

```
lock() {
 disable interrupts;
 while(value != FREE)
  {enable interrupts;
  disable interrupts};
  value = BUSY;
 enable interrupts;
}
```

```
lnlock() {
 disable interrupts;
 value = FREE;
 enable interrupts;
}
```

- Why does lock() disable interrupts in the beginning of the function?
- Does this guarantee atomicity and allow progress?

Big picture

- The critical sections are the accesses to the "lock variable" (value on previous slide)
- We use interrupt enable/disable as a "lock" for this critical section
- Can't use higher-level primitives

Lock impl. #1

- Why ok to disable interrupts in lock()'s critical section and not user-mode code?
- Do we need to disable interrupts in unlock()?
- Why does body of while enable, then disable interrupts?

Lock impl. #1

- Why ok to disable interrupts in lock()'s critical section and not user-mode code?
  - Because we are in kernel mode and control everything, won't starve anyone
- Do we need to disable interrupts in unlock()?
  - Yes, because we are accessing protected data
- Why does body of while enable, then disable interrupts?
  - So that a context switch may occur for any pending interrupts so someone else may work

Read-modify-write instructions

- Interrupt disable works on a uniprocessor
  - Does not work on multiprocessor
- Could use atomic load / atomic store instructions (Too Much Milk #3)
- Modern processors provide easier way with atomic read-modify-write instructions
  - Atomically {read value from mem into reg, then write new value to mem location}
  - Test and set

Test and set

- Var (say X) shared across all processors
- Atomically writes 1 to X (set) and returns previous value (test)

```
test_and_set(x) {
  tmp = X;
  X =1;
  return(tmp)
}
```

- Only one processor sees); all others see 1

Lock impl. #2

```
Test and set with busy waiting
lock(boolean *lock) {
while (test_and_set (lock) == 1);
}
unlock() {*value = 0;}
```

- If lock free (value = 0), test_and_set sets value to 1 and returns 0, so while loop finishes
- If lock busy (value = 1), test_and_set doesn't change value and returns 1, so loop continues

Problem with lock impl. #1 and #2

- Waiting thread uses lots of Cpu time waiting for lock to free
- Better for thread to go to sleep and let other threads run
- Strategy for reducing busy-waiting: integrate lock implementation with thread dispatcher data structures and have lock code manipulate thread queues

Interrupt disable / enable, Try 1

- What is wrong with the following?

```
lock() {
disable interrupts;
...
if(lock is busy) {
 enable interrupts;
 add thread to lock wait queue;
 switch to next runnable thread;
}
```


Interrupt disable / enable, Try 1

- What is wrong with the following?

```
lock() {
disable interrupts;
...
if(lock is busy) {
 enable interrupts;
 add thread to lock wait queue;
 switch to next runnable thread;
}
```

- If interrupt is handled as soon as enabled, lock might be freed, control returned, ante thread would go an wait queue even though lock free

Interrupt disable / enable, Try 2

- What is wrong with the following?

```
lock() {
disable interrupts;
...
if(lock is busy) {
 add thread to lock wait queue;
 enable interrupts;
 switch to next runnable thread;
}
```

Interrupt disable / enable, Try 2

- What is wrong with the following?

```
lock() {
disable interrupts;
...
if(lock is busy) {
 add thread to lock wait queue;
 enable interrupts;
 switch to next runnable thread;
}
```

- If interrupt is handled as soon as enabled, thread put on preemption queeu
- Thread on two queues! Bad – wrong queue (wait) might be used first

Making add to (queue & switch) atomic

- Adding thread to wait queue and switching to the next thread must be **atomic**
- **Solution:**
  - Waiting thread leaves interrupts disabled when it calls switch
  - Next thread to run must of re-enabling interrupts before returning to user code
  - When waiting thread wakes up, returns from switch with interrupts disabled (from last thread).
- Think of interrupt state as shared variable

Lock impl. #3

- Interrupts disabled, no busy wait

```
lock() {
disable interrupts;
if (value == FREE) {
  value = BUSY
} else {
 add thread to wait
  queue for this lock;
 switch to next thread;
};
enable interrupts
}
```

```
unlock() {
disable interrupts;
value = FREE;
if (this lock wait queue
nonempty) {
 move 1st wait thread
 to ready queue;
value = BUSY ;
};
enable interrupts
}
```

- Why do we use while on conditions?

Handoff locks

- Thread calling unlock() gives lock to waiting thread to guarantee FIFO ordering of lock usage
  - No control over scheduling algorithm, but we do control access to locks
- What does it mean for a thread to add current thread to lock wait queue?
  - Is this the point at which it saves state?
- Why do we need a separate lock queue?

Lock impl. #3

- Invariant:
  - All thread promise to have interrupts disabled when they call schedule
  - All threads promise to re-enable interrupts after they get returned to from schedule
- Should use this for MP3

Thread A	**Thread B**
`yield() {`	
`disable interrupts;`	
`switch;`	
	`back from switch`
	`enable interrupts}`
	`<user code runs>`
	`lock() {`
	`disable interrupts`
	`...`
`back from switch`	`switch`
`enable interrupts}`	
`<user code runs>`	
`unlock() // move a to readyq`	
`yield()`	
`{disable interrupts`	
`switch`	
	`back from switch`
	`enable interrupts}`

Lock impl. #4

- Can't implement locks using test&set without some amount of busy-waiting, but can minimize it
- Idea: use busy waiting only to atomically execute lock code. Give up CPU if busy

Lock impl. #4

```
lock() {
 while(test&set(guard){
 }
  if(value == FREE) {
  value = BUSY
 } else {
  add thread to wait_q
  switch to next run t
 }
 guard = 0
}
```

```
unlock() {
 while(test&set(guard)){
 }
 value = FREE
 if(wait_q nonempty) {
 move thread to ready_q
value = BUSY
}
guard = 0
}
```


Deadlock

- Resources
  - Something needed by a thread
  - A thread **waits** for resources
  - E.g., locks, disk space, memory, CPU
- Deadlock
  - A circular waiting for resources leading to threads involved not being able to make progress

Deadlock

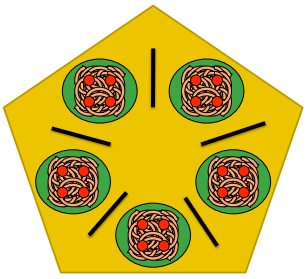
- Example:

Thread A	Thread B
`lock(x);`	`lock(y);`
`lock(y);`	`lock(x);`
…	…
`unlock(y);`	`unlock(x);`
`unlock(x);`	`unlock*(y);`

- Can deadlock occur with our code?
- Will deadlock always occur?
  - Ostrich algorithm

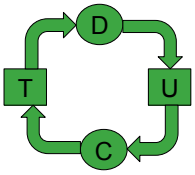
Dining Philosophers Problem

- 5 Philosophers
  - Plate for each
  - One chopstick to the left of each plate
  - Takes two chopsticks to eat
- Wait for left chopstick
- Wait for right chopstick
- Eat
- Put down chopsticks
- Deadlock?



Conditions for deadlock

- Four conditions for deadlock
  - Limited resource
    - Only 5 chopsticks
- Hold and wait
  - Grab one chopstick at a time
- No preemption
  - Grab if you can, no guarantee
- Circular chain of requests
- Wait-for graph



CPU scheduling

- How to choose next thread to run?
  - What are goals of CPU scheduler?
- Minimize average response time
  - Average elapsed time to do each job
- Maximize throughput of entire system
  - Rate at which jobs complete in the system
  - How do we maximize throughput?
- Fairness
  - Share CPU among threads in some "equitable" manner

First come, first served (FCFS)

- No preemption (run until done)
  - Thread runs until it calls yield() or blocks
  - No timer interrupts
- Pros
  - + simple
- Cons
  - - short jobs get stuck
  - - bad for interactive use

FCFS example

- Job a takes 100 seconds
- Job b takes 1 second
- Time 0: job a arrives and starts
- Time 0+: job b arrives
- Time 100: job a ends; job b starts
- Time 101: job b ends
- Average response time = (100 + 101)/2 = 100.5

Round robin

- Goal: improve average response time for short jobs
- Solution: periodically preempt running job, let next go
- Is FCFS or round robin more "fair"?

Round Robin Example

- Job a takes 100 seconds
- Job b takes 1 second
- Time slice = 1 sec
- Time 0: job a arrives and starts
- Time 0+: job b arrives
- Time 1: job b preempted; job a runs
- Time 2: job b ends; job a resumes
- Time 101: job a ends
- Average response time = (2 + 101)/2 = 51.5

Round robin

- Does round-robin always achieve lower response time than FCFS?

Choosing a time slice

- Big time slice: degrades to FCFS
- Small time slice: each context switch too expensive
- Typical compromise 10 ms or 1ms
- If context switch takes .1ms, round robin with 10ms time slice wastes 1% of CPU time

Real time scheduling

- Question is can task finish by deadline?
- Worst case analysis instead of average case analysis
- About determinism, not performance
  - Locking cache lines
- Key question: Can given tasks be scheduled and guaranteed to complete by given deadline
- Soft real time and hard real time
  - Used in multimedia systems, embedded and control systems, etc.

Real-time scheduling

- How do you schedule?

Earliest deadline first (EDF)

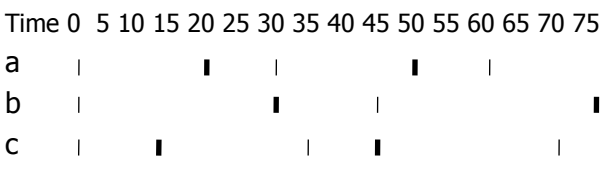
- Always run the job that has the earliest deadline
  - Dynamically adjusts the priority
- If new job arrives with earlier deadline than current job, preempt
- EDF is optimal --- we'll prove this later

Definitions

- Tasks - unit of work
- Periodic and aperiodic tasks
- Arrival time - time when task can begin execution
- Deadline – time by which task must be done
  - Can be relative or absolute
- Execution time - amount of time task needs to finish work

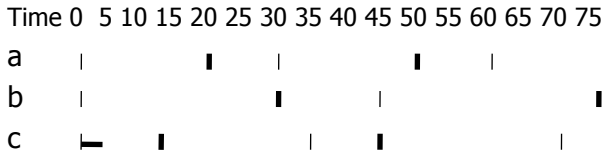
EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



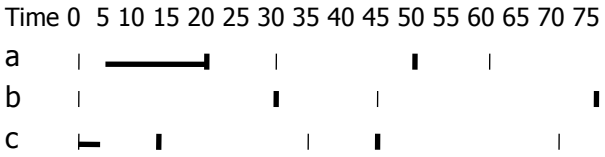
EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



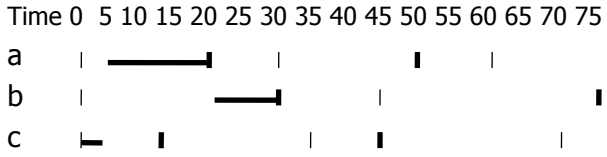
EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



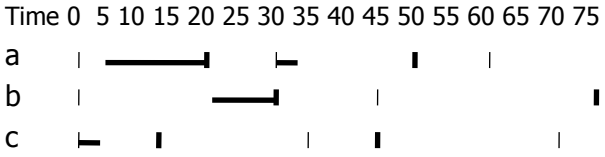
EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



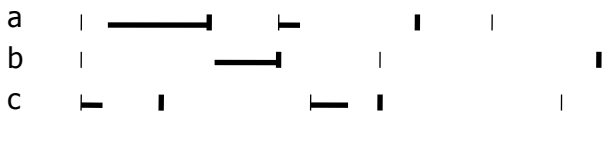
EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



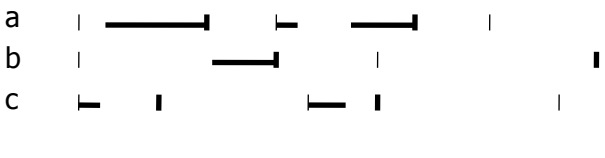
EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



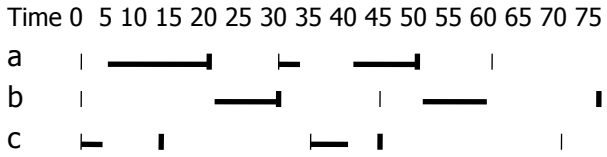
EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



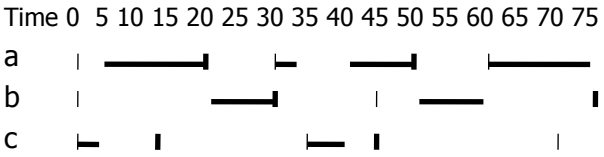
EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)



EDF example

- Job a: e(a) = 15, D(a) = 20 (period 30)
- Job b: e(b) = 10, D(b) = 30 (period 45)
- Job c: e(c) = 5, D(c) = 10 (period 35)