EECS 482, winter 2013
Midterm study guide

by Evan Hahn, Luke Anderson, Vicki Li
Add your name if you helped! Feel free to send this around.

Exam logistics

- Closed-everything
- Expect critical thinking (analysis, synthesis), not regurgitation

Introduction to operating systems

- it's a layer between hardware and application programs, providing a nicer abstraction (locks, semaphores, virtual memory)

Process, time-sharing systems, address space

- TODOt

Threads and concurrency

- TODO, atomicity, data races, synchronization (mutual exclusion, or ordering)

- Monitors (Mesa: Project 1) & semaphores, reader/writer locks (implement and understand)
- Banker's algorithm, other ways of avoiding deadlocks

CPU scheduling

First come, first served

- Simply complete jobs in the order that they're requested
- No preemption: Will run until down (thread calls yield() or is blocked, no timer interrupts)
- Simple, easy to implement
- Don’t need to know running times of processes
- Minimizes context switching

Round- Robin

- Every job gets the same amount of time (ie. # second time quantum) to try and finish (Fairness)
- Tries to improve average response time for short jobs, but need to choose appropriate time slice
- Most common in OS
- Don't need to know running time of processes
- Does not allow for a single process to monopolize the CPU
- Lots of context switching

Shortest Job (time to completion) First

- For any two pairs of jobs that you're thinking about scheduling, move the shortest job to the front (essentially a bubble sort)
- Gives optimal response time, but needs knowledge of the future
- Can lead to starvation: Maximizes throughput in terms of job per second, but jobs don't always get to run
- Unfair
- Lots of overhead
- Runtimes are hard to predict

Non-Preemptive Priority

- Jobs assigned a priority, then run with highest priority first
- Used in embedded system
- Minimizes context switches
- Uses priorities (something important isn't interrupted by a trivial process)
- Can result in starvation
- Hard to determine priorities (is this priority 3 or 4?)
- CPU vulnerable to being taken over

Earliest Deadline First

- Optimal with regard to context switches (if context switching is free, or nearly free)
- Bad with overloads
-

Example (Basic):
Job A: Takes 100 second to complete
Job B: Takes 1 second to complete
Average response time = (Time for Job A to finish + Time for Job B to finish)/(# of jobs)

With FCFS:
0: A arrives, A runs
0 + e: B arrives
100: A finishes, B runs
101: B finishes
Average response time: (100 + 101)/2 = 100.5

With RR, 10 second time quantum:

0: A arrives, A runs
0 + e: B arrives
10: A tried to finish, B runs
11: B finishes, A continues running
101: A finishes
Average response time: (101 + 11)/2 = 56

With SJF:
0: A arrives
0 + e: B arrives, B runs
1: B finishes, A runs
101: A finishes
Average response time: (101 + 1)/2 = 51

Example (Earliest Deadline First):
Job A: Takes 15 seconds, deadline is 20 seconds after entering system
Job B: Takes 10 seconds, deadline is 30 seconds after entering system
Job C: Takes 5 seconds, deadline is 10 seconds after entering system

Timeline (To show possible order given specific times jobs enter system):
+ : Job enters system
- : Job is being run
| : Job has finished

Job	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
A	+	---	----	---			+--		---	--|		+	---	---	--|	
B	+			|	---	---				+	---	--|				
C	+-|					|		+-|								

Deadlocks

Resource: Anything a thread needs to do it's job. Eg. Locks, semaphores, CPU, memory, disk
Deadlock: Acyclical waiting for resources, prevents threads involved from making progress
Deadlocks can only happen if four conditions are satisfied:

1. More than one resource
2. Request chain (circular)
3. No preemption of resources
4. Hold & Wait: Thread needs to be holding one lock while waiting for another resource

To prevent deadlocks, we can eliminate any of the four conditions. Ideas:

- Increase resources to decrease waiting
- Eliminate Hold & Wait

- Move resource acquisition to beginning):
  - phase 1a: acquire all resources
  - phase 1b: while (not done){
    work
    }
  - phase 2: release all resources
- Wait until all resources needed are free (and then grab them all at once)
- Release all acquired resources and go back to the beginning whenever a resource is found to be busy

- Allow preemption:
  - Can preempt CPU by saving state to thread control block and resuming later
  - Can preempt memory by swapping memory out to disk and loading back later

Banker's Algorithm

- Determines when it's safe for a thread to acquire a resource, blocks if unsafe
- General structure of thread code:
  - phase 1a: state maximum resource needed
  - phase 1b: while (not done){
    acquire some resources
    work
    }
  - phase 2: release all resources
- If you request all resources at the beginning to prevent a deadlock, phase 1a would be blocked (however, phase 1b can proceed without waiting)
-

Example 1:

Bank has $6000. Customers sign up with bank and establish a credit limit (maximum resource needed). They borrow money in stages (up to their credit limit). When they're done, they return all the money.

Banker's Algorithm solution: All credit limits are ok-ed, but customer may need to wait when actually asking for money.

Example 2:

```
int total_resources[NUM_RES];          // will be constant during total runtime of process
int available_resources[NUM_RES];
int curr_alloc[NUM_PROC][NUM_RES];
int max_alloc{NUM_PROC][NUM_RES];
int needed_alloc[NUM_PROC][NUM_RES];
```

	A	B	C	D
total_resources	6	5	7	6
available_resources	3	1	1	2

curr_alloc:

	A	B	C	D
p1	1	2	2	1
p2	1	0	3	3
p3	1	2	1	0

max_alloc:

	A	B	C	D
p1	3	3	2	2
p2	1	2	3	4
p3	1	3	5	0

needed_alloc:

	A	B	C	D
p1	2	1	0	1
p2	0	2	0	1
p3	0	1	4	0

	A	B	C	D
Avail	3	1	1	2
p1 need -	2	1	0	1
	1	0	1	1
p1 max +	3	3	2	2
	4	3	3	3
p2 need -	0	2	0	1
	4	1	3	2
p2 max +	1	2	3	4
	5	3	6	6
p3 need -	0	1	4	0
5	2	2	6	
p3 max +	1	3	5	0
Total	6	5	7	6

// Must have a limited number of processes for banker's algorithm
// Also needs to know the maximum for those processes in advance
// Only guarantees that something *will* finish, no guarantee for time!

Semaphores

CONSIDER MONITORS COMBINED WITH SEMAPHORES

Consider single bathroom problem:

- 3 Semaphores
  - Guard =1
  - Men = 0
  - Women = 0
- int Active_men, waiting_men
- int Active_women, waiting_women
- enum{MEN, WOMEN} turn=MEN

```
Woman_Wants_To_Enter{
	down(guard);
	if(active_men ||(waiting_men && turn == MEN)){
		++waiting_women;
		up(guard);
		down(women);
		down(guard);
	}
	active_women++;
	up(guard);
}

Woman_leaves(){
	down(guard);
	if(--active_women == 0){
		while(waiting_men){
			up(men);
			--waiting_men;
		}
	}
	turn = MEN;
	up(guard);
}
```

(mirror for men)

This study guide is licensed under a Creative Commons Attribution 3.0 Unported license. Just link back

to this if you use it at all. Its accuracy is not guaranteed.

Enjoy!