University of California, Berkeley
College of Engineering
Computer Science Division – EECS

Spring 2012 Anthony D. Joseph and Ion Stoica

Midterm Exam *Solutions*

March 7, 2012
CS162 Operating Systems

Your Name:	
SID AND 162 Login:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book and one 2-sided handwritten note** examination. You have 80 minutes to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points for that question. You should read **all** of the questions before starting the exam, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. ***Make your answers as concise as possible.*** If there is something in a question that you believe is open to interpretation, then please ask us about it!

Good Luck!!

QUESTION	POINTS ASSIGNED	POINTS OBTAINED
1	20	
2	20	
3	15	
4	20	
5	15	
6	10	
TOTAL	100	

1. (20 points total) Short answer questions.
   a. (8 points) True/False and Why? **CIRCLE YOUR ANSWER.**
      i) A lightweight process with one thread is equivalent to a heavyweight process.

TRUE FALSE

Why?

***TRUE**. A heavyweight process has only one thread. The correct answer was worth 2 points and the justification was worth an additional 2 points.*

      ii) Demand paging requires the programmer to take specific action to force the operating system to load a particular virtual memory page.

TRUE FALSE

Why?

***FALSE**. The OS automatically loads pages from disk when necessary. The correct answer was worth 2 points and the justification was worth an additional 2 points.*

   b. (8 points) Two-level Page Tables:
      i) Give a two to three sentence description of a two-level page table.

A two-level page table uses two levels of page tables where a pagetable pointer points to the top-level table. Entries in the top-level table point to the lower-level page tables. The lower level tables contain PTEs pointing to the physical locations.

      ii) Briefly (2 sentences) state one advantage AND one disadvantage of two-level page tables.

Advantage: Can map sparse address spaces efficiently
Disadvantage: Requires an additional memory reference to translate virtual to physical addresses.

c. (4 points) List the four requirements for deadlock.

Mutual exclusion, non-preemptable resources, hold and wait, circular chain of waiting.

Each requirement was worth 1 points.

2. (20 points total) Consider the following two functions implementing a producer and consumer by using monitors:

```
void send(item) {
  lock.acquire()
  enqueue(item);
  printf("before signal()\n");
  dataready.signal(&lock);
  printf("after signal()\n");
  lock.release();
}

item = get() {
  lock.acquire();
  while (queue.isEmpty()) {
    printf("before wait()\n");
    dataready.wait(&lock);
    printf("after wait()\n");
  }
  item = dequeue();
  lock.release();
}
```

a. (4 points) Use no more than three sentences to contrast Hoare and Mesa monitors.

With Hoare the signaler gives the CPU and the lock to the waiter; With Mesa the signaler schedules the waiter, and then finishes.

- *-1 if it was not clear that a thread in a Mesa monitor can hold the lock indefinitely after signaling.*
- *-1 if the signaled/woken thread is put on a wait queue instead of the ready queue.*

b. (5 points) Assume two threads T1 and T2, as follows:

```
     T1                                    T2
send(item);                          item = get();
```

What are the possible outputs if the monitor uses the Hoare implementation?

```
before signal
after_signal
```

- *We gave 2 points for the above solution.*

```
before wait
before signal
after wait
after signal
```

- *We gave 3 points for the above solution.*
- *-2 if there were more than two answers, but at least 1 point if "something" was right.*

c. (5 points) Repeat question (b) for a Mesa implementation of the monitor.

```
before signal
after_signal
```

- *We gave 2 points for the above.*

```
before wait
before signal
after signal
 after wait
```

- *We gave 3 points for the above.*
- *-2 if there were more than two answers, but at least 1 point if "something" was right.*

d. (6 points) Now assume a third thread T3, i.e.,

```
   T1                  T2                  T3
send(item);        item = get();       send(item);
```

What are the possible outputs if the monitor uses the Hoare implementation? Please specify from which thread does an output come by specifying the thread id in front of the output line, e.g., `[T1] before signal` or `[T2] after wait`.

```
[T1] before signal          [T3] before signal
[T1] after signal           [T3] after signal
[T3] before signal          [T1] before signal
[T3] after signal           [T1] after signal
```

- *1 point for each of the above ones.*

```
[T2] before wait            [T2] before wait
[T1] before signal          [T3] before signal
[T2] after wait             [T2] after wait
[T1] after signal           [T3] after signal
[T3] before signal          [T1] before signal
[T3] after signal           [T1] after signal
```

- *2 point for each of the above ones.*
- *--1 for each answer beyond four.*

3. (15 points) Design tradeoffs (15 points total):
You've been hired by Orange Computer to help design a new processor and Orange Pro laptop. After choosing the display, case, and other components, you are left with $460 to spend on the following components:

Item	Latency	Minimum Size	Cost
TLB	10 ns	256 entries	$0.10/entry
Main memory	180 ns	2 GB	$10/GB
Magnetic Disk	8 ms (8M ns)	300 GB	$0.10/GB

The page size is fixed at 64 KB. Assume you want to run up to 20 applications simultaneously. Each application has an overall maximum size of 1 GB and a working set size of 256 MB. TLB entries do not have Process Identifiers. Discuss how you would divide the available funds across the various items to optimize performance.

We start with the disk. Since the disk is the slowest component of the system, we take the minimum size, 300 GB or $30, leaving us with $430. Since the TLB does not contain process identifiers, we only need the minimum number of entries to map the working set for a single process – 256 MB / 64 KB = 4,096 entries or $409.60, leaving us with $21.40. With the remaining money, we could buy 2 GB. However, since the maximum number of applications is 20 each with a working set size of 256 MB, we should provide 5 GB of RAM ($50) to avoid paging, so we should spend $380 on 3,800 entries. While this will cause TLB misses, it does not make sense to increase the TLB any more, since that would require that we decrease the memory size below the requirements for the applications; a situation that will cause the system to start paging.

We awarded 3 points per choice, based upon the reasonableness of the choices. We used the following table:

Item / Points	3 points	2 points	1 point	0 points
(a) TLB	3,800 entries	> 3,800	< 3,800	< 256
(b) Memory	5 GB	> 5 GB	< 5 GB	< 2 GB
(c) Disk	300 GB	> 300GB, if using extra money for disk instead of memory or TLB	> 300GB	< 300GB

(d) We awarded another 3 points if the TLB answer was based upon an analysis of a single applications' working set size (i.e., since only mapping the working set matters and the TLB does not include process identifiers).
(e) We awarded an additional three points based upon the overall reasonableness of the answer. For example, a system with a small amount of paging would lose a point, while a system with a significant amount of paging (e.g., only 2 GB of memory), would lose two points.

4. (20 points) Concurrency control: Consider the following pseudocode that aims to implement a solution for the Dining Philosopher problem. Note that a philosopher can use any chopstick.

```
// assume chopstick[i].status = FREE, for 1 <= i <= N
get_chopstick(boolean hold_one_chopstick) {
  lock.acquire();
  for (i = 1; i <= N; i++) {
    if (chopstick[i].status == FREE) {
      chopstick[i].status = BUSY;
      return i;
    }
  }
  lock.release();
  return -1;
}

release_chopstick(i) {
  if (i == -1) return;
  chopstick[i].status = FREE;
}


philosopher() {
  plate = FULL;
  while (plate == FULL) {
    chopstick1 = get_chopstick(FALSE);
    if (chopstick1 != -1) {
      chopstick2 = get_chopstick(TRUE);
      plate = EMPTY;
      release_chopstick(chopstick2);
    }
    release_chopstick(chopstick1);
  }
}

main() {
  for (i = 1; i <= N; i++) {
    thread_fork(philosopher());
  }
}
```

a. (2 points) Name an error in how synchronization primitives are used in `get_chopstick()`

There is a missing `lock.release()` *before* `return i`.

b. (10 points) After fixing the error in part (a), does the program work correctly? If it does not, give a simple example to show how the program fails, and provide a fix. If it does, use no more than three sentences to argue why it works.

Is not guaranteed to work. Every philosopher can get a chopstick, fail to get the second one, release the chopstick they hold, and repeat!

- *We gave 5 points for an example.*

Add code to `get_chopstick()` *to not give the last chopstick to a philosopher if that philosopher doesn't already own a chopstick. For example:*

```
cnt = 0;
for (i = 1; i < N; i++) {
  if (chopstick[i].status == FREE) {
    cnt++;
  }
}
if (cnt == 1 && hold_one_chopstick == FALSE) {
  return -1;
}
```

- *We subtracted 3 points if you did not give the above solution.*
- *We subtracted 1 point if you said it in words, but not give the code.*
- *We subtracted 1 point if your solution was to use mutex around the entire body of* `philosopher()`*, as we considered this solution to do "excessive locking".*

In addition, you need to check for `chopstick2` *in* `philosopher()`*, i.e.,*
`if (chopstick2 != -1) { plate = EMPTY; }`

- *We subtracted 2 point if you did not provide the above fix.*

(We also need to protect the body of release_chopstick(), by lock.acquire() and lock.release(). However, we did not subtract any points for this.)

c. (8 points) Assume `main()` launches N+1 philosopher threads, instead of N. Will the program work correctly given the changes you made for parts (a) and (b)? If it does not, give a simple example to show how the program fails, and provide a fix. If it does, use no more than three sentences to argue why it works.

No change needed, as the modified code in (b) will guarantee that the last chopstick will always be picked by a philosopher that already has another chopstick.

5. (15 points total) Scheduling.
   a. (15 points) Consider the following processes, arrival times, and CPU processing requirements:

Process Name	Arrival Time	Processing Time
1	0	3
2	1	5
3	3	2
4	9	2

For each scheduling algorithm, fill in the table with the process that is running on the CPU (for timeslice-based algorithms, assume a 1 unit timeslice). For RR and SRTF, assume that an arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it. The turnaround time is defined as the time a process takes to complete after it arrives.

Time	**FIFO**	**RR**	**SRTF**
0	**1**	**1**	**1**
1	*1*	*2*	*1*
2	*1*	*1*	*1*
3	*2*	*3*	*3*
4	*2*	*2*	*3*
5	*2*	*1*	*2*
6	*2*	*3*	*2*
7	*2*	*2*	*2*
8	*3*	*2*	*2*
9	*3*	*4*	*2*
10	*4*	*2*	*4*
11	*4*	*4*	*4*
Average Turnaround Time	*3+7+7+3/ 4 =* ***5***	*6+10+4+3/ =* ***5.75***	*3+2+9+3/4=* ***4.25***

Each column is worth 5 points: 3 for correctness of the schedule (we deducted 1/2/3 points if you made minor/intermediate/major mistakes), and 2 for the average Turnaround time (1 point was deducted for minor errors).

6. (10 points total) Caching: Assume a computer system employing a cache, where the access time to the main memory is 100 ns, and the access time to the cache is 20ns.

a. (2 points) Assume the cache hit rate is 95%. What is the average access time?

*Average Access Time = Hit*cache_access_time + (1-Hit)*memory_access_time*
*= 0.95*20 ns + 0.05*100 ns = 24 ns*
*Alternatively, we accepted solutions that included the cache time in the memory access time: AAT = 0.95 * 20 ns + 0.05 * (20 ns + 100 ns) = 25 ns.*
We subtracted one point for minor errors.

b. (2 points) Assume the system implements virtual memory using a two-level page table with no TLB, and assume the CPU loads a word X from main memory. Assume the cache hit rate for the page entries as well as for the data in memory is 95%. What is the average time it takes to load X?

*The Average Memory Access Time for X (AMAT) requires three memory accesses, two for each page entry, and one for reading X: 3*24 = 72 ns. The alternate solution from (a) yields 3*25 = 75 ns. We only accepted the alternate solution for (b) if you derived the same value for (a).*

c. (3 points) Assume the same setting as in point (b), but now assume that page translation is cached in the TLB (the TLB hit rate is 98%), and the access time to the TLB is 16 ns. What is the average access time to X?

*AAT_X is TLB_hit *(TLB_access_time + AAT) + (1-TLB_hit) * (3 * AAT):*
0.98(16ns + 24ns) + 0.02*(72ns) = 0.98*40ns + 1.44ns = 40.64ns*
Alternate AAT from (a): 0.98(16ns + 25ns) + 0.02*(75ns) = 41.68ns*

It was acceptable to include the TLB time in the TLB_miss calculation:
*TLB_hit *(TLB _time + AMAT) + (1-TLB_hit) * (3 * AMAT +TLB_time).*
0.98(16ns + 24ns) + 0.02*(72ns + 16ns) = 0.98*40ns + 0.02*88ns = 40.96ns*
Alternate AAT from (a): 0.98(16ns + 25ns) + 0.02*(75ns +16ns) = 42ns*
We subtracted one point for each minor error.

d. (3 points) Assume we increase the cache size. Is it possible that this increase to lead to a decrease in the cache hit rate? Use no more than three sentences to explain your answer.

Yes, using a FIFO replacement scheme could result in Belady's anomaly. Also, using the same hash function while increasing the cache size could cause more collisions and reduce the hit rate. The correct answer was worth one point and the justification was worth two points.

This page intentionally left blank

Do not write answers on this page