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

Fall 2012 Ion Stoica

# Midterm Exam

October 15, 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 | 24 | |
| 2 | 14 | |
| 3 | 12 | |
| 4 | 24 | |
| 5 | 12 | |
| 6 | 14 | |
| TOTAL | 100 | |

1. (24 points total) True/False and Why? **CIRCLE YOUR ANSWER**. For each question: 1 point for true/false correct, 2 point for explanation. An explanation cannot exceed 2 sentences.

   a) Each thread has its own stack.

   TRUE FALSE

   Why?

   *Yes, the thread is the unit of execution, and it uses the stack to store the return addresses and arguments passed to functions/methods.*

   *One point for true/false. Two points for correct explanation.*

   b) Starvation implies deadlock.

   TRUE FALSE

   Why?

   *FALSE: A system my exit from starvation, but not from deadlock (without external intervention).*

   *One point for true/false. Two points for correct explanation.*

   *Full points required giving a situation where starvation exists but not deadlock. Note that "deadlock implies starvation" does not explain why this is false - more detail was necessary for full credit.*

   *Some partial credit was awarded if you gave correct, relevant information about deadlocks and starvation, regardless of the true/false answer.*

   c) It's generally possible to substitute a semaphore for a condition variable, because *sem.V*()/*sem.P*() have similar semantics to *cond.signal*()/*cond.wait****()***.

   TRUE FALSE

   Why?

   *cond.signal() before cond.wait() has no effect, but sem.V() before sem.P() has an effect*

   *Full credit requires explaining at least one major difference between CV and semaphore, such as*
   *1) CV is not commutative (what we were looking for)*
   *2) Semaphore has no notion of implicit lock (also accepted).*

   *Some partial credit was awarded if they gave correct, relevant information about CV's or Semaphores, regardless of the true/false answer.*

d) Shortest Run Time First (SRTF) is the "optimal" scheduling algorithm, but it is generally not implemented directly, due to excessive context switching overhead.

TRUE FALSE

Why?

*It is not implemented because it is impossible to predict the future.*

*One point for true/false. Two points for correct explanation.*

*For full credit you needed to point out the real reason why it is not correctly implemented: that it is impossible to predict the future. We also accepted "because it is unfair" and "because it can lead to starvation for long jobs" as valid reasons.*

*Some partial credit was awarded if they gave correct, relevant information about SRTF, regardless of the true/false answer.*

e) Using a smaller page size increases the size of the page table.

TRUE FALSE

Why?

*You need more pages for allocating the same memory which means more entries in the page tables.*

*One point for true/false. Two points for correct explanation.*

f) Moving from a single level page table to a two-level page table will always decrease the memory footprint (in aggregate) used by the page table.

TRUE FALSE

Why?

*If a process uses all addressable memory, you have strictly more memory used by 2-level page table.*

*One point for true/false. Two points for correct explanation*

*Some partial credit was awarded if they gave correct, relevant information about two-level page tables, regardless of the true/false answer.*

g) Unlike paging, segmentation doesn't prevent processes from accessing physical memory not allocated to them.

TRUE FALSE

Why?

*Segmentation avoids access to unallocated memory by checking that an address is within segment limits.*

*One point for true/false. Two points for correct explanation.*

*Some partial credit was awarded if they gave correct, relevant information about segmentation, regardless of the true/false answer.*

h) If you increase the size of a the page cache from 8Kb to 16Kb, and you are running a "Perfect LRU" page replacement strategy, the cache hit ratio will never get worse.

TRUE FALSE

Why?

*With LRU, you always have a superset of the original pages in cache.*

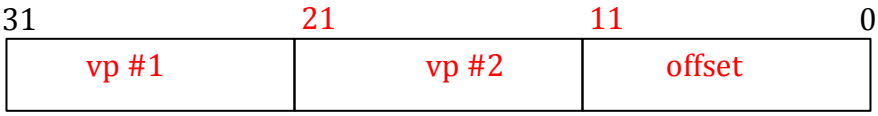
*One point for true/false. Two points for correct explanation.*

*For full credit you needed to didn't need to say much, other than to reaffirm that the hit rate increases. Explaining that the cache contents are a superset of those with the smaller cache size was even better.*

*Many people confused the page cache with hardware caches such as direct-mapped and associative, this was not a correct understanding of the question and received fewer points.*

2. (14 points) **Memory hierarch:** You are responsible for designing the memory system for a byte addressable system. The virtual memory address space is 32 bits and the physical memory address space is 16 bits.

   a) (4 points) Assume the system uses a two level page table to translate a virtual address to a physical address. Show the format of the virtual address, specify the page size (pick one size if multiple sizes are feasible), and specify the length of each field in the virtual address. Make sure that each translation table fits in a page.



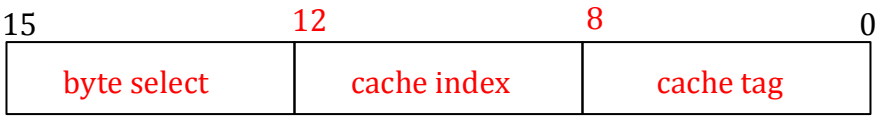

*Assume a page size of 4 KB*

- *1st field: virtual page # 1: 10 bits ($2^{10}$ entries fits in a 4 KB page)*
- *2nd field: virtual page # 2: 10 bits ($2^{10}$ entries fits in a 4 KB page)*
- *3rd field: offset: 12 bit.*

*4KB is an optimal fit while keeping the size of the VPN#1 and VPN#2 constant. If the offset is less than 12 bits, then entries of the page table will not fit in a single page.*

1 point for showing the format of the page table entry
1 point for specifying page size (and number of entries in the page)
1 point for ensuring that the each page table fits within a single page
1 point for finding optimal sized (4KB) page

   b) (4 points) Assume you add to your system a 4-way set-associative **data cache** with 16 cache blocks. Each block in the cache holds 8 bytes of data. In order to address a specific byte of data, you will have to split the address into the cache tag, cache index and byte select. Which parts of the address would you associate with each component, how long will each component be (in bits) and why? (Not: Assume there are no modifiers bits in the table.)




- *1st field: byte select: 3 bits (8 bytes of data = 2^3) [1 point]*
- *2nd field: cache index: 4 bits (16 cache blocks = 2^4) [1 point]*
- *3rd field: cache tag: 9 bits (16 – 4 – 3) [2 point]*

*1 point for the mechanism for calculating the tag size*

*1 point for identifying that the data cache indexes using physical addresses (16 bits)*

*Some students had assumed that the 16 cache blocks were for the entire cache. Which would make the cache index to be 2 bits (4 blocks per set) and the cache tag to be 11 bits.*

*It is important the structure of the address be split as Cache Tag – cache index – byte select. Switching the tag and index will reduce entropy and lead to cache misses.*

c) (3 points) The main memory access time is 100ns, and the cache lookup time is 50ns. Assuming a cache hit rate of 90%, what is the average time to read a location from main memory? (Note: Assume the cache hit rate is the same for the data and the page translation tables.)

*Each access requires three memory reads, one for 1st level page table, one for the 2nd level page tables, and one for accessing that memory data. For each memory access, the hit rate is 90%, and thus the*

*average access time = 0.9*50ns + 0.1*(50+100)ns = 60ns.*

*Since each read requires three accesses to the memory,*

*average read time = 3*60ns = 180ns*

*If the cache lookup and memory reads were assumed to be parallel, then the average access time would be 55ns.*

d) (3 points) To speed up the address translation process we introduce a TLB that has an access time of 20ns. Assuming the TLB hit rate is 95%, what is the average access time for a memory operation?

*If we have a TLB hit, then*

*average read time = 0.95*(20ns + 60ns) + 0.05*(20ns + 180ns) = 86ns*

1 point for applying the formula correctly
1 point for ensuring the address translation is 2 memory reads
1 point for adding the time for final memory location read.

3. (12 points) **Synchronization:** A common parallel programming pattern is to perform processing in a sequence of parallel stages: all threads work independently during each stage, but they must synchronize at the end of each stage at a synchronization point called a ***barrier***. If a thread reaches the barrier before all other threads have arrived, it waits. When all threads reach the barrier, they are notified and can begin the execution on the next phase of the computation.

Example:

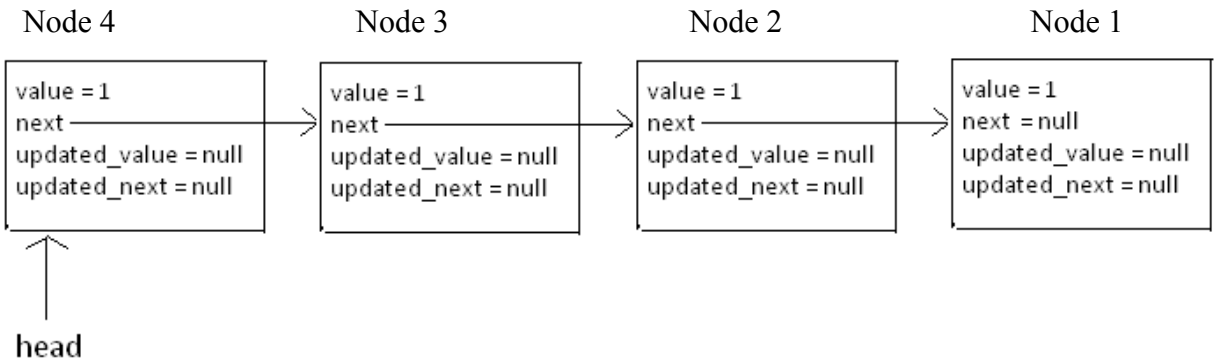
```
while (true) {
        Compute stuff;
        BARRIER();
        Read other threads results;
}
```

a) (4 points) The following implementation of Barrier is incomplete and has two lines missing. Fill in the missing lines so that the Barrier works according to the prior specifications.

```
class Barrier() {
    int numWaiting = 0;        // Initially, no one at barrier
    int numExpected = 0;       // Initially, no one expected
    Lock L = new Lock();
    ConditionVar CV = new ConditionVar();

    void threadCreated() {
        L.acquire();
        numExpected++;
        L.release();
    }
    void enterBarrier() {
        L.acquire();
        numWaiting++;
        if (numExpected == numWaiting) { // If we are the last
            numWaiting = 0;      // Reset barrier and wake threads

            CV.broadcast();      // Fill me in
        } else {     // Else, put me to sleep

            CV.wait(L);          // Fill me in
        }
        L.release() ;
    }
}
```

*2 points for CV.broadcast() or CV.wakeAll()*
*1 point for CV.signal() or CV.wake()*

b) (5 points) Now, let us use ***Barrier*** in a parallel algorithm. Consider the linked list below:




In our parallel algorithm, there are four threads (Thread 1, Thread 2, Thread 3, Thread 4). Each thread has its own instance variable ***node***, and all threads share the class variable ***barrier***. Initially, Thread 1's ***node*** references Node 1, Thread 2's ***node*** references Node 2, Thread 3's ***node*** references Node 3, and Thread 4's ***node*** references Node 4.

In the initialization steps, **barrier.threadCreated()** is called once for each thread created, so we have **barrier.numExpected == 4** as a starting condition.

Once all four threads are initialized, each thread calls its **run()** method. The **run()** method is identical for all threads:

```
void run() {
     boolean should_print = true;
     while (true) {
          if (node.next != null) {
               node.updated_value = node.value +
                                    node.next.value;
               node.updated_next = node.next.next;
          } else if (should_print) {
                System.out.println(node.value);
                should_print = false;
          }
          barrier.enterBarrier();
          node.value = node.updated_value;
          node.next = node.updated_next;
          barrier.enterBarrier();
     }
}
```

List all the values that are printed to stdout along with the thread that prints each value. For example, "thread 1 prints 777".

*Answer:*
*Thread 1 prints 1*

*Thread 2 prints 2*
*Thread 3 prints 3*
*Thread 4 prints 4*
*Note: This is a parallel list ranking algorithm where the final value of each node is its position in the linked list.*

*1 point for "Thread 1 prints 1"*
*1 point for "Thread 2 prints 2"*
*1 point for "Thread 3 prints 3" or "Thread 3 prints 2 + null" or any other answer with an explicit explanation of what happens when you add a number with null*
*1 point for "Thread 4 prints 4*

c) (3 points) In an attempt to speed-up the parallel algorithm from the previous part (2c), you notice that the line **barrier.enterBarrier()** occurs twice in **run()**'s while loop. Can one of these two calls to **barrier.enterBarrier()** be removed while guaranteeing that the output of the previous part (2c) remains unchanged? If your answer is "yes", specify whether you would remove the first or second occurrence of **barrier.enterBarrier()**.

*Answer: no*

4. (24 points total) **CPU scheduling.** Consider the following **single-threaded** processes, arrival times, and CPU processing requirements:

| Process ID (PID) | Arrival Time | Processing Time |
|---|---|---|
| 1 | 0 | 6 |
| 2 | 2 | 4 |
| 3 | 3 | 5 |
| 4 | 6 | 2 |

a) (12 points): For each scheduling algorithm, fill in the table with the ID of the process that is running on the CPU. Each row corresponds to a time unit.
- For time slice-based algorithms, assume one unit time slice.
- When a process arrives it is immediately eligible for scheduling, e.g., process 2 that arrives at time 2 can be scheduled during time unit 2.
- If a process is preempted, it is added at the tail of the ready queue.

| Time | FIFO | RR | SJF |
|---|---|---|---|
| **0** | 1 | 1 | 1 |
| **1** | 1 | 1 | 1 |
| **2** | 1 | 1 | 1 |
| **3** | 1 | 2 | 1 |
| **4** | 1 | 1 | 1 |
| **5** | 1 | 3 | 1 |
| **6** | 2 | 2 | 4 |
| **7** | 2 | 1 | 4 |
| **8** | 2 | 3 | 2 |
| **9** | 2 | 4 | 2 |
| **10** | 3 | 2 | 2 |
| **11** | 3 | 1 | 2 |
| **12** | 3 | 3 | 3 |
| **13** | 3 | 4 | 3 |
| **14** | 3 | 2 | 3 |
| **15** | 4 | 3 | 3 |
| **16** | 4 | 3 | 3 |

*-4 for using the wrong algorithm (e.g., SRTF instead of SJF)*

*-1 for those who assumed that the newly arrived processes go to the front of the queue (unlike previous years, the problem expects processes to be added to the end of the queue when they arrive)*
*-1 for each mistaken sequence (of processes).*

b) (6 points): Calculate the response times of individual processes for each of the scheduling algorithms. The response time is defined as the time a process takes to complete after it arrives.

| | PID 1 | PID 2 | PID 3 | PID 4 |
|---|---|---|---|---|
| **FIFO** | 6 | 8 | 12 | 11 |
| **RR** | 12 | 13 | 14 | 8 |
| **SJF** | 6 | 10 | 14 | 2 |

*-0.5 for each mistake*

c) (6 points) Consider same processes and arrival times, but assume now a processor with **two** CPUs. Assume CPU 0 is busy for the first two time units. For each scheduling algorithm, fill in the table with the ID of the process that is running on each CPU.

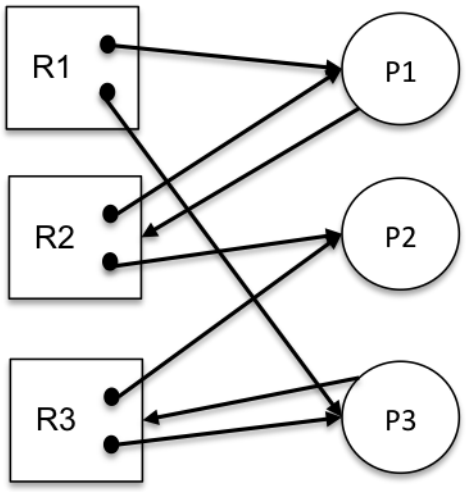
- For any non-time slice-based algorithm, assume that once a process starts running on a CPU, it keeps running on the same CPU till the end.
- If both CPUs are free, assume CPU 0 is allocated first.

| Time | CPU # | FIFO | RR | SJF |
|---|---|---|---|---|
| **0** | **0** | X | X | X |
| | **1** | 1 | 1 | 1 |
| **1** | **0** | X | X | X |
| | **1** | 1 | 1 | 1 |
| **2** | **0** | 2 | 1 | 2 |
| | **1** | 1 | 2 | 1 |
| **3** | **0** | 2 | 1 | 2 |
| | **1** | 1 | 2 | 1 |
| **4** | **0** | 2 | 3 | 2 |
| | **1** | 1 | 1 | 1 |
| **5** | **0** | 2 | 2 | 2 |
| | **1** | 1 | 3 | 1 |

| | | | | |
|---|---|---|---|---|
| **6** | **0** | 3 | 1 | 4 |
| | **1** | 4 | 2 | 3 |
| **7** | **0** | 3 | 3 | 4 |
| | **1** | 4 | 4 | 3 |
| **8** | **0** | 3 | 3 | |
| | **1** | | 4 | 3 |
| **9** | **0** | 3 | 3 | |
| | **1** | | | 3 |
| **10** | **0** | 3 | | |
| | **1** | | | 3 |

*-2 for using the wrong algorithm (e.g., SRTF instead of SJF)*
*-1 for those who assumed that the newly arrived processes go to the front of the queue (unlike previous years, the problem expects processes to be added to the end of the queue)*
*-1 for each mistaken sequence (of processes).*

5. (12 points) **Deadlock**: Consider the following resource allocation graph:




a) (3 points) Does the above allocation graph contain a deadlock? Explain your answer using no more than ***two*** sentences.

*No. A possible execution sequence is P2, P1, P3.*

*1 point for yes/no.*
*2 points for a possible execution sequence, or an explanation that the graph can not fulfill four deadlock conditions.*

b) (3 points) Assume now that P2 also demands resource R1. Does this allocation graph contain a deadlock? Explain your answer using no more than ***two*** sentences.

Yes. If P2 demands R1, none of P1, P2, or P3 will get all resource they need to be processed. Therefore, there is a deadlock.

1 point for yes/no.
2 points for the explanation that all four deadlock conditions are met, or any reasonable explanation about no thread can proceed.

c) (3 points) Assume the allocation graph at point b), and, in addition, assume that R2 has now three instances. Does this allocation graph contain a deadlock? Explain your answer using no more than ***two*** sentences.

*No. The only possible execution sequence is P1, P2, P3.*

*1 point for yes/no.*
*2 points for a possible execution sequence, or an explanation that the graph can not fulfill four deadlock conditions.*

d) (3 points) Add to the original allocation graph an additional process P4 that demands an instance of R1. Does the allocation graph contain a deadlock? Explain your answer using no more than ***two*** sentences.

*No. A possible execution sequence is P2, P1, P3, P4.*

*1 point for yes/no.*
*2 points for a possible execution sequence. Or any reasonable explanation based on (a).*

6. (14 points) **Caching**: Consider a memory consisting of four pages (frames), and consider the following reference stream of virtual pages A, B, C, D, E, C, A, B, C, D, F.

a) (4 points) Consider the LRU page replacement algorithm. Fill in the following table showing all page faults. What is the number of page faults?

| Ref Page | A | B | C | D | E | C | A | B | C | D | F |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | | | E | | | | | D | |
| 2 | | B | | | | | A | | | | F |
| 3 | | | C | | | | | | | | |
| 4 | | | | D | | | | B | | | |

*9 page faults.*

*-1 for every wrong page fault*
*-1 for not writing the number of page faults*

b) (4 points) Consider now the MIN page replacement algorithm. Assume the reference stream continues with virtual pages A, C, B, F (i.e., the entire reference stream is A, B, C, D, E, C, A, B, C, D, F, *A, C, B, F*). Fill in the following table showing all page faults. How many page faults are there?

| Ref Page | A | B | C | D | E | C | A | B | C | D | F |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | | | | | | | | | |
| 2 | | B | | | | | | | | | |
| 3 | | | C | | | | | | | | |
| 4 | | | | D | E | | | | | D | F |

*7 page faults.*

*-1 for every wrong page fault*

*-1 for not writing the number of page faults*

c) (3 points) Consider again the LRU replacement policy, and the original reference stream. What is the minimum memory size (in pages) such that the number of faults to be no larger than 6? Explain.

5 pages. Ignoring F that appears last, all pages in the reference stream fit in a 5 page memory, so for these references we only have 5 compulsory misses. Referring F will cause another page fault, thus 6 page faults in total.

*1.5 points for correct answer;*
*1.5 points for explanation.*

*We gave partial credit (1 point) if you said 6 pages.*

d) (3 points) Replace a **single** reference in the original reference stream (e.g., change the third reference from C to A) such that to reduce the number of page faults by **two** when using LRU. Show the resulting reference stream and the corresponding fault in the following table:

| Ref Page | A | B | C | D | E | C | A | C | C | D | F |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | | | E | | | | | | F |
| 2 | | B | | | | | A | | | | |
| 3 | | | C | | | | | | | | |
| 4 | | | | D | | | | | | | |

*We also gave full points if you gave an example reducing the number of page faults by **more** than two (since we made this clarification during the exam).*

*-1 point for every mistake in your page.*