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

Fall 2013 Anthony D. Joseph and John Canny

Midterm Exam #1 *Solutions*

October 21, 2013
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	8	
2	17	
3	15	
4	23	
5	18	
6	19	
TOTAL	100	

Solutions **NAME: ____**

1. (8 points total) True/False and Why? **CIRCLE YOUR ANSWER.**

i) A user-level process cannot modify its own page table entries.

TRUE FALSE

Why?

TRUE*. If a user-level process was allowed to modify its own page table entries, then it could access physical memory being used by other processes or the OS kernel. Kernel mode is required to modify page table entries. The correct answer was worth 1 points and the justification was worth an additional 1 point.*

ii) The scheduler is the part of an Operating System that determines the priority of each process.

TRUE FALSE

Why?

FALSE*. The scheduler schedules processes based on user-specified priorities.*

We accepted an answer of ***TRUE****, only if you stated that the scheduler was calculating the effective priority or performing priority donation/inheritance. The correct answer was worth 1 point and the justification was worth an additional 1 point.*

iii) Shortest Remaining Time First is the best preemptive scheduling algorithm that can be implemented in an Operating System.

TRUE FALSE

Why?

FALSE*. SRTF cannot be implemented because it requires knowledge of the future. The correct answer was worth 1 points and the justification was worth an additional 1 point.*

iv) The working set model is used to compute the *average* number of frames a job will need in order to run smoothly without causing thrashing.

TRUE FALSE

Why?

FALSE*. The working set model is used to compute the minimum (total) number of frames a job will need in order to run smoothly without causing thrashing. The correct answer was worth 1 points and the justification was worth an additional 1 point.*

2. (17 points total) Deadlock.

a. (11 points total) Recall the various deadlock detection and prevention algorithms we've discussed in this course, and consider the following snapshot of a system with five processes (P1, P2, P3, P4, P5) and four resources (R1, R2, R3, R4). There are no current outstanding queued unsatisfied requests.

Currently Available Resources

R1	**R2**	**R3**	**R4**
2	1	2	0

	Current Allocation				**Max Need**				**Still Needs**			
Process	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**
P1	0	0	1	2	0	0	3	2	0	0	2	0
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	0	3	4	6	6	5	6	6	6	2	2
P4	2	3	5	4	4	3	5	6	2	0	0	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

i) (5 points) Is this system currently deadlocked, or can any process become deadlocked? Why or why not? If not deadlocked, give an execution order.

Using the Banker's algorithm, the system is not deadlocked and will not become deadlocked. The process finishing order is: P1, P4, P5, P2, P3. We awarded no credit for saying the system is deadlocked or could become deadlocked, and we deducted 3 points for an incorrect finishing order. We deducted 1 point for minor errors.

ii) (3 points) If a request from a process P1 arrives for (0, 4, 2, 0), can the request be immediately granted? Why or why not? If yes, show an execution order.

Three points for correct answer: No, the request is ***invalid****, as it would exceed the maximum need that P1 specified. We deducted one point for saying no, but giving a different explanation (e.g., not enough available resources).*

iii) (3 points) If a request from a process P2 arrives for (0, 1, 2, 0), can the request be immediately granted? Why or why not? If yes, show an execution order.

Three points for correct answer: No, the request is valid but if granted, the resulting Currently Available Resources would be (2, 0, 0, 0) and there is no sequence of process executions that would allow the completion of all processes. This is an UNSAFE state. We only deducted 1 pt for Yes answers, if you stated that the system was in an unsafe state and the only valid order was executing P2, but deducted 2 pts if you only said Yes. We deducted 2 points if you said No and tried to adjust the Max Needs or Still Needs vectors or give a full/partial execution order.

b. (6 points) Briefly *in at most three sentences each* describe two approaches to avoiding deadlock.

Approach #1:

Deadlock can be avoided by using ordering on resource acquisition, or by using the Banker's Algorithm, or by removing any of the four requirements for Deadlock. Unacceptable answers included: using a deadlock detection algorithm, providing infinite resources, and running just one process at a time.

Approach #2:

Solutions **NAME: ____**

3. (15 points total) Demand Paging
   For each of the following page replacement policies, list the total number of page faults and fill in the contents of the page frames of memory after each memory reference.

   a. (5 points) MIN page replacement policy:

Reference	**E**	**D**	**H**	**B**	**D**	**E**	**D**	**A**	**E**	**B**	**E**	**D**	**E**	**B**	**G**
Page #1	**E**	*E*	*E*	*E*	*E*	*	*E*	*E*	*	*E*	*	*E*	*	*E*	***G***
Page #2	**-**	**D**	*D*	*D*	*	*D*	*	***A***	*A*	*A*	*A*	***D***	*D*	*D*	*D*
Page #3	**-**	**-**	**H**	***B***	*B*	*B*	*B*	*B*	*B*	*	*B*	*B*	*B*	*	*B*
Mark X for a fault	**X**	**X**	**X**	***X***				***X***				***X***			***X***

   Number of MIN page faults? ______________**<u>7 and G can be placed anywhere</u>**

   b. (5 points) LRU page replacement policy:

Reference	**E**	**D**	**H**	**B**	**D**	**E**	**D**	**A**	**E**	**B**	**E**	**D**	**E**	**B**	**G**
Page #1	**E**	*E*	*E*	***B***	*B*	*B*	*B*	***A***	*A*	*A*	*A*	***D***	*D*	*D*	***G***
Page #2	**-**	**D**	*D*	*D*	*	*D*	*	*D*	*D*	***B***	*B*	*B*	*B*	*	*B*
Page #3	**-**	**-**	**H**	*H*	*H*	***E***	*E*	*E*	*	*E*	*	*E*	*	*E*	***E***
Mark X for a fault	**X**	**X**	**X**	***X***		***X***		***X***		***X***		***X***			***X***

   Number of LRU page faults? ______________**<u>9</u>**

   c. (5 points) FIFO page replacement policy:

Reference	**E**	**D**	**H**	**B**	**D**	**E**	**D**	**A**	**E**	**B**	**E**	**D**	**E**	**B**	**G**
Page #1	**E**	*E*	*E*	***B***	*B*	*B*	*B*	***A***	*A*	*A*	*A*	***D***	*D*	*D*	*D*
Page #2	**-**	**D**	*D*	*D*	*	***E***	*E*	*E*	*	***B***	*B*	*B*	*B*	*	***G***
Page #3	**-**	**-**	**H**	*H*	*H*	*H*	***D***	*D*	*D*	*D*	***E***	*E*	*	*E*	
Mark X for a fault	**X**	**X**	**X**	***X***		***X***	***X***	***X***		***X***	***X***	***X***			***X***

   Number of FIFO page faults? ______________**<u>11</u>**
   <u>We deducted 1pt for minor errors (e.g., fault count) and 4pts for incorrect algorithms.</u>

4. (23 points) Memory management:

a. (5 points) Consider a memory system with a cache access time of 10ns and a memory access time of 110ns – assume the memory access time includes the time to check the cache. If the effective access time is 10% greater than the cache access time, what is the hit ratio *H*? (**fractional answers are OK**)

Effective Access Time $= H*T_{cache} + (1-H) * T_{memory}$
$1.1 * T_{cache} = H*T_{cache} + (1-H) * T_{memory}$
$1.1 \times 10 = H*10 + (1-H)110$
$11 = H*10 + 110 - 110* H$
$-99 = -100*H$
$H = 99/100$

We deducted 1pt for minor errors, 3 points for the correct formula but incorrect calculations, and 4 pts for the right idea but wrong formula and calculation.

b. (18 points total) Address Translation:

i) (5 points) Consider a machine with a physical memory of 8 GB, a page size of 8 KB, and a page table entry size of 4 bytes. How many levels of page tables would be required to map a 46-bit virtual address space if every page table fits into a single page? Be explicit in your explanation.

Since each PTE is 4 bytes and each page contains 8KB, then a one-page page table would point to 2048 or 2^{11} pages, addressing a total of $2^{11} * 2^{13} = 2^{24}$ bytes. Continuing this process:

Depth	*Address Space*
1	2^{24} *bytes*
2	2^{35}*bytes*
3	2^{46}*bytes*

We deducted 3 pts if your calculation uses the physical memory size to determine the bits required for virtual page number, 3pts for the right idea but wrong interpretation (leading to the wrong number of levels), and 3 to 4 pts for a major errors depending on whether you answer was on the right track or not.

ii) (4 points) List the fields of a Page Table Entry (PTE) in your scheme.

Each PTE will have a pointer to the proper page, PPN, (worth 3 pts) plus several bits – read, write, execute, (0.5 pts for protection bits), and valid (0.5 pts). This information can all fit into 4 bytes, since if physical memory is 2^{33} bytes, then 20 bits will be needed to point to the proper page, leaving ample space (12 bits) for the information bits.

If you added incorrect fields (VPN, offset, …), we subtracted one point for each incorrect field.

iii) (3 points) *Without a cache or TLB*, how many memory operations are required to read or write a single 32-bit word?

Without extra hardware, performing a memory operation takes 4 actual memory operations: 3 page table lookups in addition to the actual memory operation. We did not award partial credit for this problem. We deducted 1 pt if you omitted the actual memory operation for the 32-bit word. If your answer was incorrect, but consistent with prior parts, we deduced 2 pts.

iv) (6 points) How much physical memory is needed for a process with three pages of virtual memory (for example, one code, one data, and one stack page)?

Six physical pages totaling 48KB are needed: one for the first-level page table, one for one page of the second-level page table, one for one page of the third-level page table, and three for the process' three pages.

*Note that the second- and third-level page tables do not need 16MB (2^{11} * 8KB) each, because the top-level (and second-level) page tables enable you to only have the next level page table pages for those pages that are part of the process's virtual address space.*

For partially correct answers, we subtracted three points for not including the page table memory, 2 pts for having more than one top level page table, 1 pt for not including the data pages, 1 pt for minor errors and 1 pt if your answer was not consistent with prior parts.

5. (18 points total) Scheduling.
Assumptions: All timeslice-based algorithms have a timeslice of one unit; The currently running thread is not in the ready queue while it is running; An arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it. Turnaround time is defined as the time a process takes to complete after it arrives.

Fill in ALL blanks in EACH table – each blank has an unambiguous answer.
For the missing schedulers, the possibilities are **SRTF, RR, and Priority.**
Priority is a preemptive scheduler.
Hint: Fill in the entry time (below) for Thread C first!

Entry Times	
A	1
B	2
C	*5*
D	8

Priorities	
A	3
B	4
C	5
D	6

We deducted 6 pts for the wrong entry time for C, 4 pts for each incorrect scheduler name, 1 or 2 pts each for minor/major errors in ordering and the avg TRT calculation .

↓Current Time	**Currently Scheduled Process**		
Scheduler →	**FIFO**	***SRTF***	***Priority***
1	A	A	A
2	A	A	B
3	A	A	B
4	B	*B*	*B*
5	B	*C*	*C*
6	B	*B*	*A*
7	C	*B*	*A*
8	D	*D*	*D*
9	D	*D*	*D*
10	D	*D*	*D*
Avg Turnaround Time	3.5	*3.25*	*3.5*

6. (19 points) Concurrency Control.
   You are the organizer of a gaming exhibit at the E3 Electronic Entertainment Expo. You want to allow the attendees to play your startup's new game demo. You model the attendees as threads, called *players*, and your job is to synchronize access to a single copy of the game, as follows:
   - When a player arrives, he or she waits in a waiting area.
   - Once there are 4 or more players waiting to play, you allow *exactly* 4 of them to leave the waiting area to begin playing. These four leave the waiting area and approach the game console.
   - When a player reaches the console, the player waits until all four players are at the console, at which point all four players begin playing.
   - Players may finish playing at any time. However, you cannot allow any new players to begin playing until all four players have left.
   - You do not need to let players out of the waiting area in the order in which they arrived.
   - You cannot assume that a player will ever finish playing.

   You decide to solve this synchronization problem using two custom synchronization primitives, which have "barrier-like" semantics:

```
GameBarrier gb;
ConsoleBarrier cb;
```

   Your task is to implement these synchronization primitives according to the specifications listed below. Each player thread has access to the two global barriers, and uses them in the following sequence:

```
void Player(ThreadID tid, GameBarrier gb, ConsoleBarrier cb) {
      gb.waitToPlay();
      cb.waitAtConsole();
      play();
      gb.donePlaying();
}
```

   The `GameBarrier` can be in one of three states:
   - `GAME_NOTREADY`: There are fewer than 4 players waiting to play. When the barrier is in this state, no player can progress beyond `waitToPlay()`.
   - `GAME_FILLING`: There are (or were) at least 4 players waiting to play, and either we are in the first turn or else all four players from the prior turn have departed (via `donePlaying()`). When the barrier is in this state, a player can progress beyond `waitToPlay()`, and in fact four players must progress beyond this function. When exactly four players have progressed beyond this function, the barrier enters the `GAME_FILLED` state.
   - `GAME_FILLED`: Four players have been sent to the console, and the turn is not over (meaning that the departure of all four from the console via `donePlaying()` has not yet taken place). When the barrier is in this state, no waiting player can progress beyond `waitToPlay()` .

The `ConsoleBarrier` can be in one of two states:

- `CONSOLE_WAIT`: Four players have not yet arrived at the console in the current round. When the barrier is in this state, no player can progress beyond `waitAtConsole()`.
- `CONSOLE_ALLOW`: Four players have arrived in the current round. When the barrier is in this state, all four waiting players must progress beyond `waitAtConsole()`, after which the state reverts to `CONSOLE_WAIT`.

Your barrier will require the use of condition variables. Recall that condition variables provide three methods: `Condition.wait(Lock mutex)`, `Condition.signal()`, and `Condition.broadcast()`.
Locks provide two methods: `Lock.acquire()` and `Lock.release()`.

Note that part (but not all) of your work is to ensure that the barriers make the correct state transitions.

Below, where indicated, fill out the variables and methods for the `GameBarrier` and `ConsoleBarrier` objects.

```
public class GameBarrier {
     private static final int GAME_NOTREADY = 0;
     private static final int GAME_FILLING = 1;
     private static final int GAME_FILLED = 2;

     private Lock mutex;
     private int state;
     private Condition cv;

     /* SPECIFY ANY OTHER CLASS VARIABLES */
     private int NumWaiters;
     private int NumPlayers;




     public GameBarrier(Lock lock) {
          this.mutex = lock;
          this.state = GAME_NOTREADY;
          this.cv = new Condition();

          /* INITIALIZE ANY OTHER CLASS VARIABLES */
          this.NumWaiters = 0;
          this.NumPlayers = 0;
```

We deducted 1 pt each if your answer was missing either `NumWaiters` *or* `NumPlayers`

```
     }
```

```
public void waitToPlay() {
      /* YOU MUST FILL IN THIS FUNCTION */
      mutex.acquire();
      NumWaiters++
      if ((NumWaiters >= 4) &&
          (state ==  GAME_NOTREADY)) {
           state = GAME_FILLING;
           cv.broadcast();
      }
      while (state != GAME_FILLING) {
           cv.wait(mutex);
      }
      NumWaiters--;
      NumPlayers++;
      if (NumPlayers == 4) state = GAME_FILLED;
      mutex.release();
```

We applied the following grading rubric:

- *-3 pts for not using the mutex or using the mutex incorrectly. We deducted 1 pt if you forgot the mutex acquire OR release but not both.*
- *-1 pt for using signal instead of broadcast. We accepted answers that signaled at least the appropriate number of times.*
- *-1 pt for not calling* `cv.wait()` *from within a* `while` *loop*
- *-3 pts for using extra synchronization primitives (e.g., another condition variable or semaphores).*
- *-1.5 pts if you had incorrect or nonexistent state transitions.*
- *-1 or -2 pts for each additional minor or major error*

```
}
```

```
public void donePlaying() {
      /* YOU MUST FILL IN THIS FUNCTION */
      mutex.acquire();
      NumPlayers--;
      if (NumPlayers = 0) {
            state = GAME_NOT_READY;
            if (NumWaiters >= 4) {
                  state = GAME_FILLING;
                  cv.broadcast(mutex);
            }
      }
      mutex.release();
```

We applied the following grading rubric:

- *-3 pts for not using the mutex or using the mutex incorrectly. We deducted 1 pt if you forgot the mutex acquire OR release but not both.*
- *-1 pt for using signal instead of broadcast. We accepted answers that signaled at least the appropriate number of times. We deducted 2 pts if your answer had no signals or broadcast.*
- *-2 pts for calling `cv.wait()` and blocking `donePlaying()`*
- *-1 pt for not checking NumPlaying or NumWaiters.*
- *-1.5 pts if you had incorrect or nonexistent state transitions.*
- *-1 pt for each additional error*

```
        }
    }

    public class ConsoleBarrier {
        private static final int CONSOLE_WAIT = 0;
        private static final int CONSOLE_ALLOW = 1;

        private Lock mutex;
        private int state;
        private Condition cv;

        /* SPECIFY ANY OTHER CLASS VARIABLES */
        int NumPlayers;





        public ConsoleBarrier(Lock lock) {
            this.mutex = lock;
            this.state = CONSOLE_WAIT;
            this.cv = new Condition();

            /* INITIALIZE ANY OTHER CLASS VARIABLES */
            this.NumPlayers = 0;
```

We deducted 2 pts if your answer was missing `NumPlayers`

```
        }
```

```
public void waitAtConsole() {
      /* YOU MUST FILL IN THIS FUNCTION */
      mutex.acquire();
      NumPlayers++
      if (NumPlayers == 4) {
            state =  CONSOLE_ALLOW;
            cv.broadcast();
      }
      while (state == CONSOLE_WAIT) {
            cv.wait(mutex);
      }
      NumPlayers--;
      if (NumPlayers == 0) state = CONSOLE_WAIT;
      mutex.release();
```

We applied the following grading rubric:

- *-3 pts for not using the mutex or using the mutex incorrectly. We deducted 1 pt if you forgot the mutex acquire OR release but not both.*
- *-2 pts if your answer had no signals or broadcast.*
- *-2 pts for missing `cv.wait()`*
- *-1 pt for not checking NumPlaying or NumWaiters.*
- *-1.5 pts if you had incorrect or nonexistent state transitions.*
- *-1 or -2 pts for each additional minor/major error*

```
}
```