Southern University of Science and Technology
College of Engineering
Department of Computer Science and Engineering – CSE

Spring 2018 Dr. Bo Tang

First Midterm Exam

April 11, 2018
CS302 Operating Systems

Your Name:	
Student ID:	

General Information:
This is a **closed book and one 2-sided handwritten note (A4-size)** examination. You have 120 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
P1: CPU Scheduling	14	
P2: P/C problem	22	
P3: Synchronization	16	
P4: Multithreading	15	
P5: Dining Philosophers	18	
P6 : Ture or False	15	
TOTAL	100	

P1 (14 points) CPU Scheduling.
Consider the following **single-threaded** processes, and their arrival times, estimated CPU costs and their priorities (a process with a higher priority number has priority over a process with lower priority number):

Process	Estimated CPU Cost	Arrives	Priority
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Please note:

- Priority scheduler is preemptive.
- Newly arrived processes are scheduled last for RR. When the RR quanta expires, the currently running thread is added at the end of to the ready list before any newly arriving threads.
- Break ties via priority in Shortest Remaining Time First (SRTF).
- If a process arrives at time x, they are ready to run at the beginning of time x.
- Ignore context switching overhead.
- The quanta for RR is 1 unit of time.
- Average turn-around time is the average time a process takes to complete after it arrives.

Given the above information please fill in the following table.

Time	FIFO/FCFS	Round Robin	SRTF	Priority
1	A	A	A	A
2	A	A	B	B
3	A	B	A	A
4	A	A	C	C
5	B	C	C	C
6	C	A	A	D
7	C	D	A	D
8	D	C	D	D
9	D	D	D	A
10	D	D	D	A
Avg.Turn-around Time	18/4	19/4	16/4	17/4

P2 (22 points) Producer/Consumer.
Consider the following code that implements a synchronized unbounded queue.

```
1.      semaphore  mutex;
2.      semaphore  dataready;
3.      Queue      queue;

4.      AddToQueue(item) {
5.        sem_wait(&mutex);            // Get Lock
6.        queue.enqueue(item);         // Add item
7.        sem_post(&dataready);        // Signal any waiters
8.        sem_post(&mutex);            // Release Lock
9.      }

10.     RemoveFromQueue() {
11.       while(queue.isEmpty()){
12.             sem_wait(&dataready);      // Get dataready
13.       }
14.       sem_wait(&mutex);            // Get Lock
15.       item = queue.dequeue();      // Get next item
16.       sem_post(&mutex);            // Release Lock
17.       return(item);
18.     }
```

Please answer the following questions.

a) (6 points) Assume that we have multiple producers running AddToQueue() and multiple consumers running RemoveFromQueue(). Do you need to make any changes to the code? If yes, specify the changes in the above code by indicating the line you need to modify, the line #'s between which you need to add new code, or the line # you need to delete. If not, use no more than two sentences to explain why.

We do not need to make any changes, as the code already handles multiple producers and consumers.

b) (10 points) Change the code to implement a bounded queue, i.e., make sure that the producer cannot write when the queue is full. Add your changes in the empty space of the code below.

Solution 1:

```
semaphore    mutex;
semaphore    dataready;
semaphore    queueready;
Queue queue;
```

```
AddToQueue(item) {

   while(queue.isFull()) {      //    isFull is automic
       sem_wait(&queueready);   //    If nothing, sleep
   }
   sem_wait(&mutex);            // Get Lock
   queue.enqueue(item);         // Add item
   sem_post(&dataready);        // Signal any waiters
   sem_post(&mutex);            // Release Lock
}

RemoveFromQueue() {
   while(queue.isEmpty()){
         sem_wait(&dataready);  // Get dataready
   }
   sem_wait(&mutex);            // Get Lock
   item = queue.dequeue();      // Get next item
   sem_post(&queueready)        // Signal any waiters
   sem_post(&mutex);            // Release Lock
   return(item);
}
```

Solution 2:

```
semaphore    mutex;
semaphore    dataready;
semaphore    queueready;
Queue queue;
int queuesize;

AddToQueue(item) {
   sem_wait(&mutex);            // Get Lock
   while(queuesize==N)    {
       sem_wait(&queueready);   //    If nothing, sleep
   }
   queuesize++;
   queue.enqueue(item);         // Add item
   sem_post(&dataready);        // Signal any waiters
   sem_post(&mutex);            // Release Lock
}

RemoveFromQueue() {
   while(queue.isEmpty()){
         sem_wait(&dataready);  // Get dataready
   }
   sem_wait(&mutex);            // Get Lock
   item = queue.dequeue();      // Get next item
   queuesize--;
```

```
   sem_post(&queueready)        // Signal any waiters
   sem_post(&mutex);            // Release Lock
   return(item);
}
```

c) (6 points) Implement a new function, ReadFromQueue(), which uses the function "item=queue.read()" to read an item from the queue **without removing it**.

```
ReadFromQueue()   {
      sem_wait(&mutex)              //    Get Lock
      while (queue.isEmpty()) {
            sem_wait(&dataready);   //    If nothing, sleep
      }
      item=queue.read();            //    Get next item
      sem_post(&mutex)              //    Release Lock
      return(item);
}
```

[This page intentionally left blank]

P3 (16 points) Synchronization.

Next Saturday is the international day of Poker. As the owner of the largest poker website worldwide you expect a large number of games being played (and finishing) at any point in time in your website. Consider that players can play more than one game at a time and any two players can play against each other in more than one game simultaneously. For simplicity, we consider each game has exactly **two** players.

The backend system of your poker website contains the following multi-threaded code, please note move_chips can be called by other functions.

```
Queue           games_finished_queue;
semaphore       games_finished_lock;
semaphore       games_to_process_sem;

typedef struct Game{
        ....
}       Game;
typedef struct Player{
        semaphore       mutex;
        uint64_t        n_chips;
        uint64_t        unique_id;
}Player;

void    finish_game(Game* game)     {
        sem_wait(&games_finished_lock);
        enqueue(&games_finished_queue, game);
        sem_post(&games_finished_lock);
        sem_post(&games_to_process_sem);
}

void    process_finished_games() {
        sem_wait(&games_finished_lock);
        sem_wait(&games_to_process_sem);
        Game*  g = pop_queue_front(&games_finished_queue);
        move_chips(g->player1, g->player2, g->n_chips);
        sem_post(&games_finished_lock);
}

void    move_chips(Player* player1, Player* player2, uint64_t n_chips){
        sem_wait (&player1->mutex);
        sem_wait (&player2->mutex);
        player1->n_chips -= n_chips;
        player2->n_chips += n_chips;
        sem_post(&player2->mutex);
        sem_post(&player1->mutex);
}
```

(a) (6 points) Identify two places in the code where deadlock can occur. If deadlock occurs, use no more than two sentences to explain why it occurs.

First, in `process_finished_games sem_wait(&games_to_process_sem);` can make thread wait on a critical section.

Second, `move_chips` can deadlock if two players play against each other simultaneously. This can lead to two concurrent calls with the same (but swapped) players/arguments.

```
move_chips(player1,player2,n1)
move_chips(player2,player1,n2)
```

(b) (10 points) Use the space bellow to change process_finished_games() and move_chips () (or copy if correct) to ensure no deadlocks can occur. Explain succinctly why no deadlock can occur with the newly modified code. Note: a single lock at the beginning and end of move_chips is not an accepted solution.

```
void    process_finished_games() {
        // acquire semaphore outside of critical section
        sem_wait(&games_to_process_sem);
        sem_wait(&games_finished_lock);
        Game*  g = pop_queue_front(games_finished_queue);
        move_chips(g->player1, g->player2, g->n_chips);
        sem_post(&games_finished_lock);
}

void    move_chips(Player* player1, Player*   player2, uint64_t n_chips) {
        // acquire locks in well defined order
        If (player1->unique_id < player2-unique_id) {
                sem_wait (&player1->mutex);
                sem_wait (&player2->mutex);
        }else{
                sem_wait (&player2->mutex);
                sem_wait (&player1->mutex);
        }
        player1->n_chips -= n_chips;
        player2->n_chips += n_chips;
        sem_post(&player1->mutex);
        sem_post(&player2->mutex);
}
```

[This page intentionally left blank]

P4 (15 points) Multithreading.
Consider the following two threads, to be run concurrently in a shared memory (all variables are shared between the two threads):

Thread A	Thread B

for (i=0; i<5; i++) {
  x = x + 1;
}
	for (j=0; j<5; j++) {
	  x = x + 2;
	}

Assume a single-processor system, that load and store are atomic, that x is initialized to 0 *before either thread starts*, and that x must be loaded into a register before being incremented (and stored back to memory afterwards). The following questions consider the final value of x after both threads have completed.

a) (3 points) Give a *concise* proof why x≤15 when both threads have completed.

Each x=x+1 statement can either do nothing (if erased by Thread B) or increase x by 1.
Each x=x+2 statement can either do nothing (if erased by Thread A) or increase x by 2.
Since there are 5 of each type, and since x starts at 0, x is >= 0 and x <= (5*1)+(5*2)=15

b) (3 points) Give a *concise* proof why x≠1 when both threads have completed.

Every store into x from either Thread A or B is >= 0, and once x becomes >= 0, it stays >= 0. The only way for x=1 is for the last x=x+1 statement in Thread A to load a zero and store a one. However, there are at least four stores from Thread A previous to the load for the last statement, meaning that it could not have loaded a zero.

c) (3 points) Suppose we replace '**x = x+2**' in Thread B with an atomic double increment operation **atomicIncr2(x)** that cannot be preempted while being executed. What are all the possible final values of x? Explain.

Final values are 5, 7, 9, 11, 13, or 15. The x=x+2 statements can be "erased" by being between the load and store of an x=x+1 statement. However, since the x=x+2 statements are atomic, the x=x+1 statements can never be "erased" because the load and store phases of x=x+2 cannot be separated. Thus, our final value is at least 5 (from Thread A) with from 0 to 5 successful updates of x=x+2.

d) (3 points) What needs to be saved and restored on a context switch between two threads in the same process? What if the two threads are in different processes? Be explicit.

Need to save the processor registers, stack pointer, program counter into the TCB of the thread that is no longer running. Need to reload the same things from the TCB of the new thread.
When the threads are from different processes, need to not only save and restore what was given above, but you also need to load the pointer for the top-level page-table of the new address space. You do not need to save the old pointer, since this will not change and is already stored in the PCB.

e) (3 points) Under what circumstances can a multithreaded program complete more quickly than a non-multithreaded program? Keep in mind that multithreading has context-switch overhead associated with it.

When there is a lot of blocking that may occur (such as for I/O) and parts of the program can still make progress while other parts are blocked.

P5 (18 points) Dining Philosophers.
The goal of this exercise is to implement a solution to the Dining Philosophers problem – 5 philosophers sitting at a round table. Philosophers repeat (forever) the following three things in order: (1) think, (2) eat, and (3) sleep. Each philosopher has a plate of spaghetti in front of him or her and a chopstick between each plate.

Important rule: A philosopher must have two (2) chopsticks to be able to eat! A philosopher must acquire the chopstick to the left of their plate and the chopstick to the right of their plate; they cannot reach across the table to use other chopsticks. When they are done eating, they put down (and free) the two chopsticks for someone else to use. Multiple philosophers should be able to acquire chopsticks at the same time.

a) (5 points) Solution A: When philosopher X is hungry, she has to check if anyone is using the chopsticks that you need. If yes, she waits. If no, seize both chopsticks. After eating, put down all your chopsticks. Does this solution work fine (i.e., without deadlock), if yes, explain, if no, please give an concrete example to show in which case we will have deadlock.

No, it has deadlock.
(1) Each philosopher finishes thinking at the same time and each first grabs her left chopstick
(2) All chopsticks[i]=0
(3) When taking the right chopstick, all are waiting

b) (5 points) Solution B: First, a philosopher takes a chopstick. If a philosopher finds that she cannot take the second chopstick, then she should put it down. Then, the philosopher goes to sleep for a while. When wake up, she retries Loop until both chopsticks are seized. Does this solution will have deadlock? If yes, explain, if no, please which requirement will be broken in this solution.

No, philosophers are all busy (no deadlock), but no progress (starvation)
(1) all pick up their left chopsticks,
(2) seeing their right chopsticks unavailable (because P1's right chopstick is taken by P2 as her left chopstick) and then putting down their left chopsticks,
(3) all sleep for a while
(4) all pick up their left chopsticks, ...

c) (8 points) Please construct a synchronization protocol such that philosophers will not starve to death, and will not result in any deadlock scenarios. You are required to describe your protocol in general words first, then write down the pseudocode of your protocol.

Idea: Need to guarantee: when "Philosopher x" is eating, the left and the right of "Philosopher x" cannot eat
Use semaphore to model Philosopher instead of Chopstick (i.e., wait for Philosopher instead of wait for chopstick)

Shared objects:

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT  ((i+1) % N)
int state[N]; // eating, thinking, Hungry
semaphore mutex = 1;
semaphore p[N] = 0;
```

main function:

```
1  void philosopher(int i) {
2     think();
3     take_chopsticks(i);
4     eat();
5     put_chopsticks(i);
6  }
```

```
1  void take_chopsticks(int i) {
2     wait(&mutex);
3     state[i] = HUNGRY;
4     captain(i);
5     post(&mutex);
6     wait(&p[i]);
7  }
```

```
1  void put_chopsticks(int i) {
2     wait(&mutex);
3     state[i] = THINKING;
4     captain(LEFT);
5     captain(RIGHT);
6     post(&mutex);
7  }
```

```
1 void captain(int i) {
2       if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING) {
3       state[i] = EATING;
4       post(&p[i]);
5    }
6 }
```

```
1 void wait(semaphore *s) {
2    disable_interrupt();
3          *s = *s – 1;
4    if ( *s < 0 ) {
5      enable_interrupt();
6      sleep();
7      disable_interrupt();
8     }
9    enable_interrupt();
10  }
```

```
1 void post(semaphore *s) {
2   disable_interrupt();
3   *s = *s + 1;
4   if ( *s <= 0 )
5     wakeup();
6   enable_interrupt();
7  }
```

P6 (15 points) True/False and Why?
CIRCLE YOUR ANSWER AND WRITE AN EXPLANATION (no credit will be given if no explanation is provided). It is important that you *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!).

(a) When you type Ctrl+C to quit a program in your terminal, you are actually sending a SIGINT signal to the program, which makes it quit.

True / False
Explain
SIGINT is a vehicle for conveying a Ctrl-C requests from the terminal to the program. Assuming that the SIGINT handler has not be redirected, this will cause the program to quit. Note: we would also take "False" if you explain that the SIGINT handler might have been redirected.

(b) Two processes can share information by reading and writing from a shared linked list.

True / False
Explain
If a shared page is mapped into the same place in the address space of two processes, then they can share data structures that utilize pointers as long as they are stored in the shared page and pointers are to structures in the shared page.

(c) Suppose that a shell program wants to execute another program and wait on its result. It does this by creating a thread, calling exec from within that thread, then waiting in the original thread.

True / False
Explain

The shell program must create a new process (not thread!) before calling exec() otherwise, the exec() call will terminate the existing process and start a new process – effectively terminating the shell.

(d) Peterson's solution suffers from priority inversion problem.

True / False
Explain
A low priority process L is inside the critical region, but a high priority process H gets the CPU and wants to enter the critical region. But H cannot lock (because L has not unlock) So, H gets the CPU to do nothing but spinning

(e) If program A's real time is larger than the sum of user time and system time, it means program A is CPU bounded.

True / **False**

Explain

Real time is wall clock time, user time is the amount of time spend in user-mode within the process, system time is the CPU time spend in the kernel within the process. If real time > user time + system time, the process is I/O bound. Execution on multiple cores would be of little to no advantage.

[This remaining page left for π]

3.14159265358979323846264338327950288419716939937510582O.....