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

Fall 2016 Anthony D. Joseph

Midterm Exam #1 *Solutions*

September 28, 2016
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	21	
2	11	
3	19	
4	22	
5	27	
TOTAL	100	

Solutions **NAME: _____**

1. (21 points total) True/False and Why?
   a. (8 points) True/False and Why? **CIRCLE YOUR ANSWER.**
      i) When a user program performs a system call, it must first put the system call arguments onto the kernel stack and then run a special instruction to switch to kernel mode.

TRUE FALSE

Why?

FALSE. *Arguments are put on the thread's stack, not kernel stack. The correct answer was worth 1 point and the justification was worth an additional 1 point.*

      ii) On a system with a single processor core without hyperthreading, you can speed up a single-threaded program by utilizing multiple threads.

TRUE FALSE

Why?

TRUE. *Hypertrhreading enables the overlap of I/O and computation. The correct answer was worth 1 points and the justification was worth an additional 1 point.*

      iii) When a user program successfully calls `execv()`, its process is replaced with a new program. Everything about the old process is destroyed, including all of the CPU registers, program counter, stack, heap, background threads, file descriptors, and virtual address space.

TRUE FALSE

Why?
FALSE. *File descriptors are not destroyed by* `execv()`. *The correct answer was worth 1 points and the justification was worth an additional 1 point.*

iv) The only way a user thread can transfer control to kernel mode is through syscalls, exceptions, and interrupts.

TRUE FALSE

Why?
TRUE. *For security, threads can only transfer to kernel mode through voluntary syscalls, and involuntary exceptions and interrupts. The correct answer was worth 1 points and the justification was worth an additional 1 point.*

b. (13 points) Short answer.

i) (4 points) Briefly, in two to three sentences, explain how an Operating System kernel uses dual mode operation to prevent user programs from overwriting kernel data structures.

Dual mode operation combined with address translation prevents user programs from overwriting kernel data structures by limiting the operations that user programs can perform. In particular, they cannot execute the privileged instructions that change address translation..

ii) (3 points) Briefly, in two to three sentences, explain the purpose of the `while` loop in the context of Mesa-style condition variables.
With Mesa-style condition variables, the signaler simply places the signaled thread on the ready queue. Since another thread could run before the signaled thread and could change the condition that caused the signaled thread to wait, the signaled thread must check the condition using a `while` *loop.*

iii) (4 points) Briefly, in two to three sentences, explain the problems with the Therac-25 – be specific but brief in your answer.
Software errors caused the deaths and injuries of several patients. There were a series of race conditions on shared variables and poor software design that lead to the machine's malfunction under certain conditions.

iv) (2 points) In a typical OS, one way that a thread can transition from the 'running' state to the 'waiting/blocked' state is to wait on acquiring a lock. Other than calling sleep or using semaphores or monitors, what are **TWO** other different *types* of ways that a thread can transition from the 'running' state to the 'waiting/blocked' state?
By making an I/O call, or calling `wait()` *on a child process*

2. (11 points total) C Programming.
Consider the following C program:

```
int thread_count = 0;

void *thread_start(void *arg) {
   thread_count++;
   if (thread_count == 3) {
       char *argv[] = {"/bin/ls", NULL};
       execv(*argv, argv);
   }
   printf("Thread: %d\n", thread_count);
   return NULL;
}

int main(int argc, char *argv[]) {
   int i;
   for (i = 0; i < 10; i++) {
       pthread_t *thread = malloc(sizeof(pthread_t));
       pthread_create(thread, NULL, &thread_start, NULL);
       pthread_join(*thread, NULL);
   }
   return 0;
}
```

a. (6 points) When you run the program, what will be the output (assume that all system calls succeed)?
The key part of this question is that exec will replace the current process, so no more threads will be created. The output will be:

```
Thread: 1
Thread: 2
(contents of current directory)
```

b. (5 points) If we removed the line `pthread_join(*thread, NULL)`, could the output change? If so, explicitly describe the way(s) it might differ.
Because the threads are created but aren't waited for, they will execute in a non-deterministic order. So, we could get 0 or more printings of:

```
Thread: {1…10}
 (contents of current directory zero or one times)
```

3. (19 points total) Finite Synchronized Queue using Monitors.

In lecture we implemented an Infinite Synchronized Queue using a monitor. In this exam question you will implement a Finite Synchronized Queue (FSQ) using a monitor. The FSQ has Last-In, First-Out semantics (like a stack), a fixed maximum `capacity` and supports three operations: `CreateQueue()`, `AddToQueue()`, and `RemoveFromQueue()`.

a. (6 points) What are the correctness constraints for the Finite Synchronized Queue?

1. *If the queue is at capacity (full), then `AddToQueue()` must wait*

2. *If the queue is empty, then `RemoveFromQueue()` must wait*

3. *Only one thread may manipulate the queue at a time*

Each element of the FSQ is a `node`:

```
typedef struct {
   int data;
   node *next;
} Node;
```

b. (3 points) Modify the `FSQ struct` to support a Finite Synchronized Queue.

You may assume that you have `struct` types for the components of a monitor.

```
typedef struct {
   Node *head;           /* pointer to head of queue */
   int length;           /* number of nodes in queue */
   int capacity;         /* max size of queue */
   Lock lock;
   Condition dataAvailable;
   Condition spaceAvailable;
```

```
} FSQ;
```

We have implemented the `CreateQueue(int size)` function for you. *You may assume that the `head`, `length`, `capacity`, and monitor components are initialized.*

c. (5 points) Implement the `AddToQueue()` function. *Feel free to use pseudocode for monitor operations.*

```
void AddToQueue(FSQ *queue, Node *data) {
   LockAcquire(&queue->lock);
   while(queue->length == queue->capacity)
        wait(&queue->spaceAvailable, &queue->lock);
   data->next = queue->head;
   queue->head = data;
   queue->length++;
   signal(&queue->dataAvailable);
   LockRelease(&queue->lock);









}
```

d. (5 points) Implement the `RemoveFromQueue()` function. *Feel free to use pseudocode for monitor operations.*

```
Node *RemoveFromQueue(FSQ *queue) {
   Node *data;
   LockAcquire(&queue->lock);
   while(queue->length == 0)
        wait(&queue->dataAvailable, &queue->lock);
   data = queue->head;
   queue->head = data->next;
   signal(&queue->spaceAvailable);
```

```
    LockRelease(&queue->lock);
    return data;
}
```

4. (22 points total) PintOS questions

a. (3 points) What is the maximum number of threads you can create in a user process in PintOS?
One; each process consists of a single thread

b. (4 points) How does PintOS preempt the currently running thread in order to schedule a new thread to run, if the currently running thread never yields?
Timer interrupts occur on each tick which lets the kernel preempt a thread that has been running too long.

c. (3 points) What is the purpose of the idle thread in PintOS? *Note that simply saying "the idle thread is the thread that runs when the system is idle" is not an acceptable answer.*
The idle thread is a placeholder thread so the kernel is always running some thread which allows scheduling logic to be implemented easily OR allows `switch_threads()` *to still work when there is no thread to switch from.*

d. (4 points) Briefly explain what happens when a timer interrupt occurs while interrupts are disabled in PintOS.
When interrupts are disabled, timer interrupts will not trigger the corresponding interrupt handler. Because of this, threads running with interrupts disabled cannot be context switched. Thus, turning interrupts off allows us to create an atomic section in our code.

e. (4 points) How does `struct thread *switch_threads (struct thread *cur, struct thread *next)` work for a thread that has just been created?
Upon creation of the thread, the stack will be initialized with two dummy frames to make it seem like it has just been context switched away from. This includes the root frame for the function `kernel_thread` *and the frame for* `switch_threads`.
The dummy frame for `switch_threads` *includes space for the registers that are normally saved on the stack when a context_switch is performed.*

```
struct switch_threads_frame
  {
    uint32_t edi;               /*  0: Saved %edi. */
    uint32_t esi;               /*  4: Saved %esi. */
    uint32_t ebp;               /*  8: Saved %ebp. */
    uint32_t ebx;               /* 12: Saved %ebx. */
    void (*eip) (void);         /* 16: Return address. */
    struct thread *cur;         /* 20: switch_threads()'s
CUR argument. */
    struct thread *next;        /* 24: switch_threads()'s
NEXT argument. */
  };
```

f. (4 points) Consider the following code for the thread_exit() function in PintOS:

```
void thread_exit (void) {
   ASSERT (!intr_context ());

#ifdef USERPROG
   process_exit ();
#endif

   /* Remove thread from all threads list, set our status to
       dying, and schedule another process.  That process
```

```
        will destroy us when it calls
        thread_schedule_tail() */
    intr_disable ();
    list_remove (&thread_current()->allelem);
    thread_current ()->status = THREAD_DYING;
    schedule ();
    NOT_REACHED ();
}
```

Briefly explain what `NOT_REACHED ()` means and why this happens.
`NOT_REACHED()` panics the kernel when it is called, so the code should never reach that point. The reason why it should never reach that point is because `schedule()` will eventually remove the dying thread and clean up its memory so it can never be run again.

5. (27 points total) Santa Claus problem (from *Operating Systems: Internals and Design Principles*)

Santa Claus sleeps in his shop and can only be woken up by either:
(1) all ten reindeer being back from their vacation or
(2) some of the elves having difficulty making toys.
The elves can only wake Santa up when 5 of them have a problem. While Santa is helping the 5 elves with their problems, any other elf who needs to meet Santa must wait till the other 5 elves have been tended to first.

If Santa wakes up to 5 elves and all ten reindeer, Santa decides that the elves must wait and focuses on getting his sled ready. The last (tenth) reindeer to arrive must get Santa while the others wait before being harnessed to the sled. *You can assume that there are only ten reindeer, but you cannot make assumptions on the number of elves.*

Assume Santa can call these methods: `prepareSleigh()`, `helpElves()`
Reindeer call `getHitched()` and Elves call `getHelp()` – both `getHitched()` and `getHelp()` are thread safe, and can be called outside of critical sections.

You MUST implement the problem using semaphores.

a. (6 points) What are the correctness constraints?
1. *After the 10th reindeer arrives, Santa must call* `prepareSleigh()`*, then all ten reindeer must call* `getHitched()`

2. *After the 5th elf arrives, Santa must invoke* `helpElves()`*. All 5 elves should invoke* `getHelp()`

3. *All 5 elves must invoke* `getHelp()` *before any other elves enter*

b. (6 points) Initialize the global variables:

```
int num_elves = 0;
int num_reindeer = 0;
santaSem = Semaphore(0);
reindeerSem = Semaphore(0);
elfSem = Semaphore(1);
lock = Semaphore(1);
```

c. (5 points) Implement the `Santa()` function:

```
Santa() {
      while(1) {
            santaSem.P();
            lock.P();
            if num_reindeer == 10 {
                  prepareSleigh();
                  reindeerSem.V(); // do this 10 times
                  num_reindeer -= 10;
            } else if (num_elves == 5) {
                  helpElves();
            }
            lock.V();
      }
}
```

Santa either has to deal with all the reindeer arriving, or a group of elves who need help. If ten reindeer are waiting, Santa calls prepareSleigh() then signals reindeerSem 10 times so that the reindeer can invoke getHitched(). If elves are waiting, Santa just calls helpElves(). He doesn't need to worry about decrementing any counter because the elves do that on their own on their way out.

d. (5 points) Implement the `ReindeerComesHome()` function:

```
ReindeerComesHome() {
        lock.P();
        num_reindeer += 1;
        if (num_reindeer == 10) {
                santaSem.V();
        }
        lock.V();
        reindeerSem.P();
        getHitched();
}
```

The 10th reindeer signals Santa then joins the other reindeer waiting on the reindeerSem. When Santa signals all the waiting reindeer, they execute getHitched().

e. (5 points) Implement the `ElfRequestsHelp()` function:

```
ElfRequestsHelp() {


        elfSem.P();
        lock.P();
        num_elves += 1;
        if (num_elves == 5) {
              santaSem.V();
        } else {
              elfSem.V();
        }
        lock.V();
        getHelp();
        lock.P();
        num_elves -= 1;
        if (num_elves == 0) {
              elfSem.V();
        }
        lock.V();
```

```
}
```

The first four elves release the elfSem at the same time they release the lock, but the last elf holds the elfSem, which prevents other elves from entering until all three elves have invoked getHelp(). The last elf to leave releases the elfSem, which allows the next group of elves to enter to request help from Santa.