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

Spring 2018 Anthony D. Joseph and Jonathan Ragan-Kelley

Midterm Exam #1 *Solutions*

February 28, 2018
CS162 Operating Systems

Your Name:	*Mitsuha Miyamizu*
SID AND 162 Login:	*162162162, s162*
TA Name:	*Taki Tachibana*
Discussion Section Time:	*4-5PM, Funday*

General Information:
This is a **closed book and one 2-sided handwritten note** examination. You have 110 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	18	
2	24	
3	14	
4	21	
5	23	
TOTAL	100	

NAME: _____

1. (18 points total) Short Answer.
   a. (10 points) True/False and Why? **CIRCLE YOUR ANSWER.**
      i) One can always replace `signal()` with `broadcast()` for Hoare-style monitors.

TRUE FALSE

Why?

FALSE. *Hoare-style monitors can function correctly with if rather than while, so* `broadcast()` *for Hoare may incorrectly wake up threads. The correct answer was worth 1 point and the justification was worth an additional 1 point.*

ii) One can disable interrupts on any computer system that supports it to guarantee atomicity.

TRUE FALSE

Why?

FALSE. *Doesn't work on multiprocessor systems. The correct answer was worth 1 point and the justification was worth an additional 1 point.*

iii) In Pintos, the scheduler code runs in the idle thread and facilitates the context switch between any two threads.

TRUE FALSE

Why?

FALSE. *The idle thread only runs when there are no other threads. The scheduler is run on the two threads' kernel stacks. The correct answer was worth 1 point and the justification was worth an additional 1 point.*

iv) The kernel first validates `syscall` arguments on the user stack before copying them to its corresponding kernel stack, in order to protect kernel memory.

TRUE FALSE

Why?

FALSE. *The arguments are checked after copying to prevent malicious user code from evading checks (TOCTOU attack). However, we also accepted TRUE provided the justification mentioned checks both before and* **after** *the copy, due to the admittedly ambiguous wording. The important thing was the justification mentioning the check after to protect against TOCTOU attacks. The correct answer was worth 1 point and the justification was worth an additional 1 point.*

v) A context switch between any two threads requires saving all the old thread's registers and replacing them all with the new thread's, including the stack pointer, the segment registers, and the page table base register.

TRUE FALSE

Why?

FALSE*.Threads within the same process would share the same address space and thus would not need to change the segment/PTBR. The correct answer was worth 1 point and the justification was worth an additional 1 point.*

b. (4 points) Briefly, in one to two sentences each, give two reasons why a typical web server creates new threads to service new connections instead of creating new processes to service new connections.

i) Reason #1:

(1) Lots of requests means threads are lower overhead to create.

(2) Many requests will be for the same popular files means threads will share a cache.

(3) Lots of requests means switching between threads is less overhead

ii) Reason #2:

c. (2 points) Much of the software code used in the Therac-25 was taken from earlier models, which had functioned without the failures seen in the Therac-25. Why did this reused code suddenly fail in the Therac-25? Briefly, in one to two sentences, explain your answer.

The previous code had the same failings, but was masked by safety interlocks in hardware. Engineers reused the code on faith ("cargo cult programming") without understanding or retesting it.

d. (2 points) Suppose you are writing a multithreaded test in Pintos for project 1, and one of your test's threads dereferences a NULL pointer. Briefly, in one sentence, explain what happens to the system and why.

The entire kernel panics because we have not implemented user programs yet, so all threads are using their kernel stacks in shared kernel memory and there is no protection. A Segmentation Fault is user space and only implies a single (user) process crashing (since it is a protection fault), and is NOT equivalent to a kernel panic.

NAME: _____

2. (24 points total) Scheduling. Consider the following processes with their remaining instructions, arrival times, and priorities (A process with a higher priority number has priority over one with a lower priority number):

Process	Remaining instructions	Arrival time	Priority
A	3	2	4
B	4	1	1
C	1	3	3
D	2	1	2

There is a lock *L*:

- Process A acquires *L* in its first unit of time, and releases L in its last unit of time.
- Process D acquires *L* in its first unit of time, and releases L in its last unit of time.

- Processes busy wait when trying to acquire a lock.
- Priority donation is implemented ☺ .

Please note:

- The priority scheduler is *preemptive*.
- All processes arriving at the same time step arrive in alphabetical order.
- The quanta for RR is 1 unit of time and newly arrived processes are scheduled last for RR. When the RR quantum 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 and only 1 instruction runs at each time step.
- Total turnaround time is the time a process takes to complete after it arrives.

NAME: _____

Given the above information, fill in the following table:

Time	Round Robin	SRTF	Priority
1	*B*	*D*	*D*
2	*D*	*D*	*A*
3	*B*	*C*	*D*
4	*A*	*A*	*A*
5	*D*	*A*	*A*
6	*C*	*A*	*A*
7	*B*	*B*	*C*
8	*A*	*B*	*B*
9	*B*	*B*	*B*
10	*A*	*B*	*B*
11	*A*		*B*
Total Turnaround Time	*28*	*18*	*24*

Each column was graded separately with the same breakdown of 8 points. The sequence was 6 of the 8 points and the turnaround time was 2 of the 8 points.

NAME: _____

(14 points total) C Programming.

In the following program, we want to print out, "`Process 162 says 42`" once. Assume that the process ID of the child process is 162, the `fork()` is successful, and we want the behavior to be predictable. Do not add extra lines or try to compact your code onto the lines.

No hard-coding/assignment of values is allowed for your blanks inside of `main()`.

```
1 void helper(void) {
2   exit(42);
3 }

4 int main(void) {
5  int kirito = 0;
6  pid_t pid = fork();
7  if ( pid ) {                  // or waitpid()
8     wait( &kirito );
9  } else {
10    helper ();
11 }
12 printf("Process %d says %d\n", pid,
13   WEXITSTATUS(kirito));
14 }  // or just kirito
```

a) (8 points) Fill in the above blanks, to output "`Process 162 says 42`".

b) (4 points) If the `fork()` failed, what would be printed out, if anything?

If `fork()` fails, `pid` gets set to `-1`, which is evaluated as a `TRUE` value, so the first conditional block runs. `wait()` will return immediately because `fork()` failed and we will just print out "`Process -1 says 0`"

c) (2 points) In Linux, is the entire address space copied when `fork()` executes successfully? If not, what happens instead?

No. It is implemented with copy-on-write, which does not copy the entire address space. Copies are made when modifications are made.

NAME: _____

3. (21 points total) Pintos Priority Donation.

Suppose you have successfully completed project 1 task 2 (priority scheduling), with all tests passing (including recursive and multiple donations). You made the following changes in your thread and lock `struct`s, as indicated in your design doc:

```
/* thread.h */
struct thread
  {
    /* Preexisting members, some omitted for brevity. */
    ...
    int priority;                 /* Priority. */
    ...
    /* Your new members. */
    int base_priority;            /* Base priority without donations. */
    struct lock *wanted_lock;     /* Lock thread is waiting for, if any. */
    struct list acquired_locks;   /* List of acquired locks. */

    unsigned magic;               /* Detects stack overflow. */
  };

/* synch.h */
struct lock
  {
    /* Preexisting members. */
    struct thread *holder;      /* Thread holding lock. */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
    /* Your new members. */
    struct list_elem lock_elem; /* For storage in `acquired_locks` list. */
  };
```

In `lock_release()`, a thread currently handles multiple donations by looping over `acquired_locks` and looping over each lock's waiters to find the maximum priority among all waiters of all locks it holds. It then applies that donation if applicable, otherwise it reverts to the base priority.

One of your project partners thinks this is inefficient and suggests adding an "`int lock_priority`" member to `struct lock`, which would represent the highest priority among all of that lock's waiters so you don't have to loop over the waiters. Your friend also mentions updating `lock_priority` as needed upon any thread calling `lock_acquire()` on that lock, or calling `thread_set_priority()` on itself. However, your friend forgets that priority donations can change priorities and thus `lock_priority` is not updated upon each donation.

a. (3 points) Briefly, in two to three sentences, describe a test case that could identify this bug. *Hint: think about recursive priority donation*
Assuming A<B<C<D, have a thread (A) with 2 locks, one thread waiting on each (B & D). B should have earlier acquired a different lock and then a thread with priority between them (C) waits on B. Then A releases the lock for D and is incorrectly set to B's priority rather than C's.

b. (16 points) Your project partner still is not convinced, so you decide to write a new Pintos test. Complete the following Pintos test code so it passes without your partner's modification, but fails with it. You must use at least 1 `ASSERT()` statement in `test_priority_exam()`, and underline the `ASSERT()` that would fail with your friend's suggestion. You may assume for the purposes of this problem that `PRI_DEFAULT` is `0`. The following functions may be helpful:

```
/* thread.h */
typedef void thread_func(void *aux);
tid_t thread_create(const char *name,int priority,thread_func *,void *);
int thread_get_priority(void);
/* synch.h */
void lock_acquire(struct lock *);
void lock_release(struct lock *);
```

```
struct lock a,b,c,d,e; // May not need all locks.

/* May not need all thread functions */
static void thread_func_madoka(void *)
{
  lock_acquire(&c);
  lock_acquire(&a);
  lock_release(&a);
  lock_release(&c);
}
static void thread_func_sayaka(void *)
{
  lock_acquire(&c);
  lock_release(&c);



}
static void thread_func_homura(void *)
{
  lock_acquire(&b);
  lock_release(&b);



}
```

NAME: _____

```
static void thread_func_mami(void *)
{




}

void test_priority_exam(void *)
{
  ASSERT (thread_get_priority() == PRI_DEFAULT);
  /* Assume locks are already initialized */

  /* WRITE YOUR CODE BELOW */
  lock_acquire(&a);
  lock_acquire(&b);

  thread_create("1", PRI_DEFAULT + 1,
                thread_func_madoka, NULL);
  ASSERT (thread_get_priority () == PRI_DEFAULT+1);
  thread_create("2", PRI_DEFAULT + 2,
                thread_func_sayaka, NULL);
  ASSERT (thread_get_priority () == PRI_DEFAULT+2);
  thread_create("3", PRI_DEFAULT + 3,
                thread_func_homura, NULL);
  ASSERT (thread_get_priority () == PRI_DEFAULT+3);
  lock_release(&b);
  ASSERT (thread_get_priority () == PRI_DEFAULT+2);
                              /* !!! */
  lock_release(&a);
  ASSERT (thread_get_priority () == PRI_DEFAULT);
}
```

c. (2 points) Your project group rejects your friend's suggestion and you now pass all Task 2 tests, including your new one. However, what is one potential issue with your current design, specifically with keeping track of locks themselves in `struct thread` to implement recursive and multiple donations? Briefly, explain your answer in 1 sentence.

This design imposes the additional requirement that locks cannot be deallocated before release, which is counterintuitive but could result in a kernel panic if not properly handled.

NAME: _____

4. (23 points total) Office Hours Synchronization.

Suppose we want to use ***condition variables*** to control access to a CS162 office hours room for three types of people: students, TA's, and professors. A person can attempt to enter the room (or will wait outside until their condition is met), and after entering the room they can then exit the room. The following are each type's conditions:

- Suppose professors get easily distracted and so they need solitude, with no other students, TA's, or professors in the room, in order to enter the room.
- TA's don't care about students inside and will wait if there is a professor inside, but there can only be up to 7 TA's inside (any more would clearly be imposters from CS161 or CS186).
- Students don't care about other students or TA's in the room, but will wait if there is a professor inside. (An aside, maybe this is why more students don't come to the professors' office hours…)

a. (5 points) Specify the correctness constraints. Be succinct and explicit in your answer.

1 point for each correct constraint.
-Professor must wait if anyone else is in the room
- TA must wait if there are already 7 TA's in the room
-TA must wait if there is a professor in the room
-student must wait if there is a professor in the room
-only person can access the room synchronization at a time (one thread accesses the condition and state variables at a time).

b. (4 points) Complete the following incomplete `struct` definition for `room_lock`. Assume you have the following synchronization primitives:

```
typedef struct lock {…} lock // lock.acquire(), lock.release()
typedef struct cv {…} cv     // cv.wait(&lock), cv.signal(),
                             // cv.broadcast()
#define TA_LIMIT 7

typedef struct {
  lock lock;
  cv student_cv;
  int waitingStudents, activeStudents;
  cv ta_cv, prof_cv;
  int waitingTas, waitingProfs;
  int activeTas, activeProfs;

} room_lock;
```

NAME: _____

c. (14 points) Complete the following functions. We have partially filled in the student code to get you started. Please fill in the blanks of the student portions and fill in the entireties of the TA and professor portions. If there are multiple kinds of people waiting, prefer to wake up professors > TA's > students.

```
/* mode = 0 for student, 1 for TA, 2 for professor */
enter_room(room_lock* rlock, int mode) {
  rlock->lock.acquire();
  if (mode == 0) {
    while((rlock->activeProfs+rlock->waitingProfs) > 0){
      rlock->waitingStudents++;
      rlock->student_cv.wait(&rlock->lock);
      rlock->waitingStudents--;
    }
    rlock->activeStudents++;
  } else if (mode == 1) {
    while((rlock->activeProfs+rlock->waitingProfs) > 0
          || rlock->activeTas >= TA_LIMIT) {
      rlock->waitingTas++;
      rlock->ta_cv.wait(&rlock->lock);
      rlock->waitingTas--;
    }
    rlock->activeTas++;

  } else {
     while((rlock->activeProfs + rlock->activeTas +
          rlock->activeStudents) > 0){
      rlock->waitingProfs++;
      rlock->prof_cv.wait(&rlock->lock);
      rlock->waitingProfs--;
    }
    rlock->activeProfs++;

  }
  rlock->lock.release();
}
exit_room(room_lock* rlock, int mode) {
  rlock->lock.acquire();
  if (mode == 0) {
    rlock->activeStudents--;
    if ((rlock->activeStudents + rlock->activeTas) == 0
```

NAME: _____

```
          && rlock->waitingProfs)
        rlock->prof_cv.signal();
    } else if(mode == 1) {
      rlock->activeTas--;
      if ((rlock->activeStudents + rlock->activeTas) == 0
          && rlock->waitingProfs)
        rlock->prof_cv.signal();
      else if (rlock->activeTas < TA_LIMIT &&
               rlock->waitingTas)
        rlock->ta_cv.signal();


    } else {
      rlock->activeProfs--;
      if (rlock->waitingProfs)
        rlock->prof_cv.signal();
      else {
        if (rlock->waitingTas)
          rlock->ta_cv.broadcast();
        if (rlock->waitingStudents)
          rlock->student_cv.broadcast();
      }

    }
    rlock->lock.release();
}
```

NAME: _____

Congratulations on reaching the end of the exam!

There is no exam material on this page.

Remember that no matter how you do, someone cares for you.

We hope you enjoy operating systems so far.