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

Spring 2019 John Kubiatowicz

Midterm I

February 28th, 2019
CS162: Operating Systems and Systems Programming

Your Name:	
SID AND 162 Login (e.g. s042):	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 1 page of notes (both sides). You may use a calculator. You have 2 hours to complete as much of the exam as possible. Make sure to read all of the questions first, 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.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	**Possible**	**Score**
1	18	
2	21	
3	27	
4	21	
5	13	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [18 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation *GET NO CREDIT.*

Problem 1a[2pts]: If we disable interrupts in Pintos, the only way for another thread to run is for the current thread to die and exit, triggering the scheduler to choose another thread to run.

□ True □ False

Explain:

Problem 1b[2pts]: Threads within the same process can share data with one another by passing pointers to objects on their stacks.

□ True □ False

Explain:

Problem 1c[2pts]: After a forked child process calls exit(33), the child process will assign its exit code to the parent's wait status reference (int *status) via the code *status = exit_code (where exit_code is 33 in this case).

□ True □ False

Explain:

Problem 1d[2pts]: A system call (such `as read()`) is implemented in the C library as a function which does two separate things: (1) executes a "transition to kernel mode" instruction that changes the processor from user level to kernel level, followed by (2) a call to the correct routine in the kernel. This process is secure because the C library involves standard code that has been audited by the designers of the kernel.

□ True □ False

Explain:

Problem 1e[2pts]: Immediately after a process has been forked, the same variable in both the parent and the child will have the same virtual memory address but different physical memory addresses.

□ True □ False

Explain:

Problem 1f[2pts]:."Hyperthreading" refers to the situation in which a modern operating system allows thousands of threads to access the same address space.

□ True □ False

Explain:

Problem 1g[2pts]: Semaphores that operate across multiple processors (and at user level) can be implemented on machines in which the only atomic instruction operations on memory are loads, stores, and test-and-set.

□ True □ False

Explain:

Problem 1h[2pts]: Because the "monitor" pattern of synchronization involves sleeping inside a critical section, it requires special hardware support to avoid deadlock (i.e. permanent lack of forward progress).

□ True □ False

Explain:

Problem 1i[2pts]: Thread pools are a useful tool to help prevent an overly popular web site from crashing the hosting web servers.

□ True □ False

Explain:

Problem 2: Short Answer [21pts]

Problem 2a[3pts]: What is priority donation and why is it important?

Problem 2b[3pts]: Name three ways in which the processor can transition from user mode to kernel mode. Can the user execute arbitrary code after transitioning?

Problem 2c[3pts]: What happens when an interrupt occurs? What does the interrupt controller do?

Problem 2d[2pts]: Processes (or threads) can be in one of three states: **Running, Ready,** or **Waiting**. For each of the following examples, write down which state the process (or thread) is in:

a. Spin-waiting for a variable to become zero:

b. Having just completed an I/O, waiting to get scheduled again on the CPU:

c. Blocking on a condition variable waiting for some other thread to signal it:

d. Scanning through the buffer cache within the kernel in response to `read()`:

Problem 2e[3pts]: Disabling of interrupts can be very dangerous if done incorrectly. Is it possible to design an OS for modern computer hardware that never disables interrupts? Why or why not?

Problem 2f[3pts]: 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.

Problem 2g[2pts]: List two reasons why overuse of threads is bad (i.e. using too many threads for different tasks). Be explicit in your answers.

Problem 2h[2pts]: Why is it possible for a web browser (such as Firefox) to have 2 different tabs opened to the same website (at the same remote IP address and port) without mixing up content directed at each tab?

Problem 3: Synchronization Primitives [27pts]

Assume that you are programming a multiprocessor system using threads. In class, we talked about two different synchronization primitives: Semaphores and Monitors.

The interface for a Semaphore is similar to the following:

```
struct sem_t {
   // internal fields
};
void sem_init(sem_t *sem, unsigned int value) {
   // Initialize semaphore with initial value
   …
}
void sem_P(sem_t *sem) {
   // Perform P() operation on the semaphore
   …
}
void sem_V(sem_t *sem) {
   // Perform V() operation on the semaphore
}
```

As we mentioned in class, a Monitor consists of a Lock and one or more Condition Variables. The interfaces for these two types of objects are as follows:

```
struct lock_t {
   // Internal fields
};
void lock_init(lock_t *lock) {
   // Initialize new lock
   …
}

void acquire(lock_t *lock) {
   // acquire lock
   …
}
void release(lock_t *lock) {
   // release lock
   …
}
```

```
struct cond_t {
   // Internal fields
};
void cond_init(cond_t *cv, lock_t *lock) {
   // Initialize new condition variable
   // associated with lock.
   …
}
void cond_wait(cond_t *cv) {
   // block on condition variable
   …
}
void cond_signal(cond_t *cv) {
   // wake one sleeping thread (if any)
   …
}
void cond_broadcast(cond_t *cv) {
   // wake up all threads waiting on cv
   …
}
```

Monitors and Semaphores can be used for a variety of things. In fact, each can be implemented with the other. In this problem, we will show their equivalence.

Problem 3a[2pts]: What is the difference between Mesa and Hoare scheduling for monitors?

Problem 3b[6pts]: Show how to implement Semaphores using Monitors (i.e. the `lock_t` and `cond_t` operations). Make sure to define `sem_t` and implement all three methods, `sem_init()`, `sem_P()`, and `sem_V()`. *None of the methods should require more than five lines.* Assume that Monitors are Mesa scheduled.

```
struct sem_t {



};
void sem_init(sem_t *sem, unsigned int value) {



}
void sem_P(sem_t *sem) {



}
void sem_V(sem_t *sem) {



}
```

Problem 3c[4pts]: Show how to implement the Locks using Semaphores (i.e. the sem_t operations). Make sure to define `lock_t` and implement all three methods, `lock_init()`, `acquire()`, and `release()`. None of the methods should require more than five lines.

```
struct lock_t {



};
void lock_init(lock_t *lock) {



}
void acquire(lock_t *lock) {



}
void release(lock_t *lock) {



}
```

Problem 3d[2pts]: Explain the difference in behavior between `sem_V()` and `cond_signal()` when no threads are waiting in the corresponding semaphore or condition variable:

Problem 3e[10pts]: Show how to implement the Condition Variable class using Semaphores (and your Lock class from 2c). Assume that you are providing Mesa scheduling. You should not use lists or queues for this solution. Be very careful to consider the semantics of `cond_signal()` as discussed in (2d). *Hint: the Semaphore interface does not allow querying of the size of its waiting queue; you may need to track this yourself.* None of the methods should require more than five lines.

```
struct cond_t {



};
void cond_init(cond_t *cv, lock_t *lock) {



}
void cond_wait(cond_t *cv) {



}
void cond_signal(cond_t *cv) {



}
void cond_broadcast(cond_t *cv) {



}
```

Problem 3f[3pts]: Suppose that we implement locks using test-and-set and that we want to avoid long-term spin-waiting. Also assume that we use these locks at user level to synchronize across multiple processors. Explain why we still need to provide support for our lock implementation in the kernel and provide an interface for any system calls that might be needed.

Problem 4: Syscall Potpourri (21pts)

Problem 4a[4pts]: Assume that we have the following piece of code:

```
1.  int main() {
2.     printf("Starting main\n");
3.     int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
4.     dup2(file_fd, STDOUT_FILENO);
5.     pid_t child_pid = fork();
6.     if (child_pid != 0) {
7.        wait(NULL);
8.        printf("In parent\n");
9.     } else {
10.       printf("In child\n");
11.    }
12.    printf("Ending main: %d\n", child_pid);
13. }
```

What is the output of this program? You can assume that no syscalls fail and that *the child's PID is 1234.* Fill in the following table with your prediction of the output:

Standard out	test.txt

Problem 4b[4pts]: Next, assume that we have altered this code as follows:

```
1.  int main() {
2.     printf("Starting main\n");
3.     int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
4.     int new_fd = dup(STDOUT_FILENO);
5.     dup2(file_fd, STDOUT_FILENO);
6.     pid_t child_pid = fork();
7.     if (child_pid != 0) {
8.        wait(NULL);
9.        printf("In parent\n");
10.    } else {
11.       dup2(new_fd, STDOUT_FILENO);
12.       printf("In child\n");
13.    }
14.    printf("Ending main: %d\n", child_pid);
15. }
```

What is the output this time? Fill in the following table with your prediction of the output:

Standard out	test.txt

Problems 4c[5pts]: Consider the following fragment of a program (which is missing lines of code). Fill in the blanks in the code to ensure that the output of this program is always the following two lines in the following order (under all interleavings or schedules):

```
val = 0
val = 1
```

No more than one function call (or system call) per blank! Your solution is not allowed to make assignments to the "`val`" variable. Also, no use of `printf()` or other print statements!

```
void rectangle () {
     pid_t oval = ____________________________________;
     kill(oval, SIGCONT);
}

void circle(int i) {
     val = 1;
}
int val = 0;
int main() {
     pid_t pid = fork();

     ____________________________________________;
     if (pid == 0) {

          ____________________________________________;
     } else {

          ____________________________________________;
     }
     printf("val=%d\n",val);
}
```

Problem 4d[3pts]: If a child process sends a SIGKILL signal to its parent, what happens to the parent process and the child process? Can the parent prevent this from happening?

Problem 4e[2pts]: Consider the following scenario. A process forks a child process and the parent process waits on it. Then, the child exits normally and the parent is unblocked. Finally, the parent makes another wait call on the child process. What happens to this last wait call and to the parent process?

Problem 4f[3pts]: Why do we switch from the user's stack to a kernel stack when we enter the kernel (e.g. for a system call)? Why do we associate a *unique* kernel stack for *each* user thread?

[This page intentionally left blank!]

Problem 5: Alternate Readers-Writers Access to Database [13pts]

```
Reader() {
   //First check self into system
   lock.acquire();
   while (AW > 0) {
      WR++;
      okToRead.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // Perform actual read-only access
   AccessDatabase(ReadOnly);

   // Now, check out of system
   lock.acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okToWrite.signal();
   lock.release();
}
```

```
Writer() {
   // First check self into system
   lock.acquire();
   while ((AW + AR) > 0) {
      WW++;
      okToWrite.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // Perform actual read/write access
   AccessDatabase(ReadWrite);

   // Now, check out of system
   lock.acquire();
   AW--;
   if (WR > 0){
      okToRead.broadcast();
   } else if (WW > 0) {
      okToWrite.signal();
   }
   lock.release();
}
```

Problem 5a[5pts]: Above, we show a *modified* version of the Readers-Writers example given in class. It uses two condition variables, one for waiting readers and one for waiting writers. Suppose that *all* of the following requests arrive in very short order (while W_1 is still executing):

Incoming stream: $W_1\ R_1\ R_2\ R_3\ W_2\ W_3\ R_4\ R_5\ R_6\ W_4\ R_7\ W_5\ W_6\ R_8\ R_9\ W_7\ R_{10}$

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue). Explain how you got your answer.

Problem 5b[2pts]: How is the above version of the Readers-Writers access control different from the one give in class? Your answer here should be very short.

Problem 5c[2pts]: Explain why the above code ensures that if `AR > 0`, then `AW == 0`.

Problem 5d[2pts]: Explain why the above code makes sure that there can only be one writer at a time and that if `AW == 1`, then `AR==0`?

Problem 5e[2pts]: Finally, explain why the code for `Reader()` does not have to check to see if `WR > 0` (and as a result never needs to call `okToRead.broadcast()`)?

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]