

cs 111 Operating Systems Principles, Spring 2006

Midterm Solutions

You have 110 minutes to complete this midterm.

Write your name on this cover sheet AND at the bottom of each page of this booklet. Use the backs of the pages if you need to.

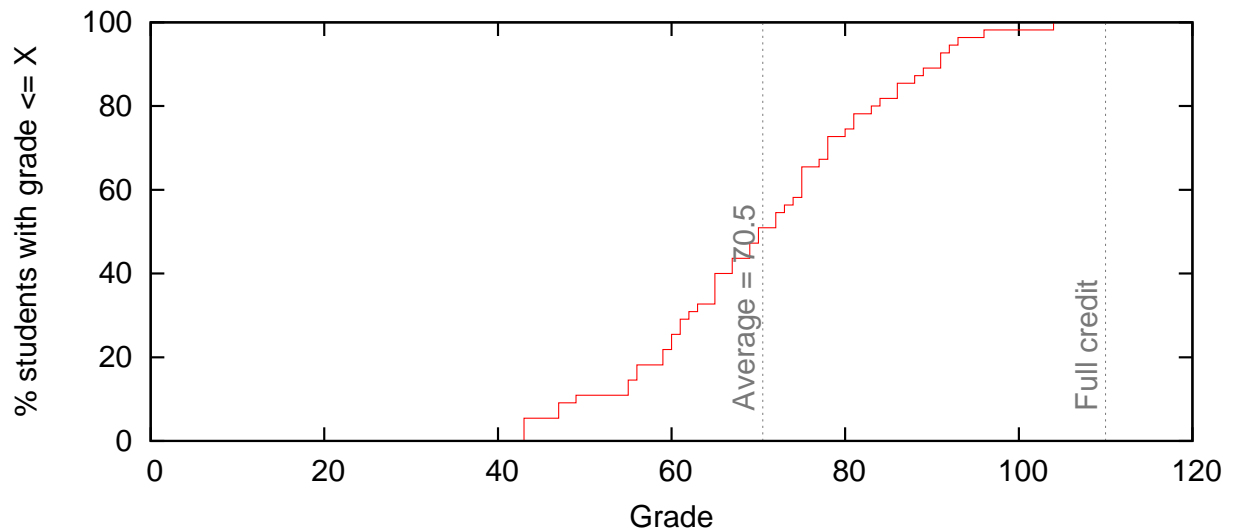
In order to receive credit you must answer the question as precisely as possible. Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Write down your reasoning; this will make it easier for us to give you partial credit.

OPEN BOOK, OPEN NOTES, CLOSED COMPUTER

I (xx/15)	II (xx/25)	III (xx/25)	IV (xx/15)	V (xx/30)	Total (xx/110)

Grade statistics

Average	70.5
Median	70
High score	104
Low score	43



I Abstract machines

1 [10 points]. Which of the following abstract machine properties are likely to be the same across *all* operating systems **for a given type of processor**, and which might differ from operating system to operating system? Circle one per row. If you're not sure or think there might be some ambiguity, pick one and justify your answer.

A. Register names	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
B. Scheduling priority	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
C. Instruction set	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
D. Maximum number of open files	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
E. Initial stack pointer	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
F. System call set	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
G. Quantum	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
H. Pointer size (e.g., 32 bits)	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
I. Protected instructions (e.g., “HSC”, Halt and Spontaneously Combust)	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different
J. Signal names	<input type="checkbox"/> Always same	<input type="checkbox"/> Sometimes different

Note on H.: Some processors do allow the operating system to choose pointer size—AMD's x86-64 architecture, for example, supports either 32-bit or 64-bit pointers. You had to mention this explicitly to get credit.

2 [5 points]. WeensyOS 1's extra credit problem #7 asked you to “Introduce a `sys_kill(threadid)` system call, which forces thread `threadid` to exit;” the system call should return 0 on success and `-1` on error. Recall that this was *after* you implemented the wait queue, where a joining thread would block on a waiting thread. Here's an attempt at the system call implementation:

```
case TRAP_SYS_KILL: {
    threadid_t t = current->t_registers.reg_eax; // get thread ID to kill
    threads[t].t_state = P_ZOMBIE; // kill it
    threads[t].t_exit_status = 0; // arbitrary exit status
    current->t_registers.reg_eax = 0; // return 0, indicating success
}
```

```
    run(current);                // return
}
```

There are at least four problems with this implementation; briefly describe at least two of them.

1. *t* is not bounds-checked; if the user passes a value < 0 or $\geq NTHREADS$, memory corruption will ensue.
2. The current state of *t* isn't checked. If the thread is empty, the system call should fail—or at the least its state shouldn't be turned to *P_ZOMBIE*.
3. If the thread kills itself, it will run nevertheless (since the `run(current)` will run the thread regardless of its state).
4. This code does not wake up the thread's wait queue.

II Interface design

Modern operating systems have added functionality to the abstract machine interface to support new uses. This question concerns one such example, *directory notification*. This supports the Open and Save dialogs ubiquitous in today's applications. Here's an example:

1. The user chooses Open in a text editor, which displays an Open dialog. The dialog shows the contents of the user's Documents directory.
2. Meanwhile, in the background, a Web browser is downloading a file `resume.txt` into that same directory.
3. As soon as the download completes and the file is created, the Open dialog changes to show `resume.txt` as well.

There are many ways to implement directory notification, each with different properties. In this section you'll compare and contrast several possible implementation strategies, namely these:

- A. Periodic refresh.** Every *T* seconds the Open dialog rereads the directory and displays what it finds.
- B. File system trace.** The operating system provides a special device file, `/dev/fstrace`. All file system changes are written to the file in a special format. Applications can open that file and read the file system changes from it. Reading from the file will block until there is a change.
- C. Directory versions.** The application can obtain any directory's current *version*. The operating system updates a directory's version when any changes to that directory are made. The application reads the directory's version, compares to its cached version, and rereads the directory when the version changes.

- D. Directory trace.** This is like **B**, the file system trace, except that a special function is used to provide a trace file for a named directory. Only changes that apply to that directory are written to the file.
- E. Provisional files.** A new function “`int wait_for_file(const char *filename)`” blocks the current process until a file named `filename` is created. This function will notify the Open dialog when `resume.txt` is created.
- F. Directory signals.** A new function “`int dirsignal(const char *dirname, int signo)`” will cause the OS to send the current process the `signo` signal on any change to the named directory. The Open dialog will reread the directory in response to the signal handler.

3 [4 points]. One of these implementation strategies just plain doesn't work: it cannot detect some file system changes. Say which one, and give an example of a change that it will not detect.

E (provisional files) won't work. It's a pretty bad interface: the Open dialog application would have to call `wait_for_file` for every possible file name. Worse, provisional files will not detect file deletions.

4 [6 points]. For each implementation strategy, say whether it is more like polling, blocking, or interrupt-driven.

- | | | | |
|------------------------------|-----------------|-----------------------------|-------------------------|
| A. Periodic refresh | <i>Polling</i> | D. Directory trace | <i>Blocking</i> |
| B. File system trace | <i>Blocking</i> | E. Provisional files | <i>Blocking</i> |
| C. Directory versions | <i>Polling</i> | F. Directory signals | <i>Interrupt-driven</i> |

5 [5 points]. Which of these implementation strategies requires a new system call (relative to the system calls covered in class, and normal system calls to open and read directory entries)? List all that apply.

C, D, E, F—although if you claimed that this could all be done with `ioctl` we gave you credit.

6 [5 points]. Which of these implementation strategies would require a new thread to specifically handle directory changes?

B, D, E: the calls that block.

7 [5 points]. Give an example scenario when **A.** (Periodic refresh) will perform better than **B.** (File system trace). Elements of the scenario might include directory size, refresh time T , the rate of changes to the directory, the rate of changes to the file system, and so forth.

If the relevant directory is small (not many entries), and the rate of changes to the directory is low, but the rate of changes to the whole file system is large, then periodic refresh will outperform file system trace—in that periodic refresh will have less overhead, so more time is spent doing useful work.

III Processes and threads

A team of archeologists discovers the following C source code inscribed on a cuneiform tablet near present-day Indianapolis. Unfortunately, a part of the source code has been garbled—hopefully you can tell which part.

```
char *buf = "X";

void child(void *) {
    buf = "A";
    write(1, buf, 1);
    exit(1);
}

int main(int c, char **v) {
    AMSDIOoie2iqidjU*(&#!#(*$)(R*AS&DHZjxnzkna1idu982u43.com
    buf = "B";
    write(1, buf, 1);
}
```

You know the garbled portion *either* creates a child process that runs the `child` function or creates a new thread that runs the `child` function, but you don't know which. (You know the garbled portion does nothing other than create a new process or thread.)

On the other side of the tablet are several inscriptions describing different possible outputs when this program is run. Assume that each inscription describes *all possible output* from the program (if no system calls return an error). Use your forensic skills to determine which of the following possibilities apply.

- A. If the inscription is correct, then the garbled portion creates a child process.
- B. If the inscription is correct, then the garbled portion creates a new thread, and the operating system uses preemptive threads.

- C. If the inscription is correct, then the garbled portion creates a new thread, and the operating system uses *cooperative* threads.
- D. The inscription is a trick: neither threads nor processes will have the given inscription. (For example, the inscription contains an output impossible for a new process, but is missing an output possible for a new thread.)

One way to think about this problem is to write out exactly what output each of A, B, and C might generate in the absence of system call errors.

- A. Processes might generate AB or BA only. Processes have isolated address spaces, so setting the value of *buf* in one process will not affect the other. Also, each process exits independently.
- B. Preemptive threads might generate any of AA, AB, BA, BB, A, and B. AA and BB can happen if the operating system preempts one thread after it sets *buf*, but before it calls *write*, and runs the other thread. A and B can happen if one thread exits the containing process before the other gets a chance to write: remember that when a process exits, all its threads are killed. (This is how *exit* and *pthread_exit* differ.) The *child* function exits explicitly with a call to *exit*; the *main* function exits implicitly by running off the end of the function.
- C. In cooperative threads, the thread implementation can switch between threads at system calls, but is not required to do so. This prevents AA and BB from happening, but all the other output from preemptive threads can still happen: AB, BA, A, and B.

That leaves us with the following answers.

8 [3 points]. AB or BA

- A. New process C. New thread, cooperative
- B. New thread, preemptive D. None of the above

9 [3 points]. AA or AB or BA or BB

- A. New process C. New thread, cooperative
- B. New thread, preemptive D. None of the above

10 [3 points]. A or AA or AB or BA or BB

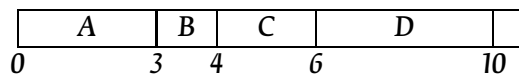
- A. New process C. New thread, cooperative
- B. New thread, preemptive D. None of the above

IV Scheduling

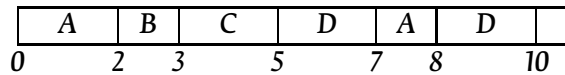
15 [8 points]. Say that the following jobs arrive at a CPU scheduler all at time 0 in alphabetical order. Draw Gantt charts for the following scheduling algorithms given this set of jobs. In addition, calculate each schedule's average turnaround time and average wait time, neglecting the cost of context switches C.

Job		A	B	C	D
Service time τ		3	1	2	4

A. First Come First Served



B. Preemptive Round Robin with Quantum $Q = 2$



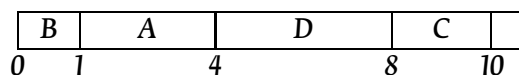
16 [4 points]. Given Problem 15's jobs and a Preemptive Round Robin scheduler with quantum $Q = 2$, what value of the context switch time C provides a utilization of $\rho = 0.8$? Recall that the utilization is the fraction of time the CPU spends doing useful (i.e., application) work.

The utilization ρ is a system-wide metric that equals the fraction of time the CPU spends doing any application's job. Say we begin measuring utilization when A starts running, and stop as soon as D completes, not counting either time 0 or time 10 as context switches. Then the schedule in Problem 15B contains 10 units of work, and 5 context switches. So:

$$\begin{aligned} \rho = 0.8 &= 10 / (10 + 5C) \\ 8 + 4C &= 10 \\ C &= 1/2 \end{aligned}$$

17 [3 points]. Jobs like those of Problem 15 are run under a real-time scheduler with strict deadlines. One deadline has been left blank. Write in the earliest deadline that allows the schedule to be met, and write out the Gantt chart for the resulting schedule using Earliest Deadline First.

Job		A	B	C	D
Service time τ		3	1	2	4
Deadline		4	2	15	10



V Semaphores

Nick Lachey has some concerns about our semaphore-based implementation of the *event* synchronization object. Recall that an event object supports two operations, `wait` and `signal`. `wait` always blocks the calling thread; `signal` wakes up one waiting thread, if there are any. If `signal` is called on an event that has no waiting threads, then the event's state does not change. The semaphore implementation is (in pseudocode):

```
typedef struct {
    semaphore_t m = 1;
    semaphore_t e = 0;
    int wc = 0;
} event_t;

wait(event_t *e) {
    P(e->m);
    ++e->wc;
    V(e->m);
    P(e->e);
}

signal(event_t *e) {
    P(e->m);
    if (e->wc > 0) {
        --e->wc;
        V(e->e);
    }
    V(e->m);
}
```

He particularly doesn't like that `signal` calls `V(e->e)` while the mutex is locked, but `wait` calls `P(e->e)` *without* locking the mutex. He's been a little distracted in class recently, but he seems to remember something about "lock ordering": shouldn't `wait` lock the mutex before accessing `e->e`? Or maybe `signal` should access `e->e` *without* locking the mutex.

Nick therefore proposes two different fixes to this code. Unfortunately, they're both wrong. Your job is to explain how.

Here's Nick's first fix; he moves `wait`'s `P(e->e)` inside the section where the mutex is held.

```

wait(event_t *e) {
W1   P(e->m);
W2   ++e->wc;
W3   P(e->e);
W4   V(e->m);
}

signal(event_t *e) {
S1   P(e->m);
S2   if (e->wc > 0) {
S3       --e->wc;
S4       V(e->e);
S5   }
S6   V(e->m);
}
```

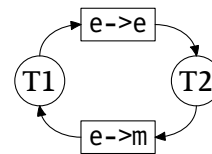
18 [3 points]. True or false: This code leads to deadlock.

True.

19 [7 points]. Give a sequence of steps that demonstrate the race condition or deadlock by filling out the following table. Mark the point where deadlock occurs or where

the race condition is triggered. If there is a deadlock, draw a resource model graph demonstrating the deadlock; if the race condition is not a deadlock, describe it briefly. You need at least two threads: T1, which is executing wait(e), and T2, which is executing signal(e). Do not jump steps.

Thread	Line	e->m	e->e	e->wc	
Initial state		1	0	0	
T1	W1	0	0	0	
T1	W2	0	0	1	
T1	W3	0	0	1	Block on e->e
T2	S1	0	0	1	Block on e->m



Here’s Nick’s second fix; he moves signal’s V(e->e) outside the section where the mutex is held. This requires a bit more rearrangement—but Nick is not so good at rearrangement.

```

wait(event_t *e) {
W1   P(e->m);
W2   ++e->wc;
W3   V(e->m);
W4   P(e->e);
}

signal(event_t *e) {
S1   if (e->wc > 0) {
S2   P(e->m);
S3   --e->wc;
S4   V(e->m);
S5   V(e->e);
S6   }
}
    
```

20 [3 points]. True or false: This code leads to deadlock.

False.

21 [7 points]. Give a sequence of steps that demonstrate the race condition or deadlock by filling out the following table. Mark the point where deadlock occurs or where the race condition is triggered. If there is a deadlock, draw a resource model graph demonstrating the deadlock; if the race condition is not a deadlock, describe it briefly. You need at least three threads: T1, which is executing wait(e), and T2 and T3, which are executing signal(e). It is OK to jump steps as indicated.

Thread	Line(s)	e->m	e->e	e->wc	
Initial state		1	0	0	
T1	W1-W4	1	0	1	Block on e->e
T2	S1	1	0	1	
T3	S1	1	0	1	Race! Both will signal
T2	S2-S6	1	1	0	
T3	S2-S6	1	2	-1	

The race condition is that the signal function checks e->wc outside of the induced critical section. Thus, thread T3 “wakes up” the signal again rather than leaving it unchanged. This puts the signal in an inconsistent state: e->wc < 0. The next thread to call wait will immediately return, even though it should block.

The following code attempts to implement an N -thread *barrier* synchronization object with semaphores. Recall that an N -thread barrier has one method, `barrier`; threads block when they call `barrier` until all N threads are blocked on the barrier, at which point all N threads unblock.

```
typedef struct {
    semaphore_t mutex = 1;
    semaphore_t barrier = 0;
    int nwait = 0;
} barrier_t;

barrier(barrier_t *b) {
    B1    P(b->mutex);
    B2    if (++b->nwait == N) {
    B3        while (b->nwait-- > 0)
    B4            V(b->barrier);
    B5    }
    B6    V(b->mutex);
    B7    P(b->barrier);
}
```

But this barrier has a problem: there is a race condition. Here's a demonstration. Say that $N = 2$, and both threads are executing the following code:

```
L1 printf("%d: before\n", sys_getthreadid());
L2 barrier(b);
L3 printf("%d: between\n", sys_getthreadid());
L4 barrier(b);
L5 printf("%d: after\n", sys_getthreadid());
```

This code should output something like on the left. Unfortunately, sometimes it will output something like on the right.

1: before	1: before
2: before	2: before
2: between	2: between
1: between	2: after
1: after	
2: after	

22 [10 points]. Fix the race condition: write a correct version of `barrier`. You will need at least one more semaphore. **This problem is hard.** Or, for partial credit, give a sequence of steps that demonstrate the race condition.

See the next page.

The problem here is that a thread can cycle through the barrier a second time before all threads blocked on the first barrier have woken up. Here's an example sequence of steps demonstrating the race:

Thread	Line(s)	b->mutex	b->barrier	b->nwait	
	Initial state	1	0	0	
T1	L1	1	0	0	"1: before"
T1	B1	0	0	0	
T1	B2-B5	0	0	1	
T1	B6	1	0	1	Release b->mutex
Context switch before B7					
T2	L1	1	0	1	"2: before"
T2	B1	0	0	1	
T2	B2	0	0	2	Wake up barrier
T2	B3-B4	0	1	1	
T2	B3-B4	0	2	0	
T2	B5-B7	1	1	0	
T2	L3	1	1	0	"2: between"
T2	B1	0	1	0	T2 calls barrier again
T2	B2-B5	0	1	1	
T2	B6-B7	1	0	1	T2's P(b->barrier) succeeds! (It steals T1's wakeup opportunity)
T2	L5	1	0	1	"2: after"
T1	B7	1	0	1	Block on b->barrier

The classical solution for this problem introduces two barrier semaphores and alternates between them. That way, we will never have a situation like the above, where T2's second barrier call steals T1's wakeup opportunity, because T2's second barrier call will wait on a different semaphore.

```

typedef struct {
    semaphore_t mutex = 1;
    semaphore_t barrier[2]
        = { 0, 0 };
    int sense = 0;
    int nwait = 0;
} barrier_t;

barrier(barrier_t *b) {
    B1 int sense;
    B2 P(b->mutex);
    B3 sense = b->sense;
    B4 if (++b->nwait == N) {
    B5     while (b->nwait-- > 0)
    B6         V(b->barrier[sense]);
    B7     b->sense = 1 - b->sense;
    B8     }
    B9     V(b->mutex);
    B10    P(b->barrier[sense]);
}

```

We also saw a different solution, which ensures that all processes leave the first barrier before any of them can enter the second. This constrains execution order more than the two-barrier solution, since processes are released one at a time, rather than in groups, but it does work!

```
typedef struct {
    semaphore_t mutex = 1;
    semaphore_t barrier = 0;
    semaphore_t outmutex = 0;
    int nwait = 0;
} barrier_t;

barrier(barrier_t *b) {
B1     int sense;
B2     P(b->mutex);
B3     if (++b->nwait == N) {
B4         while (b->nwait-- > 1) {
B5             V(b->barrier);
B6             P(b->outmutex);
B7         }
B8         b->nwait--; // this thread
B9         V(b->mutex);
B10    } else {
B11        V(b->mutex);
B12        P(b->barrier);
B13        V(b->outmutex);
B14    }
}
```