

CS 474

Midterm Solutions

October 5, 2005

The following exam is open book and open notes. You may feel free to use whatever additional reference material you wish, but **no electronic aids** are allowed. Please note the following instructions. There will be a ten point deduction for failure to comply with them:

- start each problem on a new sheet of paper
- write the number from your “take a number” slip on everything you turn in.
- show your work whenever appropriate. There can be no partial credit unless I see how you arrived at your answers.
- be succinct. You may lose points for facts that, while true, are not relevant to the question at hand

You have until 12:20 to finish the exam.

1. (10 points) Procedure calls and system calls are conceptually very similar (to the point that it isn't uncommon for people to not quite be sure which calls such as `read()` and `fread()` are. However, there are some very distinct differences in their implementations.

(a) On a procedure call, you push the parameters on the stack. For a system call, you put them in registers. Why?

It switches from the user stack to the system stack. Since the user stack isn't available (at least, not with a reasonable amount of effort), you have to use the registers.

(b) On a procedure call, you jump to the address of the procedure while pushing a return address on the stack. With a system call, you cause an interrupt to occur, which lands you in an interrupt handler (and you have no opportunity to set the address you jump to in the interrupt handler). Why?

We need to switch from user mode to kernel mode. If users were allowed to decide where in the OS to execute, they would be able to do things like jump into a system call after the parameters were checked, and bypass security.

(c) How does the OS determine what code to execute once it lands in the interrupt handler?

It decodes one of its parameters.

Points	Error
3	3 points per part

2. (25 points) The following tables give the total numbers of resources of each type R_0 through R_2 that are available, the current allocations to processes P_0 through P_3 , and the currently unsatisfied requests.

Total Resources

R_0	R_1	R_2
3	2	3

Allocated Resources

	R_0	R_1	R_2
P_0	1	0	0
P_1	1	0	1
P_2	0	2	1
P_3	0	0	1

Unsatisfied Requests

	R_0	R_1	R_2
P_0	0	0	1
P_1	0	1	0
P_2	0	0	1
P_3	0	0	0

(a) Are there currently any deadlocks? Which processes are able to proceed, and which are not?

P_3 is able to proceed.

P_0 , P_1 , and P_2 are all waiting for resources that aren't available at the moment.

We don't have a deadlock, since if P_3 continues it will release its R_2 , after which P_2 can have its request satisfied. When it finishes P_1 can proceed, and after that P_0 can finish.

(b) Suppose P_3 were to request one unit of R_1 . How would this change the situation?

Now we've got a deadlock. None of the processes can proceed.

3. (20 points) Attached to this exam is an excerpt from the Unix man page for `clone()`.

(a) Would you want to set the `CLONE_PARENT` flag if you were performing a `fork()`? How about if you were creating a thread?

When creating a new process with `fork()`, there should be a parent-child relationship between them. Two threads, however, should be regarded as siblings. Consequently the answer is "no" for `fork()`, but "yes" for threads.

(b) Would you want to set the `CLONE_VM` flag if you were performing a `fork()`? How about if you were creating a thread?

Threads share memory, but processes don't. So the answer is "no" for `fork()` but "yes" for threads.

Points	Error
-5	Per answer (so 10 per part)

4. (25 points) In a threaded application, two threads are obtaining input and placing it in a buffer. They each use the same code:

```
while (1) {
    awry[i++] = newin();
    if (i > 100) i = 0;
}
```

The `newin()` function correctly obtains one unit of input. `awry` and `i` are both shared between the threads. In this machine, the `i++` operation is not atomic.

(a) Show that it is possible for inputs to be lost using this code (assume some other code someplace is correctly removing the inputs from the array – the inputs aren't lost due to `i` wrapping around to 0).

Note that there are many different traces that will show the same behavior; this is just one of them.

- i. P_0 reads i .
- ii. P_1 reads i (and obtains the same value P_0 did).
- iii. P_0 increments its copy of i (and since `i++` is a post-increment operator, it will "remember" i 's old value for use in writing to `awry`).
- iv. P_1 increments its copy of i (likewise, P_1 will "remember" i 's old value. This is the same value P_0 is remembering).
- v. P_0 writes its copy of i back to memory.
- vi. P_1 writes its copy of i back to memory.
- vii. P_0 writes its unit of input to `awry`, at the location specified by i 's old value.
- viii. P_1 does the same thing, and writes to the same location.

At this point, the unit of input read by P_0 has been lost.

(b) Modify the code so that it will correctly put each new input in a new location in the array. You may add new local or global variables, you may restructure the code, and you may add semaphores. A correct solution will only lock the operations that actually need it (so don't just insert a `down()` at the beginning of the loop and an `up()` at the end).

The main thing here is that the increment (and remember the old value) has to be made atomic. Here's code that will do it (assuming `tmp` is local to this thread, while `sem` is a global semaphore).

```
while (1) {
    int tmp;
    down(sem);
    tmp = i++;
    if (i > 100) i = 0;
    up(sem);
    awry[n] = newin();
}
```