UCLA Computer Science 111 section 2 (Spring 2007)
Midterm
111 minutes total, open book, open notes

Name: Michael Wilson Student ID: 003508374

1	2	3	4	5	6	7	8	9	10	total
5	3	7	11	13	21	6-	10	12	7	95

1 (5 minutes). Is it possible to enforce hard modularity without using any special machine instructions or other special hardware features? Briefly explain.

5

Yes, if you use pure virtualization so that every command executed is run through an interpeter. But, this is very slow.

2 (5 minutes). Suppose you run the following shell command in an empty directory. What will happen? Describe the sequence of events.

See attached for the rest.

```
cat foo > foo < foo # phew!
```

waitpid #

3 (10 minutes). WeensyOS 1, like many operating systems, implements getpid as a system call. Would it be wise to implement it as an ordinary user-space function whose implementation simply accesses memory that is already readable by the current process? Explain the pros and cons of this alternate implementation.

4a (8 minutes). Suppose your solution to Lab 1b implements the command (CMD1 | CMD2) by creating two processes connected by a pipe, but you forgot to have CMD2 process close the write end of the pipe. What can go wrong as a result, from the user's viewpoint, and why?

4b (7 minutes). Likewise, but suppose you forgot to have the CMD1 process close the read end of its pipe.

5 (15 minutes). A child thread normally has a separate stack from its parent. But suppose we want to create a super-fast operating system, one that optimizes thread creation by having the child share the parent's stack. We intend to inform application programmers that they need to program their parent and child threads carefully to avoid race conditions, e.g., by using a mutex to avoid simultaneous access to the same part of the stack, or by having the child and parent never access the same part of the shared stack.

Is it possible to use such a programming model? If so, what would it look like? If not, why not?

6. Consider the following source code, adapted from the implementation of sys_wait in mpos-kern.c's 'interrupt' function, in WeensyOS 1.

```
 1  void
 2  interrupt(registers_t *reg)
 3  {
 4    current->p_registers = *reg;
 5    switch (reg->reg_intno) {
 6      ...
 7    case INT_SYS_WAIT: {
 8      pid_t p = current->p_registers.reg_eax;
 9      if (p <= 0
10          || p >= NPROCS
11          || p == current->p_pid
12          || miniproc[p].p_state == P_EMPTY
13          || 0)
14        current->p_registers.reg_eax = -1;
15      else if (0
16               || miniproc[p].p_state == P_ZOMBIE
17               || 0)
18        current->p_registers.reg_eax = miniproc[p].p_exit_status;
19      else
20        current->p_registers.reg_eax = WAIT_TRYAGAIN;
21      schedule();
22    }
23
24    default:
25      for (;;)
26        continue;
27    }
28  }
```

For each of the following lines in the source code, give an example of exactly what could go wrong, from the application's viewpoint, if you omitted that particular line:

6a (3 minutes). Line 4
6b (3 minutes). Line 10
6c (3 minutes). Line 11
6d (3 minutes). Line 12
6e (3 minutes). Line 16
6f (3 minutes). Lines 19 and 20 (omitting both lines, simultaneously)
6g (3 minutes). Line 21 (replacing it with "break;")

7 (10 minutes). Suppose you have a simple (non-blocking) mutex around a shared pipe object, and implement a read/write lock in the usual way. Suppose each reader and writer locks the object only for a short period of time. Can a would-be reader starve? If so, show how. If not, explain why not. If your answer depends on the implementation, explain your assumptions and why they matter.

Similarly, can a would-be writer starve?

8 (10 minutes). Suppose we modified Unix so that, instead of having fork() and execvp(file, args) system calls, it has a single system call forkexecvp(file, args) that does the work of both fork and execvp. Give an example of a Unix application that would be much harder to write in this modified Unix, and explain why it'd be harder.

9. Consider the following implementation of a shared pipe object using a blocking mutex and condition variables:

```
struct pipe {
  bmutex_t b;
  char buf[N];
  size_t r, w;
  condvar_t nonfull, nonempty;
};

void writec(struct pipe *p, char c) {
  acquire(&p->b);
  while (p->w - p->r == N)
    wait(&p->nonfull, &p->b);
  p->buf[p->w++ % N] = c;
  notify(&p->nonempty);
  release(&p->b);
}
```

9a (5 minutes). Give the implementation for 'readc' that corresponds to this implementation of 'writec'.

9b (8 minutes). Wouldn't it be better to use finer-grained locking, and have one blocking mutex for readers and another for writers, instead of having a single blocking mutex for both readers and writers? If so, rewrite the code accordingly and explain why it works; if not, explain why not by giving a scenario where the rewritten code fails.

firstcome firstserve

10 (7 minutes). Does it make sense to use FCFS in a preemptive scheduler? If so, give an example; if not, explain why it doesn't make sense.

2) ~~A~~ pipe is setup that takes foo as the input, and cat reads from that input and combines foo with itself, then writes that combination out to foo. But, when it open foo to write, it erases foo, so it just writes nothing back out to itself. Actually, two copies of nothing.

3

eg. foo = $\boxed{\text{hi}}$

then the new foo = $\boxed{}$

3) Not wise. It would save some overhead in switching into the kernel space for a system call, but then every process would need to learn its own pid and store it somehow. It saves time if you call it often, but complicates matters.

1

4)a) When CMD1 finishes writing to the pipe, it will close/disconnect from the pipe, but CMD2 will still have the same end open. Since the pipe never sees the EOF from CMD2 (since CMD2 is not writing to the pipe) the pipe remains open and CMD2 never finishes reading from the pipe. Computer hangs. Deadlock.

8

b) There won't be any writers, but there will still be 1 reader, so the pipe won't close.

3

5) ~~If~~ the shared stack is separated into two stacks so that the parent and the child never accessed the same part of the stack, this would work. But, the parent would need to start with a gigantic stack, and each child would need an equally large stack so that its children could have their own stack.

how? do you do this?

No, this is not a good idea because you don't know how many children there might be, so you don't know how large each stack should be, and you could smash any stack by forking enough times. Bad idea.

~~10~~ 13

6] a) Something in the registers could change while examining or testing the registers, besides the fact that the current process registers wouldn't have the right data.

b) There may have been a mistake whereby the process created was not supposed to be, or an erroneous value was loaded into eax, If not caught, dereference miniproc[p] is not a valid array value.

c) We might be waiting on ourselves, which could be a very long wait. That's an error.

d) We're waiting on a process that isn't running, so we can wait and someone might take that pid, but if they don't then we'll be waiting a long time. Lock.

e) If we didn't check to see if it's a zombie, then we'll just call wait forever, and never get the exit status.

f) We would return whichever value was in there. If it was the pid, then this would be interpreted as the exit_status and we would believe the child exited when it didn't.

g) We would fall out the bottom of interrupt and nothing would ever be scheduled again.

7) No, the reader would not starve, since we are guaranteed that each reader & writer lock for a short time. [We allow many readers, so most of the time, the readers will be able to lock to read.] Not necessarily true

6

In contrast, allowing many readers means that a writer might starve. If many readers read, even with high turnover the readers count might not reach 0 to let the writer take the lock.

8) Any application that uses a pipe would be hard to write, since bothen ends of the pipe need to be setup before the child executes. The shell is an application that does just that, and would be much more difficult to write with just a forkexecvp (), since the child & parent only want one end of the pipe apiece.

10

9) a)

```
char reade(struct pipe* p) {
    acquire (&p->b);
    while (p->w - p->r == 0)
        wait (&p->nonempty, &p->b);
    char c = p->buf[p->r++ % N];
    notify (&p->nonfull);
    release (&p->b);
    return c;
}
```

5

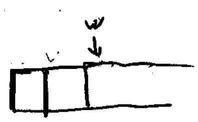
b) A finergrain lock would result in race conditions of the type that we had hoped to avoid by using mutexes in the first place.

7

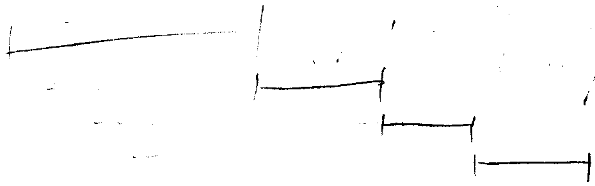
If the (?) finds that the pipe has stuff in it just after the writer increments its pointer but before the character is finished being written, then the reader pulls garbage from the pipe.

Chron. steps:

1) writer puts stuff, signals read
2) read wakes up with read_lock, starts doing stuff.
3) writer writes another character, but size_t w increments before write finishes.
4) reader finished reading the first one and moves on to read the second since the pipe is non empty
5) writer is not done writing, so reader gets garbage.



10)



~~No, it doesn't make sense if the order of arrival is the only priority metric used in scheduling. If the process is already running, then it arrived before any other process that might want to run, so that no preemption would actually occur.~~

You could have a scheduler that uses FCFS as a default priority, but allows certain processes a special flag to indicate that they have super, ultra-high priority & they need to run right away. Perhaps you could let system calls or interrupts have this highest priority that will overpower any priority based on arrival times.

7

SO: Use FCFS to break priority ties. ✓