CS 3204 Sample Midterm

Answers are shown in this style. This exam was given Fall 2006.

1. Semaphores, Monitors, and Barriers (24 pts)

a) (12 pts) Suppose a system provides monitors, but no semaphores. Assume that the monitors are Mesa-style. Assume a C-style syntax for the monitors, e.g. whereby mutexes and condition variables are separately defined. Implement a semaphore using monitors. You only need to show the sema_down/up functions, and the necessary initialization code.

The following code shows how to implement semaphores using condition variables:

```
// initialization
struct semaphore {
      int value;
      struct lock lock;
      struct condvar cond;
};

sema_init(struct semaphore *s, int initialvalue) {
      s->value = initialvalue;
      lock_init(&s->lock);
      cond_init(&s->cond);
}

// code for signal (or sema_up)
void sema_up(struct semaphore *s) {
      lock_acquire(&s->lock);
      if (++s->value == 1)
            cond_signal(&s->cond, &s->lock);
      lock_release(&s->lock);
}

// code for wait (or sema_down)
void sema_down(struct semaphore *s) {
      lock_acquire(&s->lock);
      while (s->value == 0)
            cond_wait(&s->cond, &s->lock);
      s->value--;
      lock_release(&s->lock);
}
```

b) (12 pts) *Barrier Synchronization.* In class, we had discussed how 2 semaphores can be used to implement a rendezvous between 2 threads: no thread will progress past the rendezvous point unless the other thread has also reached that point. A barrier is a synchronization primitive whereby N threads can rendezvous with each other.
Provide pseudo-code that implements an n-way barrier using semaphores only. You may assume that the number N of threads is a constant.

The solution is a straightforward generalization of the code discussed in class - use one semaphore for each thread – on a rendezvous, signal the semaphores of all other threads and wait for N-1 signals from them. The opposite approach works as well (upping your own semaphore N-1 times and downing each of the others.) This is not a very efficient solution, but it works with only semaphores.

```
semaphore s[N](0);      // all initialized to 0
barrier() {
      for (i = 0..N-1)
            if (i != self)
                  sema_up(s[i]);
      for (i = 1..N-1)
            sema_down(s[self]);
}
```

2. Protection (16 pts)

a) (8 pts) In the Pintos fault page handler, the default exception handler code reports whether the page fault occurred in a “user context” or in the “kernel context.”
Briefly explain what each means and why you are much more concerned about page faults that occur in the kernel context. (State your assumptions, if necessary.)

Assuming that the system does not use paging (as is the case up until before Project 3), a page fault in a user context means that a user program accessed memory that is not mapped, or accessed it in a way not allowed by the current mapping (e.g., a write to a read-only page, an attempt to read kernel-only pages.) The correct action is to terminate the offending process and continue.

A page fault in kernel context, however, means that kernel code accessed memory that’s not mapped (or tried to write to read-only) memory. This indicates a bug in your code. At this point, the kernel panics, forcing the termination of all running applications and a reboot. This is clearly a more severe situation that losing one user process. (In Windows, you’d see the so-called Blue Screen Of Death instead. Linux in such situations only terminates the current process and prints an “Oops” in its kernel log – however, this approach runs the risk that kernel state is already corrupted, which can cause further corruption and often leads eventually to a lockup and/or loss of data.)

b) (8 pts) Your teammate suggests that, in order to check the validity of the buffer passed to the write() system call (which is declared as `write(int fd, const void *buffer, size_t length)`), you could simply check whether the beginning of the buffer (‘buffer’) and the end of the buffer (‘buffer + length-1’) are in the user virtual address range and have valid page table entries.

Show an example user program for which this approach fails.

It is necessary to check every single page that is spanned by the range [buffer, buffer+length). Otherwise, the kernel would attempt to access the entire memory for such programs as:

```
int main() {
      static char c[3];
      write(1, &c[2], 0xffffffff);
}
```

'buffer' would be &c[2], 'buffer+length-1' would be &c[0]. Both will have valid user virtual addresses.

3. Process States and Scheduling (36 pts)

a) (8 pts) Let |RUNNING| be the number of running processes in a system, let |READY| be the number of ready processes in a system, and let |BLOCKED| be the number of blocked processes (assuming the simple 3-state model.)

1. (4 pts) What are possible values for |RUNNING|?

Possible values are 0 (if the system is idle), or 1, 2, 3, … up to the number of CPUs in the system.

2. (4 pts) At a typical moment in a typical system, how does |READY| compare to |BLOCKED|?

Typically, |READY| << |BLOCKED|. If it were otherwise, the system would not be usable – many READY processes would make very slow progress. Look in your Windows Task Manager and click the Processes Tab. You'll see numerous process waiting for either user I/O, timer events, or external events such as network I/O. In Linux, use top.

b) (6 pts) The Windows Task Manager shows you CPU utilization in its "CPU Usage History" window. The Unix tool "xload", on the other hand, plots a machine's load average over time, which is computed in the same manner as you computed Pintos's load average in Project 1. What information does the load average provide for a user that CPU utilization does not?

The CPU utilization tells a user what portion of the time the CPU is used vs. what portion it is idle. The load average tells a user in addition what the CPU demand is, i.e., how much longer it will take to complete a job, giving the current number of ready processes that are competing for the CPU or CPUs. Another way to say this is to say that load average represents the number of CPUs you could keep busy, rather than how busy the current CPUs are.

c) (6 pts) In Unix terminology, a child process becomes a ‘zombie’ if it calls exit() before its parent calls wait(). Your system administrator claims that too many zombie processes are bogging down the machine. Is he correct? Briefly justify your answer.

As the question says, a zombie process is one that has already exited(), so it doesn’t consume any CPU. It’s not bogging down the machine (unless there’s an extremely large number of them – in this case, the system may be unable to assign new PIDs because the limit on the total number of processes is reached. Also, there is a small amount of memory consumed for each zombie to store its exit status for retrieval by the parent – but this will not usually bog down a machine.)

d) (8 pts) Show a task set that demonstrates that the Earliest Deadline First (EDF) real-time scheduling algorithm does not always produce a schedule with a minimum average waiting time.

We know that Shortest Process Next (SPN) produces schedules with the minimum average waiting time. Consequently, EDF produces such schedules for task sets where shorter tasks have farther deadlines. For example,
Task 1: cost = 3, deadline 4.
Task 2: cost = 1, deadline 6.
Assume both tasks arrive at 0. EDF schedules Task 1, then Task 2, average waiting time is 1.5. SPN would schedule Task 2, then Task 1. Minimum average waiting time is 0.5.

e) (8 pts) Consider the BSD4.4 Advanced Scheduler you implemented in Project 1. Explain how a thread could ``cheat'' this scheduler and use a much larger proportion of the CPU than other user threads with the same nice value. (Assume that the thread does not have the privilege to use interrupt disable/enable – clearly, disabling interrupts would give a thread exclusive access to the CPU.)

The scheme you implemented has a fundamental weakness, which many of you learned the hard way: it relies on sampling which thread is running at each timer interrupt. A thread could make sure that no CPU time is ever charged to it by giving up the CPU right before a timer interrupt occurs – this could be accomplished, for instance, by calling “thread_sleep(1).” Most likely, when the thread is woken up at the next interrupt, it will have the highest dynamic priority and reacquire the CPU. To be practical, a thread would have to insert checks for the current time (for instance, using the processor’s cycle counter register) periodically in its execution.

4. Mutual Exclusion (24 pts)

a) (10 pts) Rewrite the following code fragment to not use locks or semaphores. You only need to sketch the differences in the right column.

Pseudo-code is ok. Your solution should not contain any race conditions, not directly disable interrupts, and should not use busy-waiting.

```
int request_counter;
struct lock l;

void server()
{
  while (!shutdown)
    {
      handle_request();
      lock(&l);
      request_counter++;
      unlock(&l);
    }
}

int main()
{
  thread t[5];
  lock_init (&l);
  // start five server threads
  for (int i = 0; i < 5; i++)
    t[i] = thread_create (server);

  // wait for them to finish
  for (int i = 0; i < 5; i++)
    thread_join (t[i]);



  printf ("Requests handled:%d\n",
           request_counter);
}
```

```
int request_counter;
int thread_counter[5];

void server()
{
  while (!shutdown)
    {
      handle_request();
      int me = current()->id;
      thread_counter[me]++;

    }
}

int main()
{
  thread t[5];

  // start five server threads
  for (int i = 0; i < 5; i++)
    t[i] = thread_create (server);

  // wait for them to finish
  for (int i = 0; i < 5; i++)
    {
       thread_join (t[i]);
       request_counter +=
          thread_counter[i];
    }
  printf ("Requests handled:%d\n",
           request_counter);
}
```

This is a classic example where state partitioning can avoid race conditions, so locks are not necessary. Give each thread its own counter, and sum them after the threads have finished – no need to lock every increment.

b) (10 pts) Suppose you have a system that uses nonpreemptive scheduling. Assume that nonpreemptive scheduling means (1) that unblocking a thread does not trigger a scheduling decision and (2) that running threads are never involuntarily preempted. In such a system, would you still need locks (or other ways of providing critical sections?) Justify your answer!

Locks are still necessary, because processes can still lose access to the CPU inside a critical section if they voluntarily yield or block due to I/O or sleep. In those cases, a nonpreemptive scheduler is invoked and may schedule another thread, which could then enter the critical section, leading to a loss of mutual exclusion. (Locks can only be omitted if it can be ensured that a thread will not yield or block – which you may be able to verify for small sections of code, but this property clearly does not always hold.)

c) (4 pts) When designing synchronization primitives that work on shared-memory multiprocessor systems, we must use atomic instructions such as “compare-and-exchange”, “test-and-set”, etc. Explain the computer-architectural reason why an OS designer would attempt to minimize the number of times those instructions are executed.

These instructions must exclude other CPUs from accessing the memory location that the atomic instruction accesses. This requirement implies some form of bus locking or interprocessor communication, which costs many more cycles than a regular memory access.