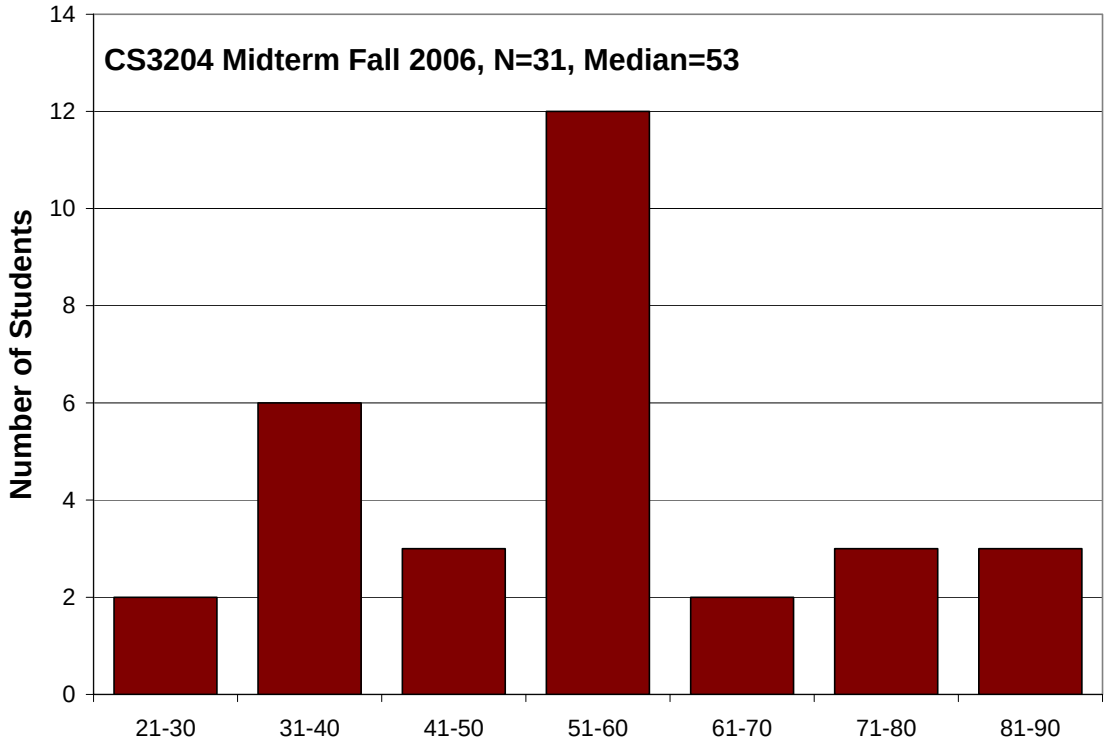
CS 3204 Midterm Solution

31 students took the midterm. The table below shows information about each problem, including who graded it. If you have questions, read this handout first, then approach who graded your question first.

Problem	1	2	3	4	Total
Possible Pts	24	16	36	24	100
Max	24	16	32	24	89
Min	0	3	6	1	24
Med	8	10	20	14	53
Avg	10.2	10.3	20.6	13.1	54.3
StdDev	6.3	2.9	7.1	5.8	16.6
Grader	Godmar	Jai	Godmar	Jai	

This midterm counts for 15% of your grade. Although your final grade will depend on both exams and the project, students who have scored lower than 30 should be aware that they are at risk of failing this class, unless their final shows a significant improvement. Students who have scored lower than 40 are at risk of getting a grade lower than the C that is required by CS classes that have CS3204 as a prerequisite.



Solutions are shown in this style.

Grading comments are shown in this style. For some problems, we list how many points of partial credit were given. We did this for problems where certain partially correct answers occurred frequently.

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);
}
```

To get full credit, you needed to implement a semaphore based on a monitor, and you needed to show the monitor in C syntax (that is, showing both locks and condition variables.) Since you are layering one synchronization construct on top of another, you should not manipulate blocked lists directly. I gave partial credit if I could find the following elements in the solution: use of condition variables, use of locks, keeping of a counter, something that shows that you understand what wait/signal are supposed to do.

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]);
}
```

Approaches that counted the number of threads that had arrived worked also, as long as they properly protected the associated counter and as long as they used only semaphores (as asked in the problem.) I gave partial credit if your code resembled a rendezvous.
A typical mistake was to initialize a single semaphore with the value N – in this case, the first thread to reach the barrier would run right through it.

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.)

Jai said he used the following scheme for this question: 3 pts for explaining what a page fault in a user context is, 3 pts for explaining what a page fault in the kernel context is, and 2 pts for making some reasonable statement of why the latter is more dangerous than the former.

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') 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[2];
      write(1, &c[1], 0xffffffff);
}
```

'buffer' would be &c[1], 'buffer+length' would be &c[0]. Both will have valid user virtual addresses.
As posed, the question allows for a second interpretation: only checking buffer and buffer+length is also wrong because you shouldn't check for buffer+length at all. Buffer+length does not need to be a valid address, only buffer+length-1 should be. [I had meant to write [buffer, buffer+length).] We gave full credit for pointing this out as well.
Jai says: pointing out the right approach – 5 pts. Showing C code with write() call, an additional 3 pts, for a total of 8.

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.

Since I had not specified that it's a uniprocessor system, the answers "0 or 1" were incomplete unless you stated you assumed a uniprocessor system. 3 pts were given for this partially correct answer.

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.

4 pts for a correct answer or guess. 0 pts for an incorrect guess. 2 pts for pointing out that the two numbers are independent of each other – although true, it skirts that the question is asking what to expect in a "typical moment in a typical system."

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.

For full credit, you had to point out the key difference, which is that the load average contains information about demand, e.g., how many CPUs would be needed to run all processes. Utilization does not have that information. Users can use this information to deduce how many longer their jobs would take to finish, given the current demand (averaged over the recent past.)

Several answers pointed out that xload displays a history, but this is also true of the Windows Task Manager – the question even points out that it has a "CPU Usage History" tab. Partial credit was only given if I could ascertain that you understood how the load average is computed.

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.)

For full credit, you needed to state that zombies don't consume CPU, and at most use up a small amount of memory for the exit status. Partial credit was given for answers that convinced me that you understood what wait/exit do in general.

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.
This question was easy to grade: any task set where the order of task execution cost doesn't match the order of deadlines yielded full credit. No partial credit was given unless you showed at least both deadlines and costs and some understanding of how EDF works and what "average waiting time" means.

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.
Some solutions suggested that the thread could simply set its recent_cpu low or its priority high, thereby forcing the scheduler to pick it. However, this would only be possible for a privileged thread that has direct access to its TCB. Because the question wasn't entirely clear on what "cheats" are allowed vs. what "cheats" are not, I gave 5 pts of partial credit for such answers. The question did have two strong indications that those cheats would not be allowed – such as using the term "user thread" and pointing out that the thread does not have access to the sti/cli instructions – this must means it is running in user mode. (I also didn't accept your saying to change the nice value, since the question clearly stated that the nice value of the threads should be the same.)

I also gave partial credit if your explanation showed a correct understanding of MLFQS, such as suggesting that a thread could sleep for some time to increase its CPU use after wakeup. However, this would not increase that thread's overall proportion of the CPU because it didn't use the CPU while it slept.

4. Mutual Exclusion (24 pts)

a) (10 pts) Rewrite the following code fragment to not use locks. You only need to sketch the differences in the right column. Pseudo-code is ok. Your solution should not contain any race conditions.

```
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.

We gave 4 pts of partial credit if you didn't actually remove the locks, but simply replaced them with another, but equivalent form, such as by mapping them to semaphores or (worst solution) disabling/enabling interrupts.

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.)

For full credit, you needed to give the correct explanation. Partial credit was given if you stated you did not need locks unless a thread blocked or yielded. However, note that an approach where you disallow blocks or yields would mean busy waiting for I/O, which isn't acceptable either. Partial credit was given if your reasoning was incomplete.

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.

For full credit, you needed to mention the bus locking or interprocessor communication requirement – the question had asked for a "computer architectural reason." Simply stating something like vague like "more CPU activity for atomic instructions" yielded only partial credit.