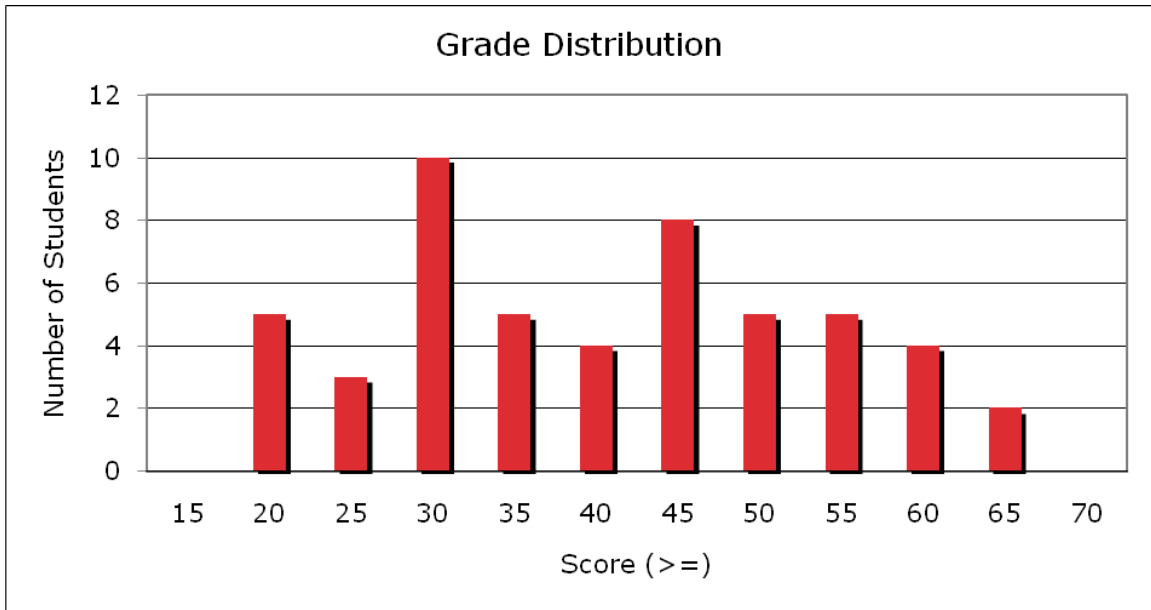


CS 3204 Midterm Exam

This is a closed-book, closed-internet, closed-cellphone and closed-computer exam. However, you may refer to your sheet of prepared notes. Your exam should have 8 pages with 4 questions totaling 100 points. You have 75 minutes. Please write your answers in the space provided on the exam paper. If you unstaple your exam, please put your initials on all pages. You may use the back of pages if necessary, but please indicate if you do so we know where to look for your solution. You may ask us for additional pages of scratch paper. You must submit all sheets you use with your exam. However, we will not grade what you scribble on your scratch paper unless you indicate you want us to do so. Answers will be graded on correctness and clarity. You may lose points if your solution is more complicated than necessary or if you provide extraneous, but incorrect information along with a correct solution.



| | Q1 | Q2 | Q3 | Q4 | Total |
|---------|-------|------|------|------|-------|
| Min | 0 | 0 | 0 | 6 | 21 |
| Max | 23 | 17 | 22 | 23 | 70 |
| Average | 11.31 | 6.63 | 15.8 | 15.8 | 43.6 |
| Median | 12 | 5 | 10 | 15 | 44 |
| Grader | Min | Butt | Min | Butt | |

1 Priority Inversion (25 pts)

In systems that use strict priority-based scheduling, synchronization techniques for shared resources may lead to priority inversion. We attempt to design a new technique for priority inversion. Here, anytime a thread acquires a lock, it is given a large priority (1 + the largest priority thread that will try to acquire the lock). The goal is to prevent the thread that holds the lock from being blocked indefinitely by other threads.

- a) (15 pts) Construct an example scenario to show whether or not this approach can help avoid priority inversion. Assume three threads A, B, and C with reasonable values for respective priorities.

Let A's priority be 1, B's 2, and C's 3.

Max priority = 3

Without new scheme

A acquires lock L
 C tries to acquire L
 C blocks on L
 B preempts A (B's 2 > A's 1)
 C cannot make progress due to B
 Although C's 3 > B's 2
 ⇒ Priority inversion

with new scheme

A acquires lock, get priority $1+3=4$
 C tries to acquire L
 C blocks on L
 B cannot preempt A (B's 2 < A's 4)
 A releases L
 C acquires L
No priority inversion

- b) (5 pts) What are some of the short-comings of this new approach, especially, compared to the priority inheritance approach you have implemented in Pintos.

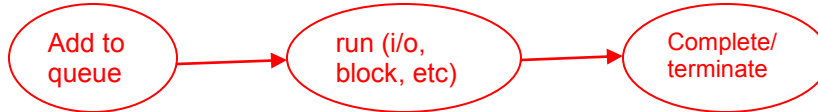
In a realistic system, it is not easy (or even possible in most scenarios) to determine the largest possible priority in advance. In contrast to this approach, priority inheritance automatically assigns successively higher priorities to a lock-holding thread if threads with higher priority try to acquire the lock, and thus do not suffer from this problem.

- c) (5 pts) A newly developed system, acme, only uses disabling interrupts to protect critical sections. Is priority inversion an issue in acme? Why or why not?

Since only one thread runs at a time when interrupts are disabled, a problem like priority inversion cannot occur in acme. Wrt. example in (a) if A is in the critical section neither B nor C can run. After A exits from the critical section, C will run as it has higher priority than B. Thus, no priority inversion issue arises.

2 Context Switching (25 pts)

- a) (5 pts) Draw the state diagram for a program thread running on a system that employs simple batch scheduling.



- b) (20 pts) Describe how you will design an experiment to measure the average context switch time between two processes on a given system. Clearly state your assumptions, provide pseudo code, and discuss any limitations of your approach.

We are interested in measuring the time between when control gets transferred from a thread A in user mode to when thread B starts running in user mode, or vice versa. Assume priority scheduling. One solution is to have two threads A & B with equal priority (higher than the main thread). Thread A runs in a very long loop and records the current value of the time. Thread B, when scheduled, simply yields the CPU. After the threads have run for several minutes, write the recorded values to a file. The file contains many time stamps of which most will differ by only a small amount as they are printed in a loop. However, each context switch will be indicated by a bigger duration between recorded timestamps. The difference between such timestamps will give the time it took to context switch from A to B, and back. Averaging these differences and dividing by 2 will give the average context switch time.

Limitations: A very fine grain timer is required. The timestamps does not capture the exact context switch start and end point, but within the time it takes to do one of the loop iterations.

Thread A

```

TR [LARGE_NUM];
Int c = 0;
While (time passed < 5 minutes)
{
    record timestamp in TR[c++];
}
  
```

Print TR to file;

Post process file to identify timestamp differences;

Thread B

```
yield;
```

Note: You cannot use printf in the loop, as the speed at which data is generated will result in the print buffer to be overrun and much of the data to be not printed.

3 Synchronization (25 pts)

A classic synchronization problem in CS is “Sleeping barber problem,” which is defined as follows. A barber has one chair where he cuts customer’s hair and 5 chairs for customers to sit in while waiting. If there are customers, the barber gives the next customer the hair-cut, and then the customer leaves. If there are no customers, the barber sits in the hair-cutting chair and goes to sleep. When a customer arrives he either wakes up the barber, or sits in the waiting chairs to wait for his turn. If all the waiting chairs are full, any new customer simply leaves. Provide pseudo code for the barber and customer thread, so that it provides proper implementation of the above activities, without synchronization issues such as race conditions and deadlock.

One possible solution: (any standard one will do)

```
Semaphore customers, barber initialized to zero;
Lock seat_access;
Int num_seats = 5;
```

Barber:

```
while(true)
{
    sema_down(customers); // wait for a customer
    lock_acquire(seat_access);
    num_seats++; // free a seat for more customers
    sema_up(barber); // barber can now cut hair, signal next customer
    lock_release(seat_access);
    // cut hair
}
```

Customer:

```
while(true)
{
    lock_acquire(seat_access)
    if ( num_seats > 0 )
    {
        num_seats--; // sit down
        sema_up(customers); // wake up/notify barber
        lock_release(seat_access);
        sema_down(barber); // wait for barber to be available
        //get haircut
    }
    else
    {
        // no free seats
        lock_release(seat_access)
        // leave without a haircut
    }
}
```

4 Synchronization II (25 pts)

Consider the code:

```
int bigBuffer[100] = { // 100 random numbers ... };
int sum;

void addMe(int start)
{
    int c;
    for (c=start; c< start+50; c++)
        sum += bigBuffer[c];
}

main()
{
    thread_create(A, addMe, 0 /*start parameter */);
    thread_create(B, addMe, 50);

    // wait for threads to finish
    print sum;
}
```

- a) (5 pts) What is the outcome of this program, and why?

There is a race for shared variable sum. The result will be some unpredictable value depending on how the two threads are scheduled.

- b) (5 pts) What is the expected correct outcome of the program? What is an easy way to ensure that the program produces correct output?

The expected outcome is the sum of all the numbers in the bigBuffer array.

For a single core using priority scheduling, one easy fix is to use different priorities for A and B. This will essentially serialize the two threads and remove the race.

For multiple cores, or for a generic solution, use a lock and protect the access to sum with it. The lock should be around the sum+= command, and not around the whole loop though.

- c) (15 pts) Rewrite the above program to produce the most efficient means for producing the correct output. State your assumptions clearly.

For multiple cores, any solution that used partitioning.

For a single core, just do this in a single loop in main! There is no true parallelism for a single core, so why pay the overhead of context switching etc.?