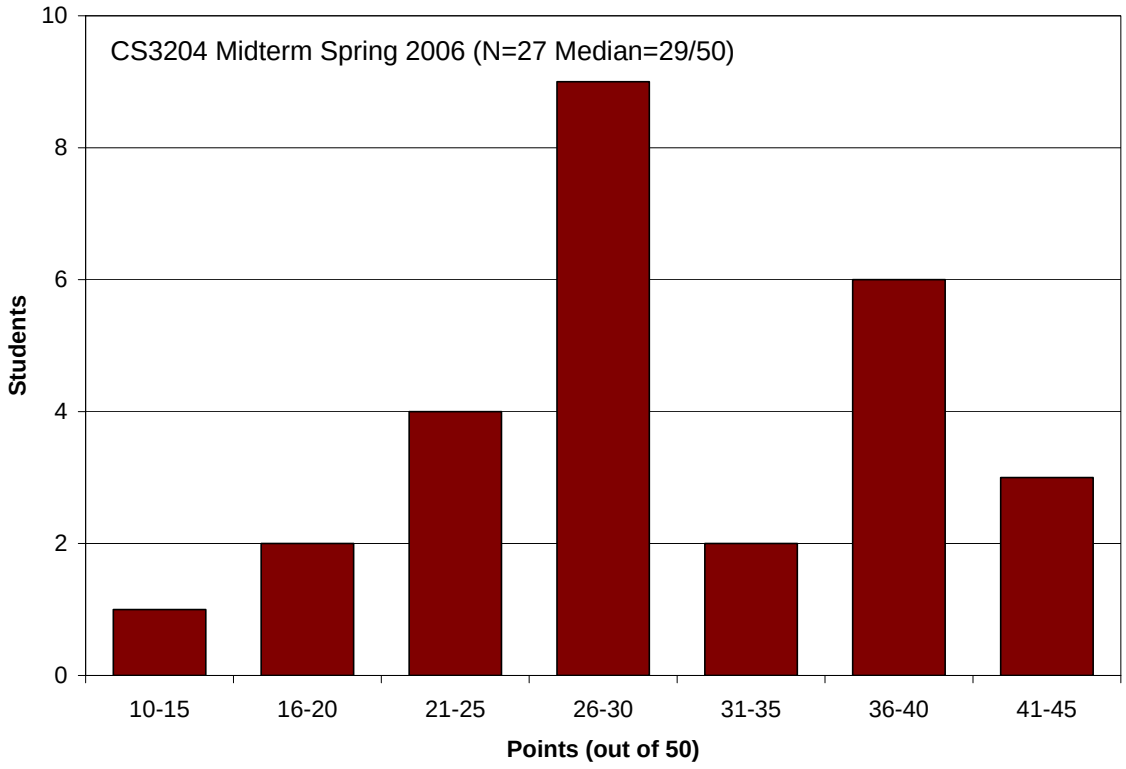
# CS 3204 Midterm Solution

27 students took the midterm. The table below shows who graded which problem and how well students did on the problem. The median was a 29. The midterm will count for approximately 15% of your final grade.

| Problem | 1 | 2 | 3 | 4 | 5 | Total |
|---|---|---|---|---|---|---|
| Points | 10 | 8 | 14 | 10 | 8 | 50 |
| Median | 8 | 3 | 12 | 8 | 4 | 29 |
| Average | 6.3 | 2.7 | 11.0 | 6.5 | 3.8 | 30.4 |
| Std Dev | 2.8 | 1.9 | 3.4 | 3.0 | 1.6 | 8.2 |
| Min | 1 | 0 | 3 | 1 | 1 | 13 |
| Max | 9 | 8 | 14 | 10 | 7 | 44 |
| Grader | Godmar | Godmar | Vijay | Vijay | a) Vijay<br>b) Godmar | |




*Answers on the following pages are shown in this style.*
[Notes regarding our grading are shown like this.]

# 1 System Calls (10 pts)

Suppose you decide to add a *gettimeofday* system call to Pintos. To user programs, *gettimeofday()* looks like an ordinary function call:

```
struct timeval {
   uint32_t tv_sec;    // seconds
   uint32_t tv_usec;   // microseconds
};

int gettimeofday(struct timeval *p);
```

If successful, gettimeofday will write the current time to *p and return 0. Suppose that the kernel has some way of reading the current time, such as a device driver that can read the PC's built-in real-time clock.

a) (2 pts) Name one change you would have you to make to Pintos's standard C library for user programs (located in lib/user/*.)

*You would have to assign a new system call number and add it to syscall-nr.h, and you would have to write a system call stub in lib/user/syscall.c for the new call.*

b) (4 pts) What changes, if any, would you have to make to your system call handling framework in userprog/syscall.c (excluding the actual implementation of *gettimeofday* itself)?

*You would at least have to add an entry to your system call handler table at the proper index that specifies the number of arguments to be copied (1 in this case) and the name of the function implementing the call.*

c) (4 pts) Suppose you implement *gettimeofday* as a function `sys_gettimeofday()` in syscall.c. Suppose a real-time clock device driver implements a function rtc_gettimeofday() that has the same signature as gettimeofday(), i.e. `int rtc_gettimeofday(struct timeval *p);` Implement `sys_gettimeofday` in terms of `rtc_gettimeofday`!

```
int
sys_gettimeofday(struct timeval *userp)
{
     if (verify_area(userp, sizeof(*userp)) != SUCCESS)
         process_terminate(-1);
     return rtc_gettimeofday(userp);
}
```

You may use the function names you used in your code without explaining them, as long as your names are descriptive.

*Note: if your verify() function takes a pointer, you would have to check the validity of "userp" and "((char *)userp) + sizeof(*userp) – 1", that is, the first and last byte of the struct since it could straddle a page boundary.*

# 2 Protection (8 pts)

a) (4 pts) Suppose an architecture does not provide software trap instructions such as the `int` instruction on the i386. Describe how you would implement system calls on such an architecture.

*Any instruction that will cause a fault can function as a system call trap. So you could use an illegal or a privileged instruction, or you could have the user program intentionally provoke an illegal memory access to some magic address when it wants to call into the kernel.*

b) (4 pts) Suppose an i386 ELF executable gets corrupted on disk in such a way that all bytes inside the code segment are overwritten with the value 0x90, which is the machine code for a NOP instruction on the i386. What would happen if a process tried to exec() this executable?

*The exec() would succeed, and the process would start executing nops until it falls off the end of its code segment. At this point, it will with high likelihood fault in one way or another. (If there's a hole following the code segment, it will page fault on the instruction fetch, otherwise, it will interpret the data in the following segments as instructions and try to execute them, which will usually quickly lead to an illegal instruction or another fault.)*

[ 2 pts for saying it will run and do nothing, and 4 pts for also realizing that it will eventually fault. No credit if you assumed that exec() would fail. ]

# 3 Semaphores (14 pts)

a) (8 pts, 2 pts each) Suppose you have 2 semaphores and 3 concurrent threads as in the example below. Assume that the threads run for a sufficiently long time under the regime of a preemptive scheduler.

```
struct semaphore S, U;
sema_init(&S, 2);
sema_init(&U, 0);
```

| // Thread 1 | // Thread 2 | // Thread 3 |
|---|---|---|
| while (1) {<br>  sema_down(&S);<br>  putchar('A');<br>  sema_up(&U);<br>} | while (1) {<br>  sema_down(&U);<br>  putchar('B');<br>  putchar('C');<br>  sema_up(&U);<br>} | while (1) {<br>  sema_down(&U);<br>  putchar('D');<br>} |

i. How many times will 'A' be printed?

*2 times because S is initialized to 2*

ii. How many times will 'D' be printed?

*2 times also (unless the scheduler starves Thread 3 indefinitely)*

iii. What is the minimum number of times that 'B' might be printed?

*0 times – it may never be printed if Thread 3 eats up the 2 signals on U before Thread 2 has a chance to run (note that this does not mean Thread 2 is starved indefinitely – just long enough for Thread 3 to eat the signals.)*

iv. Can the output string start with 'D'?

*No, because Thread 1 prints an 'A' before it first signals the semaphore U, which is initialized to 0 and which must have been signaled before Thread 3 can print 'D'.*

b) (6 pts) Suppose you have 1 semaphore and 3 concurrent threads as in the example below.

```
struct semaphore S;
sema_init(&S, 4); // suppose S represents a resource with 4 units

void resource_user_thread(void *_) {
  while (1) {
    // get two units of S
    sema_down(&S);
    sema_down(&S);
    // (use resource)
    // return two units of S
    sema_up(&S);
    sema_up(&S);
  }
}
...
tid_t t1 = thread_create(..., resource_user_thread, NULL);
tid_t t2 = thread_create(..., resource_user_thread, NULL);
tid_t t3 = thread_create(..., resource_user_thread, NULL);
```

Can this program deadlock?
If so, give the sequence of events leading to deadlock.
If not, say why not. Be specific.

*Deadlock is not possible because there are 4 resources, 3 threads, and each thread requests only 2 resources. Worst case is that each of the 3 thread holds one resource and tries to acquire the other, but even in this case there's one resource left that allows at least one thread to make progress, avoiding deadlock.*

# 4 Scheduling (10 pts)

Assume that Pintos's thread_set_priority() function is extended such that it takes an argument of type `tid_t`. `thread_set_priority(t, p)` sets the priority of thread with thread id `t` to `p`.

```
#define HIGH_PRIORITY           PRI_DEFAULT - 5
#define MEDIUM_PRIORITY         PRI_DEFAULT
#define LOW_PRIORITY            PRI_DEFAULT + 5

void printer(void *name)
{
  while (1)
    printf("%s", name);
}

int main()
{
  tid_t t[4];
  int i, c;

  ASSERT (!enable_mlfqs);

  thread_set_priority(thread_current()->tid, HIGH_PRIORITY);
  t[0] = thread_create("a-thread", LOW_PRIORITY, printer, "A");
  t[1] = thread_create("b-thread", LOW_PRIORITY, printer, "B");
  t[2] = thread_create("c-thread", LOW_PRIORITY, printer, "C");
  t[3] = thread_create("d-thread", LOW_PRIORITY, printer, "D");

  c = 0;
  for (i = 0; i < 10; i++)
    {
      thread_set_priority (t[c], MEDIUM_PRIORITY);
      timer_sleep (TIMER_FREQ);
      thread_set_priority (t[c], LOW_PRIORITY);
      c = (c + 1) % 4;
    }
}
```

a) (8 pts) What would this kernel output when run, assuming that strict priority scheduling is used?

*It would output*
AAAAA…AAAAAABBBBBB…BBBBBBCCCCCC…CCCCCDDDDDD…DDDDDDD
AAAAA…AAAAAABBBBBB…BBBBBBCCCCCC…CCCCCDDDDDD…DDDDDDD
AAAAA…AAAAAABBBBBB…BBBBBB
*switching 10 times between threads, once every second, until 10 seconds have passed.*
*[ After that, either the kernel would shut down (if –q was given) or the threads would continue to run and output*
*AAAAA…AAAAAABBBBBB…BBBBBBCCCCCC…CCCCCDDDDDD…DDDDDDD… ad infinitum*
*switching every time slice. ]*
[8 points for a fully correct answer that covered the first 10 seconds (we didn't say if the main thread would shut down the system afterwards.)
For those solutions that said the output to be ABCDABCDABCD 6 points were awarded]

b) (2 pts) What other scheduling policy for threads t[0] to t[3] would produce the same output?

*Round-robin. The main thread manipulates the priorities of the other threads such that a round-robin scheduling policy is produced.*
[2 points for the correct answer
Some of the answers said that FCFS would produce a similar output which is incorrect as FCFS does not preempt a process.]

# 5 Short Questions (8 pts)

a) (4 pts) Why are page sizes always powers of 2?

*Because then the virtual page number can be obtained from the virtual address simply by considering a number of higher-order bits. This would not work if page sizes weren't powers of 2.*
[4 pts for a fully correct answer.
Partial credits were assigned to solutions that mentioned about the virtual to physical address translations. Some of the solutions mentioned that it is efficient to have the page sizes as powers of 2 because all data is internally stored in a binary format. Such solutions received no credit. Also solutions that did not mention about the address translation and stated that the fragmentation can be avoided did not get any credits.]

b) (4 pts) Is it ever safe to pass a pointer to a local variable to another thread? If not, say why not. If so, say under what circumstances.

*It is safe if you know that the thread to which the pointer is passed won't access it after the function that defines the local variable has exited. This can be ensured using synchronization mechanisms such as semaphores, condition variables, or join()/wait(). (Potential examples of this in the projects were the timer_sleep() implementation in Project 1 and the process_execute() implementation in Project 2.)*
[4 pts for a fully correct answer.
Some assumed that the address space could switch between threads (if they're processes), in which case the pointer would point to something else. This is true, but if the variable is in kernel code on the kernel stack it would still be accessible, at least until the other process returns from kernel mode – so it could be safe. We deducted 1 pts for this answer.
Some ignored the lifetime issues at all and pointed out that locking is required. This however applies to any shared variable and does not address the question specifically.]