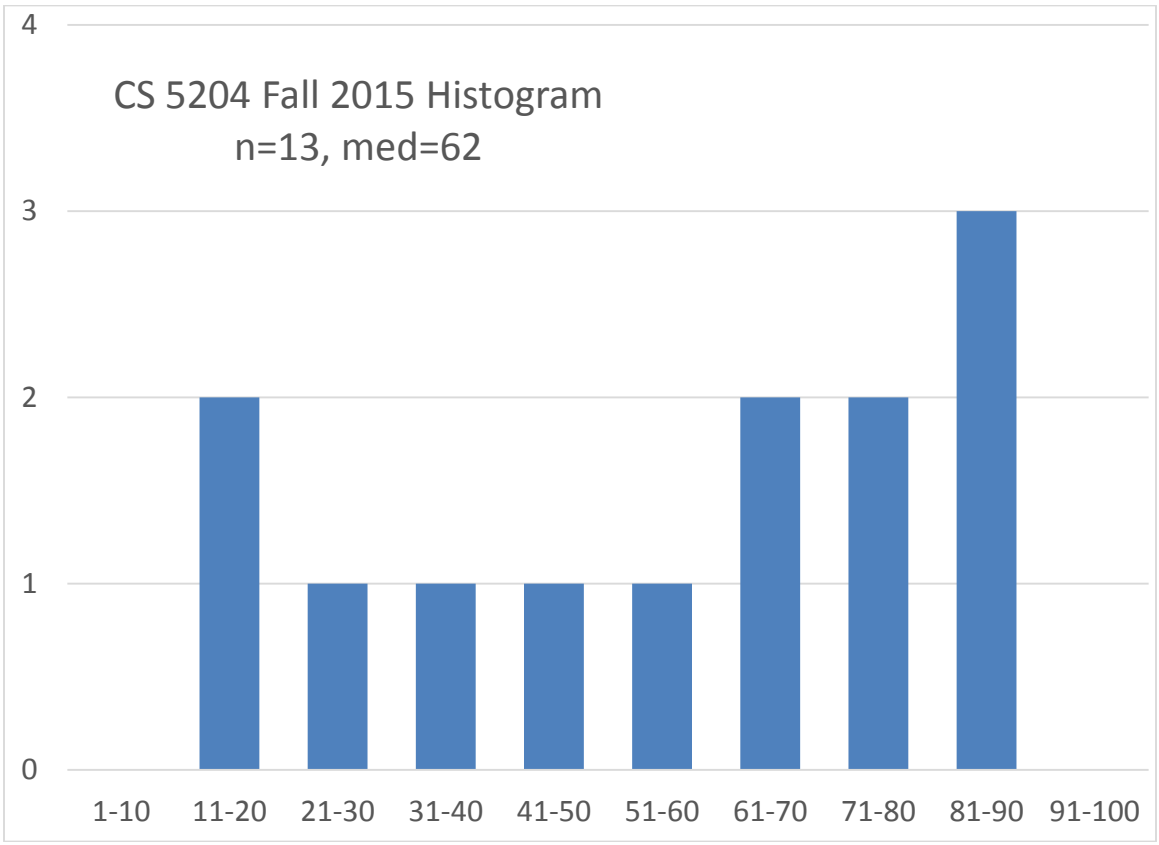
# CS 5204 Midterm Solutions

13 students took the midterm. See below for statistics.

| | P1 | P2 | P3 | P4 | P5 | P6 | Total |
|---|---|---|---|---|---|---|---|
| **Possible** | 14 | 14 | 20 | 20 | 22 | 10 | 100 |
| **Min** | 0 | 0 | 0 | 1 | 4 | 0 | 12 |
| **Max** | 12 | 13 | 20 | 20 | 22 | 10 | 87 |
| **Average** | 6.5 | 7.5 | 10.1 | 11.7 | 13.6 | 6.2 | 55.5 |
| **Median** | 8 | 8 | 9 | 14 | 14 | 10 | 62 |
| **StDev** | 4.3 | 4.1 | 6.4 | 6.3 | 7.0 | 4.5 | 27.5 |




*Solutions are shown in this style.*
*Grading comments are shown in this style.*

# 1. Mode and Context Switching (14 pts)

a) (4 pts) You are thinking about how to optimize your Pintos system call implementation. You study the interrupt entry code and notice that a fair amount of code is executed from the interrupt handler, dispatch, all the way to the system call handler. You notice that the interrupt handler saves all registers, even though on a system call only callee-saved register need saving, and that there are two table-based dispatches: one to find the syscall handler, and then a second to find the address of the code handling the system call.

You think that you could optimize it by asking the processor architects to add a special "SYS_CALL <addr>" instruction, which works as follow: (1) transition to kernel mode, (2) call the function at <addr> inside the kernel, (3) return and transition from kernel to user mode. In fact, you have heard that modern processors might already include such an instruction.

Will this idea work as described? State briefly why or why not!

*No it won't work. Allowing user processes to call arbitrary kernel functions violates an OS's protection guarantee; entry to the kernel must be controlled. The syscall/sysenter instructions that were introduced in x86_64 provide a faster way to enter the kernel, but they do not allow the invocation of arbitrary functions in the way this proposal would.*

*I gave partial credit for answers that pointed out that it would pose other issues, such as ensuring that user processes know the addresses of the kernel entry points. Some of you pointed out that arguments need to be passed; that applies to any type of instruction used to perform a system call.*

b) (3 pts) In a system that supports multiple user processes, not all mode switches lead to context switches.
Give an example of when a mode switch is not followed by a context switch!

*Two examples include:*

- *A system call that can be serviced without having to block the calling process*
- *A timer interrupt before the current process's time slice is exhausted.*
- *A minor page fault that can be serviced without having to evict a page.*

c) (3 pts) Why do traditional kernels such as Linux in their default configuration impose a limit on the number of file descriptors a process may create?

*File descriptors use up kernel memory, which is a shared resource. Limiting their number reduces the risk for one process to intentionally or inadvertently deny service to others.*

*Note that the question asked "why does kernels impose a limit," not "why does kernels use integer numbers"*

d) (4 pts) You are thinking about how to optimize file descriptor-related system calls in Pintos. You notice that the system call API says a file descriptor is a 32-bit int. Since Pintos still supports only 32-bit virtual addresses, you realize that you can cast the pointer of the underlying `struct file *` (or `struct dir *`) inside the kernel to such an integer and return it to the user on `open()`. For safety, you'll still need to ensure that the user process did not manufacture the pointer – to that end, you embed a magic constant unknown to the user in the beginning of each `struct file` or `dir`.
Critique this idea for its feasibility and practicality!

*Perhaps I shouldn't have asked for "feasibility" and "practicality" – it's certainly feasible in the sense that it can be implemented. The question is would it be correct?*
*A malicious user process could simply try all possible kernel virtual addresses, hoping to hit a file descriptor allocated by any process. On a 32-bit system, this would require only a small number of tries. It is not correct to perform file operations on a file descriptors not opened by a process.*

## 2. Scheduling (14 pts)

a) (4 pts) Consider an embedded system that uses a strict priority-based scheduler. It runs multiple recurring tasks which are assigned different priorities. Occasionally, tasks need to synchronize with one another about results they have computed. One of the developers of the system has read that certain synchronization constructs are expensive, and wants to optimize inter-task communication by avoiding the use of locks or semaphores. He proposes the following code:

```
// atomically set by thread producing result
volatile int result_received;
…
while (!result_received)
   thread_yield();
```

Critique this idea for its feasibility and practicality!

*The solution is busy-waiting, which interacts badly with a priority-based scheduler. In particular, if the thread waiting for the result has a higher priority than the thread producing the result, the latter will never be scheduled. Inter-thread communication must be reliable no matter the policy choice of the scheduler.*
*Note that the question specifically referred to a strict priority scheduler where the possibility arises that the result is never produced.*

b) (4 pts) In the Mars pathfinder story (as well as in a number of project 1 Pintos tests), you had seen how priority inversion can affect systems in which there are threads of three different priority classes: high, medium, and low.

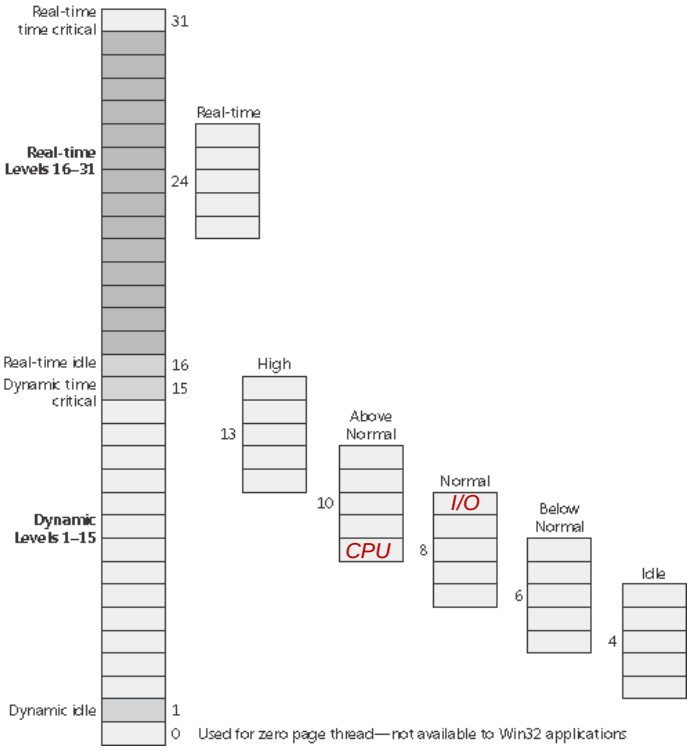
Can priority inversion also occur in systems that use only two priority levels (say high and low)? Justify your answer!

*Yes it can. Priority inversion describes the effect that a high priority task is waiting on a resource held by a low priority thread, which clearly can occur in a system with just high/low priorities. If there are only two threads, this priority inversion might go undetected since the lower priority thread is the one being scheduled. If there is a third high priority thread, it may cause starvation just like if there were a 3rd medium priority thread in the Mars example.*

c) (6 pts) Consider the scheduling classes in a Windows kernel (shown to the right), which implements dynamic priority adjustment on top of a strict priority-based scheduler in a fashion similar to BSD 4.4's advanced scheduler you have implemented.

i. (3 pts) Write the word "CPU" in the figure to mark the dynamic level at which the OS would most often place a CPU-bound thread that was given "Above Normal" priority class by the user!

ii. (3 pts) Write the word "I/O" in the figure to mark the dynamic level at which the OS would most often place an IO-bound thread that was placed in the "Normal" priority class by the user!




## 3. Synchronization (20 pts)

a) (12 pts) Pintos implements its condition variables on top of semaphores using a technique invented by Andrew Birrell when he implemented POSIX condition variables on top of Microsoft Windows semaphores.
Sketch how you would do the reverse, i.e., implement semaphores using condition variables!

```
struct semaphore {
      int value;
      condvar cond;
      mutex lock;
};
```

```
void sema_down(struct semaphore * s) {
      mutex_lock(&s->lock);
      while (s->value == 0)
           condwait(&s->cond, &s->lock);
      s->value--;
      mutex_unlock(&s->lock);
}

void sema_up(struct semaphore *s) {
      mutex_lock(&s->lock);
      if (s->value++ == 0)
          condsignal(&s->cond);
      mutex_unlock(&s->lock);
}

bool sema_try_down(struct semaphore * s) {
     mutex_lock(&s->lock);
     bool result = false;
     if (s->value > 0) {
          success = true;
          s->value--;
     }
      mutex_unlock(&s->lock);
     return success;
}
```

*Some of you used condition variables without a lock. That's an antipattern.*

b) (8 pts) Hardware Lock Elision (HLE). Locking remains the most widely used strategy for mutual exclusion. However, on a multiprocessor, acquiring a lock can be expensive as it involves at least one costly atomic instruction that can take many cycles. Intel introduced TSX (Transactional Synchronization Extensions) that allow programmers to use instruction prefixes that optimistically elide the cost of an atomic instruction. Intel describes these extension thusly:

*These extensions can help achieve the performance of fine-grain locking while using coarser grain locks. These extensions can also allow locks around critical sections while avoiding unnecessary serializations. (…) Even though the software uses lock acquisition operations on a common lock, the hardware is allowed to recognize this, elide the lock, and execute the critical sections on the two threads without requiring any communication through the lock if such communication was dynamically unnecessary.*

Based on what you know about the nature of race conditions against which locks can defend, describe how the hardware would detect if "such communication was dynamically unnecessary!" Note that when Intel refers to "threads" they mean "cores"!

*The part omitted from the text above states:*

*These extensions can help achieve the performance of fine-grain locking while using coarser grain locks. These extensions can also allow locks around critical sections while avoiding unnecessary serializations.* ***If multiple threads execute critical sections protected by the same lock but they do not perform any conflicting operations on each other's data, then the threads can execute concurrently and without serialization.*** *Even though the software uses lock acquisition operations on a common lock, the hardware is allowed to recognize this, elide the lock, and execute the critical sections on the two threads without requiring any communication through the lock if such communication was dynamically unnecessary.*

Source: https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell

*The key is that the processor must monitor for potentially conflicting accesses to data. Once a hardware-elided lock is taken, the processor will start to monitor what it accesses (separately for reads and writes); if it detects that there is overlap between its write set and that the read sets of other processors that were attempting to acquire a lock at the same address (or vice versa), and if this overlap means that a conflict could occur, it will undo any updates, reroll execution to the point where the lock would be taken, and take the lock.*
*Note that the transaction will not be aborted simply because there is an overlap in terms of the data being accessed, so just saying that the hardware detect data overlap is incorrect.*

## 4. Virtual Memory (20 pts)

a) (12 pts) Pintos, as you implemented it in project 3, does not provide a `malloc()` function for its user processes. Describe the steps you would need to take to implement `malloc()` for user processes!

   i. Describe which changes, if any, would be required to the user's C library (e.g. lib/user/*.c)!

*You would need to add an implementation of a user-level memory allocator whose task is to serve client requests for malloc/free. For instance, an explicit list-based or tree-based allocator. The allocator would obtain memory in larger chunks from the OS. An example is sbrk(), which asks the OS to allocate virtual memory contiguously above the data segment.*

   ii. If not already discussed in i), describe specifically which changes, if any, would be required in lib/user/syscall.[ch]!

*A new system call stub would need to be added for the new system call, passing the appropriate parameters and receiving information about the virtual address being allocated by the kernel.*

iii. Describe which changes, if any, will be required in your kernel! Include in your discussion any new system calls you might introduce, and how those system calls would affect your virtual memory implementation. Your implementation should allocate physical memory on demand, not eagerly, and be fully integrated with your page replacement strategy!

*sbrk() can be implemented by adding entries for additional anonymous pages to the supplemental page table (the same type as say pure bss pages). Depending on how the API is chosen, the kernel may also have to keep track of the process's current heap boundary (the "break"). On the first page fault on any so allocated page, the page is mapped to newly allocated physical memory. Subsequently, it may be evicted to swap and can be restored from there – the logic for this is identical to bss pages and is already in your kernel.*

*Generally, I was looking for a basic understanding as to what is done on the user side and what is done on the kernel side, but also for an understanding that malloc() requires a two-level implementation: an allocator on the user side, and some way of getting larger chunks from the kernel. Lastly, I was looking for a correct description how this will integrate into your virtual memory implementation.*

b) (4 pts) Suppose an adversary discovered an undocumented, non-privileged CPU instruction that allowed read-only access to memory using a physical, rather than a virtual address.
What kind of gains could an adversary derive from such a discovery?

*The adversary would achieve the ability to inspect both kernel memory and physical memory used by other processes, violating a core tenet of isolation in OS. Note that some architectures have instructions for accessing physical memory directly, but they are privileged instructions intended for use by the kernel.*

c) (4 pts) Suppose you are implementing virtual memory on an architecture that is otherwise similar to the x86, but which does not provide a "dirty bit" in its hardware page table entries.
How could you emulate the dirty bit in this case?

*The dirty bit can be implemented in software by mapping the page read-only, and setting it on a page fault caused by a write access, which maps the page read-write before returning.*

*Some of you proposed checksumming on eviction, but that seems out of proportion given that the purpose of the dirty bit is to aid in making eviction decisions.*

## 5. File Systems (22 pts)

a) (10 pts) Unix-based file systems implement so-called 'hard links' as a feature. For instance, if you create a file named **"a"**, then the Unix command **"`ln a b`"** will make it so that subsequently the names **"a"** and **"b"** refer to the same file on disk. You can open file **"a"**, write data into it, then open file **"b"** and

read the data you have written. You can remove the 2 names for the file separately: for instance, removing `"a"` will still retain the file in the file system (so `open("b")` will still work); the file will be deleted only after "a" and "b" have been removed.
Describe how you would implement this facility in your project 4 file system implementation! State any assumptions specific to your implementation as necessary!

i. Describe what changes, if any, to your on-disk data structures you would require!

*You would need a reference count in the on-disk inode to keep track of how many names exists for a file.*

ii. Describe what changes, if any, to your in-memory data structures inside your file system you would require!

*It should not require any. The open file table does not associate open files with their names (otherwise, the ability to keep already deleted files open would not work.)*

iii. Sketch how you would implement the following new system call:

```
int link(const char *oldpath, const char *newpath);
```

which creates a second name 'newpath' for an existing file 'oldpath'.

- *Look up oldpath and make sure it exists*
- *Find the inode number corresponding to oldpath from its containing directory*
- *First the directory in which newpath would be created and ensure that it does not exist*
- *Add an entry for newpath to the directory, referring to oldpath's inode number*
- *Increment the inode's reference number by 1*

*Same of you suggested cloning the file's metadata – this will not work when future writes happen to the file.*

iv. Describe what changes, if any, you would need to make to the `remove()` system call!

*Instead of removing the inode, it would merely decrement its reference count. A file would be deleted only once the last name disappears.*

b) (4 pts) Suppose you are asked to design a new file system that will be used for promotional DVD-ROMs. Advertisers will compile a set of promotional

materials, including videos, that will be burned onto a DVD-ROM in its entirety. Once burned, the DVD-ROM cannot be modified further.
What kind of block allocation strategy would you use, and how will this affect your metadata design?

*A contiguous layout will do, allowing a metadata design that uses as little as a single extent per file. It reduces metadata overhead and increases sequential read performance. ISO 9660 works like that.*

c) (4 pts) Traditional file systems in the heritage of the original Unix v7 file system use in-place updates – once a disk block is allocated to hold some part of the data of one file, it will not change even when the file data is overwritten. Some recent file system designs avoid "in-place" updates to files. Describe *one* advantage and one disadvantage of the in-place strategy!

i. Advantage of in-place strategy:

*Simple. Once allocated, it will never change. Any pointers need to be updated only once, never on subsequent operations.*

ii. Disadvantage of in-place strategy:

*Can easily lead to fragmentation, and subsequent discontinuous accesses that limit performance.*
*I accepted various other answers as long as they were related to whether the allocation was in-place or not and represented a unique advantage/disadvantage.*

d) (4 pts) Some of you implemented read-ahead in your p4 buffer cache implementation, but didn't notice significant, or any, performance improvement.
Describe a necessary characteristics of a workload for which properly implemented read-ahead results in performance improvement!

*For read-ahead to be effective, a) accesses must be sequential and b) the accessing process must do some work between accesses so that I/O and computation can overlap. If the accessing process does not do computation, benefits will be marginal as it will have wait for the read-ahead thread to complete.*

## 6. Virtualization (10 pts)

a) (5 pts) You used the QEMU virtual machine emulator to run your Pintos kernels on a shared computer system (the rlogin nodes) to which you have neither administrative access (you are not root) nor do you have the ability to install your own custom kernel to run code in privileged mode.

Given that the Pintos kernel you implemented makes use of privileged CPU

instructions, and given that you used standard compiler/linker tools to build your Pintos kernel images, describe how this is possible!

*This can be done in 2 ways: traditional deprivileging, in which the instruction is executed, causing a trap that is then relayed as a SIGSEGV Unix signal to the qemu process, which will then emulate the instruction. The second way (which is what Qemu actually does) is to rewrite the guest's kernel so that any privileged instructions are replaced with calls to Qemu's runtime system. Neither of the two options requires privileged access to the host machine (or the ability to execute privileged instructions on the hardware), making Type II hypervisors such as Qemu feasible.*

b) (5 pts) You are hired by a 3-letter agency to implement zero-day exploit software to attack ISIL. The software will be delivered using a spear-phishing attack. Once deployed, the software needs to run undetected. In particular, if the software can sense that it is run inside a virtual machine, it should quietly delete itself! You are responsible for writing the code that determines whether the exploit runs inside a virtual machine or not.

Describe how you would approach this assignment!

*Strictly speaking, it should not be possible: Popek's equivalence postulate states that programs running inside a VMM should exhibit essentially identical behavior as when run on real hardware.*
*In practice, though, there may be telltale signs to look for: for instance, you could check if the machine has a device driver loaded that is used only when the OS is run as a VM guest, or you could use a timer to attempt to measure the performance of instructions that are typically emulated in a VM. In practice, it can be extremely difficult for a hypervisor to hide its presence to inquisitive guests.*