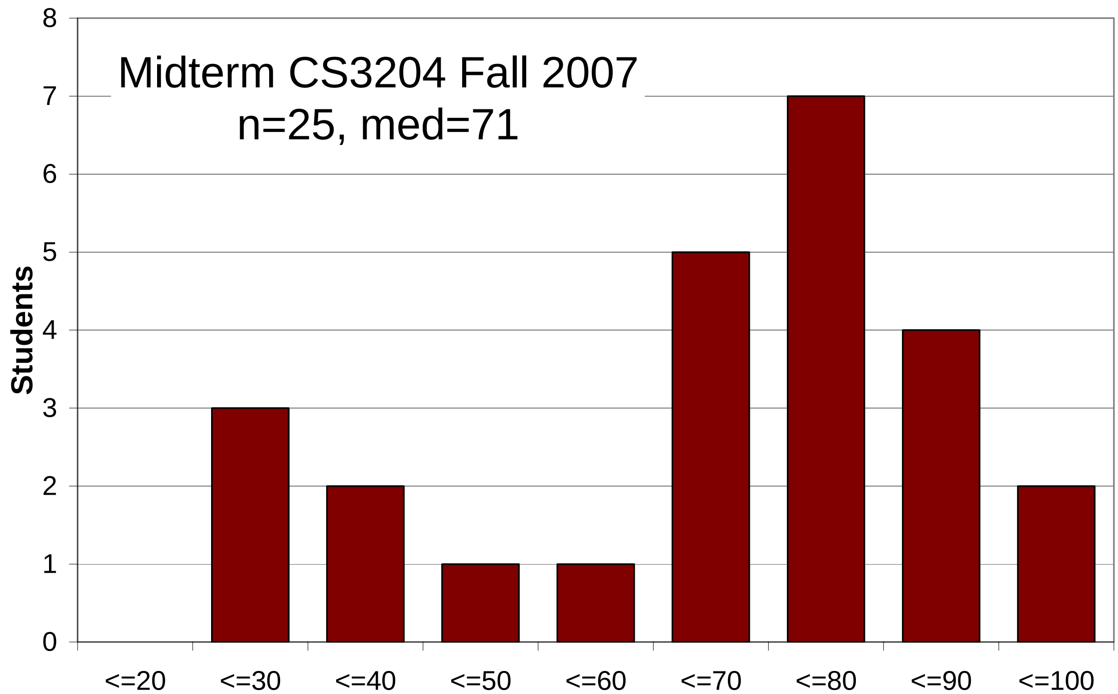
CS 3204 Midterm Solutions

25 Students took the midterm. The table below shows who graded which problem. If you have questions, read this handout first, then address your question to the person who graded it. If you can't find a resolution, come and see me.

Problem	***1***	***2***	***3***	***4***	***Total***
Median	15	21	10	26	**71**
Average	13.9	19.3	9.4	21.9	**64.5**
StdDev	7.4	5.4	4.8	9.7	**22.1**
Min	0	8	0	1	**22**
Max	20	24	18	34	**92**
Possible	20	24	20	36	**100**
Grader	Godmar	Xiaomo	Godmar	Jai	

This midterm counts for about 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. They will need to show significant improvement in the final exam. Students who have scored lower than 55 are at risk of receiving a grade lower than the C that is required by CS classes that have CS 3204 as a prerequisite.



Solutions are shown in this style.
Grading Comments are shown in this style.

1. Semaphores (20 pts)

a) (20 pts) Semaphores.
Write a program using 2 threads that will ***always deadlock*** (independent of which scheduling policy is in effect). Your program should ***use only semaphores*** (that is, no timer_sleep() or thread_yield() or similar functions).

The idea is to use two semaphores l1 and l2 as resources (initialized to 1), which the two threads acquire in a different order. To ensure that deadlock always happens, the threads each must know that the other has acquired its resource, which can be easily accomplished by using a rendezvous after acquiring the first resource acquire. This rendezvous requires the use of two additional semaphores s1 and s2, initialized to 0.

// Insert your definitions here (show initial values for your semaphores!) *semaphore s1(0), s2(0), l1(1), l2(1);*	
// Thread 1 *l1.down();* *s1.up();* *s2.down();* *l2.down();*	// Thread 2 *l2.down();* *s2.up();* *s1.down();* *l1.down();*

This question ended up being a lot more difficult to grade than I had expected.

The difficulty centers around the meaning of "deadlock." What I had intended was a deadlock that is a (static) resource deadlock, which is a situation that arises due to a sequence of requests, acquisitions and subsequent releases of static, identifiable resources by processes. Only this type of deadlock can be described by the 4 necessary conditions discussed in class.

However, in lecture (as well in the example exams and even in the OS literature), the term deadlock is often used more informally to describe a number of situations in which progress is prevented because (static or dynamic) resources will not become available to the processes waiting for them.

An interesting discussion of this topic can be found in Levine's paper on "Defining Deadlock" http://doi.acm.org/10.1145/881775.881781.See in particular Sections 1 and 2, where Levine points out that even OS textbooks sometimes contain examples that don't strictly follow (their own) definition of deadlock. For instance, our textbook (pg 243) gives the example of two trains meeting at a crossing, and a Kansas law that requires each train to give preference to the other. Obviously, this example is not a deadlock – there is no hold-and-wait, for instance. Instead, it is a simultaneous request for a single resource which can be handled by arbitrating the request in some way.

Semaphores can be used in two roles: initialized as 0, in which case they represent a signaling facility that can be viewed as a dynamic resource: sema_up() creates it, sema_down() consumes the resource. If the semaphore is initialized to 1, however, sema_down() acquires the resource, and sema_up() releases it.

I ended up deciding on the following grading scheme:
Full credit was given to any code that, based on semaphores, reliably brings about a state in which progress to both threads is prevented.

I deducted if you included extraneous information, if your solution only sometimes deadlocked, or if you included information that made me conclude you don't fully understand the interaction of initial values and down/up; I also deducted if only one thread would reach a state in which progress was prevented.

I did not assign credit if your program used a means other than semaphores to prevent progress (such as infinite loops or the like.), or if it never deadlocked.

2. Process States and Priorities (24 pts)

a) (14 pts) Assume a preemptive, strictly priority-based scheduler that implements priority inheritance. Under this assumption, write down the output of the following program. You may assume that thread_create and lock_* functions work as in Pintos.

```
void
test_priority (void)
{
  struct lock lock;

  /* Make sure our priority is the default. */
  ASSERT (thread_get_priority () == PRI_DEFAULT);

  lock_init (&lock);
  lock_acquire (&lock);
  thread_create ("thread1", PRI_DEFAULT + 1, thread1_func, &lock);
  msg ("A");

  thread_create ("thread2", PRI_DEFAULT + 2, thread2_func, &lock);
  msg ("B");

  lock_release (&lock);
  msg ("C");
}

static void
thread1_func (void *lock_)
{
  struct lock *lock = lock_;

  lock_acquire (lock);
  msg ("D");
  lock_release (lock);
  msg ("E");
}
```

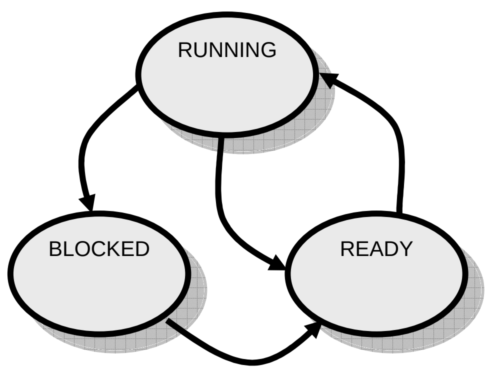
```
static void
thread2_func (void *lock_)
{
  struct lock *lock = lock_;

  lock_acquire (lock);
  msg ("F");
  lock_release (lock);
  msg ("G");
}
```

The output is:

A B F G D E C

(You should already be familiar with this program as priority-donate-one)

b) (6 pts) Recall the simple 3-state process diagram:



Explain why there are no arrows from BLOCKED to RUNNING and from READY to BLOCKED!

i. (3 pts) BLOCKED → RUNNING is missing because:

Blocked processes first transition into the READY state because this model separates the act of unblocking a processing from the act of scheduling that process. The former is dictated by the occurrence of an event – the latter is a policy decision that is under the purview of the scheduler.

Xiaomo says: If (they) mention that BLOCKED threads have to go to READY status because they need the scheduler to decide which thread should be run/executed at next, then full points. It will be even better if mentioned there is only one RUNNING thread at any time.

ii. (3 pts) READY → BLOCKED is missing because:

A process blocks if it encounters a situation in which it cannot continue execution. Ready processes do not currently execute, therefore, they cannot block.

c) (4 pts) Suppose that only two threads are in the READY state in a system at any given time. Is there a need for priority inheritance in such a system? Say why or why not!

As posed, the question does not say anything about how many threads are in the BLOCKED state. If higher-priority threads are BLOCKED on locks held by lower-priority threads, then priority inversion can certainly occur if this lower-priority thread is preempted by a second, medium priority thread. If priority inversion can occur, there is a need for priority inheritance.

What I had meant to ask was: "Suppose that there are only two threads in a system, and suppose both are in the READY or RUNNING state". Under this assumption, there is no need for priority inheritance, because priority inversion cannot occur since even if a high-priority thread eventually blocks on a resource held by a low-priority thread, the low-priority thread will be run. (If the low-priority thread is also blocked on a resource, this would be deadlock, which we ignore here.)

I instructed Xiaomo to grade the question as posed. But if you interpreted the question to mean that there are only two threads in total, I told him to accept that as well. Under either assumption, your answer needed to be correct.

Xiaomo pointed out that some equated priority inversion with deadlock, which is not correct. Also, the simple fact that a thread may block while another thread holds a resource does not necessarily imply priority inversion. Rather, priority inversion occurs only if the resource holder cannot make progress towards releasing the resource.
Finally, note that the problem is priority inversion. The solution is priority inheritance (or donation, which means the same thing.)

3. Deadlock (20 pts)

a) (4 pts) While debugging your Pintos kernel, you notice that it's getting stuck. Your teammate says that Pintos must have deadlocked. Is his inference correct? Justify your answer!

No, this inference is incorrect. "Getting stuck" could happen for any number of reasons that do not indicate deadlock: processes could loop infinitely, processes could sleep, could wait for I/O, or you may have a bug in your scheduler in which you incorrectly manage your ready queue.

See also question 1 a). However, even under an extended view of deadlock that includes static and dynamic resources, not all reasons for getting stuck will be qualifiable as deadlock.

b) (8 pts) The following comment can be found in a file that is part of the Linux kernel (linux/mm/rmap.c):

```
/*
 * Lock ordering in mm:
 *
 * inode->i_mutex  (while writing or truncating, not reading or faulting)
 *   inode->i_alloc_sem (vmtruncate_range)
 *   mm->mmap_sem
 *     page->flags PG_locked (lock_page)
 *       mapping->i_mmap_lock
 *         anon_vma->lock
 *           mm->page_table_lock or pte_lock
 *             zone->lru_lock (in mark_page_accessed, isolate_lru_page)
 *             swap_lock (in swap_duplicate, swap_info_get)
 *               mmlist_lock (in mmput, drain_mmlist and others)
 *               mapping->private_lock (in __set_page_dirty_buffers)
 *               inode_lock (in set_page_dirty's __mark_inode_dirty)
 *                 sb_lock (within inode_lock in fs/fs-writeback.c)
 *                 mapping->tree_lock (widely used, in set_page_dirty,
 *                           in arch-dependent flush_dcache_mmap_lock,
 *                           within inode_lock in __sync_single_inode)
 */
```

i. (4 pts) Was the intention of the developer when writing this comment related to deadlock recovery, deadlock avoidance, or deadlock prevention? Define the term you choose as an answer.

The intention is deadlock prevention. Deadlock prevention removes one of the four necessary conditions for deadlock to occur.

Note that the question lists deadlock avoidance and prevention as two of three possible answer alternatives. This was a multiple choice question with 3 choices. I assign partial credit if you provided a correct definition.

ii. (4 pts) Depending on your answer in i., explain how this comment can help recover from deadlocks, avoid deadlocks, or prevent deadlocks. Be specific.

Specifying a locking order avoids circularities in the resource allocation graph, removing one of the necessary conditions for deadlock.

c) (8 pts) Generally, a better method to convey information about the assumptions that underlie a correct program is the use of assertions (e.g. ASSERT()). Could the information that is conveyed in the comment given in part b) of this question also have been expressed as assertions? If so, sketch how by example. If not, justify why not. Be precise.

*In theory, yes. In **all** places where a lock is acquired, place an assertion that the current thread does not already hold **any** of the locks that are lower in the locking order. In practice, this is less practical because a) there may be many objects, b) not all objects that are sources of deadlock have a notion of ownership, or implement ownership, and c) because not all objects may be easily accessible in the current scope.*

Example:

```
/* [ would have to place assertions that none of the locks that could be
acquired following inode->i_mutex are already held, for instance: ] */
assert(!lock_held_by_current_thread(inode->i_alloc_sem));
/* and many more, which likely makes this approach impractical. */
lock_acquire (inode->i_mutex);
```

Note that locking order does not mean that all lower-order locks must be acquired before a higher-order lock is requested. Rather, it means that a lower-order lock cannot be requested once a higher-order lock is held. It would be wrong to assert() you're holding a lower-order lock before acquiring a higher-order lock. For full credit, you needed to state that clearly.
Some of you pointed out that halting the system due to a potential deadlock is harsh; but that's not what the question asked. Some of you pointed out that you can't know who holds a lock – while this can be true for certain types of lock implementations, it's certainly not always true – otherwise, you couldn't have implemented project 1.

4. System Calls and Protection (36 pts)

a) (8 pts) Virtual machine monitors (VMM) such as VMware ESX are a software layer that can multiplex the simultaneous execution of multiple operating systems, along with the user processes running inside them, on a single physical machine. VMM virtualize physical hardware components, such as the CPU, memory, or I/O devices for the guest operating systems running inside them, very much like ordinary operating systems virtualize such resources for their user processes.
To provide isolation between different virtual machines, a technique called deprivileging is being used in which the kernel code of each guest operating system is run in user mode.
As a specific example, assume that the guest OS contains an idle thread which it schedules when there are no other process to run. The idle thread executes the x86 "hlt" instruction in a loop, which suspends the CPU in a low-power state until an interrupt arrives.

i. (4 pts) What would happen, specifically, if a guest OS's idle thread were run in user mode? Explain why!

The hlt instruction is a privileged instruction because ordinary user process must not allow a machine to be halted. If such a privileged instruction is executed while the CPU is in user mode, a trap or exception will occur. (I demoed this in class.)

ii. (4 pts) How could the underlying virtual machine monitor make it so that the guest OS that has scheduled an idle thread would still function as designed (that is, the guest must not be aware that it is running deprivileged inside a virtual machine)?
Sketch what the VMM would have to do! Consider also how the guest OS regains control eventually.

When the trap occurs, the hypervisor will realize that this guest machine is idle, and it will stop assigning the real CPU to it; it may then move on to schedule another guest machine. When an interrupt occurs for this guest machine, the hypervisor will resume execution in that virtual machine after the hlt instruction, then deliver the interrupt. To the guest, it will appear that the hlt instruction had the same effect as on real hardware.

For full credit, you needed to realize that fault resumption is being used here, as discussed in lecture. The hypervisor in its trap handler can emulate the effect of this instruction. It uses a fault resumption technique to continue execution – very much like, for instance, Pintos can use fault resumption to resume the execution of a process after a page fault.

b) (20 pts, 4 pts each.) In Unix-like OS, processes are usually terminated when they access virtual memory addresses that are either not mapped or that are mapped as being accessible in kernel-mode only. If these processes are started from a shell, you then see a message that says “Segmentation Fault.” Typically, such invalid memory accesses are the results of a programming error.
Based on your knowledge of a typical runtime environment, state, for each of the following erroneous programs, under which conditions they would be terminated with a segmentation fault! State your assumptions if necessary. You may use C-style short-hand notation.

		Would be terminated if:

i)	`int overflow1(void) {`
	`  int buf[10];`
	`  return buf[20000];`
	`}`
		*buf + 20000 * sizeof(int) > PHYS_BASE (or upper end of stack segment)*

ii)	`int underflow1(void) {`
	`  int buf[10];`
	`  return *(buf - 20000);`
	`}`
		*buf – 20000 * sizeof(int) < Current lower end of stack segment*

iii)	`int overflow2(void) {`
	`  int *b = malloc(80);`
	`  return b[20000];`
	`}`
		*b + 20000 * sizeof(int) > Current upper end of heap segment*

iv)	`static int buf1[20];`
	`int overflow3(void) {`
	`  return buf1[20000];`
	`}`
		*b + 20000 * sizeof(int) > Current upper end of heap segment (assuming data segments and heap segments are adjacent.)*

v)	`int arbitrary(void) {`
	`  return *(int*)0x9000000;`
	`}`
		0x9000000 is not in any valid segment.

Just mentioning that a fault occurs if the address is question is not mapped yielded partial credit for parts i) through iv) (the question asked to relate the condition to your knowledge of a typical runtime environment, which consists of stack, heap, data, and code segments.

c) (8 pts) In a previous offering of CS 3204, a student criticized Pintos for not being a real OS because unlike "real OS," Pintos does not pass system call arguments in registers. By comparison, Linux passes arguments to system calls in registers: %eax is the syscall number on entry; %ebx, %ecx, %edx, %esi, %edi and %ebp are the six generic registers used as parameters 0 to 5. The return value is also transferred in register %eax.

Sketch what your syscall handler function would look like if Pintos used this method of passing system call parameters!

Assuming that at most 3 parameters are used, the system call handler could look as simple as:

```
static void
syscall_handler (struct intr_frame *f)
{
      if ((unsigned)f->eax >= SYS_CALLNO_MAX)
            thread_exit();
      f->eax = syscall_table[f->eax](f->ebx, f->ecx, f->edx);
}
```

Note that since the stack is not used, no checking on f->esp needs to be done.