

Midterm I
SOLUTION

October 2nd, 2026

CS162: Operating Systems and Systems Programming

Your Name:	
SID:	
CS162 Login:	
TA Name/Section Time	
Person to Left, Right, Front, and Back of you:	

This is a **closed book** exam. You are allowed 1 page of notes (both sides). You have 170 minutes to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	Possible	Score
1	18	
2	18	
3	22	
4	21	
5	21	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [18 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Keep answers within the box. Also, answers without an explanation *GET NO CREDIT*.

Problem 1a[2pts]: In Linux, when a thread wakes up after a call to `futex()` with `op= FUTEX_WAIT`, the condition that caused `futex()` to be called in the first place is guaranteed to be met.

☐ True ☒ False

Explain: *There are multiple reasons this is false. First, when a thread that was sleeping is awoken, it is merely scheduled to run; other threads may get in and alter the condition before the futex call returns. Second, it is also possible for there to be spurious wakeups to occur, independent of being explicitly awoken by another thread.*

Problem 1b[2pts]: Assume that you read 10 bytes from a large file immediately after opening it.. You could do this with either the high-level or low-level file API (i.e. via `fread()` or `read()` respectively). Both methods of access will copy the file's data the same number of times on the way to your user buffer during this first read.

☐ True ☒ False

Explain: *The high-level API has additional buffering that occurs at user-level. When reading a 10-byte chunk of data, the high-level API will request a full block of data (i.e. 1024 or 4096 bytes) from the low-level API, copy this data into the user-level buffer, then return 10 bytes to the caller.*

Problem 1c[2pts]: Assume that there are two threads running under the same parent process. If thread A calls `malloc` (which succeeds) and communicates the returned address to thread B, thread B can access this memory without causing a segmentation fault.

☒ True ☐ False

Explain: *Both threads are running in the same parent process, so have access to the same address space. Consequently, the fact that malloc succeeded means that the returned address is valid – for both Thread A and Thread B.*

Problem 1d[2pts]: “Hyperthreading” is a mechanism supported by modern processors that allows an individual thread to run twice as fast when there is a sufficient power budget.

☐ True ☒ False

Explain: *Hyperthreading is a mechanism that allows two or more hardware threads to share resources from the same pipelining. Thus, an individual thread will run at the same or slower pace as before, but the advantage is that more than one thread gets to run (with a combined number of instructions/cycle that is greater than before).*

Problem 1e[2pts]: The `dup2()` system call can be used to change the behavior of the `printf()` function.

☒ True ☐ False

Explain: *Since `printf(...)` is really an alias for `fprint(stdout,...)`, we can change the destination of `printf(...)` by using `dup2(...)` to redirect `stdout` to write to a different file descriptor, i.e. if `out` is a file descriptor for an open file, `dup2(out,1);` will redirect `printf(...)` to that file.*

Problem 1f[2pts]: In PintOS using the x86 ABI, after the stack for a new program has been initialized, the stack pointer (ESP) is 16B aligned.

☐ True ☒ False

Explain: *The last argument is 16B aligned, but to make space for the return address, the ESP will never be 16B aligned.*

Problem 1g[2pts]: In Base and Bound address translation, a user program can directly modify the “Base” and “Bound” registers.

☐ True ☒ False

Explain: *Allowing a user program to directly modify the Base and Bound registers would completely defeat the protection provided by the mechanism. These registers can only be modified when the processor is running in supervisor mode, i.e. by kernel code.*

Problem 1h[2pts]: Processes created by fork may access the same physical memory.

☒ True ☐ False

Explain: *Although processes operate in independent virtual address spaces, this constraint does not prevent the OS from mapping virtual addresses to the same physical pages. The result is an ability for different processes to communicate with one another through memory. [We mentioned this in class as one of many advantages of page-level address translation.]*

Problem 1i[2pts]: When using the Monitor synchronization pattern, a programmer must remember to release the lock before going to sleep (waiting) on a condition variable.

☐ True ☒ False

Explain: *In fact, exactly the opposite is true: you must still hold the lock when you wait on a condition variable. The reason for this rule is that you need to check the sleep condition and potentially sleep **within** a critical section (holding a lock) or you might end up with a race condition with other threads.*

Problem 2: Multiple Choice [18pts]

For the multiple choice, below, selecting an item when it is inappropriate or not selecting an item when it is appropriate will result in loss of points.

Problem 2a[2pts]: Suppose we were implementing a shell and wanted to redirect a program's output to a file. What should we do? (*choose one*):

- A: ☐ Create a child process via `fork()` to execute the program, then replace the `stdin` file descriptor of the child, using `dup2()`.
- B: ☒ Create a child process via `fork()` to execute the program, then replace the `stdout` file descriptor of the child, using `dup2()`.
- C: ☐ Create a child process via `fork()` to execute the program, then replace the `stdout` file descriptor of the parent, using `dup2()`.
- D: ☐ Create two child processes, one to execute the program and one to handle file output. These two processes should be connected by a pipe.
- E: ☐ None of the above.

A: *FALSE. Replacing the `stdin` file descriptor does nothing to redirect the output.*

B: *TRUE. This option runs the program in a child process (for security), and redirects the output of the child to a different file descriptor (the open file).*

C: *FALSE. Redirecting `stdout` in the parent process does nothing to redirect output of the program, which is running in the child process.*

D: *FALSE. This option is unnecessarily complex and clearly not the right solution – when compared with option (B).*

E: *FALSE. Obviously.*

Problem 2b[2pts]: Select all true statements about the x86 instruction set (*choose all that apply*):

- A: ☒ “`pushl eax`” subtracts 4 from `esp`
- B: ☐ “`popl eax`” subtracts 4 from `esp`
- C: ☒ Using proper calling convention, a ‘caller’ pushes the arguments to a ‘callee’ in reverse order on the stack.
- D: ☐ Using proper calling convention, we store the base pointer (`ebp`) of the ‘caller’ function frame in a saved register like `eax`, so that we can restore it after the ‘callee’ has completed.
- E: ☐ None of the above.

A: *TRUE. Since the stack grows downward in memory, pushing on the stack decreases the stack pointer. The “l” in `pushl` means “long”, which in Intel-speak is a 32-bit integer.*

B: *FALSE. Since the stack grows downward in memory, popping from the stack increases the stack pointer.*

C: *TRUE. Refer to Project0 / early discussions about calling sequence.*

D: *FALSE. The caller's base pointer is stored on the stack, not in `eax`. `eax` is used to store the return value of the callee and is thus not a saved register.*

E: *FALSE. Obviously.*

Problem 2c[2pts]: Select all of the following that are true about semaphores (*choose all that apply*):

- A: ☒ Semaphores can be allocated on the stack.
 B: ☐ A semaphore's value can become negative if multiple threads simultaneously execute `p()`.
 C: ☒ Semaphores can be used to implement condition variables.
 D: ☒ You can implement a semaphore in user-level code using compare-and-swap and `futex()`.
 E: ☐ None of the above.

A: *TRUE. Semaphores are stored in memory, so can be allocated as local variables in a function (i.e. on the stack).*

B: *FALSE. By definition, semaphores must never become negative. If some implementation of semaphores allows them to become negative, then the implementation of `p()` is buggy*

C: *TRUE. We indicated in class that this was possible. The trick is that for each condition variable, you need a semaphore and an integer. The semaphore will be used for sleeping/signaling, and the integer will count the number of threads potentially waiting. To implement `condition_wait()`, you (1) increment the count, (2) release the lock, (3) do `p()` on semaphore, while will wait (4) reacquire lock, and (5) decrement the count. This way, you can implement `signal()` by checking for `count > 0` before calling `v()`.*

D: *TRUE. The key is the `p()` operation:*

```
void p(int* mysemi) {
    int value;
    do {
        value = *mysemi;
        while (value == 0) {
            futex(mysemi, FUTEX_WAIT, 0);
            value = *mysemi;
        }
    } while (!CAS(mysemi, value, value-1));
}
```

E: *FALSE. Obviously.*

Problem 2d[2pts]: Which of the following are true about interrupt handling in PintOS?

- A: ☒ The x86 hardware saves the values for the user's flags register and program counter before jumping to the interrupt handler.
 B: ☐ After interrupting a user thread, interrupts create a new kernel thread to run the interrupt handler.
 C: ☒ Disabling interrupts is guaranteed to implement mutual exclusion.
 D: ☒ Lower priority interrupts may be put on hold by a higher priority interrupt.
 E: ☒ All of the hardware interrupts (from the hardware interrupt vector table) feed into the same general prolog code before being dispatched to interrupt-specific PintOS handlers.

A: *TRUE. See discussion from class.*

B: *FALSE. The interrupt handler runs on the stack of the thread that was interrupted.*

C: *ANY. Disabling of interrupts prevents the scheduler from switching threads on a single processor. If there are other hardware threads, they will not be prevented from executing by disabling interrupts. However, PintOS is a single-core OS. Thus, we accepted both true and false.*

D: *TRUE. The interrupt controller has a masking mechanism and built-in priority encoder, that can be set up to disable lower-priority interrupts while allowing higher-priority interrupts to interrupt the processor.*

E: *TRUE. See the discussion of the code in Lecture 10.*

Problem 2e[2pts]: Consider the following program (assuming that we are running this code on Unix and that `fork()` does not fail):

```
int mystery(int arg) {
    if (arg <= 0)
        return 0;
    else if (arg == 1)
        return 1;
    else {
        // Assume that fork() does not fail.
        if (!fork())
            exit(mystery(arg-2));
        else {
            int childcode;
            wait(&childcode);
            return (mystery(arg-1) + childcode);
        }
    }
}
```

Which of the following statements are true (here “child processes” refer to all processes created when running the function—children, grandchildren, etc)? (*choose all that apply*):

- A: ☒ The return values of `mystery(n)` from $n = 2$ to 7 are {1, 2, 3, 5, 8, 13}
- B: ☐ If `numchild(n)` is the number child processes created by `mystery(n)`, then we know that `numchild(n) <= n+2`
- C: ☒ If `numchild(n)` is the number child processes created by `mystery(n)`, then we know that for $n > 1$, `numchild(n) = 1 + numchild(n-2) + numchild(n-1)`.
- D: ☐ Ignoring the calling process, this function can have up to `numchild(n)` of its child processes alive (running or waiting) at the same time.
- E: ☒ Ignoring the calling process, this function can have up to `floor(n/2)` of its child processes alive (running or waiting) at the same time.

The final else clause computes (mystery(arg-2)+mystery(arg-1)). We know this because the parent launches a child process to compute mystery(arg-2). The parent waits for the result, then computes mystery(arg-1) and adds them together. Consequently, mystery(n)=mystery(n-2)+mystery(n-1), with mystery(0)=0, mystery(1)=1. In other words, mystery(n) => fibonacci(n)

- A: TRUE. *mystery(2) = mystery(0)+mystery(1) = 1, mystery(3) = mystery(2)+mystery(1) = 2, remainder from mystery(n) = mystery(n-2)+mystery(n-1)*
- B: FALSE. *See result in (C). This grows faster than linear. For instance: numchild(6)=12 > 6+2*
- C: TRUE. *With $n \geq 2$, mystery(n) enters the final else clause and performs one fork() and also calls mystery(n-2) and mystery(n-1). Thus, numchild(n) = 1 + numchild(n-2) + numchild(n-1)*
- D: FALSE. *Assume we compute mystery(n). Since the parent in the else clause executes wait() before computing mystery(n-1), the child process and all of the grandchild processes launched by the child's computation of mystery(n-2) will have exited before the parent computes mystery(n-1). So the number of simultaneous children must be < numchild(n)*
- E: TRUE. *Let cmax(n) be the maximum number of launched progeny (children/grandchildren/etc) running at the same time. We know: cmax(0)=0, cmax(1)=0, since no fork for $n < 2$. For $n \geq 2$, $cmax(n) = \max(1 + cmax(n-2), cmax(n-1))$, because the parent waits for the child to complete so, max of child or parent. Proof by induction to show that $cmax(n) = \text{floor}(n/2)$.*

Problem 2f[2pts]: Which of the following statements about files are true? (*choose all that apply*):

- A: ☒ The same file descriptor number can correspond to different files for different processes.
- B: ☐ The same file descriptor number can correspond to different files for different threads in the same process.
- C: ☒ Files can be stored on a variety of different physical devices because each device driver exports an array of common interface functions (such as `open`, `close`, `read`, and `write`). A pointer to this `fop` array is stored in the file description.
- D: ☒ File descriptions keep track of the file offset.
- E: ☒ An `lseek()` within one process may be able to affect the writing position for another process.

- A: *TRUE. Every process has its own unique file descriptor table mapping file descriptors to file descriptions. Thus, each process can open different files and make this statement true.*
- B: *FALSE. Because the file descriptor table is a per-process table, all threads must share the same mapping of file descriptors to files.*
- C: *TRUE. We discussed the `fop` table in class when discussing device drivers.*
- D: *TRUE. We discussed the file description contents in class when discussing device drivers.*
- E: *TRUE. During `fork`, the child process receives a copy of the parent's file descriptor table. This means that they will share file descriptions for any files open at the time of `fork`.*

Problem 2g[2pts]:

Select all true statements about monitors (*choose all that apply*):

- A: ☐ A monitor cannot have more than one condition variable for each lock.
- B: ☒ The `cond_wait()` function (internally) releases the monitor lock and puts the thread to sleep until the `cond_signal()` or `cond_broadcast()` function has been called.
- C: ☒ In a system with Mesa semantics, there is no guarantee that a signaled condition will still be true when the signaled (and awoken) thread gets around to checking the condition.
- D: ☐ Because the monitor programming pattern is not very powerful, very few programming environments provide support for condition variables.
- E: ☐ None of the above.

- A: *FALSE. In class and discussion, we gave several examples of monitor patterns with one lock and more than one condition variable (e.g. readers/writers).*
- B: *TRUE. This is how the monitor pattern (always hold lock while waiting) can avoid deadlock – under the covers there is no sleeping while holding the lock.*
- C: *TRUE. When a thread is signaled, it simply gets put on the ready queue. By the time it reacquires the lock and gets to run, another thread could easily have performed an action that changes the condition.*
- D: *FALSE. The monitor pattern is extremely powerful. It is widely recommended over use of semaphores, and, as shown in class, there is support for monitors in many languages (pthreads for C, directly in Java, etc).*
- E: *FALSE. Obviously.*

Problem 2h[2pts]: Suppose we wanted to redirect the output of one program, as input to another program (piping). The parent process P1 creates a pipe, and then forks to create P2. (*choose all that apply*):

- A: ☒ The pipes in both P1 and P2 have the same file descriptor numbers and point to the same file descriptions.
- B: ☐ P1 can safely keep its pipe's write end open while waiting for P2 to exit, while P2 closes its own write end and reads until EOF.
- C: ☐ If P1 closes the read end of pipe and P2 closes the write end, then P2 may pass information unidirectionally to P1.
- D: ☒ It is acceptable for neither P1 nor P2 to close any descriptors.
- E: ☐ None of the above.

A: *TRUE. This is true because fork() duplicates the file descriptor table, which points at file descriptions.*

B: *FALSE. This scenario results in a deadlock, since P1 won't finish/exit until P2 does, which means that the pipe's write end stays open. Further, P2 won't receive an EOF (and thus exit) until P1 closes its write end.*

C: *FALSE. Since P2 has closed the write end of the pipe, it cannot send (write) information through the pipe to P1.*

D: *TRUE. There is nothing that forces P1 or P2 to close any file descriptors: you might imagine that some application needed bidirectional communication.*

E: *FALSE. Obviously.*

Problem 2i[2pts]: In Pintos, every user-level thread has both a user-level stack and a kernel-level stack. The kernel stack is sometimes called a "kernel thread" because it is manipulated by the scheduler when multiplexing the CPU. What is true about this arrangement (*choose all that apply*):

- A: ☐ The physical memory for the kernel stack must be at a lower address than the physical memory for the user stack so that the stacks can be considered together (treating a system call like a procedure call into the kernel).
- B: ☒ The kernel gains safety because it does not have to rely on the correctness of the user's stack pointer register or validity of user's stack memory for correct behavior.
- C: ☐ While the thread is executing within the kernel, it has access to an arbitrarily large stack, allowing deeply recursive handling of the scheduling algorithm.
- D: ☒ Threads which run exclusively within the kernel (and have no associated user-level stack) can be scheduled by the same scheduler as regular (user) threads.
- E: ☒ When the user-thread makes a system call, the thread can be blocked at any time (and at any call depth within the kernel) by saving current state on the kernel stack and/or TCB, putting the kernel stack on a wait queue.

A: *FALSE. The user and kernel stacks are separate and disconnected from one another. Further, the addresses of the stacks are virtual and mapped to arbitrary physical pages.*

B: *TRUE. Entry into the kernel (e.g. during system calls, interrupts, traps, etc) involves an automatic replacement of the user stack pointer with a pre-allocated kernel stack.*

C: *FALSE. Kernel stacks are typically limited in size, since kernel routines are not deeply recursive. As a case in point, Pintos kernel stacks are <4KB. Linux kernel stacks < 8KB.*

D: *TRUE. Normal user threads must enter the kernel and save their information on their paired kernel stack before they can be scheduled. Kernel-only threads operate exclusively on a kernel stack. Consequently, the scheduler performs a context switch by dealing only with kernel stacks—thus, it views all threads as identical.*

E: *TRUE. Any time a user thread enters the kernel, it will save the user-level information on the kernel stack and can then continue using the kernel stack. Thus, the kernel stack contains all the information necessary to continue the user thread. Since the context-switch process operates entirely between kernel stacks, it can suspend a thread simply by putting its kernel stack on a wait queue.*

[This page intentionally left blank]

Problem 3: Saving Us From The Agents [22pts]

Coding agents now routinely run shell commands on their host machines. This brings up an important question of how much trust should we give the agent! There has been a lot of hype about AI Agents “escaping” from their environments and hacking into other companies, so the answer is that we probably shouldn’t trust them at all.

So far there have been 4 solutions the AI trust issue:

1. yolo mode: trust the agent is smart enough to not do anything bad
2. approve for me: ask the agent if what the agent is doing is bad
3. manual approval: trust the user knows what is dangerous
4. sandboxing: use the operating system itself to prevent dangerous behavior

This last approach is the one that has been tried in various ways and to various levels of success. Let’s take a step toward sandboxing. One way we can build cheap sandboxes is ensure untrusted commands can not write or delete files. Linux provides this mechanism with **seccomp** (secure computing) and we will add a simple version to PintOS. **seccomp**’s goal is to filter system calls allowing you multiple complex filters that run before any future system call executes.

In this problem, the essential mechanism is a *context*, which is a per-process list of filtering rules that are used during handling of system calls. Once a context has been activated, then every system call is matched against the filtering rules. If there are any matches, then the corresponding action is performed. If there are no matches, then the system call will be processed as the default.

On the next page we have given an example implementation of a `safe_run_tool_call()` which runs a shell command with read-only access to the filesystem. You **do not** need to understand all of the syntax, only note how these three library calls are used:

seccomp_init(default): Create a **context** with a **default** action to perform when the process attempts to execute a syscall that does not match any rules. For instance, when **default** is **SCMP_ACT_ALLOW**, the default action is to allow any syscall that is unmatched by a rule. Conversely, when default is **SCMP_ACT_KILL**, the default action is to kill the process if it attempts to perform a syscall that isn’t matched by a filtering rule.

seccomp_rule_add(ctx, action, syscall, n, ...): Add a **rule** to a **context**. This rule says what action to take when the program makes a syscall that matches the pattern (including, at least, the syscall number and zero or more matching arguments). In the code below, for instance, the action is always **SCMP_ACT_KILL**, which means “kill the process”. Other options exist such as **SCMP_ACT_ALLOW** (let the syscall run). The argument **n** indicates the number of the syscall’s arguments the rule also inspects, and the argument tests follow.

seccomp_load(ctx): activates the context on the calling process. A loaded filter cannot be removed or modified. Before load is called, the filters are not active.

Note that we have identified several “LOCATIONS” which will apply to questions to follow.

Sample use of our new filtering contexts for Problem 3. We are going to run the command string for the agent, but we are going to put restrictions onto the system call behavior before we exec the shell. Assume that `fork()`, `exec()`, and file descriptors work as in Unix (not Pintos).

```
1.  /*
2.   * Run a shell command from the agent, but prevent it from deleting
3.   * files or directories, and force all files to be opened read-only.
4.   */
5.  int safe_run_tool_call(const char *cmd, const char *out) {
6.      // LOCATION A
7.      pid_t pid = fork();
8.      if (pid == 0) {
9.          // LOCATION B
10.         // allow everything by default
11.         seccomp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);
12.
13.         // Prevent deleting of files and directories
14.         // note: unlink/unlinkat is Linux's remove syscall
15.         seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(unlink), 0);
16.         seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(unlinkat), 0);
17.         seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(rmdir), 0);
18.
19.         // Prevent opening of files in WRITE or READ/WRITE modes
20.         seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(openat), 1,
21.                           SCMP_A2(SCMP_CMP_MASKED_EQ, O_ACCMODE, O_WRONLY));
22.         seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(openat), 1,
23.                           SCMP_A2(SCMP_CMP_MASKED_EQ, O_ACCMODE, O_RDWR));
24.
25.         // activate the context with its filters
26.         seccomp_load(ctx);
27.
28.         // LOCATION C
29.         // run the command now that these filters have been installed
30.         execl("/bin/sh", "sh", "-c", cmd, (char *) NULL);
31.         // LOCATION D
32.     }
33.     // LOCATION E
34.     int status;
35.     waitpid(pid, &status, 0);
36.     return status;
37. }
```

Problem 3a[3pts]: How can we make only the agent's standard output be redirected to the file whose file name is given by the input parameter 'out'? Indicate the location name (LOCATION A–LOCATION E) and explain in one sentence why this location is better than the other options. Further, complete the missing lines of code (you do not have to use all lines and may use STDIN_FILENO and STDOUT_FILENO as predefined macros):

Location (pick one): ☐A ☒B ☐C ☐D ☐E

Answer: *The other locations don't work because they either redirect too much (A), run afoul of the new syscall restrictions (C), or happen too late (D, E).*

1. `int fd = open(out, D_WRONLY | O_CREAT, 0644);`
2. `if (fd < 0 || dup2(fd, STDOUT_FILENO) < 0)`
3. `exit(-1);`

Problem 3b[2pts]: Should filters be passed down to children when fork is called? Justify in one or two sentences.

Answer (pick one): ☒Yes ☐No

Explain: *Otherwise, filters could be bypassed by simply calling fork().*

Problem 3c[2pts]: Under the Unix implementation of exec(), why do the filters stay in place after the command is run?

Answer: *exec() reuses the same process. Consequently, the filters stay in place because they are part of the kernel state of the process, not the program.*

Problem 3d[1pts]: Why should we not add a syscall to remove filters?

Answer: *Because a malicious program would simply remove them.*

Now we will implement seccomp in Pintos, modeled on the Linux API above.

- A filter matches some syscalls and carries a verdict: allow or deny. The implementation details of filters are provided by a filter library.
- A context is an ordered (FIFO) list of filters, a default verdict, and an active flag. A context's verdict for a syscall is the verdict of the first filter that matches, or the context's default if no filter matches.
- A process has a list of contexts. Inactive contexts are ignored. A syscall runs only if every active context allows it; otherwise the process is killed.

The kernel provided data structures and functions we will be using are as follows:

```

struct process {
    uint32_t* pagedir;
    char process_name[16];
    struct thread* main_thread;
    /* ... other fields ... */
    struct list contexts; /* All contexts, active or not */
    int next_ctx_id;      /* Next id to hand out. */
};

// State of context/storage for filters
struct context {
    struct list_elem elem;
    int id; /* Handle given to user. */
    bool active; /* false until seccomp_load. */
    bool default_allow; /* Verdict if nothing matches. */
    struct list filters; /* In order, first match wins. */
};

// Find context with this id, NULL if none (searches list of contexts).
struct context* find_context(struct process* pcb, int id);

// Prints "<name>: exit(-1)", calls process_exit(), never returns. //
void kill_process(void) NO_RETURN;

```

We will be using a third-party **filter** library with the following provided structures and functions:

```

// Most of this structure managed by 3rd party library
struct filter {
    struct list_elem elem;
    bool allow; /* Verdict if this matches. */
    /* ... rest of struct, assume no pointers/filter can be freed with free */
};

// True if FILTER matches this syscall. ARGS points at the syscall's
//first argument on the user stack. */
bool filter_matches(struct filter* filter, int syscall_number, void* args);

// Validates the user-space filter at UFILTER and returns a newly
// malloc()'d struct filter, or NULL if bad. */
struct filter* filter_from_user(const void* ufilter);

```

We will start you off with the kernel-side handler for **seccomp_load()**:

```
static void sys_seccomp_load(struct intr_frame* f, uint32_t* args) {
    struct context* ctx = find_context(thread_current()->pcb, args[1]);
    if (ctx == NULL) {
        f->eax = false;
    } else {
        ctx->active = true;
        f->eax = true;
    }
}
```

Problem 3e[3pts]: Implement the kernel-side handler for **seccomp_init()**. Create a new, inactive context with no filters, add it to the process, and return its id. Assume that functionality for copying contexts, wherever needed, and freeing them in **process_exit()** is already implemented for you. Assume also that **malloc()** never fails, all user pointers are valid (i.e. no need to validate **f->esp**), and every process has one thread. Use only provided lines and semicolons:

```
static void sys_seccomp_init(struct intr_frame* f, uint32_t* args) {
    struct process* pcb = thread_current()->pcb;
    struct context* ctx = malloc(sizeof *ctx);

    ctx->id = pcb->next_ctx_id+;
    ctx->active = false;
    ctx->default_allow = args[1];
    List_init(&ctx->filters);
    List_push_back(&pcb->contexts, &ctx->elem);
    f->eax = ctx->id;
}
```

Problem 3f[3pts]: Implement the kernel-side handler for **seccomp_rule_add(ctx, filter)**. Return false in **f->eax** if the context does not exist or is already active; kill the process if filter is invalid; otherwise append filter and return true. Use only provided lines and semicolons:

```
static void sys_seccomp_rule_add(struct intr_frame* f, uint32_t* args) {
    struct context* ctx =
        find_context(thread_current()->pcb, args[1]);

    if (ctx == NULL || ctx->active) {
        f->eax = false;
        return;
    }
    struct filter* flt = filter_from_user((const void*)args[2]);
    if (flt == NULL)
        kill_process();

    List_push_back(&ctx->filters, &flt->elem);
    f->eax = true;
}
```

Problem 3g[8pts]: Finally, fill in `syscall_handler()` so that it kills the process unless every **active** context allows the syscall (first matching filter wins, else the default). Use only provided lines and semicolons:

```
static void syscall_handler(struct intr_frame* f) {
    uint32_t* args = ((uint32_t*)f->esp);
    struct process* pcb = thread_current()->pcb;
    /* Check the syscall against each of the process's contexts. */
    struct list_elem* e;

    for ( e = list_begin(&pcb->contexts)
          e != list_end(&pcb->contexts)
          e = list_next(e) ) {
        struct context* ctx = list_entry(e, struct context, elem);
        if ( !ctx->active
              continue;
            /* This context's verdict. */

            bool allow = ctx->default_allow;
            struct list_elem* fe;

            for ( fe = list_begin(&ctx->filters)
                  fe != list_end(&ctx->filters)
                  fe = list_next(fe) ) {
                struct filter* flt = list_entry(fe, struct filter, elem);
                if ( filter_matches(flt, args[0], &args[1]) ) {
                    allow = flt->allow;
                    break;
                }
            }
            /* Enforce the verdict. */

            if ( !allow
                  kill_process();
            )
        }
    }
    /* ... rest of syscall_handler (not shown) ... */
}
```

Problem 4: Short Answer Potpourri [21pts]

For the following questions, provide a concise answer of NO MORE THAN 2 SENTENCES per sub-question (or per question mark), unless instructed otherwise.

Problem 4a[2pts]: In general, busy-waiting for a lock to be released is an incredible waste of resources. Can you name and explain a circumstance in which waiting for a lock to be freed by spin-waiting using `test&set` might be better than sleeping on the lock?

In general, using a vanilla `test&set` lock, complete with busy-waiting, will be better whenever the expected time wasted busy-waiting is less than the time to enter the kernel and put the thread to sleep on some queue; this is particularly true on a multicore in which cores can be dedicated to particular threads so that one busy-waiting thread does not impact other threads. The primary case we discussed in class is when a multicore processor is used for an application that utilizes multiple threads, each of which is allocated a unique core. Such applications often utilize a barrier-style computation in which each thread computes some piece of a puzzle, then waits for everyone else to finish, after which all the waiting threads are quickly released.

Problem 4b[2pts]: Why is it possible for a web browser (such as Firefox) to have 2 different tabs opened to the same website (at the same remote IP address and port) without mixing up content directed at each tab?

Each connection is distinguished (made unique) by a 5-tuple of information: Source IP, Source Port, Destination IP, Destination Port, and protocol. So, even though the two tabs share Source IP, Destination IP, Destination Port, and protocol, they can be distinguished by their Source Port.

Problem 4c[3pts]: The linked list nodes you have seen before (in 61B) stored each value alongside the previous and next pointers. However, this is not the case in PintOS. Below are the definition of the `list_elem` and `list` structures for PintOS:

```
struct list_elem
{
    struct list_elem *prev;
    struct list_elem *next;
};
struct list
{
    struct list_elem head;
    struct list_elem tail;
};
```

Provide justification for why PintOS lists are implemented this way. In an actual list built using these definitions, where are the related values stored?

Lists are implemented this way to provide a type independent, object-oriented (or “generic”) list mechanism in C (which is decidedly not object-oriented). Lists and list elements can be manipulated by functions that do not need to know anything about the nodes that are being linked in the list. In an actual list built this way, the “`list_elem`” structure is embedded in an enclosing structure, which actually contains the values.

Problem 4d[3pts]: Name three ways in which the processor can transition from user mode to kernel mode. Can the user execute arbitrary code after the transition?

The processor transitions from user mode to kernel mode when (1) the user executes a system call, (2) the processor encounters an synchronous exception such as divide by zero or page fault, (3) the processor responds to an interrupt. The user cannot execute arbitrary code because entry into the kernel is through controlled entry points (not under control of the user).

Problem 4e[3pts]: What is the difference between Mesa and Hoare scheduling for monitors? Include passing of locks between signaler and signalee, scheduling of CPU resources, and impact on the programmer.

With Hoare scheduling, a signal() operation from one thread immediately wakes up a sleeping thread, hands the lock to the sleeping thread, and starts the sleeping thread executing; control returns to the signaling thread after the signaled thread attempts to release the lock (which is then handed back to the signaling thread). With Mesa scheduling, the signaling thread simply places the signaled thread on the run queue and continues executing with the lock.

The practical consequence is that Mesa-scheduled monitors require programmers to recheck conditions after waking (typically with a “while” loop):

```
while (condition not satisfied)
    condition.wait();
```

With Hoare-scheduled monitors, the “while” statement can often be replaced with an “if” statement.

Problem 4f[2pts]: Provide 2 possible console outputs that can result from running the main function below. Assume all calls to pthread_create/pthread_join succeed.

```
void subtask(void* args_) {
    int idx = (int) args_;
    printf("%d ", idx);
    return;
}
int main(void) {
    pthread_t threads[6];
    for (int i = 1; i <= 6; i++) {
        pthread_create(&threads[i - 1], NULL, subtask, (void*) i);
        if (i % 3 == 0) {
            for (int j = i - 2; j <= i; j++) {
                pthread_join(threads[j - 1], NULL);
            }
        }
    }
    return 0;
}
```

Any ordering of 1...6 where each “chunk” of three successive integers (1-3, 4-6) are in order (e.g. 6 can come before 5, but 6 cannot come before 3).

Problem 4g[3pts]: How does the OS prevent malicious user programs from executing arbitrary code in kernel mode or from accessing other programs' memory? Name 3 mechanisms (limit 1 sentence per mechanism):

There are a number of possible answers here. For instance: (1) Each user process has a separate address space that is mapped to unique physical memory, thereby making it impossible for a malicious user program from reading or writing to memory from the kernel or other processes. (2) System calls into the kernel go through an explicit system call vector, so that all user-initiated entry to the kernel goes through preauthorized entry points that check all arguments for validity. (3) Dual-mode execution (in which the processor has a different hardware mode for system vs user mode means that the OS can protect the virtual memory mappings from modification in user mode. (4) file system protections enforced by the kernel prevent the user from modifying the kernel binary or SUID root programs and thereby gaining unauthorized access to user or kernel data.

Problem 4h[3pts]: Wilson and Zhuowen both want to use a shared TV to watch a show; Wilson wants to watch House of the Dragon, while Zhuowen wants to watch The Rings of Power. To decide which show to watch, they have decided on the following protocols for each to follow (If statements act like if statements in C, so they execute the following line if the condition is true).

Protocol A (Wilson) {

1. Leave a note on the TV saying "Wilson was here".
2. Wait until there is no note on the TV saying "Zhuowen was here".
3. If there is no note on the TV saying "Rings":
4. Leave a note saying "HOTD" on the TV.
5. Remove the "Wilson was here" note.

}

Protocol B (Zhuowen) {

1. Leave a note on the TV saying "Zhuowen was here";
2. If there is no note saying "Wilson was here":
3. If there is no note saying "HOTD":
4. Leave a note saying "Rings";
5. Remove the "Zhuowen was here" note;

}

Assume that only one person can take an action at a time (i.e., they can't both leave/remove notes at the same time). Provide a brief explanation why once both Wilson and Zhuowen have run through their protocols in any order, there will be exactly one note on the TV saying either "HOTD" or "Rings". Limit your explanation to 4 sentences (longer answers may not be graded).

Hint: Split into two cases. What happens if Line A1 happens before Line B2?

Split into two cases:

Case 1: Line A1 happens before Line B2 → Zhuowen cannot leave a note saying "Rings", so Wilson will leave a note saying "HOTD".

Case 2: Line A1 happens after Line B2 → Zhuowen will leave a note saying "Rings" and Wilson will wait until B has completed, so Wilson will not leave a note saying "HOTD".

[This page intentionally left blank]

Problem 5: Synchronization and the RCU primitive [21pts]

In class, we discussed a number of *atomic* hardware primitives that are available on modern architectures. In particular, we discussed “test and set” (TSET), SWAP, and “compare and swap” (CAS). They can be defined as follows using c-style function syntax. Assume that the body of each of these functions will operate as an atomic unit, i.e. can’t be interleaved or interrupted.

Test and Set (TSET)	Atomic Swap (SWAP)	Compare and Swap (CAS)
<pre>TSET(int* addr){ // atomic group int result = *addr; *addr = 1; return result; }</pre>	<pre>SWAP(int* addr, int expr) { // atomic group int result = *addr; *addr = expr; return (result); }</pre>	<pre>CAS(int* addr, int expr1, int expr2) { // atomic group if (*addr == expr1) { *addr = expr2; return true; } else return false; }</pre>

Both TSET and SWAP return values (from memory), whereas CAS returns either true or false. As a canonical use of atomic operations, TSET could be used to implement a spin-lock acquire as follows:

```
int lock = 0; // lock is initially free

// Simple implementation of lock_acquire(&lock)
while (TSET(&lock));
```

CAS is general enough as an atomic operation that it can be used to implement both TSET and SWAP. For instance, consider the following implementation of TSET with CAS:

```
TSET(int *addr) {
    int temp;
    do {
        temp = *addr;
    } while (!CAS(addr,temp,1));
    return temp;
}
```

Problem 5a[3pts]: What is the problem with the above implementation of lock_acquire() that uses TSET()? Provide a better implementation using futex() by filling in the lines below:

Problem: *The issue with the simple TSET version of lock_acquire() is that it busy-waits when it cannot acquire right away.*

```
void futex_lock_acquire(int* lock) {
    while (TSET(lock)) {
        futex(lock, FUTEX_WAIT, 1);
    }
}
```

Problem 5b[2pts]: Provide a corresponding release function for the above `futex_lock_acquire()` by filling in the lines, below:

```
void futex_lock_release(int* lock) {
```

```
    *lock = 0;
```

```
    futex(lock, FUTEX_WAKE, 1);
```

```
}
```

Problem 5c[2pts]: Using `CAS()`, implement an atomic increment operation. This operation will take a memory address and atomically increase the memory location's contents by one (similar to the C increment operator, `++`, but atomic so that it can be used by multiple threads simultaneously). You do not have to use every line.

```
void atomic_incr(int* addr) {
```

```
    int old = *addr;
```

```
    while(!CAS(addr, old, old+1) {
```

```
        old = *addr;
```

```
    }
```

```
}
```

Problem 5d[2pts]: Using `CAS()`, implement an atomic fetch-and-add function. This function will atomically fetch the contents of a memory location, update memory by adding an arbitrary value, then returning the initial value to the caller. You do not have to use every line.

```
int fetch_and_add(int* addr, int amt) {
```

```
    int old = *addr;
```

```
    while(!CAS(addr, old, old+amt) {
```

```
        old = *addr;
```

```
    }
```

```
    return old;
```

```
}
```

Implementing an RCU synchronizing primitive: Our next task is to implement a simple version of the read-copy-update (RCU) primitive used in Linux. The high level idea of this primitive is to allow many readers to access the data at once, similar to a reader/writer lock (as discussed in class). However, *unlike* the reader/writer lock discussed in class, we will continue to allow readers to access the data while the writer is making updates. In fact, readers do not have to acquire a lock – instead our RCU provides lock-free access to readers. Of course, we don't want readers to see partially updated data. Thus, we will optimize for reading by forcing the writer to make changes to a copy of the data, then atomically swapping this new version for the old one and eventually freeing the old copy after it makes sure that no readers are still examining it.

The controlled data can be anything, such as a string, a hash table, etc – the RCU will keep a pointer to the most recent exported version of the data to hand to readers. When a writer makes an update, it will generate a new version of the data and submit it to the RCU lock to distribute to readers. The user is responsible for allocating and initializing the RCU structure with an initial version of the data. Our API includes the following structure and functions:

```
/*
 * This is the basic RCU structure. It must be initialized. Although
 * the data pointer can point to an arbitrary data idem, it must be to
 * something that is on a 2 byte boundary (we use the low bit of the
 * address as a one-bit "epoch" counter).
 */
typedef struct RCU {
    void* tagged_data;    // Combined address and 1-bit epoch (lsb)
    int num_readers[2];    // Count of readers in each epoch (i.e. 0 or 1)
    int writer_mutex;      // For writers to avoid overlap with each other
} RCU_t;

// Initialize RCU structure. Takes pointer to RCU and initial data.
// Data should be from malloc(), i.e. freeable at a later date.
void rcu_init(RCU_t* rcu, void* data);

// Free RCU structure and verify that there are no readers (with assert)
Void rcu_deinit(RCU_t* rcu);

// Check out latest version of data for reading. Returns a "tagged"
// pointer to data (combining an address and epoch together).
void* read_start(RCU_t* rcu);

// Decode tagged pointer to extract address of data.
void* get_ptr(void* tagged_data);

// Finish using data. Use tagged pointer from matching read_start().
void read_end(RCU_t* rcu, void* tagged_data);

// Update RCU with new data and free old data
// Readers will see new data next time that they execute a read_start().
void write_replace(RCU_t* rcu, void* new_data);
```

Example usage for RCU primitive: Here is an example of using the above API for strings stored in the heap. The following allocates a bunch of readers and a smaller number of writers. Each write appends its ID to the end of the current data, then writes it back. Note: this code assumes we do not overflow our buffers, so is not robust! We have highlighted the API functions in bold:

```
#define NUM_THREADS 100
#define BUFFSIZE 1024 // Must be big enough to avoid overflow
RCU_t rcu; // Our RCU

int main() {
    pthread_t threads[NUM_THREADS];

    // Initialize RCU with string
    char* ptr = (char*)malloc(BUFFSIZE);
    strcpy(ptr, "Initial String");
    rcu_init(&rcu, (void*) ptr);

    // Start some threads accessing data
    for (size_t i = 0; i < NUM_THREADS; i++) {
        if (i % 10 == 0) // 1 of 10 threads is a writer
            pthread_create(&threads[i], NULL, &writer, (void*)i);
        else
            pthread_create(&threads[i], NULL, &reader, (void*)i);
    }
    for (size_t i = 0; i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);
    rcu_deinit(&rcu);
}

// Reader gets and prints data, then exits
void reader(void* arg) {
    void* tagged_data = read_start(&rcu); // Check out data
    char* ptr = (char*) get_ptr(tagged_data); // Get pointer to data
    printf("Thread %d says: rcu currently holds: %s", (int)arg, ptr);
    read_end(&rcu, tagged_data); // Finish with data
}

// Writer gets data and adds to it
void writer(void* arg) {
    char* new_data = (char*)malloc(BUFFSIZE); // new buffer

    // Copy old data to our new buffer and append ",Wxxx", where xxx is ID
    void* tagged_data = read_start(&rcu); // Get data
    char* old_data = (char*) get_ptr(tagged_data);

    // Just like printf(), but puts result into new_data buffer
    sprintf(new_data, "%s,W%d", old_data, (int)arg);
    read_end(&rcu, tagged_data);

    // Replace old data in RCU with our new_data
    write_replace(&rcu, new_data);
}
```

How RCU works: To implement the RCU primitive, we have two *epochs*, an even one (0) and an odd one (1). At any given time, the RCU will be in a particular epoch, which we will call the “current epoch”. The epoch changes only when data is written.

When readers check out data (by calling `read_start()`), they are registered as operating in the current epoch by incrementing the epoch-specific count in the RCU structure. They will then receive a pointer to the data that is tagged with the current epoch. Later, when they finish with the data and call `read_end()`, they will be unregistered in that epoch by decrementing the counter they incremented earlier. The two epoch-specific counters can be seen in the RCU structure (see `num_readers[]`):

```
typedef struct RCU {  
    void* tagged_data;      // Combined address and 1-bit epoch (lsb)  
    int num_readers[2];     // Count of readers in each epoch (i.e. 0 or 1)  
    int writer_mutex;       // For writers to avoid overlap with each other  
} RCU_t;
```

The current epoch is actually encoded in the least-significant bit of the `tagged_data` and can always be retrieved by casting `tagged_data` to an `int` and masking out other bits.

The writer is straightforward. Because of the mutex, there can be only one writer at a time. The writer swaps in a new `tagged_data` pointer for the old one, flipping to the new epoch in the process. All new readers will operate in the new epoch and increment the counter for the new epoch. The writer then simply waits until all reads using the old epoch have completed, after which it can free the old data from the previous epoch.

Problem 5e[2pts]: Implement the `rcu_init()` function by filling in the missing lines. Assume that we will be using our futex lock implementation from (5a) and (5b) as our `writer_mutex`. We have repeated the RCU structure definition for your convenience:

```
void rcu_init(RCU_t* rcu, void* data) {  
    rcu->tagged_data = data;  
    rcu->num_readers[0] = 0;  
    rcu->num_readers[1] = 0;  
    rcu->writer_mutex = 0;  
}
```

Problem 5f[3pts]: Next, implement the `read_start()` function by filling in the missing code lines. We have included the helper function `get_ptr()` that extracts an actual pointer from a tagged pointer. Before you can return a tagged_data pointer to the caller, you need to make sure that epoch reflected in the tagged_data is the same as the one whose counter you incremented (need to handle rare case of overlap with writer):

```
void* read_start(RCU_t* rcu) {
    void* tagged_data = rcu->tagged_data;

    uint epoch = (uintptr_t)tagged_data & 1;

    atomic_incr(&rcu->num_readers[epoch]);

    while (tagged_data != rcu->tagged_data) {
        // Writer overlapping, so need to roll back and start again
        // Possibly wake up waiting writer who might be sleeping on counter

        if (fetch_and_add(&rcu->num_readers[epoch], -1) == 1) {
            futex(&rcu->num_readers[epoch], FUTEX_WAKE, 1);
        }
        tagged_data = rcu->tagged_data;

        epoch = (uintptr_t)tagged_data & 1;

        fetch_and_add(&rcu->num_readers[epoch], 1);
    }
    return tagged_data;
}

// Extract pointer to data from tagged_data returned by read_start()
void* get_ptr(void* tagged_data) {
    return (void*)((uintptr_t)tagged_data & ~(uintptr_t)1);
}
```

Problem 5g[2pts]: Next, implement the `read_end()` function. Keep in mind that the writer might be waiting (sleeping) for readers to finish with data from old epoch. They will be using futex to sleep. If we are the last, then we may need to wake them up.

```
// Return data/unregister reader of data and possibly wake sleeping writer
void* read_end(RCU_t* rcu, void* tagged_data) {

    uint epoch = (uintptr_t)tagged_data & 1;

    if (fetch_and_add(&rcu->num_readers[epoch], -1) == 1) {
        futex(&rcu->num_readers[epoch], FUTEX_WAKE, 1);
    }
}
```

Problem 5h[5pts]: Next, implement the `writer_replace()` function. Keep in mind that we need to enforce the invariant that there is only a single writer allowed to run at a time. Also keep in mind that we cannot free the old data until we are sure that no readers are still using it.

```
// Handle write operation
void* write_replace(RCU_t* rcu, void* new_data) {
    futex_lock_acquire(&rcu->writer_mutex);
    void* old_tagged_data = rcu->tagged_data;
    int old_epoch = (uintptr_t)old_tagged_data & 1;
    rcu->tagged_data = (void*)((uintptr_t)new_data | (old_epoch ^ 1));
    int remaining_readers = rcu->num_readers[old_epoch];
    while(remaining_readers != 0) {
        futex(&rcu->num_readers[old_epoch], FUTEX_WAIT, remaining_readers);
        remaining_readers = rcu->num_readers[old_epoch];
    }
    // If we get here, no one is still using the old data.
    free(get_ptr(old_tagged_data));
    futex_lock_release(&rcu->writer_mutex);
}
```

[This page intentionally left blank]

[Function Signature Cheat Sheet]

```

/***** pthreads *****/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_cond_init(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

/***** Processes *****/
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);

/***** High-Level I/O *****/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);

/***** Sockets *****/
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);

/***** Low-Level I/O *****/
int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
    whence=>SEEK_SET, SEEK_CUR, or SEEK_END
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);
```

[*Function* Signature Cheat Sheet (con't)]

```
/****** Futex *****/
int futex(int *uaddr, int futex_op, int val);
```

futex_op is one of:

FUTEX_WAIT, FUTEX_WAKE, FUTEX_FD, FUTEX_WAKE_OP, FUTEX_CMP_REQUEUE

```
/****** Pintos *****/
void list_init(struct list *list);
struct list_elem *list_head(struct list *list)
struct list_elem *list_tail(struct list *list)
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
struct list_elem *list_pop_front(struct list *list);
struct list_elem *list_pop_back(struct list *list);
```

```
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);
```

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]