

Midterm I

February 15th, 2024

CS162: Operating Systems and Systems Programming

Your Name:	
SID AND 162 Login (e.g. s042):	
TA Name:	
Discussion Section Time:	

General Information:

This is a **closed book** exam. You are allowed 1 page of notes (both sides). You have 110 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	20	
2	18	
3	22	
4	24	
5	16	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [20 pts]

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

Problem 1a[2pts]: Because the “monitor” pattern of synchronization involves sleeping inside a critical section, it requires a special hardware support to avoid deadlock (i.e. a permanent lack of forward progress).

True False

Explain:

Problem 1b[2pts]: In UNIX, Pipes are implemented with a buffer in user space.

True False

Explain:

Problem 1c[2pts]: Any two processes in a running system have a common ancestor process.

Clarification during exam: a process is considered an ancestor of itself.

True False

Explain:

Problem 1d[2pts]: After the main thread in a process calls the `exit()` system call (either implicitly or explicitly), the other threads are allowed to run to completion before the process terminates.

True False

Explain:

Problem 1e[2pts]: A user-level library implements each system call by executing a “transition to kernel mode” instruction followed by a procedure call to an appropriate system call handler in the kernel. To make this work, the user-level library must be built with access to a symbol table that contains the kernel addresses of each system call.

True False

Explain:

Problem 1f[2pts]: Not every call to `fread()` must make a call the kernel using the `read()` system call to retrieve data from the underlying file.

True False

Explain:

Problem 1g[2pts]: The `test&set` instruction consists of *two* independent operations: (1) read the value from memory and (2) replace it with the value “1”. Thus, it must be used in a loop to protect against the case in which multiple threads execute `test&set` simultaneously and end up interleaving these two operations from different threads.

True False

Explain:

Problem 1h[2pts]: It is possible for a client machine (with IP address X) and server machine (with IP address Y) to have multiple simultaneous but independent communication channels between them – despite the fact that network packets get intermixed in the middle.

True False

Explain:

Problem 1i[2pts]: When a process issues a system call in Pintos, the kernel must change the active page table (i.e., update the page table base register) so that the system call handler can access kernel memory.

True False

Explain:

Problem 1j[2pts]: A user-level application can build an implementation of `lock acquire()` and `release()` by using the `interrupt enable` and `disable` functions of the `pthread` library.

True False

Explain:

Problem 2: Multiple Choice [18pts]

Problem 2a[2pts]: Which of the following statements about Base&Bound style address translation are true? (*choose all that apply*):

- A: Base&Bound cannot protect kernel memory from being read by user programs.
- B: Sharing memory between processes is difficult.
- C: Context switches incur a much higher overhead compared to use of a page table.
- D: Base&Bound will lead to external fragmentation.
- E: With Base&Bound, each process can have its own version of address “0”.

Problem 2b[2pts]: What are some things that can cause a transfer from user mode to kernel mode? (*choose all that apply*):

- A: User code divides by zero.
- B: The user executes a system call.
- C: A packet is received from the network.
- D: The application uses malloc() to allocate memory from the heap.
- E: The timer goes off.

Problem 2c[2pts]: Consider the following pseudocode implementation of a lock_acquire().

```
lock_acquire() {
    interrupt_disable();
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    interrupt_enable();
}
```

Which of the following are TRUE? Assume we are running on a uniprocessor/single-core machine. (*choose all that apply*):

- A: For this implementation to be correct, we should call interrupt_enable() before sleeping.
- B: For this implementation to be correct, we should call interrupt_enable() before putting the thread on the wait queue.
- C: For this implementation to be correct, sleep() should trigger the scheduler and the next scheduled thread should enable interrupts.
- D: It is possible for this code to be run in user mode.
- E: None of the above.

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

- A: Semaphores can be initialized to any 32-bit value in the range -2^{31} to $2^{31}-1$
- B: If there is at least one thread sleeping on a given semaphore, then the result of performing a `Semaphore.V()` will be to wake up one of the sleeping threads rather than incrementing the value of the semaphore.
- C: Semaphores cannot be used for locking.
- D: The *interface* for `Semaphore.P()` is specified in a way that prevents its *implementation* from busy-waiting, even for a brief period of time.
- E: The pure semaphore interface does not allow querying for the current value of the semaphore.

Problem 2e[2pts]: In PintOS, every “user thread” has a chunk of memory that serves as a user-level stack and that is matched one-for-one with a corresponding kernel stack in kernel-protected memory. What is true about this arrangement (*choose all that apply*):

- A: When a user-thread makes a system call that must block (*e.g.* a read to a file that must be retrieved from disk), the user thread can be put to sleep at any time by pushing CPU state onto its kernel stack and putting that stack onto an appropriate wait queue inside the kernel.
- B: The kernel stack is often identified in figures as a “kernel thread” because it represents an independent computational entity that can run at the same time as the user thread.
- C: The kernel stack can grow to an arbitrary size (on demand) using virtual memory.
- D: The kernel gains safety by the presence of the kernel stack because it does not have to rely on the correctness of the user’s stack pointer register for correct behavior.
- E: None of the above.

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

- A: If a process calls `execv()`, it does not need to free any of its allocated heap variables.
- B: Using IPC (such as a pipe), a child process is able to share the address of a stack-allocated variable directly with its parent process so that the two processes can communicate directly using load and store instructions.
- C: Each process has its own instance of user and kernel memory.
- D: If a parent process has multiple running threads at the time of `fork()`, then the child process will have multiple running threads immediately after `fork()`.
- E: Immediately after `fork()`, a child has a *duplicate* file descriptor table with pointers to *duplicated* file description structures for files that are open in the parent.

Problem 2g[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: Reserved 0, 1, and 2 (`stdin`, `stdout`, `stderr`) file descriptors cannot be overwritten by a user program.
- 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.

Problem 2h[2pts]: Select all true statements about the x86 Calling sequence (*choose all that apply*):

- A: Parameters are pushed onto the stack in the order that they are declared in the function.
- B: Right before the Caller jumps to the Callee function, the ESP must be 16-byte aligned.
- C: If padding is necessary for alignment, it should be added at a lower address than the parameters.
- D: In the Callee, space is allocated for local variables via subtracting from the ESP.
- E: None of the above.

Problem 2i[2pts]: Which of the following are true about condition variables? (*choose all that apply*):

- A: `cond_wait()` can only be used when holding the lock associated with the condition variable.
- B: In practice, Hoare semantics are used more often than Mesa semantics.
- C: Mesa semantics will lead to busy waiting in `cond_wait()`.
- D: Each condition variable has its own wait queue for sleeping threads.
- E: All of the above.

[This page intentionally left blank]

Problem 3: Synchronization [22pts]

Consider the following two threads, to be run concurrently in a shared memory (all variables are shared between the two threads):

Thread A	Thread B
<pre>for (i=0; i<5; i++) { x = x + 1; }</pre>	<pre>for (j=0; j<5; j++) { x = x + 1; }</pre>

Assume a single-processor system, that load and store are atomic, and that x must be loaded into a register before being incremented (and stored back to memory afterwards). *Assume that x is initialized to zero before either Thread A or Thread B start.*

Problem 3a[3pts]: Give a *concise* proof why $x \neq 1$ when both threads have completed. *Hint: consider the fact that each store is one greater than the value it loaded, even when threads interleave.*

Problem 3b[2pts]: What is a *critical section* and what is the role of locking in enforcing correct behavior for a multithreaded program?

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 (let “expr” be an expression, “&addr” be an address of a memory location, and “M[addr]” be the actual memory location at address addr):

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

Both TSET and SWAP return values from memory, whereas CAS returns either true or false. Note that our &addr notation is similar to a reference in C, and means that the &addr argument must be something that can be stored into. For instance, TSET could implement a spin-lock acquire as follows:

```
int lock = 0;          // lock is free
while (TSET(&lock)); // Later: acquire lock
```

Problem 3c[2pts]: Show how to implement a spinlock `acquire()` with a single while loop using CAS instead of TSET. Fill in the arguments to CAS below. *Hint: need to wait until can grab lock.*

```
void acquire(int *mylock) {
    while (!CAS(mylock, _____, _____ ));
}
```

Problem 3d[3pts]: In class we argued that spinlocks were a bad idea because they can waste a lot of processor cycles (busy waiting). There is, however, one case we mentioned in which spinlocks would be more efficient than blocking locks. When is that? Can you say why the spinlock of **Problem 3c** might be even better in this circumstance than the standard spinlock built with TSET?

Problem 3e[2pts]: Fill in the blanks, to construct a lock-free implementation of a singly-linked list “push” operation. *Hint: We need to retry if someone changes the root point out from under us.*

```
typedef struct node {
    node_t *next;
    data_t mydata;
} node_t
void push(node_t **rootp, node_t *newnode) {
    do {
        node_t *oldfront = *rootp;
        newnode->next = oldfront;

        } while (!CAS(rootp, _____, _____));
}
// Sample usage:
node_t *root = NULL;          // Empty list
push(&root, mynewnode);      // push new node onto list
```

The issue with constructing locks using only user-level atomic instructions such as TSET or CAS is that we cannot block a thread (i.e. put it to sleep) when it is unable to acquire a lock. As discussed in class, `futex()` is a *system call* that allows a thread to *put itself to sleep*, under certain circumstances, on a queue associated with a user-level address. It also allows a thread to ask the kernel to wake up threads that might be sleeping on the same queue. Recall that the function signature for `futex` is:

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

In this problem, we focus on two `futex_op` values:

1. For `FUTEX_WAIT`, the kernel checks atomically as follows:

if (`*uaddr == val`): the calling thread will sleep and another thread will start running

if (`*uaddr != val`): the calling thread will keep running, i.e. `futex()` returns immediately

2. For `FUTEX_WAKE`, this function will wake up to `val` waiting threads. ~~You can assume in this problem that `val` will always be \leq the actual number of waiting threads.~~

Problem 3f[2pts]: Fill out the missing blanks, in the `acquire()` function, below, to make a version of a lock that does not busy wait. The corresponding `release()` function is given for you:

```
void acquire(int *thelock) { // Acquire a lock
    while (TSET(thelock)) {
        futex(thelock, _____, _____);
    }
}
void release(int *thelock) { // Release a lock
    *thelock = 0;
    futex(thelock, FUTEX_WAKE, 1);
}
```

Problem 3g[2pts]: Under which circumstances is the lock implementation in **Problem 3f** suboptimal enough to look for a different implementation?

Problem 3h[3pts]: To address the problem in **Problem 3g**, we can build a lock with three states:

```
typedef enum { UNLOCKED, LOCKED, CONTESTED } Lock;
Lock mylock = UNLOCKED; // Initialize the lock in UNLOCKED state
```

Explain why we might want three states (rather than just `LOCKED` and `UNLOCKED`). In your explanation, make sure to say what each of the states indicate and under what circumstances the presence of the third state (`CONTESTED`) allows us to optimize performance.

Problem 3i[3pts]: Fill in the missing blanks, below, for the three-state solution, using constants from the Lock enum. Hint: You are optimizing your lock based on the answer to **Problem 3h**:

```
typedef enum { UNLOCKED, LOCKED, CONTESTED } Lock;
Lock mylock = UNLOCKED; // Initialize the lock in UNLOCKED state

// Acquire a lock
void acquire(Lock *thelock) {
    // Fast case acquire:

    if (CAS(thelock, _____, _____))
        return;

    // Contested acquire:

    while (SWAP(thelock, _____) != _____)
        futex(thelock, FUTEX_WAIT, CONTESTED);
}

// Release a lock
void release(Lock *thelock) {

    if (SWAP(thelock, _____) == _____)
        futex(thelock, FUTEX_WAKE, 1);
}
```

Problem 4: Short Answer Potpourri [24pts]

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[3pts]: What happens when an interrupt occurs? What does the interrupt controller do?

Problem 4b[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?

Problem 4c[3pts]: What needs to be saved and restored on a context switch between two threads in the same process? What if the two threads are in different processes? Be explicit.

Problem 4d[2pts]: What was the problem with the Therac-25 radiation therapy machine? Your answer should involve one of the topics of the class.

Problem 4e[2pts]: What is “hyperthreading”? Does it allow threads to execute “concurrently” or “in parallel”? Explain.

Problem 4f[2pts]: When a process thread executes `fork()`, this creates a new child process with an address space that is *separate* from the parent and that has *identical contents* to the parent process. Does an implementation of `fork()` require the operating system to copy all of the data of the parent process into the child process? Explain.

Problem 4g[2pts]: When handling PintOS syscalls in `userprog/syscall.c`, how can we tell what syscall the user called, since there is only one `syscall_handler` function?

Problem 4h[4pts]: Your friend tells you that they can open a file once but read it twice. Although you are skeptical, you decide to give it a try. Without utilizing another call to open, finish the following `double_read()` function. This function should read the specified file twice, storing the appended result as a duplicated, null-terminated string in the given buffer 'buffer'. Note: *You must actually read() the file contents twice – do not just copy the data after reading the file once!*

In the following, assume that all system calls succeed and that the buffer is big enough to hold the result. Read the file in a maximum of `CHUNK_SIZE` bytes at a time. *While you do not necessarily need to use every line here, you are also not allowed to add semicolons to existing lines.*

```
#define CHUNK_SIZE 1024
void double_read(char *filename, char *buffer) {
    int fd = open(filename, O_RDONLY);

    _____;

    _____;

    for (_____ ; _____ ; _____) {
        while(1) {
            _____;

            if (_____) break;

            _____;

        }
        _____;
    }

    _____;

    close(fd);
}
```

Problem 4i[3pts]: The “high-level” file I/O interface (i.e `fopen()`, `fclose()`, `fread()`, `fwrite()`, and other associated functions) is often called the “streaming I/O interface” in contrast to the “low-level” interface (i.e. `open()`, `close()`, `read()`, `write()`). Why is this terminology/distinction appropriate? (*Hint: start by explaining what a streaming communication pattern might is*).

[This page intentionally left blank]

Problem 5: Stock Trading [16]

Stock trading is a very dynamic process with multiple traders simultaneously issuing buy and sell requests. As a result, any system that supports trading must deal with synchronization to provide correct behavior. In this problem, we will build a system to match sell requests to buyers. One essential element of our solution is that each particular stock has a `match_queue` that is used for coordinating the sales of that stock. We will build a fully synchronized solution using the monitor pattern with pthreads (i.e. using `pthread_mutex_t` and `pthread_cond_t` variables). Both sellers and buyers will be put to sleep while their corresponding trades are matched and executed.

Problem 5a[2pts]: We will represent a sell request with a `sell_request_t` structure and the queue of pending sell requests with a `match_queue_t` structure. Complete the following definitions. Assume that we will use a *Pintos list* to link sell requests into the `match_queue_t` structure. Further assume that buyers will sleep on one condition variable and sellers will sleep on a separate one, both of which are stored within the `match_queue`. Do not add any semicolons!

```
typedef struct sell_request {
    int waiting_sell;           // Remaining # shares for this seller
    struct list_elem mylink;    // Link for Pintos list
} sell_request_t;

typedef struct match_queue {
    char *stock_symbol;        // String describing stock
    int waiting_buy;           // Number waiting buyers
    _____;
    _____;
    _____;
    _____;
} match_queue_t;
```

Problem 5b[2pts]: Complete the following `match_queue` allocator, assuming `calloc` succeeds). Please do not add any semicolons. Error codes for syscalls and pthread functions do not need to be checked.

```
match_queue_t *match_queue_alloc(char *stock_symbol) {
    match_queue_t *new_match_queue =
        (match_queue_t *)calloc(1, sizeof(match_queue_t));
    new_match_queue->stock_symbol = stock_symbol;
    _____;
    _____;
    _____;
    _____;
    return new_match_queue;
}
```

Problem 5c[6pts]: Fill in the missing blanks for the buyer's routine, below. Assume that `buy_stock_match()` should not return until the total requested shares have been matched with a seller. However, busy-waiting is strictly forbidden. Make sure that sellers (represented by `sell_request_t` structures with a non-zero `sell_waiting` value) are handled strictly in order. Error codes for syscalls and pthread functions do not need to be checked. Note: The `matchq` argument represents a pointer that has gone through your `match_queue_alloc()` routine, so should be properly initialized. *While you do not necessarily need to use every line here, you are also not allowed to add semicolons to existing lines.*

Hint: The buyers and sellers work together. Thus, as you will handle in **Problem 3d**, sellers will sleep waiting for their shares to be completely matched with buyers. And buyers will sleep waiting for enough shares to become available.

```
// Buy num_shares of a stock through the given match_queue
// Assume num_shares > 0 and matchq is valid (from match_queue_alloc())
void buy_stock_match(match_queue_t *matchq, int num_shares) {
    _____;
    while(num_shares) {
        if (list_empty(_____)) {
            matchq->waiting_buy += 1;
            _____;
            matchq->waiting_buy -= 1;
        } else {
            struct list_elem *fe = list_pop_front(_____);
            sell_request_t *first =
                list_entry(_____, _____, _____);
            if (first->waiting_sell > num_shares) {
                _____;
                _____;
                _____;
            } else {
                _____;
                _____;
                _____;
            }
        }
    }
}
_____;
```

Problem 5d[4pts]: Fill in the missing blanks for the seller's routine. Assume that `calloc()` always succeeds and that the seller should be put to sleep until all of their stock shares have been matched to buyers. Make sure that new sellers are not allowed to jump in front of existing sellers. Error codes for syscalls and pthread functions do not need to be checked. Note: The `matchq` argument represents a pointer that has gone through your `match_queue_alloc()` routine, so should be properly initialized. Busy-waiting is not allowed. Further, you should make sure that there are no memory leaks (it is up to the seller to free any sell structures they have allocated). *While you do not necessarily need to use every line here, you are also not allowed to add semicolons to existing lines.*

```
// Sell num_shares of a stock through the given match_queue
// Assume num_shares > 0 and matchq is valid (from match_queue_alloc())
void sell_stock_match(match_queue_t *matchq, int num_shares) {
    sell_request_t *req =
        (sell_request_t *)calloc(1, sizeof(sell_request_t));
    req->waiting_sell = num_shares;

    _____;

    list_push_back(_____, _____);
    if (matchq->waiting_buy) {

        _____;
    }
    while ( _____ ) {

        _____;
    }
    _____;
    _____;
}
}
```

Problem 5e[2pts]: Suppose that many sellers show up all at once, before buyers arrive. Then, suppose buyers arrive one at a time. Although still correct, the above solution will incur a lot of scheduler overhead. Explain the problem in three sentences or less and how you would fix it.

[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, structure 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)]

```
/****** 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_push_front(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!]