University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2022 Joseph & Kubiatowicz

Midterm I

February 17th, 2022
CS162: Operating Systems and Systems Programming

Your Name:	
SID AND Autograder Login (e.g. student042):	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 1 page of notes (both sides). You may use a calculator. 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.

PLEASE WRITE YOUR ANSWERS ON THE ANSWER SHEET, NOT IN-LINE HERE. *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	16	
3	12	
4	23	
5	15	
6	16	
e	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!). Also, answers without an explanation *GET NO CREDIT*.

Problem 1a[2pts]: If Thread A and B are in the same process, then Thread A can access local variables stored in Thread B's stack.

□ True □ False

Explain:

Problem 1b[2pts]: Let *n* be the size of the virtual address space. On a `fork()` call, the OS does O(*n*) work to duplicate the parent's address space for the child process.

□ True □ False

Explain:

Problem 1c[2pts]: Trying to use `lseek()` on file descriptor #1 will always result in an error.

□ True □ False

Explain:

Problem 1d[2pts]: A child process can communicate with its parent by utilizing a data structure on its heap that was allocated before the parent performed a `fork()` system call.

□ True □ False

Explain:

Problem 1e[2pts]: If we have a queue of multiple waiters on a condition variable with Mesa scheduling, and we execute a `cond_broadcast()`, the waiters would be processed in FIFO order.

□ True □ False

Explain:

Problem 1f[2pts]: There are situations where disabling interrupts *must* be used as opposed to other synchronization primitives.

□ True □ False

Explain:

Problem 1g[2pts]: Inside the Pintos kernel, we can find the `struct thread` of the current running thread by rounding down `%esp` to the nearest page boundary.

□ True □ False

Explain:

Problem 1h[2pts]: The FPU registers must be saved and restored on every entry into the kernel.

□ True □ False

Explain:

Problem 1i[2pts]: Context switching is implemented in Pintos by swapping user stacks.

□ True □ False

Explain:

Problem 2: Multiple Choice [16pts]

Problem 2a[2pts]: Which of the following are true about syscalls in Pintos? (*choose all that apply)*:

A: ☐ User programs directly call the syscall functions in the file src/userprog/syscall.c.

B: ☐ Arguments passed into system calls must be validated before they are used.

C: ☐ Syscalls in Pintos are run in user mode.

D: ☐ The wait syscall returns the exit code of the child process specified in the argument to the function.

E: ☐ None of the above.

Problem 2b[2pts]: What are some reasons that overuse of threads is bad (*i.e.* using too many threads at the same time in a single process)? (*choose all that apply*):

A: ☐ The thread name-space becomes too large and fragmented, making it very difficult to efficiently track the current executing thread.

B: ☐ The overhead of switching between too many threads can waste processor cycles such that the overhead outweighs actual computation (i.e. thrashing)

C: ☐ Excessive threading can waste memory for stacks and TCBs

D: ☐ The number of page tables becomes too large, thus overloading the virtual memory mechanisms.

E: ☐ All of the above.

Problem 2c[2pts]: What are the disadvantages of disabling interrupts to serialize access to a critical section? (*choose all that apply*):

A: ☐ User code cannot utilize this technique for serializing access to critical sections.

B: ☐ Interrupt controllers have a limited number of physical interrupt lines, thereby making it problematic to allocate them exclusively to critical sections.

C: ☐ This technique would lock out other hardware interrupts, potentially causing critical events to be missed.

D: ☐ This technique is a very coarse-grained method of serializing, yielding only one such lock for each core.

E: ☐ This technique could not be used to enforce a critical section on a multiprocessor.

Problem 2d[2pts]: In Pintos, every user thread is matched with a corresponding kernel thread (complete with a kernel stack). What is true about this arrangement (*choose all that apply*):

A: ☐ When the user-thread makes a system call that must block (*e.g.* a read to a file that must go to disk), the thread can be blocked at any time by putting the kernel thread (with its stack) to sleep and waking another kernel thread (with its stack) from the ready queue.

B: ☐ The presence of the matched kernel thread makes the user thread run twice as fast as it would otherwise.

C: ☐ The kernel thread helps to make sure that the page table is constructed properly so that the user thread's address space is protected from threads in other processes.

D: ☐ While user code is running, the kernel thread manages cached data from the file system to make sure the most recent items are stored in the cache and ready when the user needs them.

E: ☐ The kernel gains safety because it does not have to rely on the correctness of the user's stack pointer register for correct behavior.

Problem 2e[2pts]: What are some disadvantages of Base&Bound style address translation? (*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 2f[2pts]: Which of the following are true about condition variables? (*choose all that apply):*

A: ☐ `cond_wait()`, and `cond_signal()` 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: ☐ `cond_signal()` can only be called after setting the boolean condition associated with the condition variable to true.

E: ☐ All of the above.

Problem 2g[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 2h[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 3: Readers-Writers Access to Database [12 pts]

<pre>Reader() {
   //First check self into system
   lock.acquire();
   while ((AW + WW) > 0) {
      WR++;
      okToRead.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();
 	
   // Perform read access
   AccessDatabase(ReadOnly);
 	
   // Now, check out of system
   lock.acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okToWrite.signal();
   lock.release();
}</pre>
	<pre>Writer() {
	   // First check self into system
	   lock.acquire();
	   while ((AW + AR) > 0) {
	      WW++;
	      okToWrite.wait(&lock);
	      WW--;
	   }
	   AW++;
	   lock.release();
	
	   // Perform read/write access
	   AccessDatabase(ReadWrite);
	
	   // Now, check out of system
	   lock.acquire();
	   AW--;
	   if (WW > 0){
	      okToWrite.signal();
	   } else if (WR > 0) {
	      okToRead.broadcast();
	   }
	   lock.release();
	}</pre>

Problem 3a[2pts]: Above, we show the Readers-Writers example given in class. What are the correctness constraints described by the Reader/Writer model above? (Hint: When can writers/readers access the database)?

Problem 3b[2pts]: The above code uses two condition variables, one for waiting readers and one for waiting writers. Suppose that *all* of the following requests arrive in very short order (while R_1 and R_2 are still executing):

Incoming stream: R_1 R_2 W_1 R_3 W_2 W_3 R_4 R_5 R_6 W_4 R_7 W_5 W_6 R_8 R_9 W_7 R_{10}

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue). Explain how you got your answer.

NOTE: Each of the following subparts are independent. Describe in 2-3 sentences the conceptual changes needed to support the new feature, *including the positions where the described logic should be added or changed.*

Problem 3c[4pts]: Suppose we have now upgraded our storage system to Google Sheets, which can now handle multiple writers accessing the database at the same time. How do we modify the existing logic to have multiple writers access the database concurrently? Assume all other correctness constraints are the same.

Problem 3d[4pts]: Suppose funds have run low, and our database can only handle a certain number of readers at a time. How do we modify the logic so that **at most 10** readers may access the database at any time?

Problem 4: Atomic Synchronization Primitives [23pts]

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)

`TSET(&addr) {`
`  int result = M[addr];`
`  M[addr] = 1;`
`  return (result);`
`}`
	`SWAP(&addr, expr) {`
	`  int result = M[addr];`
	`  M[addr] = expr;`
	`  return (result);`
	`}`
		`CAS(&addr, expr1, expr2) {`
		`  if (M[addr] == expr1) {`
		`     M[addr] = expr2;`
		`     return true;`
		`  } else {`
		`     return false;`
		`  }`
		`}`

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 be used to implement a spin-lock acquire as follows:

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

// Later: 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(&addr) {
   int temp;
   do {
      temp = M[addr];
   } while (!CAS(addr,temp,1));
   return temp;
}
```

Problem 4a[2pts]:
Show how to implement a spinlock acquire with a single while loop using CAS instead of TSET. You must only fill in the arguments to CAS below:

```
// Initialization
int lock = 0;  // Lock is free

// acquire lock

while (  !CAS( ________________________________ )  );
```

Problem 4b[2pts]:
Show how SWAP can be implemented using CAS. Don't forget the return value.

```
        SWAP(&addr, reg1) {

            Object result;

            Do {
Line 1:         ______________________________;
Line 2:     } while ( _________________________________ );
            return result;

        }
```

Problem 4c[2pts]:
With spinlocks, threads spin in a loop (busy waiting) until the lock is freed. In class we argued that spinlocks were a bad idea because they can waste a lot of processor cycles. The alternative is to put a waiting process to sleep while it is waiting for the lock (using a blocking lock). Contrary to what we implied in class, there are cases in which spinlocks would be more efficient than blocking locks. Give a circumstance in which this is true and explain why a spinlock is more efficient.

Using Atomic Primitives to Implement a LOCK-Free Queue:

A queue is considered "lock-free" if multiple threads can operate on this object simultaneously without the use of locks, busy-waiting, or sleeping. In this problem, we construct a lock-free FIFO queue using atomic CAS operation. We need both an `Enqueue()` and `Dequeue()` method.

We are going to do this in a slightly different way than normally. Rather than `Head` and `Tail` pointers, we are going to have "`PrevHead`" and `Tail` pointers. `PrevHead` will point at the last object returned from the queue. Thus, we can find the head of the queue (for dequeuing). If we don't worry about simultaneous `Enqueue()` or `Dequeue()` operations, the code is straightforward:

```
/*** Queue Entries and Structure ***/
typedef struct QueueEntry {
     struct QueueEntry *next;
     void *stored;
} QueueEntry;
typedef struct Queue {
      QueueEntry *prevHead;
      QueueEntry *tail;
} Queue;
/*** Queue initialization ****/
void initQueue(Queue *newqueue)
{
     newqueue->prevHead = allocQueueEntry(NULL);
     newqueue->tail = newqueue->prevHead;
}
/*** Allocate a QueueEntry to hold pointer to item ***/
QueueEntry *allocQueueEntry(void *myItem)
{
     QueueEntry *newqueue = (QueueEntry *)malloc(sizeof(QueueEntry));
     newqueue->stored = myItem;
     newqueue->next = NULL;
     return newqueue;
}

/*** Enqueue operation ****/
void Enqueue(Queue *myqueue, void *newobject)  {
     QueueEntry *newEntry = allocQueueEntry(newobject);
     QueueEntry *oldtail = myqueue->tail;
     myqueue->tail = newEntry;
     oldtail->next = newEntry;
}
/*** Dequeue operation ***/
void *Dequeue(Queue *myqueue) {
     QueueEntry *oldprevHead = myqueue->prevHead;
     QueueEntry *nextEntry = oldprevHead->next;
     if (nextEntry == NULL)
          return NULL;
     myqueue->prevHead = nextEntry;
     free(oldprevHead);
     return nextEntry->stored;
}
```

Problem 4d[3pts]:
For this non-multithreaded code, draw the state of a queue with 2 queued items on it:

Problem 4e[3pts]: For each of the following potential context switch points, state whether or not a context switch at that point could cause incorrect behavior of Enqueue(); Explain!

```
        void Enqueue(Queue *myqueue, void *newobject)  {
 1 ----->   QueueEntry *newEntry = allocQueueEntry(newobject);
 2 ----->   QueueEntry *oldtail = myqueue->tail;
 3 ----->   myqueue->tail = newEntry;
            oldtail->next = newEntry;
        }
```

Point 1:

Point 2:

Point 3:

Problem 4f[3pts]: Rewrite code for Enqueue(), using the CAS() operation, such that it will work for any number of simultaneous Enqueue and Dequeue operations. You should never need to busy wait. **Do not use locking (i.e. don't use a test-and-set lock).** Fill in each of the empty lines below. We will be grading on conciseness. Do not use more than one CAS(). *Hint: wrap a do-while around vulnerable parts of the code identified above.*

```
      void Enqueue(Queue *myqueue, void *newobject)  {
         QueueEntry *newEntry = allocQueueEntry(newobject);
         QueueEntry *oldtail;

         // Insert missing code (shown by lines)

         do {  // One of following 3 lines will have a 'while'!

Line 1:      ___________________________________;

Line 2:      ___________________________________;

Line 3:      ___________________________________;

      }
```

Problem 4g[3pts]: For each of the following potential context switch points, state whether or not a context switch at that point could cause incorrect behavior of Dequeue(); Explain! (Note: *Assume that the queue is not empty when answering this question, since we have removed the null-queue check from the original code):*

```
        void *Dequeue(Queue *myqueue) {
 1 ───────►  QueueEntry *oldprevHead = myqueue->prevHead;
 2 ───────►  QueueEntry *nextEntry = oldprevHead->next;
 3 ───────►  myqueue->prevHead = nextEntry;
             free(oldprevHead);
             return nextEntry->stored;
        }
```

Point 1:

Point 2:

Point 3:

Problem 4h[5pts]: Rewrite code for Dequeue(), using the CAS() operation, such that it will work for any number of simultaneous Enqueue and Dequeue operations. You should never need to busy wait. **Do not use locking (i.e. don't use a test-and-set lock).** Fill in each of the empty lines below. We will be grading on conciseness. Do not use more than one CAS(). You should correctly handle an empty queue by returning "null". *Hint: wrap a do-while around vulnerable parts of the code identified above and add back the null-check from the original code. Also, assign "*`result`*" inside loop.*

```
        void *Dequeue(Queue *myqueue) {
           QueueEntry *oldprevHead, *nextEntry;
           void *result;

           // Insert missing code here (shown by lines)
           do {    // One of following 5 lines will have a 'while'!
Line 1:        ______________________________;

Line 2:        ______________________________;

Line 3:        ______________________________;

Line 4:        ______________________________;

Line 5:        ______________________________;
           free(oldprevHead);
           return result;
        }
```

Problem 5: Short Answer Potpourri [15 pts]

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

Problem 5a[3pts]: What is the difference between Mesa and Hoare scheduling for monitors? How does this affect the programming pattern used by programmers (be explicit)?

Problem 5b[3pts]: Explain the key difference between the low-level and high-level file APIs in C as discussed in lecture. Why might the high-level file API be higher performance than the low-level API?

Problem 5c[2pts]: Recent Linux kernel versions introduce the new *`sendfile(2)`* system call that transfers data from one file descriptor to another. It's signature is:

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

Why might it be faster to use `sendfile()` instead of *read*-ing from `in_fd` and then *write*-ing to `out_fd`? Your answer must focus on the location of data in memory.

Problem 5d[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?

Problem 5e[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 5f[2pts]: How does a modern OS regain control of the CPU/core from a program stuck in an infinite loop?

Problem 6: Adding `dup()` to Pintos [16 pts]

As an operating systems fanatic, Nathan has recently been using Pintos as his daily driver. However, when writing some C programs, he's finding the lack of some file operation syscalls quite frustrating to work with. Specifically, Nathan would really like to use dup. As a superb CS 162 student, you have been tasked with implementing it. The descriptions for each are given below. Note that there are subtle differences from the Unix versions, mainly for sake of simplicity.

```
/* Creates a copy of the file descriptor FD such that the copy and original
   point to the same file description. The new file descriptor must be one
   above the max of the current file descriptors. For instance, if the
   existing file descriptors were [1, 5, 7], 8 would be used as the new file
   descriptor. Return new file descriptor on success or -1 for an invalid FD.
*/
int dup(int fd);
```

Complete the blanks in the skeleton for `syscall_handler()` to implement this syscall. (Blanks are labeled with capital letters). For simplicity, you may assume that Nathan will not write any malicious user programs, so you may simply access syscall arguments passed in through the user stack (i.e. no need to copy them to the kernel stack).

Below are the structure definitions within the kernel that might be helpful for this problem:

```
/* Process control block */
struct process {
  ...
  struct list fdt;     /* File descriptor table of struct fd. */
  ...
};
/* File description */
struct file {
  ...
  off_t pos;           /* Current position. */
  ...
};
/* File descriptor */
struct fd {
  int num;
  struct file* file;
  struct list_elem elem;
};
/* Interrupt stack frame */
struct intr_frame {
  ...
  void* esp;
  uint32_t eax;
  ...
};
```

Below are function signatures that might be helpful when solving this problem.

```
/****** List Operations ******/
struct list_elem* list_begin(struct list*);
struct list_elem* list_next(struct list_elem*);
struct list_elem* list_end(struct list*);
void list_push_back(struct list*, struct list_elem*);
#define list_entry(LIST_ELEM, STRUCT, MEMBER)

/****** Lock Operations *******/
void lock_acquire(struct lock*);
void lock_release(struct lock*);

/****** Memory Operations ******/
void *malloc(size_t size);
void free(void *ptr);

/******* Global file system lock *******/
struct lock fs_lock;

/*******
 * The following function retrieves the struct file* corresponding to file
 * descriptor NUM in the current process's file descriptor table.
 * Returns NULL if file descriptor NUM is invalid.
 *******/
struct file* get_file(int num);
```

Here is the skeleton of `syscall_handler()` to complete. You will fill in the missing lines on your answer sheet. Your code must be written in proper C code with the given APIs. Pseudocode or comments will not be given any credit. You may only use methods and data structures given in this question as well as any built-in ones. You may assume calls to any given method will succeed. Only one piece of code must be written per blank (i.e. no multiple statements with semicolons). Each blank must contain code and cannot be a blank line. Assume `fs_lock` is initialized.

```
    void syscall_handler(struct intr_frame *f) {
      uint32_t* args = (uint32_t*) f->esp;

      switch (args[0]) {
        ...
        case SYS_DUP:

A:        ___________________;

B:        struct file* file = __________________;

C:        if (___________________) {

D:          ___________________;
          } else {
            int max_fdnum = -1;
            struct list* fdt = &thread_current()->pcb->fdt;
            struct list_elem *e;

E:          for (____________;_____________;____________) {

F:            struct fd* fd = ___________________;

G:            if (___________________) {

H:                ___________________;
              }
            }

I:          struct fd* newfd = ___________________;

J:          newfd->num = ___________________;

K:          newfd->file = ___________________;

L:          ___________________;
          }
M:        ___________________;
          break;
        ...
      }
    }
```

Answers for question 6 can be filled in here (or put on answer sheet if there is one):

Problem 6A [1pt]: ____ ;

Problem 6B [1pt]: ____ ;

Problem 6C [1pt]: (____)

Problem 6D [1pt]: ____ ;

Problem 6E [2pt]: **for (__________________;___________________;__________________)**

Problem 6F [2pt]: ____ ;

Problem 6G [1pt]: (____)

Problem 6H [1pt]: ____ ;

Problem 6I [1pt]: ____ ;

Problem 6J [1pt]: ____ ;

Problem 6K [1pt]: ____ ;

Problem 6L [2pt]: ____ ;

Problem 6M [1pt]: ____ ;