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

Spring 2020 John Kubiatowicz

Midterm I

February 27th, 2020
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 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.

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	12	
3	19	
4	26	
5	15	
6	10	
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!). Also, answers without an explanation *GET NO CREDIT.*

Problem 1a[2pts]: Threads within the same process *can* share data (since they live in the same address space), but threads in different processes *cannot* share data.

□ 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]: A zombie process is one that got hung up trying to access a disk which is experiencing permanent read errors.

□ True □ False

Explain:

Problem 1d[2pts]: The pipe() system call creates a file and returns file descriptors to read and write to it. Child processes created with fork() share file descriptors with their parents, so both parent and child now have access to this file through the pipe file descriptors.

□ True □ False

Explain:

Problem 1e[2pts]: Dual mode operation is an important mechanism for providing memory-level isolation between processes on the same machine.

□ True □ False

Explain:

Problem 1f[2pts]: One of the first things that Pintos (running on an Intel processor) does after a user executes a system call is to replace the user's stack with the kernel's special "system call stack."

□ 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]: When using a *Monitor*, a programmer must remember to release the lock before going to sleep (waiting) on a condition variable.

□ True □ False

Explain:

Problem 1i[2pts]: TCP/IP makes it possible for a server with IP address X and well-known port Y to simultaneously communicate with multiple remote clients without intermixing response streams.

□ True □ False

Explain:

Problem 2: Multiple Choice [12pts]

Problem 2a[2pts]: What needs to be saved and restored on a context switch between two threads in two different processes (*choose all that apply)*:

A: ☐ Computational registers (integer and floating point)

B: ☐ The Page-table root pointer

C: ☐ Buffered data in streams that have been opened with `fopen()`

D: ☐ Contents of the file descriptor table

E: ☐ Stack pointer register

Problem 2b[2pts]: What are some reasons that overuse of threads is bad (*i.e.* using too many threads at the same time)? (*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: ☐ Most schedulers have a scheduling cost (overhead) that is quadratic in the number threads, thus making the mechanism for choosing the next thread to run extremely expensive.

E: ☐ There might not be enough parallelism in the algorithm to justify the additional threads.

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 the whole machine.

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]: Kernel mode differs from User mode in the following ways (*choose all that apply)*:

A: ☐ The CPL (current processor level) is 0 in kernel mode and 3 in user mode for Intel processors

B: ☐ In Kernel mode, additional instructions become available, such as those that modify page table registers and those that enable/disable interrupts.

C: ☐ Specialized instructions for security-related operations (such as for cryptographic signatures) are only available from Kernel mode.

D: ☐ Control for I/O devices (such as the timer, or disk controllers) are only available from kernel mode

E: ☐ Pages marked as Kernel-mode in the PTEs are only available in kernel mode.

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

A: ☐ Semaphores can be initialized to any 32-bit values in the range -2^{31} to $2^{31}-1$

B: ☐ `Semaphore.V()` increments the value of the semaphore and wakes a sleeping thread if the value of the semaphore is > 0

C: ☐ Semaphores cannot be used for locking.

D: ☐ The interface for `Semaphore.P()` is specified in a way that prevents its implementation from busywaiting, even for a brief period of time.

E: ☐ The pure semaphore interface does not allow querying for the current value of the semaphore.

Problem 3: Readers-Writers Access to Database [19 pts]

```
Reader() {
   //First check self into system
   lock.acquire();
   while ((AW + WW) > 0) {
      WR++;
      okToRead.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // Perform actual read-only access
   AccessDatabase(ReadOnly);

   // Now, check out of system
   lock.acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okToWrite.signal();
   lock.release();
}
```

```
Writer() {
   // First check self into system
   lock.acquire();
   while ((AW + AR) > 0) {
      WW++;
      okToWrite.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // Perform actual 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();
}
```

Problem 3a[2pts]: Above, we show the Readers-Writers example given in class. It used 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.

Problem 3b[2pts]: Let us define the *logical arrival order* by the order in which threads first acquire the monitor lock. Suppose that we wanted the results of reads and writes to the database to be the same as if they were processed one at a time – in their logical arrival order; for example, W_1 W_2 R_1 R_2 R_3 W_3 W_4 R_4 R_5 R_6 would be processed as W_1 W_2 {R_1 R_2 R_3} W_3 W_4 {R_4 R_5 R_6} *regardless of the speed with which these requests arrive.* Explain why the above algorithm does not satisfy this constraint.

Assume that our system uses Mesa scheduling and that condition variables have FIFO wait queues. Also assume that the only reason that a thread will wake up on a condition variable is because of a `signal()` or `broadcast()`.

Below is a sketch of a solution that uses only two condition variables and that *does* return results as if requests were processed in logical arrival order. Rather than separate methods for Reader() and Writer(), we have a single method which takes a "NewType" variable (0 for read, 1 for write):

```
 1.  Lock MonitorLock;                 // Methods: acquire(),release()
 2.  Condition waitQueue, onDeckQueue; // Methods:wait(),signal(),broadcast()
 3.  int Queued = 0, onDeck = 0;       // Counts of sleeping threads
 4.  int CurType = 0, NumAccessing = 0;// About current DataBase Accessors

 5.  Accessor (int NewType) {          // type = 0 for read, 1 for write
 6.     /* Monitor to control entry so that one writer or multiple readers */
 7.     MonitorLock.acquire();

 8.     /* Missing wait condition */
 9.     {  Queued++;
10.        waitQueue.wait();
11.        Queued--;
12.     }

13.     /* Missing wait condition */
14.     {  onDeck++;
15.        onDeckQueue.wait();
16.        onDeck--;
17.     }

18.     /* Missing code */
19.     MonitorLock.release();

20.     // Perform actual data access
21.     AccessDatabase(NewType);

22.     /* Missing code */
23.  }
```

The `waitQueue` condition variable keeps *unexamined* requests in FIFO order. The `onDeckQueue` keeps a *single* request that is currently incompatible with requests that are executing. W*e want to allow as many parallel requests to the database as possible, subject to the constraint of obeying logical arrival order.* Here, logical arrival order is defined by the order in which requests acquire the lock at line #7.

Problem 3c[2pts]: Explain why you might not want to use a "while()" statement in Line #8 – *despite the fact that the system has Mesa scheduling*:

Problem 3d[2pts]: What is the missing code in Line #8? *Hint: it is a single "if" statement.*

Problem 3e[2pts]: What is the missing code in Line #13? *This should be a single line of code.*

Problem 3f[2pts]: What is the missing code in Line #18? *You should not have more than three (3) actual lines of code.*

Problem 3g[3pts]: What is the missing code in Line #22? *You should not have more than five (5) actual lines of code.*

Problem 3h[4pts]: Suppose that condition variables did not have FIFO ordered wait queues. What changes would you have to make to your code? Be explicit (you should not need more than 6 new or changed lines). *Hint: consider assigning a sequentially increasing ID to each thread.*

[This page intentionally left blank]

Problem 4: Pintos Word Count System Calls [26pts]

In this question you will partially implement a word count feature for Pintos! Word count is meant to be a feature for user programs. Here's how it works.

There are 3 syscall handler functions with the following signatures:

```
void wc_open(char *word); // Open a word for the given thread
int wc_inc();             // Increment opened-word's count return result
void wc_close();          // Close the current word
```

The count for a word is shared between all user programs. A user must open a word to get access to it. Each program must open a word before it can increment it and a program can only have one word open. When a word is incremented, the count is incremented for all user programs on the system. When a process exits, it should implicitly close the word. If all processes have closed the word, then it should no longer use any system resources.

Before we start, we'll provide you with a few modifications to the Pintos TCB.

```
struct thread {
   ...
#ifdef USERPROG
   /* You can assume init_thread will initialize this to null */
   word_count_t *wc;
#endif
   unsigned magic;
}
```

Remember, word counts are shared across programs. We will also use 2 global variables which are provided below. You may assume they are initialized in `syscall_init`.

```
/* Global Variables */
struct list word_counts;           // List of all current words
struct lock word_count_list_lock;  // Lock for modifying the list
```

Problem 4a[4pts]: We'll start by defining our word count structure. There will be one of these per word that is currently being counted. Define the word-count structure (shown below). Keep in mind that each `word_count` structure could be open (and shared) by multiple threads at once.

```
struct word_count {




   struct list_elem elem;
} word_count_t;
```

Problem 4b[2pts]: Consider the following skeleton for the system call handler:

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

   switch (args[0]) {
   case SYS_EXIT:
      ...
      thread_exit ();
      break;
   case SYS_WC_OPEN: // open new word
      validate_wc_open(args);
      wc_open(args[1]);
      break;
   case SYS_WC_INC:  // increment opened word's count, return new count


      _____________________________________
      break;
   case SYS_WC_CLOSE:// close word
      wc_close();
      break;

   ...
   }
}
```

Fill in the blank, above, to finish the increment system call handler.

Problem 4c[2pts]: Assuming that the above `syscall_handler()` is written correctly, which of the following MUST be true when the SYS_WC_OPEN case begins to execute? (*choose all that apply)*:

A: ☐ The syscall number is valid

B: ☐ The pointer to the word is valid

C: ☐ The array of characters that constitutes the word is valid

D: ☐ The syscall number is stored on the user stack

Problem 4d[4pts]: Now write the `validate_wc_open()` function. Assume you have access to a function `void validate_region(void *ptr, size_t len)` which correctly validates a fixed size region of memory. You should be able to write this function in six(6) or less lines.

```
void validate_wc_open(uint32_t *args) {









}
```

Problem 4e[4pts]: Define a helper function to aid in opening a word. This function should attempt to find a pre-existing word_count_t structure (if it exists) or allocate a new one. Remember, word count structures are shared across programs. (Hint: this function doesn't need to be thread safe):

```
word_count_t *find_or_add(struct list *list, char *word) {
   // See if word already exists
   for (struct list_elem *e = list_begin(list);
            e != list_end(list); e = list_next(e)) {

      ____________________________________


      if (_________________________________) {
         return wc;
      }
   }
   // Allocate new word structure
   word_count_t *wc = malloc(sizeof(word_count_t));
   wc->word = malloc(strlen(word) + 1);


   ____________________________________


   ____________________________________


   ____________________________________


   ____________________________________


   list_push_front(list, &wc->elem);
   return wc;
}
```

Problem 4f[3pts]: Now, please write `wc_open()`, given your implementation of `find_or_add()`. This function *does* need to be thread-safe. You should be able to write this function in six(6) lines or less. Hint: Don't forget synchronization!

```
void wc_open(char * word) {
   ASSERT(!thread_current()->wc);









}
```

Problem 4g[3pts]: Next, write `wc_inc()`, keeping in mind that this function must be thread safe. You should be able to write this function in six (6) lines or less. Hint: Don't forget synchronization!

```
int wc_inc() {
   ASSERT(thread_current()->wc);









}
```

Problem 4h[4pts]: Finally, write `wc_close()`, keeping in mind that this function must be thread safe. You should be able to write this function in eight (8) lines or less. Remember that a word_count_t structure is deallocated when the last thread closes it. Hint: Don't forget synchronization!

```
void wc_close() {
   ASSERT(thread_current()->wc);









}
```

Problem 5: Scheduling Potpourri [15pts]

Problem 5a[3pts]: Prove that SRTF is the optimal algorithm to reduce average response time (wait time):

Problem 5b[2pts]: Which scheduling algorithm maximizes throughput and why?

Problem 5c[2pts]: Define Hyperthreading and explain why it might improve the performance of two concurrently executing threads over that of one thread.

Problem 5d[2pts]: Which of the following scheduling algorithms can suffer from starvation? (*choose all that apply):*

A: ☐ FIFO Scheduler

B: ☐ Shortest Remaining Time First (SRTF) Scheduler

C: ☐ Lottery Scheduler

D: ☐ Round Robin Scheduler

E: ☐ Priority Scheduler

Problem 5e[6pts]: Suppose we have the following set of processes, with their respective CPU burst times and priorities (here, processes with a higher priority value have higher priority).

Process ID	Arrival Time	CPU Burst (running) Time	Priority
A	0	3	4
B	1	1	2
C	3	5	5
D	6	2	6
E	8	2	1

For each of the three scheduling algorithms, fill in the the table below with the processes that are running on the CPU at each time tick. Assume the following:

- The time quantum for RR is 1 clock tick and when RR quantum expires, the currently running thread is added to the end of the ready list before any newly arriving threads.
- *Assume that context switch overhead is 0, and that new processes are available for scheduling as soon as they arrive.*

Time	RR	SRTF	Priority
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
Average Wait Time			

Problem 6: FileSystems/Sockets API [10pts]

You have been tasked with implementing the Central Galactic Floopy Network (CGFN), to send happy messages to clients (to ensure that clients remain satisfied with their service and leave high reviews). The CGFN trial implementation is simple. It works as a basic TCP/IP network server running on port 555. When clients send requests, the server is guaranteed to accept the requests. The server then writes back a random happy phrase to the client and closes the connection, leaving a happy client.

As their principle engineer, the CGFC requests you to implement the trial run for them. They have already tested a random happy message generator with signature:

```
char *get_happy_message();  // return statically allocated string
```

This function returns a statically allocated message (which does not need to be freed).

Problem 6a[5pts]: Assume no failures occur in any function calls and that reads and writes to sockets will always result in the entire message being received or sent. Complete the missing code in the following (you should only write one statement of code per line):

```
struct addrinfo *setup_address(char *server_name, char *port_name) {
   struct addrinfo *server;
   struct addrinfo hints;
   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;
   hints.ai_flags = AI_PASSIVE;
   getaddrinfo(server_name, port_name, &hints, &server);
   return server;
}
int main() {
   struct addrinfo *server = setup_address(NULL,“555”);
   struct sockaddr *address = server->ai_addr;
   socklen_t addrlen = server->ai_addrlen;

   int serverfd =
      socket(server->ai_family, server->ai_socktype, server->ai_protocol);

   bind(_____________________________________________________);

   listen(_____________________________,1024);

   while (1) {
      char *happy = _____________________________________________;

      _____________________________________________________;

      _____________________________________________________;

      _____________________________________________________;
   }
   close(serverfd);
   return 0;
}
```

Problem 6b[5pts]: To test if the trial implementation works, the CGFC wants trial clients to save the happy message they received in an existing text file in the local file called "/messages/`happy_message.txt`". Assume that the entire contents of a single happy message is no larger than 1000 bytes (including null terminator).

Complete the following client program to contact the floopy server ("galactic.floopy.net") on port 555. Assume no failures occur in any function calls and that reads and writes to sockets will always result in the entire message being received or sent.

You should only write one statement per line.

```
#define BUFFER_SIZE _____________________

int main() {
   char *server_name = _____________________________________;
   struct addrinfo *server = setup_address(server_name,"555");
   struct sockaddr *address = server->ai_addr;
   socklen_t addrlen = server->ai_addrlen;

   int newsockfd =
      socket(server->ai_family, server->ai_socktype, server->ai_protocol);

   connect(newsockfd, address, addrlen);
   send(newsockfd, NULL, NULL, 0);

   // Read happy message from server

   _____________________________________________;

   _____________________________________________;


   // Write happy message to disk

   _____________________________________________;

   _____________________________________________;

   _____________________________________________;

   close(newsockfd);
   return 0;
}
```

[Function Signature Cheat Sheet]

```
/****************************** Threads ******************************/
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 *restrict mutex,
const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);

/***************************** 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);
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);


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!]