

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

John Kubiawicz

Fall 2015

Midterm I  
October 14<sup>th</sup>, 2015  
CS162: Operating Systems and Systems Programming

Your Name:	
SID Number:	
Discussion Section:	

General Information:

This is a **closed book** exam. You are allowed 1 page of **hand-written** notes (both sides). You have 3 hours 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	20	
4	24	
5	20	
Total	100	

[ This page left for  $\pi$  ]

3.141592653589793238462643383279502884197169399375105820974944

**Problem 1: TRUE/FALSE [18 pts]**

In the following, it is important that you *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]:** When you type Ctrl+C to quit a program in your terminal, you are actually sending a SIGINT signal to the program, which makes it quit.

True / False

Explain:

**Problem 1b[2pts]:** The function `pthread_intr_disable()` is a crude but viable way for user programs to implement an atomic section.

True / False

Explain:

**Problem 1c[2pts]:** If the banker's algorithm finds that it's safe to allocate a resource to an existing thread, then all threads will eventually complete.

True / False

Explain:

**Problem 1d[2pts]:** The lottery scheduler can be utilized to implement strict priority scheduling.

True / False

Explain:

**Problem 1e[2pts]:** Locks can be implemented using semaphores.

True / False

Explain:

**Problem 1f[2pts]:** Two processes can share information by reading and writing from a shared linked list.

True / False

Explain:

**Problem 1g[2pts]:** In Pintos, a kernel-level stack can grow as large as it needs to be to perform its functions.

True / False

Explain:

**Problem 1h[2pts]:** Suppose that a shell program wants to execute another program and wait on its result. It does this by creating a thread, calling `exec` from within that thread, then waiting in the original thread.

True / False

Explain:

**Problem 1i[2pts]:** A network server in Linux works by calling `bind()` on a socket, and then calling `listen()` on the socket in a loop to wait for new clients.

True / False

Explain:

## Problem 2: Short Answer [18pts]

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

**Problem 2b[2pts]:** Is it possible for an interrupt handler (code triggered by a hardware interrupt) to sleep while waiting for another event? If so, explain how. If not, explain why not.

**Problem 2c[2pts]:** Why is it important for system calls to be vectored through the syscall table (indexed by an integer syscall number) rather than allowing the user to specify a function address to be called by the kernel after it transitions to kernel mode?

**Problem 2d[3pts]:** Name two advantages and one disadvantage of implementing a threading package at user level (e.g. “green threads”) rather than relying on thread scheduling from within the kernel.

**Problem 2e[2pts]:** List two reasons why overuse of threads is bad (i.e. using too many threads for different tasks). Be explicit in your answers.

**Problem 2f[2pts]:** What was the problem with the Therac-25? Your answer should involve one of the topics of the class.

**Problem 2g[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?

**Problem 2h[3pts]:** What are some of the hardware differences between kernel mode and user mode? Name at least three.

### Problem 3: Boy-Girl Lock [20pts]

A boy-girl lock is a sort of generalized reader-writer lock: in a reader-writer lock there can be any number of readers or a single writer (but not both readers and writers at the same time), while in a boy-girl lock there can be any number of boys or any number of girls (but not both a boy and a girl at the same time). Assume that we are going to implement this lock at user level utilizing pThread monitors (i.e pThread mutexes and condition variables). Note that the assumption here is that we will put threads to sleep when they attempt to acquire the lock as a Boy when it is already acquired by one or more Girls and vice-versa. *You must implement the behavior using condition variable(s). Points will be deducted for any spin-waiting behavior.*

Some snippets from POSIX Thread manual pages showing function signatures are shown at end of this problem. They may or may not be useful.

Our first take at this lock is going to utilize the following structure and enumeration type:

```

/* The basic structure of a boy-girl lock */
struct bglock {
    pthread_mutex_t lock;
    pthread_cond_t wait_var;

    // Simple state variable
    int state;
};

/* Enumeration to indicate type of requested lock */
enum bglock_type {
    BGLOCK_BOY = 0;
    BGLOCK_GIRL = 1;
};

/* interface functions: return 0 on success, error code on failure */
int bglock_init(struct bglock *lock);
int bglock_lock(struct bglock *lock, enum bglock_type type);
int bglock_unlock(struct bglock *lock);

```

Note that the lock requestor specifies the type of lock that they want at the time that they make the request:

```

/* Request a Boy lock */
if (bglock_lock(mylock, BGLOCK_BOY) {
    printf("Lock request failed!");
    exit(1);
}
/* . . . Code using lock . . . */

/* Release your lock */
bglock_unlock(mylock);

```

**Problem 3a[3pts]:** Complete the following sketch for the initialization function. Note that initialization should return zero on success and a non-zero error code on failure (e.g. return the failure code, if you encounter one, from the various synchronization functions). *Hint: the state of the lock is more than just “acquired” or “free”.*

```
/* Initialize the BG lock.
 *
 * Args: pointer to a bglock
 * Returns: 0 (success)
 *         non-zero (errno code from synchronization functions)
 */
int bglock_init(struct bglock *lock) {

}
}
```

**Problem 3b[5pts]:** Complete the following sketch for the lock function. Think carefully about the state of the lock; when you should wait, when you can grab the lock.

```
/* Grab a BG lock.
 *
 * Args: (pointer to a bglock, enum lock type)
 * Returns: 0 (lock acquired)
 *         non-zero (errno code from synchronization functions)
 */
int bglock_lock(struct bglock *lock, enum bglock_type type) {

}
}
```



**Problem 3c[5pts]:** Complete the following sketch for the unlock function.

```
/* Release a BG lock.
 *
 * Args: pointer to a bglock
 * Returns: 0 (lock acquired)
 *         non-zero (errno code from synchronization functions)
 */
int bglock_unlock(struct bglock *lock) {

}
}
```

**Problem 3d[2pts]:** Consider a group of “nearly” simultaneous arrivals (i.e. they arrive in a period much quicker than the time for any one thread that has successfully acquired the BGlock to get around to performing `bglock_unlock()`). Assume that they enter the `bglock_lock()` routine in this order:

B<sub>1</sub>, B<sub>2</sub>, G<sub>1</sub>, G<sub>2</sub>, B<sub>3</sub>, G<sub>3</sub>, B<sub>4</sub>, B<sub>5</sub>, B<sub>6</sub>, B<sub>7</sub>

How will they be grouped? (Place braces, namely “{ }” around requests that will hold the lock simultaneously). This simple lock implementation (with a single state variable) is subject to starvation. Explain.

**Problem 3e[5pts]:** Suppose that we want to enforce fairness, such that Boy and Girl requests are divided into phases based on arrival time into the `bglock_lock()` routine. Thus, for instance, an arrival stream of Boys and Girls such as this:

B<sub>1</sub>, B<sub>2</sub>, G<sub>1</sub>, G<sub>2</sub>, G<sub>3</sub>, G<sub>4</sub>, B<sub>3</sub>, G<sub>5</sub>, B<sub>4</sub>, B<sub>5</sub>

will get granted in groups such as this:

{B<sub>1</sub>, B<sub>2</sub>}, {G<sub>1</sub>, G<sub>2</sub>, G<sub>3</sub>, G<sub>4</sub>}, {B<sub>3</sub>}, {G<sub>5</sub>}, {B<sub>4</sub>, B<sub>5</sub>}

Explain what the minimum changes are that you would need to make to the `bglock` structure to meet these requirements and sketch out what you would do during `bglock_init()` and `bglock_lock()` and `bglock_unlock()` routines. You do not need to write actual code, but should be explicit about what your `bglock` structure would look like and how you would use its fields to accomplish the desired behavior.

## Assorted POSIX Thread Manual Snippets for Problem 3

PTHREAD\_MUTEX\_DESTROY(3P): initialization/destruction of mutexes

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

PTHREAD\_MUTEX\_LOCK(3P): use of mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

PTHREAD\_COND\_DESTROY(3P): initialization/destruction of condition variables

```
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_init(pthread_cond_t *restrict cond,
                     const pthread_condattr_t *restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

PTHREAD\_COND\_TIMEDWAIT(3P): sleeping on condition variables

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);

int pthread_cond_wait(pthread_cond_t *restrict cond,
                     pthread_mutex_t *restrict mutex);
```

PTHREAD\_COND\_BROADCAST(3P): signaling of threads waiting on condition variables

```
int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_signal(pthread_cond_t *cond);
```

## Problem 4: Scheduling and Deadlock [24 pts]

**Problem 4a[3pts]:** What is priority inversion and why is it an important problem? Present a priority inversion scenario in which a lower priority process can prevent a higher-priority process from running (assume that there is no priority donation mechanism):

**Problem 4b[3pts]:** How does the Linux CFS (“Completely Fair Scheduler”) scheduler decide which thread to run next? What aspect of its behavior is “fair”? (You can ignore the presence of priorities or “nice” values in your answer):

<pre> void main (void) { 1   thread_set_priority(10); 2   struct lock a, b, c; 3   lock_init(&amp;a); 4   lock_init(&amp;b); 5   lock_init(&amp;c); 6   lock_acquire(&amp;a); 7   lock_acquire(&amp;b); 8   lock_acquire(&amp;c); 9   printf("1"); 10  thread_create("a", 15, func, &amp;a); 11  printf("6"); 12  thread_create("b", 20, func, &amp;b); 13  printf("2"); 14  thread_create("c", 25, func, &amp;c); 15  lock_release(&amp;c); 16  lock_release(&amp;a); 17  lock_release(&amp;b); 18  printf("!"); } </pre>	<pre> void func(void* lock_) { 19  struct lock *lock = lock_;  20  lock_acquire(&amp;lock); 21  lock_release(&amp;lock);  22  printf("%s", thread_current()-&gt;name);  23  thread_exit(); } </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Problem 4c[2pts]:** Consider the above Pintos test that exercises your priority scheduler. Assume that no priority donation has been implemented. What does it output to the terminal? Is the output affected by priorities in any way? Explain.

**Problem 4d[5pts]:** Next, assume that the code from (4c) is executed utilizing priority donation. Fill in the following table to detail execution. This table includes 7 columns as following:

- 1) The current executing thread
- 2) Which line this thread was executing when it yielded
- 3) To which thread it yielded
- 4-7) The priorities of each thread (N/A if a thread is not created or has exited)

thread_current()	Line at which yielded	Thread which it yielded to	Main	a	b	c
main	10	a	10	15	N/A	N/A

**Problem 4e[2pts]:** What is printed according to the order of execution in (4d)? Is the output affected by priorities in any way? Explain.

**Problem 4f[4pts]:**

Suppose that we have the following resources: A, B, C and threads T1, T2, T3, T4. The total number of each resource is:

Total		
A	B	C
12	9	12

Further, assume that the processes have the following maximum requirements and current allocations:

Thread ID	Current Allocation			Maximum		
	A	B	C	A	B	C
T1	2	1	3	4	9	4
T2	1	2	3	5	3	3
T3	5	4	3	6	4	3
T4	2	1	2	4	8	2

Is the system in a safe state? If “yes”, show a non-blocking sequence of thread executions. Otherwise, provide a proof that the system is unsafe. Show your work and justify each step of your answer.

**Problem 4g[3pts]:**

Assume that we start with a system in the state of (4f). Suppose that T1 asks for 2 more copies of resource A. Can the system grant this if it wants to avoid deadlock? Explain.

**Problem 4h[2pts]:** Assume that a set of threads ( $T_1, T_2, \dots, T_n$ ) contend for a set of non-preemptable resources ( $R_1, R_2, \dots, R_m$ ) that may or may not be unique. Name at least two techniques to *prevent* this system from deadlocking:

[ This page intentionally left blank ]



## Problem 5: Address Translation [20 pts]

**Problem 5a[3pts]:** In class, we discussed the “magic” address format for a multi-level page table on a 32-bit machine, namely one that divided the address as follows:

Virtual Page # (10 bits)	Virtual Page # (10 bits)	Offset (12 bits)
-----------------------------	-----------------------------	---------------------

You can assume that Page Table Entries (PTEs) are 32-bits in size in the following format:

Physical Page # (20 bits)	OS Defined (3 bits)	0	Large Page	Dirty	Accessed	Nocache	Write Through	User	Writeable	Valid
------------------------------	---------------------------	---	---------------	-------	----------	---------	------------------	------	-----------	-------

What is particularly “magic” about this configuration? Make sure that your answer involves the size of the page table and explains why this configuration is helpful for an operating system attempting to deal with limited physical memory.

**Problem 5b[2pts]:** Modern processors nominally address 64-bits of address space both virtually and physically (in reality they provide access to less, but ignore that for now). Explain why the page table entries (PTEs) given in (5a) would have to be expanded from 4 bytes and justify how big they would need to be. Assume that pages are the same size and that the new PTE has similar control bits to the version given in (5a).

**Problem 5c[2pts]:** Assuming that we reserve 8-bytes for each PTE in the page table (whether or not they need all 8 bytes), how would the virtual address be divided for a 64-bit address space? Make sure that your resulting scheme has a similar “magic” property as in (5a) and that all levels of the page table are the same size—with the exception of the top-level. How many levels of page table would this imply? Explain your answer!

**Problem 5d[3pts]:** Consider a multi-level memory management scheme using the following format for *virtual addresses*, including 2 bits worth of segment ID and an 8-bit virtual page number:

Virtual seg # (2 bits)	Virtual Page # (8 bits)	Offset (8 bits)
---------------------------	----------------------------	--------------------

Virtual addresses are translated into 16-bit *physical addresses* of the following form:

Physical Page # (8 bits)	Offset (8 bits)
-----------------------------	--------------------

Page table entries (PTE) are 16 bits in the following format, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory):

Physical Page # (8 bits)	Kernel	Nocache	0	0	Dirty	Use	Writeable	Valid
-----------------------------	--------	---------	---	---	-------	-----	-----------	-------

1) How big is a page? Explain.

2) What is the maximum amount of physical memory supported by this scheme? Explain

**Problem 5e[10pts]:** Using the scheme from (5d) and the Segment Table and Physical Memory table on the next page, state what will happen with the following loads and stores. Addresses below are virtual, while base addresses in the segment table are physical. If you can translate the address, make sure to place it in the “Physical Address” column; otherwise state “N/A”.

The return value for a load is an 8-bit data value or an error, while the return value for a store is either “ok” or an error. If there is an error, say which error. Possibilities are: “**bad segment**” (invalid segment), “**segment overflow**” (address outside segment), or “**access violation**” (page invalid/attempt to write a read only page). A few answers are given:

Instruction	Translated Physical Address	Result (return value)
Load [0x30115]	0x3115	0x57
Store [0x10345]	0x3145	Access violation
Store [0x30316]		
Load [0x01202]		
Store [0x31231]		
Store [0x21202]		
Load [0x11213]		
Load [0x01515]		

**Segment Table (Segment limit = 3)**

Seg #	Page Table Base	Max Pages in Segment	Segment State
0	0x2030	0x20	Valid
1	0x1020	0x10	Valid
2	0xF040	0x40	Invalid
3	0x4000	0x20	Valid

**Physical Memory**

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x0000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
0x0010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
....																
0x1010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
0x1020	40	03	41	01	30	01	31	01	00	03	00	00	00	00	00	00
0x1030	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0x1040	10	01	11	03	31	03	13	00	14	01	15	03	16	01	17	00
....																
0x2030	10	01	11	00	12	03	67	03	11	03	00	00	00	00	00	00
0x2040	02	20	03	30	04	40	05	50	01	60	03	70	08	80	09	90
0x2050	10	00	31	01	F0	03	F0	01	12	03	30	00	10	00	10	01
....																
0x3100	01	12	23	34	45	56	67	78	89	9A	AB	BC	CD	DE	EF	00
0x3110	02	13	24	35	46	57	68	79	8A	9B	AC	BD	CE	DF	F0	01
0x3120	03	01	25	36	47	58	69	7A	8B	9C	AD	BE	CF	E0	F1	02
0x3130	04	15	26	37	48	59	70	7B	8C	9D	AE	BF	D0	E1	F2	03
....																
0x4000	30	00	31	01	11	01	F0	03	34	01	35	00	43	38	32	79
0x4010	50	28	84	19	71	69	39	93	75	10	58	20	97	49	44	59
0x4020	23	03	20	03	E0	01	E1	08	E2	86	28	03	48	25	34	21
....																
0xE000	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55
0xE010	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A
....																
0xF000	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0xF010	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00
0xF020	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	11
....																

[ This page intentionally left blank ]

[ This page left for scratch ]