NAME ____

FINAL EXAM, COMP 3080 OPERATING SYSTEMS
May 5, 2016

Print your name on this exam. Please check off the boxes below to indicate which assignments you have passed in and any additional information I need to know regarding your current assignment status. Print your answers (or write very clearly) in the space provided. If you need more space write on the back of the page, and indicate a continuation in the main answer space. There are 10 questions with points as shown for a total of 100 points. Please keep your answers BRIEF and to the point. System call prototypes and scratch sheets are included in a second handout; they do not have to be passed in with the rest of the exam. The exam will run from 11:30 AM until 2:30 PM.

Please check off the appropriate boxes to indicate your assignment status:

1. Parent - child process info	submitted	☐
	not submitted	☐
2. 2-Way pipe to UNIX command	submitted	☐
	not submitted	☐
3. Shared memory donuts	submitted	☐
	not submitted	☐
4. Pthreads donuts	submitted	☐
	not submitted	☐
5. First fit, Best fit, Buddy memory	submitted	☐
	not submitted	☐
6. UNIX File System, INODE listing	submitted	☐
	not submitted	☐

Any additional information I should know about your homework assignments:

10 POINTS

1. Using the buddy system of memory allocation, fill in the **starting addresses** for each of the following **memory allocation requests** as they enter an initially empty memory region which has a memory size of 2^{16} (64K) bytes. Addresses run from 0 to 64k -1, and can be given in K form (i.e. location 4096 = 4K.) Assume that when memory is allocated from a given block-size list, the available block of memory closest to address 0 (shallow end of memory) is always given for the request. Give the <u>**address**</u> of each allocation in the space provided below <u>**if the allocation can be made**</u>, or write in **"<u>NO SPACE</u>" if the allocation <u>cannot</u> be made** at the time requested.

TIME	JOB REQUESTING	JOB RETURNED	REQUEST SIZE(BYTES)
1	A		12K
2	B		3K
3	C		17K
4		A	
5	D		5K
6	E		4K
7		B	
8		D	
9	F		13K
10	G		2K
11		E	
12		C	
13		G	
14	H		15K

ANSWERS

Request A at 0

Request B at 16

Request C at 32

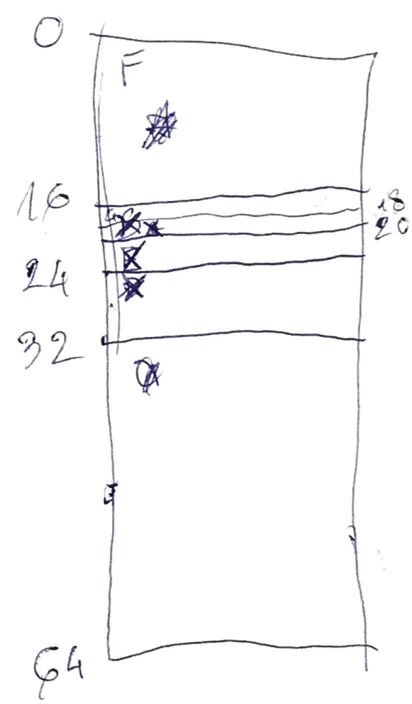
Request D at 24

Request E at 20

Request F at 0

Request G at 16

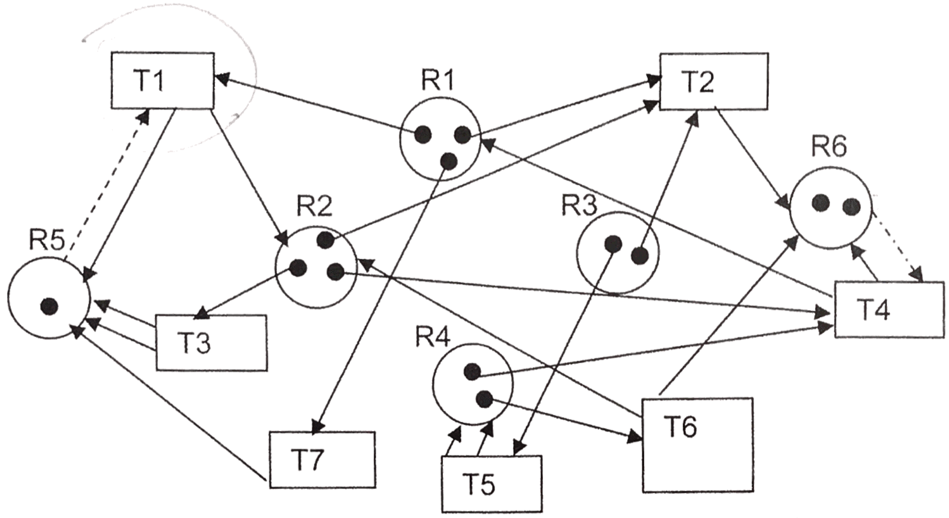
Request H at 16



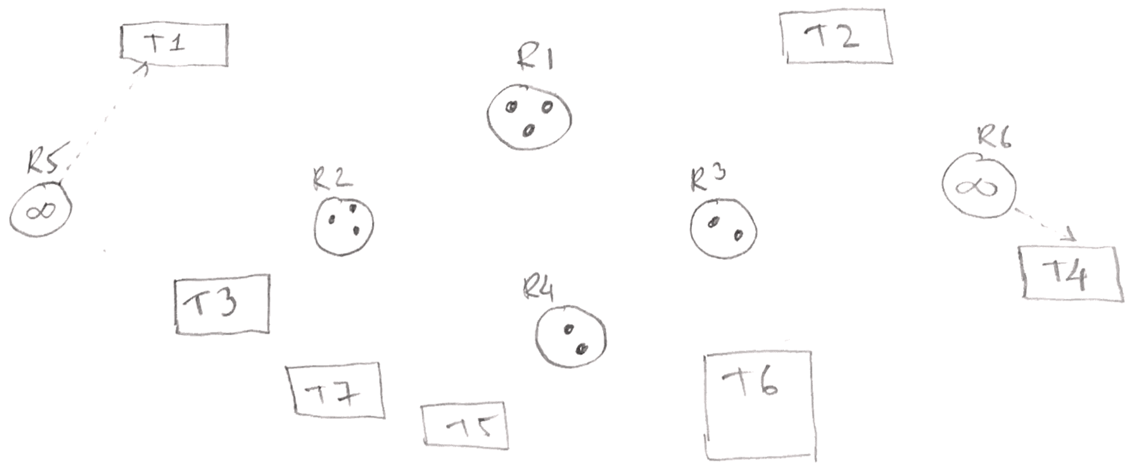
7 POINTS

2. The following resource allocation graph shows the state of a **7 thread** system using **6 types of resources** at a particular instant.

A. Using graph reduction, determine whether any deadlock exists, and if there **is deadlock** indicate the **process(es) and resources involved.** You must draw the **final reduced graph** whether or not there is a deadlock.



DRAW THE FINAL, REDUCED RESOURCE GRAPH HERE AND STATE YOUR CONCLUSION ABOUT ANY DEADLOCK ON THE LINE BELOW:



YOUR CONCLUSION: There is NO DL.

B. What is the **complexity** of the graph reduction algorithm for the general system described above **(i.e. O(?)).** Is such an algorithm **more complex**, **less complex** or of the **same complexity** as an algorithm for a system of **re-usable resources only** ? **Explain.**

COMPLEXITY, etc. ??

General graph O(mn!), more complex than reusable-only which is O(mn)

10 POINTS

3. Both the **UNIX** and the **WindowsXP** operating systems require that **some unique thread** be in the **RUN** state for each **processor** in the system at all times.

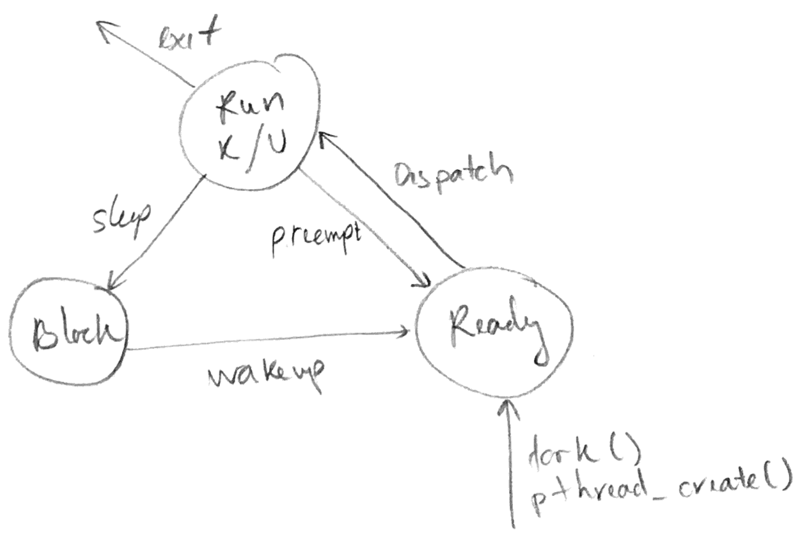
A. If all available user threads are **blocked** at a time when the current running thread **enters the context switch to block as well**, how are these systems able to find a thread to run ??

Thread priority.

B. When a UNIX or WindowsXP thread is **running in user mode** it is constrained to its own private address space. **All threads**, from time to time however, **must leave** their address spaces and **execute kernel code** in the kernel's address space. In **what ways** do threads leave their private address space and execute in the **kernel's address space** ??

System call and exceptions.

C. Draw the **thread state diagram** we've been looking at in class, using a **labeled circle** for each **thread state** and **directed labeled arcs** for each possible transition.



10 POINTS

4. The following simple program (headers not shown) named `th_run` compiles and links (using –lpthread) with no errors, but on one particular execution on a multi-core Linux machine it produced the following output but **never completed** (i.e. no shell prompt ever prints again until a **ctl C** is typed in):

```
bash-3.00$ ./th_run
THREAD 1 IS RUNNING
THREAD 5 IS RUNNING
```

SOURCE CODE FOR th_run:

```
#define N 5
pthread_mutex_t lock;
void    *th(void *arg){
    pthread_mutex_lock(&lock);
    printf("THREAD %d IS RUNNING\n", *((int *)arg));
    return NULL;
} // end th

int  main(int argc, char *argv[]){
    pthread_t        thread_id[N];
    int              arg[N];
    int              i;
    pthread_mutex_init(&lock, NULL);
    pthread_mutex_lock(&lock);
    for(i=0; i<N; i++){
      arg[i] = i + 1;
      if(pthread_create(&thread_id[i],NULL, th, (void*)(&arg[i]))!=0){
                  perror("thread create failed ");
                  exit(1);
      }
    }
    for(i=0; i<N; i++){
      pthread_mutex_unlock(&lock);
      pthread_join(thread_id[i], NULL);
    }
    printf("\nProgram with %d threads is done\n", N);
} // end main
```

Problem 4 continued on next page:

Problem 4 continued:

A. Provide a **detailed explanation** of why this program **never finishes**:

The main thread is hung in join, since it joins in create sequence, but threads may finish in any sequence (here 1 is followed by 5). If it can't join, it cannot unlock the lock for the next thread

B. If we run this program **repeatedly**, could we ever expect a **particular execution** to complete ? **Explain:**

Yes, if in a given threads could finish in any order, we could expect that they would occasionally complete in create sequence and satisfy the join loop.

5 POINTS

5. The following complete program shows a parent process creating **a single pipe and child.** As you can see, the child is programmed to run the `ls -l` command and redirect its standard output to the pipe. The parent will read the child's output from the pipe and write it all out to a local file that the parent has opened for this purpose. (Assume all necessary include files are available, line numbers are for your reference)

```
1.  int main(void){
2.     int pchan[2], pid, nread, file_channel;
3.     char buf[100];
4.     if(pipe(pchan) == -1){
5.       perror("pipe");
6.       exit(1);
7.     }
8.     switch( pid = fork() ){
9.       case -1: perror("fork");
10.               exit(2);
11.      case  0: close(1);
12.               if( dup(pchan[1] ) != 1 ){
13.                   perror("dup");
14.                   exit(3);
15.               }
16.               execlp( "ls", "ls", "-l", NULL );
17.               perror("exec");
18.               exit(4);
19.      default: if((file_channel =
20.                     open("/tmp/data", O_CREAT|O_WRONLY, 0600)) == -1){
21.                   perror("open");
22.                   exit(5);
23.               }
24.               while(nread =  read(pchan[0], buf, 100)){
25.                   write(file_channel, buf, nread);
26.               }
27.               close(file_channel);
28.               if(close(pchan[1])== -1 || close(pchan[0])== -1){
29.                   perror("close");
30.                   exit(4);
31.               }
32.               wait(NULL);
33.               return(0);
34.     }
35. }
```

A. In the above example, even though there are **NO programming errors** and **NO system call errors,** the **parent** process **never completes. Explain** why the parent never finishes, and show **where** (using line numbers) and **what code changes** are necessary **for the parent to complete.**

The child never close the write side of the pipe.

Add " close(pchan[1]) after execlp (before line 17)

B. When a singly threaded process receives an **unblocked SIGSEGV** signal, most often the process **will terminate**, but there is a way that the thread in the process can take steps to **arrange for the process to continue executing**. **Explain** what steps the thread must take so the thread in the process can continue executing when an **unblocked SIGSEGV** is delivered to the process.

Set up a single handler

.0 POINTS

6. For **Linux Ext style** file system using the **i-NODE** organization discussed in class (e.g. for Ext3 there are 15 pointers; 12 direct plus 1st, 2nd and 3rd level indirect):

A. What is the size limit on a file created in a **2048 Byte (2 KB or 4 disk sectors) per allocation unit Ext file system**, assuming that **4 bytes** are required for all allocation unit pointers ?

NOTE:

- You may express the answer in KBytes, MBytes, GBytes or powers of 2 as well as decimal, octal or hex, **just make it clear what kind of answer** you provide.

(12 direct * 2k) + ($512^1 \cdot 2k$) + ($512^2 \cdot 2k$) + ($512^3 \cdot 2k$) = 268 MB

FILE SIZE LIMIT IS: ______________________

B. To support a file of the **max size** that you calculated above in **Part A**, the amount of **meta data** required in the form of **pointers needed within index blocks** for the **2 KB sized allocation unit described above** is on the order of **.1 %** of the file data size. If the **same size file** was allocated using a file system with a **64 KB allocation unit** (instead of a 2 KB **allocation unit**), how would this affect the **amount of index block meta data** needed to support this file (i.e. would we need **more**, the **same**, or **less** total meta data ?). **Explain your answer**.

Same

C. One of the major **new features** that the Ext3 file system introduced (as an enhancement to Ext2 functionality) was **journaling**. While journaling can help **minimize data loss** across power-fail crashes, **what is the most valuable feature of journaling ? Explain.**

journaling provide

10 POINTS

7. The following information depicts a system consisting of 3 threads (**a, b, and c**) and **10 tape drives** which the threads must share. The system is currently in a **"safe"** state with respect to deadlock:

thread	max tape demand	current allocation	outstanding claim
a	4	2	2
b	6	3	3
c	8	2	6

Following is a sequence of events, each of which occurs a short time after the previous event, with the first event occurring at time one (t(1)). The exact time that each event occurs is not important except that each is later than the previous. I have marked the times **t(1), t(2),** etc. for reference. Each event either **requests or releases** some tape drives for one of the threads. If a system must be kept **"safe"** at all times, and if a request can only be met by providing all the requested drives, indicate the time at which each request will be granted using a **first-come-first-served** method for any threads that may have to wait for their requests (e.g. request 5 granted at t(x)), or indicate that a request will not be granted any time in the sequential time listed. (Note: if a thread releases one or more drives at time(x) that a waiting thread needs, that waiting thread will get its drives **at that time(x)** provided the system remains in a safe state. Put your final answers in the space provided below.

TIME	ACTION	
t(1)	request #1	a requests 1 drive
t(2)	request #2	c requests 2 drives
t(3)	release	b releases 1 drive
t(4)	request #3	a requests 1 drive
t(5)	release	c releases 2 drives
t(6)	release	a releases 1 drive
t(7)	request #4	b requests 3 drives
t(8)	request #5	c requests 2 drive
t(9)	release	a releases 2 drives

ANSWERS:

Request #1 granted at T1

Request #2 granted at T 3

Request #3 granted at T 4

Request #4 granted at never

Request #5 granted at T 8

18 POINTS

8. Consider the following details regarding a collection of UNIX objects:

Process A
- Euid 0
- Egid 1

Process B
- Euid 300
- Egid 20

Process C
- Euid 310
- Egid 20

Process D
- Euid 320
- Egid 30

Process E
- Euid 330
- Egid 20

Directory **/**
- uid 0
- gid 1
- Permissions `d rwx r-x r-x`

Directory **/abc**
- uid 310
- gid 20
- Permissions `d rwx r-x ---`

File **/abc/file_one**
- uid 310
- gid 20
- Permissions `- r-s rwx rwx`

File **/abc/file_two**
- uid 310
- gid 30
- Permissions `- --- r-- rw-`

File **/abc/file_three**
- uid 330
- gid 20
- Permissions `- rw- r-- rw-`

A. Can process D **write** on **/abc/file_three** ? **Explain.**

Yes, the write bit is on for public group

B. Can process A **write** on **/abc/file_one** ? **Explain.**

Yes, process A has Euid/uid 0 (root)

C. Can process C use the **chmod 644 /abc/file_two** shell command successfully ? **Explain.**

Yes, it is the owner of the file

D. Can process B use the **ls /abc** shell command successfully ? **Explain.**

Yes, group ID match, has read access

Problem #8 continued on the next page ➔

Problem #8 continued:

E. Assume that **/abc/file_one** is a program which attempts to create a file in **/abc** called **file_x**. If process **B** uses the **execl() system call** to load and run **/abc/file_one** , can process **B** succeed in creating **file_x in /abc** ?? **Explain.**

No, it does match group ID, but that group doesn't have write permissions for file /abc

F. Can process A **send** process E a **termination signal** ? **Explain.**

Process A can do whatever it wants due to being root

G. Can process E **send** process C a **termination signal** ? **Explain.**

Only process A (root) can kill other process.

H. Can process B **write** on **/abc/file_one** ? **Explain**

Yes, same group

I. Can process C **write** on **/abc/file_two** ? **Explain**

No, the owner has no read and no write.

10 POINTS

9. In class we examined the need for **concurrent execution paths** like a **consumer** and a **producer** to synchronize their access to a **shared ring buffer**. Below is a **global ring buffer** accessible to a **single producer** thread and **multiple consumer** threads (just as in assignment #3). You must write a solution that uses **event counters and sequencers (only if needed)** using the format shown:

The following types and operations are available:

```
ec_t  ec_name;            /* declare EC initialized with 0  */
seq_t seq_name;           /* declare SEQ initialized with 0 */
void  await (ec_t * , int);   /*  await event            */
void  advance (ec_t *);       /*  advance EC             */
int   ticket (seq_t *);       /*  get a SEQ ticket       */
```

You must declare however many **event counters** and **sequencers** you need to solve this problem efficiently. The shared ring buffer is an array of **10 integer locations**. The single producer thread must execute a **forever loop** using a random number function (like **random()**) to create an integer, and then place the integer into an appropriate slot in the shared ring buffer **when it's safe to do so**. The consumer threads must each execute a **forever loop** taking numbers out of the shared ring buffer and printing them to standard out (with a **printf()** type function) **when it's safe to do so**. Using **C code**, make the necessary **EC** and **SEQ** (if needed) declarations in the box below and then write the **producer function** and the **consumer function** as described above.

GLOBAL TO PRODUCER AND CONSUMER THREADS:

DECALRE YOUR EC(s) AND SEQ(s) (IF NEEDED) HERE:

```
ec_t pEC , cEC ;
seq_t cSEQ ;
```

ALREADY DECLARED GLOBAL OBJECT SHOWN BELOW:

```
int          ring buf[50];
```

WRITE PRODUCER FUNCTION HERE

```
void producer() {
  int in = 0;
  while(1)
    await(&pEC, in -10+1);
    ring_buff[in % 10]=random()
    in =(in +1);
    advances (&cEC)
}
```

WRITE CONSUMER FUNCTION HERE

```
void consumer() {
  int val, t ;
  while(1) {
    t = ticket (cSEQ);
    await ( pEC , t) ;
    await ( &cEC , t+1);
    val = ring_buf[t % 10];
    // print val somewhere
    advance (&pEC)
}.
```

10 POINTS

10. Let ω = **2 3 1 3 2 4 3 2 4 5 1 6 7 5 6 7 4 5 6 7 2 1**, be a **page reference stream** for a given system. You are asked to work with **the Least Frequently Used algorithm** (referred to as the NFU algorithm in our book) below. You must determine the **number of page faults** that will occur for the stream shown above with an **LFU replacement algorithm** for a memory with **3 physical frames** and a memory with **5 physical frames**. (Please use the **grid help sheet** on the next page for this problem.)

A. Assuming the primary memory is initially empty, how **many page faults** will the given reference stream have using the page replacement algorithm **LFU** for :

1. A memory with **3 physical frames** 17/3
2. A memory with **5 physical frames** 15/5

B. In our discussion of **memory objects**, we described some objects as being **anonymous** (e.g. stack objects) and some as being **file based** (e.g. text objects). **Explain how** the pages of an **anonymous** object are **managed differently** from a **file based** object with respect to **page replacement of dirty pages that must be backed up for possible re-use.**

Problem #10 continued on the next page →

Problem #10 continued:

Grid Help Sheet

LFU

ω	2	3	1	3	2	4	3	2	4	5	1	6	7	5	6	7	4	5	6	7	2	1	
1	2	3	1	3	2	4	3	2	4	5	1	6	7	5	6	7	4	5	6	7	2	1	
2		2	3	1	3	2	2	3	2	2	2	2	2	2	2	2	2	4	5	6	7	2	
3			2	2	1	3	4	4	3	3	3	3	3	3	3	3	3	2	4	5	6	7	
4						1	1	1	1	4	4	1	1	1	5	6	7	3	2	4	5	6	
5										1	5	4	4	4	1	5	6	7	3	2	4	5	
6												5	6	7	4	1	5	6	7	3	3	4	
7													5	6	7	4	1	1	8	1	1	3	
c1																							
c2				1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
c3					1	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	17/3
c4																							
c5											1	1	1	1	1	1	1	1	1	1	2	2	15/5
c6																		1	2	3	3	3	
c7														1	2	3	4	4	4	4	4	5	
∞	1	2	3	3	3	4	4	4	4	5	5	6	7	7	7	7	7	7	7	7	7	7	

1 ||

2 ||||

3 |||

4 |||

5 |||

6 |||

7 |||