ECE 391 Final, Fall 2012

Saturday, Dec 15, 2012

Name: _____

NetID: _____

Discussion Section:

◯ 11am (Adam)
◯ 1pm (Adam)
◯ 2pm (Zak)
◯ 3pm (Puskar)

- **Be sure that your exam booklet has 13 pages.**
- **Write your net ID at the top of each page.**
- **This is a closed book exam.**
- **You are allowed three** 8.5×11" **sheet of notes.**
- **No other materials are allowed**
- **Answers to questions must be precise, but need not be closed form. (E.g.,** $2 \cdot 3 + \frac{7 \cdot 11}{3}$ **is acceptable.)**
- **Absolutely no interaction between students is allowed.**
- **Show all of your work.**
- **Don't panic, and good luck!**

Page	Points	Score
3	11	
4	9	
5	8	
6	5	
7	6	
9	8	
10	6	
11	12	
12	9	
Total:	74	

Question 1: Short Answer (28 points)

Please answer concisely. If you find yourself writing more than a sentence or two, your answer is probably wrong.

(2) (a) What is the difference between masking a signal and ignoring it?

Mask means signal will be pended and will be handled later

Ignore means the signal has been handlered and handler does nothing

(3) (b) In what cases is it useful to force a program to receive a particular signal, even though the program has asked that the signal be masked out?

When an exception has occured, we must use signal to kill program, or it will be undefined behavior or dead lock

(4) (c) When does a signal get *generated*? Check all that apply.

- ✓ During a `kill` system call
- ✓ When an exception occurs
- ◯ During a timer interrupt
- ◯ When the receiving process gets scheduled
- ✓ After a terminal event (e.g., Ctrl-C)

(2) (d) When does a signal get *delivered*? Circle best option.

A. During a `kill` system call
B. After a timer interrupt
C. After any interrupt
D. When the receiving process gets scheduled

(4) (e) Describe one advantage and one disadvantage of lazy algorithms, such as those used for demand paging.

(2) (f) Explain what this code does:

```
movl %cr3,%eax
movl %eax,%cr3
```

(3) (g) When would you want to use the above code? Explain why.

(4) (h) When does the CPU set the accessed and dirty flags in the page table? And how are they used by the operating system?

(2) (i) Which of these is typically the smallest?

A. Rotational latency
B. Seek latency
C. Transfer latency

(2) (j) The use of block groups in EXT2 is primarily to reduce:

A. Rotational latency
B. Seek latency
C. Transfer latency
D. Internal fragmentation

Question 2: MP3 (11 points)

(5) (a) Your MP3 partner implemented the following system call entry routine:

```
1  syscall_enter:
2          pushfl
3          cli
4          decl %eax          # eax=eax-1
5          cmpl $10,%eax
6          jae badcall
7          cmpl $0,%eax
8          jb badcall
9          pushl %ebp
10         pushl %edi
11         pushl %esi
12         pushl %edx
13         pushl %ecx
14         pushl %ebx
15         call *jump_table(,%eax,4)
16         popl %ebx
17         popl %ecx
18         popl %edx
19         popl %esi
20         popl %edi
21         popl %ebp
22 exit:   popfl
23         iret
24 badcall:
25         movl $-1, %eax
26         jmp exit
```

You realize that some of the code in here is redundant and can be eliminated. Explain what you would remove and why. (You may use the next page to continue your answer)

(2) (b) What's the best way to avoid synchronization conflicts that may arise when accessing the MP3 file system?

?

(4) (c) To implement terminal switching, your partner created a separate buffer for each terminal. When switching terminals, the contents of the screen gets copied into the previous terminal's buffer, and the contents of the new terminal's buffer gets copied onto the screen. To make this work, you also need to update process page tables of the process during a terminal switch. Explain what you need to change and why. Be detailed.

video memory map

Question 3: Synchronization (14 points)

Below is an implementation of synchronization using wait queues.

```
char pipe_buf[4096];
unsigned int head, size;
DECLARE_MUTEX(pipe_lock);
DECLARE_WAIT_QUEUE_HEAD(readQ);
DECLARE_WAIT_QUEUE_HEAD(writeQ);

void init_pipe(void) {
        down(&pipe_lock);
    head = 0;
    size = 0;
    up(&pipe_lock);
}

/* read single char */
void write(char c) {
        down(&pipe_lock);

        DECLARE_WAIT(wait);
        while (1) {
                /* add self to wait queue */
                prepare_to_wait(&writeQ, &wait, TASK_UNINTERRUPTIBLE);
                if (size < 4096)
                        break;
                /* go to sleep (potentially) */
                schedule();
        }
        /* remove self from queue */
        finish_wait(&writeQ, &wait);

        pipe_buf[(head + size) % 4096] = c;
        size++;

        wake_up(&readQ);        /* wake up one sleeping reader */

        up(&pipe_lock);
}
```

```
char read(void) {
    char c;
        DECLARE_WAIT(wait);
        down(&pipe_lock);

        while (1) {
                /* add self to wait queue */
                prepare_to_wait(&readQ, &wait, TASK_UNINTERRUPTIBLE);
                if (size > 0)
                        break;
                /* go to sleep (potentially) */
                schedule();
        }
        /* remove self from queue */
        finish_wait(&readQ, &wait);

        c = pipe_buf[head];
        head = (head + 1) % 4096;
        size--;

        wake_up(&writeQ);        /* wake up one sleeping writer */

        up(&pipe_lock);

        return c;
}
```

(3) (a) There is a critical mistake in how the wait queues are used. What is it?

(2) (b) Why is a `while(1)` loop used? Explain how a writer could end up going to sleep multiple times.

(3) (c) `wake_up` wakes up a single waiting task (i.e., it is similar to `notify`). What would happen if `wake_up_all` (equivalent to `broadcast`) were used instead?

(6) (d) Re-implement the scheme using two semaphores. Complete the code below.

```
char pipe_buf[4096];
int head, size;
DECLARE_MUTEX(pipe_lock);
struct semaphore readsem, writesem;

void init() {
        down(&pipe_lock);
        head = 0;
        size = 0;
        sema_init(&readsem, 0);
        sema_init(&writesem, 4096);
        up(&pipe_lock);
}

void write(char c) {









}

char read(void) {









}
```

Question 4: Scheduling (21 points)

Consider the following set of processes:

Process	Arrival time	Run time
A	0	4
B	2	3
C	4	5
D	6	1

(8) (a) Show the scheduling order of these processes under the following schedulers:

- First-in First-out (FIFO)
- Shortest remaining time first (SRT)
- Round-robin with a quantum of 2 time units (RR)

Assume that the context switch overhead is 0 and, all else being equal, new processes should be scheduled first. A process arriving at time t can first be scheduled for slot t

Time Slot	FIFO	SRT	RR
0	A	A	A
1	A	A	A
2	A	B	B
3	A	B	B
4	B	B	A
5	B	A	A
6	B	D	C
7	C	A	C
8	C	C	B
9	C	C	D
10	C	C	C
11	C	C	C
12	D	C	C
13			
14			
15			
16			
17			
18			

(4) (b) Calculate the turnaround time of each process, as well as the average.

Scheduler	A	B	C	D	Average
FIFO					/4
SRT					/4
RR					/4

(6) (c) For each scheduler, explain one advantage of using it.

i. FIFO

ii. SRT

iii. RR

(3) (d) Explain the tradeoff in selecting the round-robin quantum size. In particular, what benefit do you get from increasing or decreasing the quantum?

Synchronization API reference

`spinlock_t lock;`	Declare an uninitialized spinlock
`spinlock_t lock1 = SPIN_LOCK_UNLOCKED;` `spinlock_t lock2 = SPIN_LOCK_LOCKED;`	Declare a spinlock and initialize it
`void spin_lock_init(spinlock_t* lock);`	Initialize a dynamically-allocated spin lock (set to unlocked)
`void spin_lock(spinlock_t *lock);`	Obtain a spin lock; waits until available
`void spin_unlock(spinlock_t *lock);`	Release a spin lock
`void spin_lock_irqsave(spinlock_t *lock, unsigned long& flags);`	Save processor status in `flags`, mask interrupts and obtain spin lock (note: flags passed by name (macro))
`void spin_lock_irqrestore(spinlock_t *lock, unsigned long flags);`	Release a spin lock, then set processar status to `flags`
`struct semaphore sem;`	Declare an uninitialized semaphore
`static DECLARE_SEMAPHORE_GENERIC (sem, val);`	Allocate statically and initialize to `val`
`DECLARE_MUTEX (mutex);`	Allocate on stack and initialize to one
`DECLARE_MUTEX_LOCKED (mutex);`	Allocate on stack and initialize to zero
`void sema_init(struct semaphore *sem, int val);`	Initialize a dynamically allocated semaphore to `val`
`void init_MUTEX(struct semaphore *sem);`	Initialize a dynamically allocated semaphore to one.
`void init_MUTEX_LOCKED(struct semaphore *sem);`	Initialize a dynamically allocated semaphore to zero.
`void down(struct semaphore *sem);`	Wait until semaphore is available and decrement (P)
`void up(struct semaphore *sem);`	Increment the semaphore
`struct rw_semaphore rwsem;`	Declare an uninitialized reader/writer semaphore
`void init_rwsem(struct rw_semaphore *rwsem);`	Initialize a reader/write semaphore
`void down_read(struct rw_semaphore *rwsem);` `void down_write(struct rw_semaphore *rwsem);`	Obtain the semaphore for reading/ writing
`void up_read(struct rw_semaphore *rwsem);` `void up_write(struct rw_semaphore *rwsem);`	Release the semaphore after reading/ writing