NAME: ______________________________ (FIRST NAME FIRST) **SCORE**: _______

COSC 4330 SECOND MIDTERM NOVEMBER 3, 2008

*This exam is **closed book**. You can have **one page** of notes. UH expels cheaters.*

1. *Questions with short answers:* (6×5 points)

a) How do you pass a ***linked list*** to a ***remote procedure***?

You store the linked list into an array and pass it to the remote server along with instructions describing how to rebuild the list.

b) What is the major disadvantage of the ***round-robin*** scheduling policy?

The round-robin scheduling policy does not discriminate between interactive processes, which need quick access to the CPU, and CPU-bound processes, which just need a lot of CPU time. In order to provide quick access to the CPU for all processes, it tends to use small time slices that result in too many context switches.

c) What is the difference between ***blocking sends*** and ***non-blocking sends***?

A blocking send does not return until the receiving process has actually received the message.

A non-blocking send returns as soon as the message has been accepted for later delivery by the OS.

d) What happens when a process executes a **`signal()`** system call?

When a process executes a signal() system call, it sets up a procedure to follow the next time it receives a signal.

e) What is the major disadvantage of the ***at-most-once*** semantics for remote procedure calls?

It does not prevent partial executions of remote procedures.

f) What is the main advantage of ***streams*** over ***datagrams***?

Streams ensure that all messages will arrive in the order they have been sent, without errors and without duplicates.

2. Which of the following statements apply to (a) kernel-supported threads, (b) user level threads and (c) all threads? (5 points per correct line)

	Kernel-supported	***User-level***	***Both types***
They do not require kernel modifications.	_____	__*X*__	_____
They share the address space of their parent.	_____	_____	__*X*__
They allow the kernel to allocate several processors to the threads sharing an address space.	__*X*__	_____	_____
Switching between threads in the same address space requires two context switches	__*X*__	_____	_____

3. How **processes** will be required by the execution of the following program? (5 points)

```
main(){
     fork();
     printf("Hi!\n");
     fork();
     printf("How are you?\n");
} // main
```

Executing the program will require exactly __**4**__ processes.

4. When should the System V Release 4 scheduler: (3×5 points)

a) ***Decrease*** the priority of a process?

Each time a process returns to the ready queue having exhausted its time slice.

b) ***Increase*** that priority?

- Each time a process returns to the ready queue from the waiting state.
- Whenever a process has been in the ready queue for more than ts_maxwait and not getting any CPU time.

5. Complete the following fragment of code in such a manner that the standard output of the process will be redirected to the pipe `thispipe`. (2×5 points)

```
int thispipe[2];

pipe(thispipe);

close(1);

dup(thispipe[1]);

close(thispipe[0]);close(thispipe[1]);
```

6. Why is `fork()` a ***very expensive system call***? (5 points) Was this always true? (5 points)

Fork() is a very expensive system call because it copies into the child address space the entire contents of its parent address space.

This cost was tolerable as long as UNIX was running on 16-bit architectures because these architectures limited the size of process address space to 64 KB. The switch to 32-bit architectures has changed that.

7. How can you simulate a ***blocking send*** using ***only non-blocking sends*** and ***non-blocking receives***? (10 points)

(Simulating a blocking send using a non-blocking send means that the sender process must wait until the destination process has acknowledged the message. Since we are to use non-blocking receives, we must use busy waits)

Sender side:
```
// sends message
nbsend(destination, message, nb1);
// waits for reply
while (nbreceive(destination, reply, &nb2) == NO_MSG);
```

Destination side:
```
// waits for message
while (nbreceive(sender, message, &nb1) == NO_MSG);
// sends reply
nbsend(sender, reply, nb2);
```