List of Topics for Review

This list not intended to be a complete, compressive list. There may be topics listed here that are not covered by an exam question and other topics that aren't covered here that will be covered by an exam question. However, this does cover most of the topics that will be covered on the exam.

Review Topics List:

I. C Programming

1. What is the * operator? What does it do?
2. What is the & operator? What does it do?
3. What's the difference between char c[80] and char *c?
   …what about when they're used in sizeof()?
4. What do common C functions do? How are they called?
   String Functions: strcpy(), strlen(), strcmp(), strcat()
   Binary Functions: memcmp(), memcpy(), memset()
   Memory Functions: malloc(), remalloc(), free()
   I/O Functions: printf(), fgets(), read(), write(), fread(), fwrite(), fopen(), fclose()
   Thread-related Functions: pthread_create(), pthread_join(), pthread_detacth()
   Semaphore-related Functions: sem_init() /_destroy(), sem_post(), sem_wait()
   Signal-related Functions: sigwait(), sigaction(), sigemptyset(), …
   … and others.
5. What is NULL?
6. Understand how to trace and write pointer code.
7. What's the difference between a stack and a heap variable? What about global and static variables?
8. What is stdout? What is stdin? Is stderr even real?

II. Operating Systems

1. What is an operating system?
2. What is the difference between a function call and a system call?
3. A call such as malloc() may make a system call some times, but other times doesn't. Why is that?
4. What is the difference between a process and a thread? What state information is shared between all threads in a process?
5. How do a user-thread and a kernel-thread differ?
6. What are the different states that a process may be in within in operating system?
7. What is a zombie thread? What is an orphan process?
8. What are the return values of fork()?
9. Why is fork() nearly always used in conjunction when an exec() call is going to be used?
10. What parts of memory are retained when fork() is called? …when exec() is called?
11. How do you abandon a thread? How do you wait for it to finish? How do you exit from the current thread you're in?
12. What does it mean for a function to be thread-safe?
13. What is reentrant?

III. Scheduling

1. Why do processes need to be scheduled?
2. How do you schedule with a FIFO policy? FCFS? SJF? Round robin?
3. What does it mean for a scheduling algorithm to be preemptive?
4. How does bounded wait apply to scheduling? What is starvation? What is the convey effect?
5. What is response time? Initial response time? What other metrics do we use?
6. Which scheduling algorithm minimizes average initial response time? Waiting time? Total response time?
7. How does Round Robin differ in nature when it has a small quantum vs. a large quantum?
8. Why is SJF/PSJF hard to implement in real systems?
9. Why is FIFO not considered a good algorithm in interactive systems?

IV. Synchronization / Semaphores

1. When is synchronization needed?
2. What is the difference between a semaphore and a mutex? Can you use mutexs in place of a semaphore?
3. What is a critical section?
4. What is deadlock? What other properties are there in relation to synchronization?
5. How does semaphores and testandset() differ?
6. What are algorithms learned in class to deal with synchronization?
7. How do you define a POSIX semaphore in C? What are the two main function calls to use it? How to you clean up the memory associated with a semaphore?

V. Signals / Clock

1. What are signals? Why are they needed?
2. What is the signal sent with Ctrl+C at a terminal?
3. Many programmers often use SIGALARM. How is that signal generated?
4. What is a signal mask? What is a signal set?
5. Are the signals that cannot be caught? If so, what is the need for those signals in an OS?
6. It is documented that read() is not signal-safe. Why?
7. What is a standard C function call that is signal-safe?
8. What call sets up a signal to be sent at set intervals?
9. What does the time() function return? Why is there a Y2K+38 problem?
10. What does gettimeofday() do? Why can it return the same value twice in a row on some operating systems?
11. How does timer resolution effect alarm()? If a timer is scheduled for (2.01 * RESOLUTION), what is the actual effect?

Section I: Processes and Threads

1.1) Consider the following code:

```
01:  int i;
02:  for (i = 0; i < 5; i++)
03:    fork();
04:  printf("Count me!");
```

How many different types does the text "Count me!" become printed to stdout?

Answer:	

1.2) Answer the following questions with either "process"/"processes", "user-level threads"/"user-level thread", or "kernel-level threads"/"kernel-level thread":

A single **[a]**_____ requesting I/O will block all other **[b]**_____ in the same **[c]**_____ from executing.

A **[d]**_____ doesn't not require a context switch when switching tasks.

A **[e]**_____ cannot access the heap variables of another **[e]**.

1.3) Explain the following POSIX thread calls:

[a]: pthread_create()

[b]: pthread_join()

[c]: pthread_exit()

Section II: Scheduling

2.1) Given the following set of tasks:

	Arrival Time	Job Length	Priority
Task #1:	0	4	Low
Task #2:	2	8	High
Task #3:	3	1	Medium
Task #4:	6	3	Low
Task #5:	8	5	High

Assuming that ties are broken by the earliest arriving job, complete the scheduling of the given tasks given the following scheduling algorithms:

[a]: First In First Out (FIFO)																				
Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Task:																					

[b]: Non-Preemptive Shortest Job First (SJF)																					
Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Task:																					

[c]: Non-Preemptive Priority Scheduling (PRI)																					
Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Task:																					

[d]: Preemptive Shortest Job First (PSJF)																					
Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Task:																					

[e]: Preemptive Priority Scheduling (PPRI)																					
Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Task:																					

Section III: Memory Allocation

3.1) Consider the following code:

```
01: #include <stdlib.h>
02:
03: int main(int argc, char** argv)
04: {
05:    int* a = malloc(sizeof(int));
06:
07:    void* b = &a;
08:    void* c = a;
09:    void* d = *a;
10:
11:    return 0;
12: }
```

Let us assume that the stack memory starts at the address 1000 (decimal) and counts up while the heap memory starts at the address 2000 (decimal) and counts down. Assuming no padding or extra memory allocations has been done before reaching Line #5:

	Where is the variable stored in memory?	What is the memory address that the variable is pointing to?
a		
b		
c		
d		

3.2) Consider the following structure:

```
01 typedef struct __myStruct
02 {
03     int myValue, yourValue;
04     char myChar;
05     char *yourString;
06     double *yourDouble;
07 } myStruct;
```

The following assumptions are given:

```
A1 sizeof(char) = 1
A2 sizeof(int) = 4
A3 sizeof(float) = 4
A4 sizeof(void*) = 8
A5 sizeof(double) = 8
```

What is the return value of sizeof(myStruct)?

Answer:	