CPS 310 midterm exam #2, 4/10/2017

Your name please: _____ NetID:_____

Sign for your honor: _____

P0	/50
P1	/35
P2	/15
P3	/25
P4	/25
P5	/50
P6	/50
P7	/50
	/300

Answer all questions. Please attempt to confine your answers to the space provided. Allocate your time carefully: you have 75 minutes plus grace. This exam has 300 points, but it will be normalized to weight both midterms evenly.

P0. A little Unix (50 points)

These questions deal with basic Unix system calls. Answer each question by listing one or more system calls. Please refrain from any added explanation: just list the system calls in the box.

1. What calls does your shellcode use in your p2 attack?
2. What calls increment the reference count on an I/O object?
3. What calls decrement the reference count on an I/O object?
4. What calls does a server process use to obtain a socket to exchange data with a client that connects to the server?
5. What calls can set or change the user ID (uid) of a process?
6. What calls do not return (if they succeed)?
7. A few system calls create and return a new I/O descriptor. But there is one call that returns **two** descriptors. Which is it?
8. What calls free virtual memory resources for a process?
9. What calls allow a parent process to operate on a child?
10. What calls take a port number as an argument?

P1. Sweet Dreams (35 points)

These questions deal with blocking: what causes a process (more precisely its main thread) to sleep? In each case the blocking results from an operation on an object. List the operation(s) and object(s). There may be more than one pair. Please just list them in the boxes: don't add any explanation. For Unix cases the operation is a system call.

1. What makes a cat sleep? (cat, the standard Unix utility)	
2. How about a netcat, what makes it sleep?	
3. Your p2 attack shellcode?	
4. Your p2 attack shell (on the server)?	
5. The middle process of a three-process shell pipeline?	
6. The main/UI thread of an Android app?	
7. An RPC server thread (e.g., with RMI or binder)?	

P2. Forks (15 points)

Consider the C/Unix code fragment below using the fork() system call. Assume that it runs within a program, with no errors. What does it print? Explanation optional.

```
for (j =0; j < 4; j++) {
    if (fork() == 0)
        printf("%d", j);
}
```

P3. Back of Napkin (25 points)

These questions deal with quantitative performance for servers that meet the assumptions of the basic queue models and performance laws that we discussed, including Little's Law. Consider a simple server S that behaves according to these assumptions, on average, in aggregate (i.e., the server behaves as a single "service center"). Presume that S does not saturate. Answer each question with a **number** (e.g., a factor, ratio, or percentage).

1. What is the utilization level at which the average response time of S is 100x more than it is when S is idle?

2. If my goal is to double the delivered throughput of S, by what factor must I increase its offered load?

3. How does this increase in offered load for S in (2) affect the utilization of S?

4. If S is CPU-bound, and I speed up the code by a factor of two (so it takes half as much CPU time on average to serve a request), how does this change affect the peak throughput (peak rate) of S?

5. Consider a fixed offered load (workload) for which S ran at 80% utilization before I improved the code. How much does the improvement in (4) reduce the average response time of S under this fixed workload?

P4. Android (25 points)

These questions have short full-credit answers (a word or phrase) in the "lingo" of Android. Keep it short enough to fit.

1. How does Android determine the intent filters for a component?

2. How does Android determine which application component(s) should receive a fired explicit intent?

3. How does Android determine which application component(s) should receive a fired implicit intent?

4. Under what conditions does Android create a process for the receiving component in (2) or (3)?

5. Under what conditions does Android destroy an application process for a component?

P5. A little synchronization (50 points)

This problem asks you to implement a common bit of kernel synchronization. Suppose T is a thread that requests an I/O operation and waits for it to complete, and H is an interrupt handler that wakes each thread T when T's operation is complete. T calls issue() to place an I/O operation request on a queue and await completion. H calls iodone() to wake T when an issued operation is complete.

Show how to implement issue() and iodone() using monitors (mutex+CV) and using semaphores. As always, any kind of pseudocode is fine as long as its meaning is clear. For this problem you may assume that condition variables are FIFO (as in p1t) and also that semaphores are FIFO. You may assume that all I/O operations are completed by H in FIFO order—the order in which they are issued. You may assume basic queue primitives: don't implement those.

Monitors:

```
issue(request r)
{




}
```

```
iodone(request r)
{




}
```

Semaphores:

```
issue(request r)
{




}
```

```
iodone(request r)
{




}
```

P6. C sells: deja vu (50 points)

Consider the following C code, which is divided into two separate C source files as shown. It builds and runs without errors (header file #includes are omitted). Answer the questions below. Feel free to write down any assumptions you make (in the space at the bottom).

main.c

```
int main()
{
  char* buffer = (char*)malloc(8);
  fill(buffer, 7);
  printf("%s\n", buffer);
  exit(0);
}
```

fill.c

```
void
fill(char* buf, int i)
{
  for(int j=0; j<i-1; j++)
     buf[j] = (char)(((int)'b')+j);
  buf[j] = '/0';
  return;
}
```

stack

1. Draw the stack contents at the point of the **return** statement in fill(), in the box above.
2. What is the value of buffer[] just before the program exits? i.e., what is its output?
3. How many external symbol references does the linker resolve to link this program? List them in the box→
4. What symbols does the linker resolve from the C standard library?
5. How much memory space (in pages) does this program use on its stack when it runs?
6. How much memory space (in pages) does it use in its global data segment?
7. How much memory space (in pages) does it use on its heap?
8. How much memory space (in pages) does it use in its code segment?
9. How many system call traps does it take?
10. How many page faults does it take? (Assume the OS allocates page frames only on demand.)
11. How large is a virtual memory page (in bytes)?

P7. Raft (50 points)

Assume a Raft replica group that is operating correctly, unless the question specifies otherwise. By "operating correctly" I mean that the assumptions of Raft (and classical **Consensus**) are always met, even if some portion of the group has failed/partitioned. These questions are yes/no with limited elaboration (except the last two).

1. Could a replica ever add entries to its log with a term lower than the current term?
2. Could a replica ever have entries in its log with a term higher than the current term?
3. Could a leader in a term be unaware that some entry from a previous term has committed?
4. Could the replica group lose a committed update if a majority of servers fail?
5. Could the state machines of different replicas diverge if a majority of servers fail?
6. Could the replica group ever have two leaders at once? (A *leader* is a replica in the Leader state.)
7. Could a leader continue as a leader on the minority side of a network partition?
8. If an attacker subverts (takes over, e.g., as in p2) a follower, could it take over leadership of the group?
9. If no replicas in the group have failed, does the slowest follower limit the rate at which the group commits entries?
10. In log repair mode (e.g., after a view change), could a follower ever overwrite an entry that committed in a previous term?

View change. Suppose that at the time of a view/term change (a change in leadership) in Raft to a new leader/primary P, exactly one of the replicas R that voted for P (i.e., the members of P's quorum) has voted for a value *v* for slot (index) *s* in a previous view.

11. Is it possible that *v* has committed for *s*?
12. Is it possible that *v* has not committed for *s*?
13. Is it possible that some other value *w* has committed for *s*?
14. What should P propose for *s* in the new view?
15. Suppose now that another replica R_2 has also voted in the past for a value *w* for *s*. What should P propose for *s* in the new view?