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. **See grading notes on next page.**

1. What calls does your shellcode use in your p2 attack?	socket, bind, listen, **accept**, dup*, **exec***
2. What calls increment the reference count on an I/O object?	**open**, **fork**, socket, pipe, dup*, …
3. What calls decrement the reference count on an I/O object?	**close**, **exit**, dup2
4. What calls does a server process use to obtain a socket to exchange data with a client that connects to the server?	socket**, bind,** listen**, accept**
5. What calls can set or change the user ID (uid) of a process?	**setuid**, **exec*** (for a program, with setuid bit set), fork
6. What calls do not return (if they succeed)?	**exit**, exec* (returns to process, but not program)
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?	**pipe**
8. What calls free virtual memory resources for a process?	**exit**, **exec***, munmap
9. What calls allow a parent **process** to operate on a child?	(fork)**, wait*,** kill
10. What calls take a port number as an argument?	**bind, connect**

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

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.

Each of these P0 questions was worth 5 points.

My grade scrawlings follow the standard scheme:

- **Check mark**: you got it. (might have a minus or a plus)
- **Horizontal slash**: half credit. Means the answer had part of what I was looking for but was missing one or more elements. In general I ignored extra calls in the list (may have crossed some out). A few wrong answers didn't hurt you if you your answers had at least something I was looking for.
- **Backward slash**. Probably no credit.

The essentials are in bold on the answer key on the previous page. Generally you get full credit if you hit all the bold ones. More is better.

The point of these P0 questions is to gauge your understanding of the core Unix system calls (a dozen or so). The focus is on the essentials, e.g., I don't require you to distinguish between variants of dup*, exec*, wait*.

In general, on this question and others, I may have rounded up a bit to limit any nitpicking. I generally score questions in increments of 5 points. You will see that your scores on the page 1 box may have "+" or "-". A "+" means I could have scored you higher on that question: I might have been a little hard, but it should balance with scores for other questions. A "-" means I could have hit you a little harder. The +/- usually balance out. But a few of you have received a little bump of points at the end if there were more + than – in your scores.

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)	**read** from stdin (or a file), **write** to stdout
2. How about a netcat, what makes it sleep?	**connect** to server, **read**/write on socket, read stdin, write stdout
3. Your p2 attack shellcode?	**accept** on socket to await connection from attacker
4. Your p2 attack shell (on the server)?	**wait** on children, read stdin, write stdout
5. The middle process of a three-process shell pipeline?	**read** stdin (left pipe), **write** stdout (right pipe)
6. The main/UI thread of an Android app?	await **next event** (UI event or upcall), block for no other reason
7. An RPC server thread (e.g., with RMI or binder)?	await **next request**, and perhaps block while handling it

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);
}
```

The fork syscall creates a child process and returns zero in the child and some non-zero value (process ID) in the parent. So only the children print. Since each child is a clone of its parent, each child continues exactly as its parent (after the child prints). So there is one child for j=0, two for j=1, four for j=2, and eight for j=3. There can be some non-determinism in the order of prints due to concurrency: not mentioning that cost 5 points (for a 10+).

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? 99%
   R = D/(1-U), and R = 100D → 1-U == 0.01 → U=0.99
2. If my goal is to double the delivered throughput of S, by what factor must I increase its offered load? 2x (double)
   Throughput X == offered load (until the server saturates)
3. How does this increase in offered load for S in (2) affect the utilization of S? 2x (double)
   U = XD, and D is constant but X has doubled.
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? 2x (double)
   U = XD, and U=1 at peak throughput. Since we cut D in half, we can double X before we hit U=1.
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? 6x, i.e R is now 1/6 what it was.
   U=XD → at fixed X halving D also halves U → U =0.4. Then just grind it out with R = D/(1-U). About 20% did so and got close enough. Don't forget to halve D as well!

P4. Android (25 points)

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?

The filters are declared at app install time in the app's **manifest**.

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

An explicit intent **names the target** component explicitly/directly by its classname.

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

Search the **intent filters** for components with matching subscriptions and adequate permission.

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

The target component is **not yet active**, and its app has **no process yet**. Then Android creates a process and activates the component within that process. It does not create a process if an app process is already running and can activate the component.

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

The component has **no references** (e.g., a Service component with no bound clients), and there are no other active components in the app process. Then Android might destroy the process if it needs the memory. There is no concept of an app "exiting": apps run as long as they are in active use, and fade away when they become inactive.

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)
{

    lock();
    enqueue(r);
    wait(r.cv);   /* per-request CV, all in same lock */
    unlock();



}
```

```
iodone(request r)
{

     signal(r.cv);



}
```

Semaphores:

```
issue(request r)
{

     mx.P();  /* mx starts at 1 */
     enqueue(r);
     mx.V();
     r.P();      /* r starts at 0 */

}
```

```
iodone(request r)
{

    r.V();



}
```

P5. A little synchronization (50 points)

Note on grading. For this question I did not sweat the details too much. I looked for proper locking and proper wakeup behavior. Monitors and semaphore solutions are each 25 points.

For **monitors**, that means a lock on at least the left side for the wait(), and a signal or broadcast on the right side. The main way to lose points was to leave state operations unlocked, or to forget how to use monitors. I didn't even bother to enforce that the signal goes to the right thread. Superfluous (e.g., waiting on the right side) was OK, but if you put code in it had to make sense.

For **semaphores**, I looked for a P (decrement) to block on the left and a V (increment) to wake up on the right, and (ideally) a per-request semaphore to coordinate the wakeup. State operations were unnecessary for this problem but if you have them they must be locked, which is easy to do with a second binary semaphore acting as a mutex.

A few students lost points because the P and V on the wakeup semaphore were covered by the mutex. That was a 10 point error. It causes a deadlock, because semaphores don't release any locks when they sleep, in contrast to monitors. The good news is that semaphores, like monitors, are inherently safe against missed wakeups. It doesn't matter if the P or V comes first: the V always wakes up the thread blocked in P, because semaphores "remember" any posted Vs: a V can wake up a P that happens later. So you can do the sleep (P) and wakeup (V) outside the mutex. In contrast, a signal/notify on a condition variable can never affect a subsequent wait(), and the wait() must always be covered by the mutex.

Some students confused P and V. I was OK with that if the answer made sense when I switched them.

Some students regurgitated a producer/consumer example from the slides in some form. That was mostly close enough, but the issue has to P at the end, after enqueueing. I knocked 10 points for that one too.

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). **See note on grading on next page. Here are the answers:**

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

```
[top: low addresses]
[fill frame]
int j
saved FP
saved RA
int i
char* buf
[main() frame]
char* buffer
...
```

↑

Full credit for giving me all elements of these two frames. I didn't care too much about the order or direction. Maybe a few points off for missing something. I wanted to see a saved RA but a missing FP is OK.

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? 'bcdefg', with trailing null
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? printf, exit, malloc
5. How much memory space (in pages) does this program use on its stack when it runs? 1
6. How much memory space (in pages) does it use in its global data segment? 0
7. How much memory space (in pages) does it use on its heap? 1
8. How much memory space (in pages) does it use in its code segment? 1 (or 2 or…)
9. How many system call traps does it take? mmap/sbrk for malloc, write for printf, exit → 3+
10. How many page faults does it take? (Assume the OS allocates page frames only on demand.) 3+…
11. How large is a virtual memory page (in bytes)? 4KB or 8KB or…

```
fill
printf
exit
malloc
(main)
```

P6. C sells: deja vu (50 points)

Grading: 10 points for the stack, and 4 points off for each missed question. The total is 54, but nobody got a negative score. ;-)

I was a little bit tougher on partial credit for this question, since it was present in much the same form on an earlier exam. I did make some changes to the question, and if you gave me the "old" answers I didn't give you the points. This repeated question does give me some good information: if you didn't get it this time then I know that you chose to skip a high-priority study source.

For the "external symbols" and "standard library" questions, I really wanted to see both printf() and exit(). If you regurgitated the list from the old solution, without the exit(), I gave you the big back slash. (But count the points, I rounded up to give you something.) Note that the full list includes malloc() too. The "buffer" variable is not on the list: it is not a global symbol and its value (a pointer to the buffer where the data resides) is passed by value as a parameter, so the linker does not have to resolve that reference.

If you put the contents of buffer[] on the stack, a big slash for you. It resides on the heap--- allocated with *new* or *malloc*. A surprising number of answers had it on the stack(maybe 10%). In the last version of the question the buffer resides in global data, and there was no heap data. In this version there is heap data but no global data.

One thing that is a little weird is a lot of students listed constants (like "%s\n" or 'b') as external symbols. These are constants. They are resolved by the C compiler into values stored in a section/segment for immutable global data. All the linker has to do is combine all these sections from the different object files into a single section in the output executable, and then fix the offsets accordingly anywhere in the code that references them.

Page sizes are platform-dependent, but I wanted to see reasonable and specific answers. This is a detail, but it is fundamental and it is on the slides. So that was 4 points. Note that pages are much bigger than the code/data in any of the segments (global data, text, stack, heap), and if a segment uses any part of a page it must use the full page. So the answer to the "how many pages" questions 5-8 is always 1 (or 0) for any reasonable page size. The "how many faults" question 10 presumes that every page is faulted exactly once on first reference.

As the exam was given, I had forgotten to null-terminate the buffer string. An extra + for pointing it out, and noting that the printf might print garbage and/or segfault. But few did, and I didn't put any points on that. (And malloc was also a "new".)

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? Yes, during log repair.
2. Could a replica ever have entries in its log with a term higher than the current term? No
3. Could a leader in a term be unaware that some entry from a previous term has committed? Yes. It knows the entry for sure though.
4. Could the replica group lose a committed update if a majority of servers fail? Yes, if they all fail and forget (or don't recover).
5. Could the state machines of different replicas diverge if a majority of servers fail? No, never; always safe.
6. Could the replica group ever have two leaders at once? (A *leader* is a replica in the Leader state.) Yes, but at most one has a quorum.
7. Could a leader continue as a leader on the minority side of a network partition? Yes, but it can't commit entries without a quorum.
8. If an attacker subverts (takes over, e.g., as in p2) a follower, could it take over leadership of the group? Yes, just declare as leader.
9. If no replicas in the group have failed, does the slowest follower limit the rate at which the group commits entries? No, quorum commits.
10. In log repair mode (e.g., after a view change), could a follower ever overwrite an entry that committed in a previous term? Yes, leader could send an AppendEntries that includes it, and it will overwrite, although the value itself cannot change.

View change. Suppose that at the time of a view/term change (a change in leadership) in Raft to a new leader/primary P, one of the replicas R that voted for P has voted for a value *v* for slot (index) *s* in a previous view, and no other replica has voted for *s*.

The set of replicas that voted for P as leader (including P itself) is its *quorum*.

11. Is it possible that *v* has committed for *s*? Yes, replicas outside this quorum could have voted for s:v too.
12. Is it possible that *v* has not committed for *s*? Yes
13. Is it possible that some other value *w* has committed for *s*? No, or at least one of the replicas in this quorum would have voted for it.
14. What should P propose for *s* in the new view? v, which will be in P's own log (for Raft).
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? v or w, whichever has the highest term (i.e., for Raft, whatever is in P's own log).

These are worth 3 points off for each incorrect answer. 5 points free.

Note that questions (4) and (8) do explicitly violate the assumptions of Raft, and so may produce results that "cannot happen" in Raft if the algorithm's assumptions are true.