CPS 310 midterm exam #1, 2/17/2017

Your name please: ___________________________________________  NetID: ____________

Sign for your honor: ___________________________________________

Answer all questions. Please attempt to confine your answers to the boxes provided. If you don’t know the answer to a question, then just say something else relevant for partial credit. Allocate your time carefully: you have 75 minutes plus grace.

P0. p0 me (T/F, 20 points)

These questions deal with memory allocation, e.g., as in programs that use the p0 heap manager and its versions of malloc() and free() routines. Indicate whether each statement is true or false by placing a T or F to the → left ← of the statement. You can place additional notes or conditions to the right.

Answer here.

T 1. Padding can help to reduce external fragmentation.
F 2. Padding can help to reduce internal fragmentation.
T 3. There is no external fragmentation for data allocated on the stack.
F 4. There is no internal fragmentation for data allocated on the stack.
T 5. malloc aligns the requested size of a heap block (by rounding up as needed) to ensure that blocks at higher addresses are aligned.
T 6. The compiler ensures that data items within a heap block are stored at aligned addresses, if and only if the data portion of the block starts at an aligned address.
F 7. If a program or thread accesses a free heap block, the access might generate a segmentation fault.
T 8. If a program or thread frees a heap block that it has already freed, then free might return an error, or it might not return an error.
F 9. Pointers to heap blocks are always stored on the stack.
T 10. A buggy application program can cause a correct heap manager to crash in malloc or free.
P1. Git’s baa-ack (T/F, 20 points)

Alice and Bob communicate over a network and use git (the program) and gitlab (the service) to develop software. They want to keep their communications safe from Molly, a malicious adversary. Assume that Alice and Bob have logged into gitlab with their NetIDs and have set things up so that they can use git without a password, as you have done for this class. Indicate whether each statement is true or false by placing a T or F to the left of the statement. You can place additional notes or conditions to the right. **Note:** we say that a running process “knows” a value if it handles that value at some point during the protocol.

Answer here (best 10 of 11)

<table>
<thead>
<tr>
<th></th>
<th>Statement</th>
<th>T/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1. Authorization for gitlab is based on asymmetric cryptography.</td>
<td>T/F</td>
</tr>
<tr>
<td>F</td>
<td>2. Authentication for gitlab is based on symmetric cryptography.</td>
<td>T/F</td>
</tr>
<tr>
<td>T</td>
<td>3. Git knows Alice’s private key.</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>4. Git knows Alice’s NetID password.</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>5. Gitlab knows Alice’s private key.</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>6. Gitlab knows Alice’s NetID password.</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>7. Alice can send a private (secret) message to Bob by encrypting it with her private key, i.e., so Molly cannot read her message. <em>Anyone with Alice’s public key can decrypt it, so the message is not secret.</em></td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>8. Alice can send an integrity-protected message to Bob by encrypting it with her public key, i.e., so Molly cannot modify Alice’s message without detection.</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>9. Alice may create many keypairs for herself and use them for different reasons if she chooses.</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>10. Alice gives her public key to gitlab only after gitlab knows her NetID.</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>11. If Molly discovers Alice’s private key, she can decrypt messages encrypted with Alice’s private key. <em>Only if she has Alice’s public key too.</em></td>
<td>F</td>
</tr>
</tbody>
</table>

**Note:** This question was “best 10 of 11” because I was curious to see how many students understand that your NetID logins are handled by a trusted central login server (shib.oit.duke.edu) and not by individual sites (such as gitlab). All users should understand this to stay safe: never type your NetID/password except in a secure session with shib.oit.duke.edu.

Optional bonus question: In the exam archive, many file names have the format **midtermX-Ys** or **midtermX-Yf**, where X is an integer in range [1-2] and Y is an integer in range [10-16]. What are the meanings of X, Y, f, and s? What will be the name/prefix of the file(s) for this midterm when it is placed in the archive? **This question was just a little joke in reference to a piazza exchange. This file will be midterm1-17s (actually midterm1-310-17s) and midterm1-310-17s-sol for the solution. “Everyone” got two points.**
P2. Tracing to the barriers (40 points)

Consider the following pseudocode for a barrier for N threads. This problem asks you to trace execution of this code in detail for the given main program. The thread primitives behave as in lab p1: all queues are FIFO. Assume that the code runs on a uniprocessor (single core) with no involuntary preemptions. Trace the schedule as an ordered list containing every thread state change (ready, running, blocked, exited) and only the thread state changes. The threads are named C, 1, and 2. Fill in the contents of the thread queues at each step.

```c
int absent = 2; /* N = 2 gate threads */

main()
{
    thread_libinit(controller, null);
}

void* controller(void* arg) {
    /* C */
    thread_lock(1);
    thread_create(gate, null); /* 1 */
    thread_create(gate, null); /* 2 */
    thread_wait(1, 1);
    thread_broadcast(1, 2);
    thread_unlock(1);
}

void* gate(void* arg)
{
    thread_lock(1);
    absent = absent - 1;
    if (absent == 0)
        thread_signal(1, 1);
    thread_wait(1, 2);
    thread_yield(); /* a little twirl for fun */
    thread_unlock(1);
}
```

1. C runs …[continue below]
2. 1 is ready [-5 if it runs too soon]
3. 2 is ready
4. C waits on CV #1
5. 1 runs, acquires
6. 1 waits on CV #2
7. 2 runs
8. 2 signals CV #1: C is ready
9. 2 waits on CV #2
10. C runs
11. C broadcasts: 1 and 2 are ready
12. C unlocks and exits
13. 1 runs
14. 1 yields: 1 is ready
15. 2 runs
16. 2 blocks on lock [-5 if omitted]
17. 1 runs
18. 1 unlocks: 2 is ready
19. 1 exits
20. 2 runs
21. 2 yields: 2 is ready
22. 2 runs
23. 2 unlocks and exits

I was pretty generous about messes with the queues if you got the basic idea. If you filled out the queues properly it was easy to grade. But you lose 5-10 points if you left a chunk blank or clearly misunderstood the queues in some way.
P3. Less is more (40 points)

Here is the core of the same barrier code from P2 for N threads, with a new and different yield. Show how to modify the code to implement the barrier correctly (without assuming any deterministic thread behavior!) with a single condition variable instead of two. Just fill-in or cross out as needed and write any new code in the central box (but leave the yield in place). Then answer the questions below, using the same assumptions as for P2.

```c
int absent = N;
boolean open = false;

void* controller(void* arg) {
    thread_lock( 1);
    …
    while (absent != 0)
        thread_wait( 1,1);
    open = true;
    thread_yield();
    thread_broadcast(1,1);
    thread_unlock(1);
}

With a single condition variable, it is necessary to “overload” the CV: threads are waiting on the same CV for different reasons. Given that we can’t assume deterministic FIFO behavior in general, this requires that all signals become broadcasts (since signal could wake the “wrong” thread), and all waiters must loop before leaping until the “right” condition allows them to move past the wait.

The correct behavior of this barrier is: The controller waits until all “gated” threads arrive at the barrier, and then the controller thread opens the gate and releases the waiting threads (broadcast). The last gated thread to arrive at the barrier wakes the controller, which decides when to open the gate.

```

```c
void* gate(void* arg) {
    thread_lock(1);
    absent = absent - 1;
    if (absent == 0)
        thread_broadcast(1,1);
    while (!open)
        thread_wait(1,1);
    open = true;
    thread_yield();
    thread_broadcast(1,1);
    thread_unlock(1);
}
```

The intent of the question is that overloading a condition variable requires use of broadcast instead of signal. That creates a thundering herd, and it requires some extra state and logic (looping before leaping) to put threads back to waiting when a broadcast is not intended for them. Recommended changes are in bold above. The intent was that the need to broadcast adds N context switches for the thundering herd to stampede through.

Unfortunately, the problem was more confusing than intended. The barrier problem is arguably “too simple” for the point: e.g., once the last gated thread arrives at the barrier, its broadcast is sufficient to wake the others, so it’s not clear (from the code given) if the “controller” is even still needed. The yield merely confuses the issue. For this question I gave credit (up to 20) for clear indications that you understood the need to use broadcast (not signal) and loop before leaping on both sides. Sorry for the confusion!

Despite the warning, some students assumed properties of the FIFO p1t implementation. Take care to distinguish the behavior of the abstraction—the “contract” for CVs—and the behavior of a particular implementation (p1t). Your code should never rely on implementation-dependent behaviors, because it will break in unpredictable ways when the implementation changes.
P3. Less is more (40 points)

Here is the core of the same barrier code from P2 for N threads, with a new and different yield. Show how to modify the code to implement the barrier correctly (without assuming any deterministic thread behavior!) with a single condition variable instead of two. Just fill-in or cross out as needed and write any new code in the central box (but leave the yield in place). Then answer the questions below, using the same assumptions as for P2....

How many context switches does this original code take for N=2 threads?

How many context switches does this original code take for N threads, as a function of N?

How many context switches does your modified code take for N=2 threads?

How many context switches does your modified code take for N threads, as a function of N?

This part of the problem was also botched, since the intended handling of the yields was not clear to many students. I said during the exam to ignore the yields as a superfluous complicating detail when counting context switches, but not everyone did, and that made the math more challenging and time consuming. Also, the confusion about how the barrier “should” behave called the correct answers into doubt. And many were unclear about what to do for the “modified” code.

For this part of the problem I gave credit (20 points) for any plausible answers. For the N=2 cases I looked for low fixed numbers (e.g., 5 or 6 for the original code without the yield, depending on whether you count the first switch into the controller), and for the N cases I looked for a reasonable function of N. Almost everyone got the credit unless it was blank or unreadable.
P4. C sells (40 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).

```c
main.c
#define SIZE 10
char buffer[SIZE];
int main()
{
    fill(buffer, SIZE);
    printf("%s\n", buffer);
}

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

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? abcdeghi
3. How many external symbol references does the linker resolve to link this program? List them in the box above.
4. What symbols does the linker resolve from the C standard library? `printf`
5. How much memory space (in pages) does this program use on its stack when it runs? 1: see above
6. How much memory space (in pages) does it use in its global data segment? 1 for `buffer[]`
7. How much memory space (in pages) does it use on its heap? 0: no heap allocations
8. How much memory space (in pages) does it use in its code segment? 1, plus whatever for `stdlib`
9. How many system call traps does it take? 1 for `printf`, write, + exit
10. How many page faults does it take? (Assume the OS allocates page frames only on demand.) 3, + `stdlib` code faults

What’s important here is that each segment is very small, much smaller than the typical page size of 4K or 8K. If the VM is demand-paged, there is one fault for each page referenced. Other reasonable answers were accepted. Note: `buffer[]` is global data, so it’s not on the stack, and a pointer to `buffer[]` is passed on the stack as argument `buf`, so the linker doesn’t have to resolve an external reference to `buffer[]`. `SIZE`, "%s\n", and ‘a’ are constants handled at compile time.
Part P5. Virtual tollbooth (40 points)

Cars (threads) exiting the airport parking lot must pass through a tollbooth with N pay lanes. Cars queue up to pay as they arrive: when a booth/lane is free, the next car selects the free lane and occupies it until its payment transaction is complete, then departs and goes on its merry way. You are asked to use a monitor to implement a multi-threaded simulation of the tollbooth system. Each car is represented by a thread that calls arrive() and depart() methods. Threads block in arrive() if no booth is free. When a thread’s turn comes it selects a free lane/booth and pays; arrive() returns the selected lane number, and the thread passes its lane number to depart(). To allow up to N cars/threads to pay at the same time, no thread should hold the monitor/lock as it pays. As always: “any kind of pseudocode is fine as long as its meaning is clear”.

```java
int paying = 0;

int arrive() {
    lock();
    while (paying == N)
        wait();
    paying++;
    lane = findFreeLane();
    unlock();
    return(lane);
}

depart(int lane) {
    lock();
    paying--;
    freeLane(lane);
    signal();
    unlock();
}

eachCarThreadDoesThis() {
    int lane = arrive();
    pay(lane);
    depart(lane);
}
```

For this easy synchronization problem I just looked for proper locking, a wait on the left with looping before leaping, a signal or broadcast on the right, and reasonable state logic.
Grade Statistics for Midterm 1

Average (Mean) Score: 145.66 / 200 (72.83%)
Median Score: 149.5 / 200 (74.75%)
Standard Deviation: 22.36
Lowest Score: 72 / 200 (36%)
Highest Score: 195 / 200 (97.5%)
Total Graded Scores: 158
The view of the class seemed to be that this one was “harder” than usual. But the score distribution is the same as it always is. The mean dropped a few points because the tail of the distribution fell a little harder than usual. But the median is right on the target of 150. Half the exam—P0, P2, and P5—was straight from the labs and recitation practice problems, and were easier than usual.

The challenge problems on this test were P3 and P4. P3 was hard (and time-consuming) for students who didn’t see what the question was getting at. The use of a single condition variable and the need for broadcast and looping before leaping (LBL) was discussed with the Soda and SharedLock examples, which illustrate thundering herd. The soda trace on the midterm1-310-16s illustrates the risk of deadlock due to lost wakeups with a single condition variable. I put 20 points on being familiar with that issue and how to get around it.

P4 was a challenge for everyone because it asked specific questions about linking and paging, which is material we have not exercised and that people feel uncomfortable about. Most were able to puzzle out that this simple program uses one page each of stack, data, and code, and no heap. But I learned that only about 25% of students understand the nature of linking and the concept of a “symbol”, or really understand how the stack works. Now *that* I did not know! But the damage was limited because those were only 15 points total. And 4 more for understanding that ASCII characters are “just numbers”.

And of course, some students lost on P2 because they didn’t study keypairs or understand what they were doing with keypairs in git.