CPS 310 midterm exam #1, 2/19/2018

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. Where code appears, any kind of pseudocode is fine (yours and mine) as long as its meaning is clear.

P0. The kernel (30 points)

Answer each question with a short word or phrase in the box to the left of each question. The answer refers identifies an action, operation, or condition. Example: "system call" or "context switch".

Answer here

<table>
<thead>
<tr>
<th>Process create</th>
<th>1. When does the kernel create a new page table?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process exit/death</td>
<td>2. When does the kernel destroy a page table?</td>
</tr>
<tr>
<td>Thread context switch to a thread in a different VAS</td>
<td>3. When does the kernel modify the page table base register?</td>
</tr>
<tr>
<td>Page fault / resolved</td>
<td>4. When does the kernel modify the PFN in a PTE (one example)?</td>
</tr>
<tr>
<td>Return from trap, fault, or interrupt, or thread launch</td>
<td>5. When does the kernel set the core to user mode?</td>
</tr>
<tr>
<td>sbrk</td>
<td>6. What Unix system call grows the heap segment?</td>
</tr>
<tr>
<td>mmap</td>
<td>7. What Unix system call adds a new segment to the virtual address space?</td>
</tr>
<tr>
<td>exit</td>
<td>8. When does the kernel deallocate virtual memory from the heap segment?</td>
</tr>
<tr>
<td>Timer interrupt</td>
<td>9. How does the kernel gain control of a core to preempt the current thread?</td>
</tr>
<tr>
<td>Index in a designated register</td>
<td>10. How does the kernel know which system call to execute on a trap?</td>
</tr>
</tbody>
</table>
P1. Virtual memory and the metal (20 points)

Consider a 32-bit machine with a 40-bit physical address space. This machine interprets a virtual address as having two fields: a 20-bit virtual page number (VPN) and a 12-bit offset. Give a short answer in the space to the left of each question.

<table>
<thead>
<tr>
<th></th>
<th>Answer here (best 10 of 11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>What is the page size (in bytes)?</td>
</tr>
<tr>
<td>2**12 = 4K = 4096</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>What is the highest page frame number (PFN) in physical memory?</td>
</tr>
<tr>
<td>2**28 = 256M</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>How many bits are needed to store a VPN-&gt;PFN translation in a page table entry (PTE)?</td>
</tr>
<tr>
<td>28, for the PFN only</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>How many bits are needed to store page protection/permissions in a PTE?</td>
</tr>
<tr>
<td>3 for read, write, exec</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>What permissions are allowed for a page in a heap segment?</td>
</tr>
<tr>
<td>rw</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>What permissions are allowed for a page in a text segment?</td>
</tr>
<tr>
<td>rx</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>How large is a C pointer (in bytes) on this machine?</td>
</tr>
<tr>
<td>4 = 32 bits</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>What does the machine do if an instruction references a C pointer with a VPN that has no PFN in its PTE?</td>
</tr>
<tr>
<td>fault</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>How many bits are needed to represent the page table base in the page table base register?</td>
</tr>
<tr>
<td>28 for PFN</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>What does the machine do if an instruction modifies the page table base register in user mode?</td>
</tr>
<tr>
<td>fault</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>How many PTEs are needed to represent the largest possible virtual address space on this machine?</td>
</tr>
<tr>
<td>2**20 = 1M</td>
<td></td>
</tr>
</tbody>
</table>

P2. Virtual memory and the kernel (True/False, 15 points)

<table>
<thead>
<tr>
<th></th>
<th>1. A process/thread can modify its own page tables in the kernel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2. A process/thread blocks if it references a missing page that must be fetched from disk.</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3. The kernel reads the virtual address ranges for the text and data segments from the executable program file.</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>4. The kernel can arrange to share physical pages among processes without their knowledge.</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>5. The kernel can grow the main thread’s stack storage dynamically in response to page faults.</td>
<td></td>
</tr>
</tbody>
</table>
P3. 3-way ping/pong (50 points)

Consider the following pseudocode for a ping-pong variant for N threads. This problem asks you to trace execution of this code in detail for the given start function (called from thread_libinit). The thread primitives behave as in lab p1: all queues are FIFO, with handoff locks. 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. Fill in the contents of the thread queues at each step. Trace only one round in which all three threads pass through the loop.

```c
void* rotate(int n)
{
    thread_lock(1);
    while(1) { /* forever */
        printf("%d", n);
        thread_signal(1, 1);
        thread_wait(1, 1);
    }
    thread_unlock(1);
}
```

```c
void* start()
{
    thread_create(rotate, 1); /* t1 */
    thread_create(rotate, 2); /* t2 */
    thread_create(rotate, 3); /* t3 */
    /* Return: let your children play. */
}
```

1. start thread runs
   R[          ], L[         ], CV[         ]
   R[t1        ], L[         ], CV[         ]
   R[t1 t2     ], L[         ], CV[         ]
   R[t1 t2 t3], L[        ], CV[        ]
   R[t2 t3     ], L[         ], CV[         ]
   R[t2 t3     ], L[         ], CV[t1      ]
   R[t3        ], L[         ], CV[t1      ]
   R[t3 t1     ], L[         ], CV[         ]
   R[t3 t1     ], L[         ], CV[t2      ]
   R[t1        ], L[         ], CV[t2      ]
   R[t1 t2     ], L[         ], CV[         ]
   R[t1 t2     ], L[         ], CV[t3      ]
   R[t2        ], L[         ], CV[t3      ]
   R[t2 t3     ], L[         ], CV[         ]
   R[t2 t3     ], L[         ], CV[t1      ]
   R[          ], L[         ], CV[         ]
   R[          ], L[         ], CV[         ]
   R[          ], L[         ], CV[         ]
   R[          ], L[         ], CV[         ]
   R[          ], L[         ], CV[         ]
   R[          ], L[         ], CV[         ]
```

1. start thread runs
   t1 runs, prints “1”
   t1 waits (on CV 1, the only CV)
   t2 runs, prints “2”
   t2 signals → t1 ready
   t2 waits
   t3 runs, prints “3”
   t3 signals → t2 ready
   t3 waits
   t1 runs, prints “1”
   t1 signals → t3 ready
   t1 waits
   t2 runs, prints “2”
   t2 signals → t1 ready
   t2 waits
   t3 runs, prints “3”
   t3 signals → t2 ready
   t3 waits
   t1 runs, prints “1”
   t1 signals → t3 ready
   t1 waits
P4. Now what? (50 points)

Here is the rotate() code from the previous example again. What does the program print under the following conditions? Each question proposes a change to the rotate() function or to the start function in the original program. The changes are all independent: consider them as changes to the original program.

```c
1 void* rotate(int n) {
2     thread_lock(1);
3     while(1) { /* forever */
4         printf("%d", n);
5         thread_signal(1, 1);
6         thread_wait(1, 1);
7     }
8     thread_unlock(1);
9 }
```

1. **Start creates another (fourth) rotate thread.**

2. **Change the order of the printf and signal.**

3. **Move the printf to just after the wait.**

4. **Change the signal to a broadcast.**

5. **Move the lock and unlock to inside the loop, at the start and end of the loop.**

6. **Change the order of the signal and wait.**

7. **Add a call to thread_yield before the lock?**

8. **Add a call to thread_yield after the printf?**

9. **Add another signal+wait pair on a second CV (1,2) under the same lock (after line 6)?**

No change to cycle for #9 with three or more threads: each thread released from wait on CV1 wakes the previous thread blocked on CV2, which cycles to wake the previous thread blocked on CV1.
P5. N-way ping/pong (35 points)

Here is the original code again. Now consider the behavior of the same code without the specific FIFO behaviors of p1t:
Suppose the thread system can choose threads to wakeup or run in any order, or preempt threads at its whim. Show how to modify rotate() to enforce the desired ordered of rotation under these conditions: i.e., thread n=1 prints, then thread n=2 prints, and so on up to thread n=N for some N (assume is given as a global variable), and then the cycle repeats.

```c
void* rotate(int n)
{
    thread_lock(1);
    while(1) {     /* forever */
        printf("%d", n);
        thread_signal(1, 1);
        thread_wait(1, 1);
    }
    thread_unlock(1);
}
```

```c
int next = 1;

void* rotate(int n)
{
    thread_lock(1);
    while(...) {
        while (next != n)
            wait(1, 1);
        printf("%d", n);
        next = next_thread(n); /* n++ with wrap-around */
        thread_broadcast(1, 1);
    }
    thread_unlock(1);
}
```

void* start()
{
    thread_create(rotate, 1); /* t1 */
    thread_create(rotate, 2); /* t2 */
    thread_create(rotate, 3); /* t3 */
    ....
    /* Return: let your children play. */
}