Final Exam Review

15-213: Introduction to Computer Systems
August 3, 2018

Instructor: TA(s)
Outline

- Exam Details
- Thread Synchronization
- Signals
- Processes
- Virtual Memory
Final Exam Details

- **Review server**
- **Exam format**
  - Eight problems, similar in format to midterm
  - Five (5) hours to complete exam
  - Problems cover the entire semester, focus on second half
- **Final Exam on Wednesday, August 8**
  - You may bring two (2) double-sided, 8.5” x 11” sheets of notes
  - TA will verify your notes and CMU ID
  - Navigate to exam server and use special exam password
Final Exam Topics

- Potential areas we can test you on
  - IO
  - Malloc
  - Multiple Choice/General Knowledge
    - From lecture, labs, textbook, ...
  - Processes
  - Signals
  - Threads
  - Thread Synchronization
  - Virtual Memory
Thread Synchronization

- Three types of locks
  - Mutex
  - Semaphore
  - Reader-Writer lock

- When would you want to use one over the others?

- Rule of thumb: protect shared variables and IO to the same file descriptor

- Avoid deadlocks: acquire locks in the same order in each thread
Threads Questions

- What is a scenario where a reader-writer lock would be a more appropriate choice than a mutex?
- What happens when you join on a detached thread?
Threads Questions

- How many characters does “hello.txt” contain after this example?

```c
void *work(void *data)
{
    write(*(int *) data, "a", 1);
    return NULL;
}

int main(void)
{
    int i, fd = open("hello.txt", O_RDWR);
    pthread_t tids[NTHREADS];
    for (i = 0; i < NTHREADS; ++i) {
        pthread_t tid;
        pthread_create(&tid, NULL, work, &fd);
        pthread_detach(tid);
    }
}
```
Signals and Handling Reminders

- Signals can happen at any time
  - Control when through blocking signals

- Signals also communicate that events have occurred
  - What event(s) correspond to each signal?

- Write separate routines for receiving (i.e., signals)
  - What can you do / not do in a signal handler?
Signal Blocking

- We need to block and unblock signals. Which sequence?

```c
pid_t pid;    sigset_t mysigs, prev;
sigemptyset(&mysigs);
sigaddset(&mysigs, SIGCHLD);
sigaddset(&mysigs, SIGINT);
// need to block signals. what to use?
// A. sigprocmask(SIG_BLOCK, &mysigs, &prev);
// B. sigprocmask(SIG_SETMASK, &mysigs, &prev);

if ((pid = fork()) == 0) {
    // need to unblock signals. what to use?
    /* A. sigprocmask(SIG_BLOCK, &mysigs, &prev);
    * B. sigprocmask(SIG_UNBLOCK, &mysigs, &prev);
    * C. sigprocmask(SIG_SETMASK, &prev, NULL);
    * D. sigprocmask(SIG_BLOCK, &prev, NULL);
    * E. sigprocmask(SIG_SETMASK, &mysigs, &prev);
    */
```
Signal Delivery

Child calls kill(parent, SIGUSR{1,2}) between 2-4 times. What sequence of kills may only print 1? Can you guarantee printing 2?

- What is the range of values printed?

```c
int counter = 0;

void handler(int sig) {
    counter++;
}

void fun(pid_t parent) {
    /* insert code here */
}

int main(int argc, char** argv) {
    signal(SIGUSR1, handler);
    signal(SIGUSR2, handler);
    int parent = getpid();
    int child = fork();
    if (child == 0) {
        fun(parent);
        exit(0);
    }
    sleep(1);
    waitpid(child, NULL, 0);
    printf("Received %d USR{1,2} signals\n", counter);
    return 0;
}
```

See
http://www.comlab.ox.ac.uk/rjini/teaching/OS/Notes/Signalling.pdf for more information.
Processes

- Parent and child run in parallel as different processes
- fork(): call once, return twice
  - Initial memory contents are same
  - Afterwards, no changes are shared between the two
- execve(): never returns (except on error)
Processes Question

- What is printed to the terminal?

```c
const char *msg = "hello there";
pid_t cpid;
int fd = open("hello.txt", O_RDWR);
char contents[12];
ssize_t nbytes;
if ((cpid = fork()) == 0) {
    write(fd, msg, strlen(msg));
    close(fd);
    exit(0);
}
waitpid(cpid, NULL, 0);
nbytes = read(fd, contents, strlen(msg));
contents[nbytes] = '\0';
close(fd);
printf("%s\n", contents);
```
Virtual Memory

- Virtual to physical address conversion (TLB lookup)
- TLB miss
- Page fault, page loaded from disk
- TLB updated, check permissions
- L1 Cache miss (and L2 ... and)
- Request sent to memory
- Memory sends data to processor
- Cache updated
Virtual Memory Example

- Translate 0x15213, given the contents of the TLB and the first 32 entries of the page table below.

1MB Virtual Memory

256KB Physical Memory

4KB page size

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>05</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>3F</td>
<td>15</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0F</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>18</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1F</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1F</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>2B</td>
<td>1</td>
</tr>
<tr>
<td>1D</td>
<td>23</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>0B</td>
<td>0</td>
</tr>
<tr>
<td>04</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>06</td>
<td>0F</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>08</td>
<td>1C</td>
<td>0</td>
</tr>
<tr>
<td>09</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>0B</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>0C</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>0E</td>
<td>0C</td>
<td>0</td>
</tr>
<tr>
<td>0F</td>
<td>1E</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0E</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>04</td>
<td>0</td>
</tr>
<tr>
<td>1A</td>
<td>0C</td>
<td>1</td>
</tr>
<tr>
<td>1B</td>
<td>2B</td>
<td>0</td>
</tr>
<tr>
<td>1C</td>
<td>1E</td>
<td>0</td>
</tr>
<tr>
<td>1D</td>
<td>3E</td>
<td>1</td>
</tr>
<tr>
<td>1E</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>1F</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third
IO Recap

- How does read offset?

- How does dup2 work?
  - What is the order of arguments?
IO Recap

- How does read offset?
  - Incremented by number of bytes read
  - Important: read and write offset the same fd

- How does dup2 work?
  - What is the order of arguments?
  - `dup2(oldfd, newfd)`
    - Example: `dup2(fd2, fd3)`
    - Any read/write from fd3 now happen from fd2
    - All file offsets are shared
IO and Processes

//foo.txt = “abcdefg”

fd1 = open(“foo.txt”, O_RDONLY);
pid = fork();
fd2 = open(“foo.txt”, O_RDONLY);

if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf(“%c”, c);
    dup2(fd1, fd2);
    //NOTE: the child did not exit here!
}
wait(NULL);
read(fd2, &c, sizeof(c));
printf(“%c”, c);
read(fd1, &c, sizeof(c));
printf(“%c”, c);

• How are fd shared between processes?
• How does dup2 work from parent to child?
• How are file offsets shared between processes?

Take out a piece of paper and draw out a process diagram. What is printed?
IO and Processes

//foo.txt = “abcdefg"

fd1 = open(“foo.txt”, O_RDONLY);
ipd = fork();
fd2 = open(“foo.txt”, O_RDONLY);

if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf(%c”, c);
    dup2(fd1, fd2);
    //NOTE: the child did not exit here!
}
wait(NULL);
read(fd2, &c, sizeof(c));
printf(“%c”, c);
read(fd1, &c, sizeof(c));
printf(“%c”, c);

Outcome

• Child always runs first. Parent cannot run until child has terminated
• fd1 is shared between parent and child, but parent and child have separate fd2
• Printed out: abcd