Final Exam Review

15-213: Introduction to Computer Systems
Recitation 15: Monday, Dec. 2nd, 2013

Marjorie Carlson
Section A

Agenda

- News & Exam Information
- Brief Review of Topics
  - Important System Calls
  - Virtual Address Translation
  - Threading vs. Forking
- Practice Questions

News

- Proxylab is due Thursday Dec. 5th at 11:59 PM
  - Last day to submit late is Sunday Dec. 8th
  - Make sure you've downloaded the tarball since your "Thanksgiving gift" from Dr. O'Hallaron.

Exam Information

- Monday December 9th – Thursday December 12th
  - Online, like the midterm.
  - Exact times will be sent out in an email and updated on the website later this week.
- You can bring **2** double-sided sheets of notes.
  - **No pre-worked problems.**
  - Must be your own work.
- What to study:
  - Chapters 8-12 + everything from the first half!
- How to study:
  - Read each chapter 3 (more?) times.
  - Work practice problems from the book.
  - **Do problems from previous exams (including newly posted finals).**

Agenda

- News & Exam Information
- Brief Review of Topics
  - Important System Calls
  - Virtual Address Translation
  - Threading vs. Forking
- Practice Questions

Important System Calls

- **fork**
  - Called once, returns twice (unless it fails)
    - Returns **0** in the child process
    - Returns the **pid** of the child in the parent process
    - Returns **-1** on failure
  - Makes an exact copy of the entire address space
  - Processes get unique copies of file descriptors, but share open files
  - Execution order of parent and child is arbitrary
- **execve**
  - Called once, doesn't return (unless it fails)
    - Returns **-1** on failure
  - Replaces the currently running process with the specified program

Important System Calls

- **`wait/waitpid`**
  - Reaps one child process
    - By default, blocks until a child process can be reaped
    - **`wait`** will wait for any child
    - **`waitpid`** waits for the specified child process
  - Returns the pid of the child that was reaped, or -1 on error
  - `waitpid` can be passed additional arguments to modify its behavior
    - WNOHANG will prevent waitpid from blocking
    - WUNTRACED will report stopped children
- **`signal`**
  - A simplified (but easier to understand) interface to `sigaction`
  - Installs a signal handler that is run when the specified signal is triggered

Important System Calls

- **`sigprocmask`**
  - Can block signals, unblock signals, or set the signal mask
    - SIG_BLOCK adds the given signals to the set of blocked signals
    - SIG_UNBLOCK removes the given signals
    - SIG_SETMASK replaces the blocked signals with the given signals
- **`sigsuspend`**
  - Replaces the signal mask with the specified mask
  - Blocks until one signal that isn't masked is handled
  - After the one signal is handled, the signal mask is restored

Agenda

- News & Exam Information
- Brief Review of Topics
  - Important System Calls
  - Virtual Address Translation
  - Threading vs. Forking
- Practice Questions

Virtual Address Translation

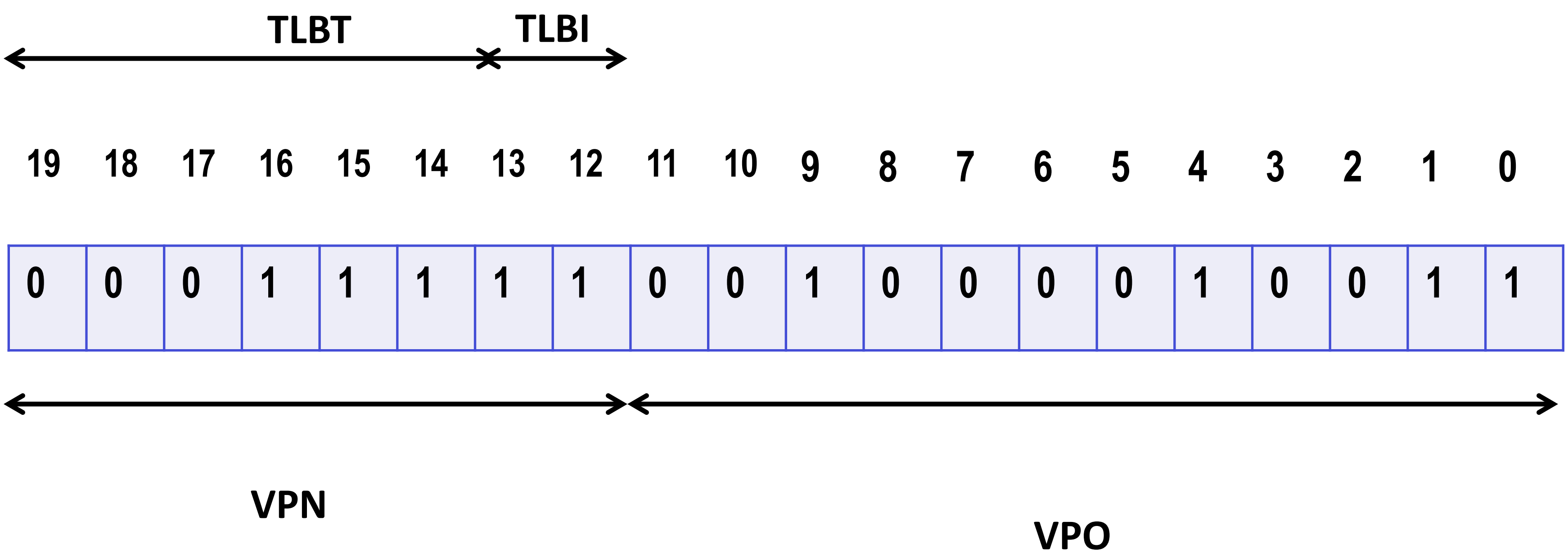
- Translates a process's virtual address into a physical address in main memory.
- Page tables store mappings from virtual addresses to physical addresses.
- Page directories store mappings from virtual addresses to page table addresses, adding an additional layer of indirection.
- Address translation is like cache lookup:
  - Split up the binary representation of a virtual address.
  - Use the parts as indices into pages, page tables, or the TLB.

Virtual Address Translation

- Know your acronyms (there are probably more in the book)
  - **TLB** Translation lookaside buffer
  - **TLBI** TLB Index
  - **TLBT** TLB Tag
  - **VPO** Virtual page offset
  - **VPN** Virtual page number
  - **PPO** Physical page offset
  - **PPN** Physical page number
  - **PTBE** Page table base address
  - **PTE** Page table entry
  - **PDE** Page directory entry
  - **CI** Cache index
  - **CT** Cache tag

Virtual Address Translation

- Refer to this diagram, blatantly copied from recitation 10



Virtual Address Translation

- A simplified overview of the translation process
  - Write out the virtual address in binary; divide it up into the relevant offset, indexes and tags.
  - Check the TLB (if there is one) to see if the page is in memory.
  - If there's a TLB miss, check the top level page directory to see if the page is in memory.
  - If the top level page directory entry is present, continue following to the next page table. If not, a page fault is triggered.
  - If you make it all the way down to the deepest page table without triggering a page fault, you will get a physical address.
  - After you have a physical address, you may have to check a cache to see if the requested data is already available.

Agenda

- News & Exam Information
- Brief Review of Topics
  - Important System Calls
  - Virtual Address Translation
  - Threading vs. Forking
- Practice Questions

Threading vs. Forking

- How they're the same
  - Both allow you to run code concurrently
- How they're different
  - Threads in the same process share memory
  - Threads share file descriptors
    - If you close a file descriptor in one thread, it's closed for all of the threads in the same process
  - Threads share signal handlers and masks
    - If you install one signal handler in one thread, and a different one in another, the most recent one will be the one that is called.

Agenda

- News & Exam Information
- Brief Review of Topics
  - Important System Calls
  - Virtual Address Translation
  - Threading vs. Forking
- Practice Questions

Process Control

```
int main() {
  if (fork() == 0) {
      printf("a");
  }
  else {
      printf("b");
      waitpid(-1, NULL, 0);
  }
  printf("c");
  exit(0);
}
```

What are the possible outputs for this program?

File I/O

```
int main() {
  char buf[3] = "ab";
  int r = open("file.txt", O_RDONLY);
  int r1, pid;
  r1 = dup(r);
  read(r, buf, 1);

  if((pid=fork())==0)
    r1 = open("file.txt", O_RDONLY);
  else
    waitpid(pid, NULL, 0);

  read(r1, buf+1, 1);
  printf("%s", buf);
  return 0;
}
```

Assume that file.txt contains the string of bytes **15213**. Also assume that all system calls succeed.

What will be the output when this code is compiled and run?

Code Snippet 1

```c
int i = 0;

void handler(int sig) {
  i = 0;
}

int main() {
  int j;
  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  exit(0);
}
```

Code Snippet 1

```c
int i = 0;

void handler(int sig) {
  i = 0;
}

int main () {
  int j;
  sigset_t s;

/* Assume that s has been
initialized and declared
properly for SIGINT */

  signal(SIGINT, handler);
  sigprocmask(SIG_BLOCK, &s, 0);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  sigprocmask(SIG_UNBLOCK, &s, 0);
  printf("i = %d\n", i);
  exit(0);
}
```

Code Snippet 3

```c
int i = 0;

void handler(int sig) {
  i = 0;
  sleep(1);
}
int main () {
  int j;
  sigset_t s;

/* Assume that s has been
initialized and declared
properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  sigprocmask(SIG_UNBLOCK, &s, 0);
  exit(0);
}
```

For each of the above code snippets, assume an arbitrary number of SIGINTs—and only SIGINTs—are sent to the process. What are the possible values of *i* that are printed out?

Processes vs. Threads

```
#include "csapp.h"

/* Global variables */
int cnt;
sem_t mutex;


* Helper function */
void *incr(void *vargp) {
  P(&mutex);
  cnt++;
  V(&mutex);
  return NULL;
}
```

What is the output?

Procs: cnt = ___

Threads: cnt = ___

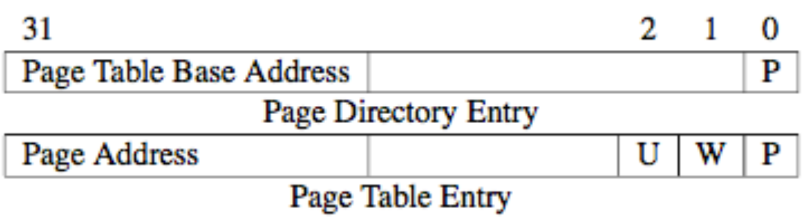
```
int main() {
  int i;
  pthread_t tid[2];

  sem_init(&mutex, 0, 1); /* mutex=1 */

  /* Processes */
  cnt = 0;
  for (i=0; i<2; i++) {
    incr(NULL);
    if (fork() == 0) {
      incr(NULL);
      exit(0);
    }
  }
  for (i=0; i<2; i++)
    wait(NULL);
  printf("Procs: cnt = %d\n", cnt);

  /* Threads */
  cnt = 0;
  for (i=0; i<2; i++) {
    incr(NULL);
    pthread_create(&tid[i], NULL, incr, NULL);
  }
  for (i=0; i<2; i++)
    pthread_join(tid[i], NULL);
  printf("Threads: cnt = %d\n", cnt);
  exit(0);
}
```

Address Translation

- 32-bit machine; 4-byte words.
- Memory is byte-addressable.
- 4 GB of virtual address space.
- 64 MB of physical memory.
- 4 KB page size.
- Two-level page tables. Tables at both levels are 4096 bytes (one page) and entries in both tables are 4 bytes, as shown to the right.

The page table base address for process 1 is **0x0021A000.** Translate virtual address **0xBFCF0145** into a physical address.

The page table base address for process 2 is **0x0021B000.** Translate virtual address **0x0804A1F0** into a physical address.



- P = 1 ⇒ Present
- W = 1 ⇒ Writable
- U = 1 ⇒ User-mode

Address	Contents
001AC021	07693003
001AC084	00142003
0021A020	0481C001
0021A080	04A95001
0021A2FF	06128001
0021A300	05711001
0021ABFC	05176001
0021AC00	001AC001
0021B020	01FAC9DA
0021B080	052DB001
0021B2C0	0B2B36C2
0021B2FF	05A11001
0021B300	01FCF001
0021BBFC	06213001
0021BC00	001AC001
01FCF021	00382003
0481C048	0523A005
04A95048	048B8005
04A95120	07D6A005
051760F0	0E33F007
051763C0	08BF1007
052DB04A	09A62006
052DB128	0D718006
05711021	00113003
05A110F0	01133007
061280F0	0A114007
0614504A	0B183006
062133C0	052F1007

Synchronization

- A producer/consumer system with a FIFO queue of 10 data items.
- Producer threads call `insert` to add to the rear of the queue; consumer threads call `remove` to put something at the front.
- The system uses three semaphores: mutex, items, and slots. Your task is to use P and V semaphore operations to correctly synchronize access to the queue.
- What is the initial value of each semaphore?

  mutex = ______ items = _______ slots = _______
- Write the pseudocode:

```
void insert(int item)
{


  add_item(item)


}
```

```
void remove()
{


  item = remove_item()


  return(item)
}
```

Questions?

- Good luck on proxy lab, and on your final exam!
- I hope you have learned half as much from me as I have from TAing you. ☺