Midterm Review

CS 170 Section 5, 2/5/2021
Karl Wang
Adopted from Nik Chaconas' Slides
Remind me to record
Agenda

- Midterm Review
  - Threads & synchronization
    - Race Conditions
    - Thread Synchronization
    - Semaphores
  - System Calls
    - Files
    - Fork/Exec/Wait
    - Pipes/Dup
  - Study Hints
- Lab 3 Q&A
Hints for Studying

● Go through all of Rich’s lecture notes
  ○ Ensure you understand all key concepts/definitions and code examples

● Be prepared to explain any key concept from his notes or mentioned in class
  ○ Practice explaining concepts in your own words rather than just reading through definitions

● Brush up on C
  ○ Be able to understand C-syntax, Linux syscalls, and pthread functions
  ○ Be able to interpret key concepts within code examples
Any question about the structure of the midterm?
Threads & Synchronization
Threads

What is a **Thread**? (in the context of this course)
Threads

What is a Thread? (Three Things)

- Sequential List of Instructions to be executed
- A set of local variables that belong to the thread
- A set of shared global variables that all threads can read and write

**synchronization**: the state of a concurrent program is consistent across two or more concurrent events
Thread Marshal and Unmarshal

- P_thread entry point is a function which returns void pointer and takes as argument a void pointer
- Void * can be thought of as generic package
- Can **marshal** any pointer into a void *
- Thread can then **unmarshal** void *
- pthread_join() is the main threads method for synchronizing with other threads and waiting for them to complete before utilizing their return values
- Pthreads demonstrates parallelism
  - Difference between parallelism and concurrency?
Race Conditions

- How would you define the term "race condition"?
Race Conditions

- **race condition**: The possibility in a program consisting of concurrent threads that all legal instruction orderings do not result in exactly the same output.
  - Any interleaving of instructions that preserves the sequential order of each individual thread is legal and may occur
- **pre-emption**: the process of pausing one thread to run another
- **context switching**: The process of switching to a new thread (as a result of a pre-emption event)
- **context**: the saved machine state that is necessary to get a thread running again
Race Conditions (continued)

- **critical section**: A segment of code that must be executed sequentially without the potential interleaving of other threads
  - **Atomic Execution**: the notion that all instructions within a section will be executed without being interleaved
  - *The instructions within a critical section are executed atomically*
- **mutual exclusion**: a form of synchronization that ensures at most one thread can be executing in a critical section
  - Normally implemented with lock or semaphore
- To prevent race conditions, you must lock before updating shared variable
- To prevent threads waiting indefinitely, unlock after updating
Semaphores

Semaphores are highly versatile:

● Ensuring the atomic execution of critical sections
● Determining whether resource is filled (Think nslots from console buffer)
● Synchronizing with other threads (Think of Interrupt handler telling do_write thread that char is ready to be written to console)
Basic Definitions for Semaphores:

- Must be executed atomically

```c
initialize(i) {
    s->value = i
    return
}
```

```c
P(Sem s) {
    s->value--;  
    if(s->value < 0)  
        block on semaphore  
    return
}
```

```c
V(s) {
    s->value++;  
    if(s->value <= 0)  
        unblock one process or thread that is  
        blocked on semaphore  
    return
}
```
The Dining Philosophers Problem

deadlock: when multiple threads of execution are blocked and cannot make progress because each is waiting for a condition in order to continue that only the others can make true

- Can progress on a thread be perpetually blocked without deadlock?
System Calls
Libraries vs System Calls

What’s the difference between a library call and system call?
Libraries vs System Calls

What’s the difference between a library call and system call?

System Calls: Calls directly to the operating system (These are the calls which you are handling within KOS!). By default are accessible within your program. (write, sbrk, fork...)

Library Calls: Helpful code which you can include in your program. Usually helps to abstract some useful functionality. May ultimately call System Calls. (printf, strlen...etc)

Understand the difference between these two!
What are the five system calls associated with files?

- `open(pathname, flags)`
  - Opens a file according to the flags, which includes the access modes
- `close(fd)`
  - Closes a file telling the OS that your process will no longer access the file
- `read(fd, buf, count)`
  - Reads a specific number of bytes from a file into a buffer
- `write(fd, buf, count)`
  - Writes a specified number of bytes from a buffer into a file
- `lseek(fd, offset, whence)`
  - Changes the current read/write offset
Fork/Exec/Wait

- **fork()** Clones the running process
  - Child PID is returned by fork call in parent process
  - Zero is returned in child process
- Can be followed by an **exec()** call to load new process into child process
- Parent can call **wait()** to wait until child terminates, stops, or resumes.
  - If a parent does not call wait and the child exits, Linux will create a zombie process to wait for the parent so that it can report the exit status of the child to the parent.
  - If parent never calls wait, the zombie will not die until parent dies or calls wait!
Pipes/Dup

- `pipe(pipefd[2])` allocates two file descriptors. Any data written to the write file descriptor will be available for reading on the read file descriptor
  - Read End
    - Read blocks if there is nothing available to read from a pipe
    - Read unblocks after receiving EOF or reading specified number of bytes or all data has been read from the pipe
  - Write End
    - Writes into write fd and then closes it when complete to signal EOF
- `dup(fd)` will take lowest available file descriptor and make it the same file descriptor as fd
- `dup2(fd1, fd2)` will make fd2 the same file descriptor as fd1
Last week, we said that the child field in PCB_struct was to keep track of pids. But it's actually so that a parent can pass its live children to init if it exits.

```c
struct PCB_struct
{
    int mem_base;
    int mem_limit;
    int data_end;
    int sbrk;
    unsigned short pid;
    struct PCB_struct *parent;
    kt_sem *waiter_sem;
    Dlist waiters;
    Rb_node children;
    int exit_status;
    int registers[NumTotalRegs];
};
```
Any Question about the midterm or Lab 2?