CS 423
Operating System Design: Systems Review

Professor Adam Bates
Spring 2017
Goals for Today

- **Learning Objective:**
  - Review the prerequisite skills needed to be successful in this course
  - I.E., make sure you’re all in the right place : - )
  - If something in this lecture is unclear to you, review ASAP: [http://www.lysator.liu.se/c/bwk-tutor.html](http://www.lysator.liu.se/c/bwk-tutor.html)

**Reminder:** Please put away devices at the start of class
Your Teaching Assistants!

Saad Hussain

Bo Teng

Yisong Yue
Function Calls

System Calls

Caller and callee are in the same process:
- Same user
- Same “domain of trust”
Function Calls

- Caller and callee are in the same Process
  - Same user
  - Same “domain of trust”

System Calls

- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse
Example System Calls?
Example System Calls?

Example:
- `getuid()`  //get the user ID
- `fork()`     //create a child process
- `exec()`     //executing a program
Example System Calls?

Example:

getuid()    //get the user ID
fork()      //create a child process
exec()      //executing a program

Don’t confuse system calls with stdlib calls

Differences?

Is printf() a system call?
Is rand() a system call?
Syscalls vs. I/O Lib Calls

Each system call has analogous procedure calls from the standard I/O library:

<table>
<thead>
<tr>
<th>System Call</th>
<th>Standard I/O call</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>fopen</td>
</tr>
<tr>
<td>close</td>
<td>fclose</td>
</tr>
<tr>
<td>read/write</td>
<td>getchar/putchar</td>
</tr>
<tr>
<td></td>
<td>getc/putc</td>
</tr>
<tr>
<td></td>
<td>fgetc/fputc</td>
</tr>
<tr>
<td></td>
<td>fread/fwrite</td>
</tr>
<tr>
<td></td>
<td>gets/puts</td>
</tr>
<tr>
<td></td>
<td>fgets/fputs</td>
</tr>
<tr>
<td></td>
<td>scanf/printf</td>
</tr>
<tr>
<td></td>
<td>fscanf/fprintf</td>
</tr>
<tr>
<td>lseek</td>
<td>fseek</td>
</tr>
</tbody>
</table>
Processes

Possible process states
- Running (occupy CPU)
- Blocked
- Ready (does not occupy CPU)
- Other states: suspended, terminated

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
Possible process states
- Running (occupy CPU)
- Blocked
- Ready (does not occupy CPU)
- Other states: suspended, terminated

Question: in a single processor machine, how many processes can be in the running state?
Creating a Process

• What UNIX call creates a process?
• What UNIX call creates a process?
  - `fork()` duplicates a process so that instead of one process you get two.
    • The new process and the old process both continue in parallel from the statement that follows the `fork()`
Creating a Process - fork()

• What UNIX call creates a process?
  ▪ fork() duplicates a process so that instead of one process you get two.
    ▪ The new process and the old process both continue in parallel from the statement that follows the fork()
  ▪ How can you tell the two processes apart?
• What UNIX call creates a process?
  ▪ `fork()` duplicates a process so that instead of one process you get two.
    ▪ The new process and the old process both continue in parallel from the statement that follows the `fork()`
  ▪ How can you tell the two processes apart?
    ▪ `fork()` returns
      ▪ 0 if child
      ▪ -1 if fork fails
      ▪ Child’s PID if parent process
What UNIX call creates a process?
- `fork()` duplicates a process so that instead of one process you get two.
  - The new process and the old process both continue in parallel from the statement that follows the `fork()`
- How can you tell the two processes apart?
  - `fork()` returns
    - 0 if child
    - -1 if fork fails
    - Child’s PID if parent process
- If the parent code changes a global variable, will the child see the change?
• What UNIX call creates a process?
  ▪ `fork()` duplicates a process so that instead of one process you get two.
    ▪ The new process and the old process both continue in parallel from the statement that follows the `fork()`
  ▪ How can you tell the two processes apart?
    ▪ `fork()` returns
      ▪ 0 if child
      ▪ -1 if fork fails
      ▪ Child’s PID if parent process
  ▪ If the parent code changes a global variable, will the child see the change?
    ▪ Nope! On fork, child gets new program counter, stack, file descriptors, heap, globals, pid!
• What if we need the child process to execute different code than the parent process?
What if we need the child process to execute different code than the parent process?

- Exec function allows child process to execute code that is different from that of parent
- Exec family of functions provides a facility for overlaying the process image of the calling process with a new image.
- Exec functions return -1 and sets errno if unsuccessful
• What is the difference between a thread and a process?
• What is the difference between a thread and a process?
  ▪ Both provided independent execution sequences, but...
  ▪ Each process has its own memory space
    ▪ *Remember how child processes can’t see changes to parent’s global variable??*
  ▪ Threads run in a shared memory space
Threads vs. Processes

- What is POSIX?
- How do you create a POSIX thread?
Threads vs. Processes

- What is POSIX?
- How do you create a POSIX thread?

<table>
<thead>
<tr>
<th>POSIX function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>create a thread</td>
</tr>
<tr>
<td>pthread_detach</td>
<td>set thread to release resources</td>
</tr>
<tr>
<td>pthread_equal</td>
<td>test two thread IDs for equality</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>exit a thread without exiting process</td>
</tr>
<tr>
<td>pthread_kill</td>
<td>send a signal to a thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>wait for a thread</td>
</tr>
<tr>
<td>pthread_self</td>
<td>find out own thread ID</td>
</tr>
</tbody>
</table>
• (a) Three processes each with one thread
• (b) One process with three threads
What is the difference between kernel and user threads? Pros and Cons?
Threads: Kernel v. User

- What is the difference between kernel and user threads? Pros and cons?
- Kernel thread packages
  - Each thread can make blocking I/O calls
  - Can run concurrently on multiple processors
- Threads in User-level
  - Fast context switch
  - Customized scheduling
M:N model multiplexes N user-level threads onto M kernel-level threads

Good idea? Bad Idea?
Processes and threads can be preempted at arbitrary times, which may generate problems.

Example: What is the execution outcome of the following two threads (initially x=0)?

Thread 1: Read X
          Add 1
          Write X

Thread 2: Read X
          Add 1
          Write X

How do we account for this?
Process {
    while (true) {
        ENTER CRITICAL SECTION
        Access shared variables;
        LEAVE CRITICAL SECTION
        Do other work
    }
}
Mutex

- Simplest and most efficient thread synchronization mechanism
- A special variable that can be either in
  - locked state: a distinguished thread that holds or owns the mutex; or
  - unlocked state: no thread holds the mutex
- When several threads compete for a mutex, the losers block at that call
  - The mutex also has a queue of threads that are waiting to hold the mutex.
- POSIX does not require that this queue be accessed FIFO.
- Helpful note — Mutex is short for “Mutual Exclusion”
POSIX Mutex Functions

- `int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);`
  - Also see `PTHREAD_MUTEX_INITIALIZER`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
Pseudocode for a blocking implementation of semaphores:

```c
void wait (semaphore_t *sp)
    if (sp->value >0) sp->value--;
    else {
        <Add this process to sp->list>
        <block>
    }

void signal (semaphore_t *sp)
    if (sp->list != NULL)
        <remove a process from sp->list, put it in ready state>
    else sp->value++;
```
Basic scheduling algorithms

- FIFO (FCFS)
- Shortest job first
- Round Robin
- Priority Scheduling
Basic scheduling algorithms
- FIFO (FCFS)
- Shortest job first
- Round Robin
- Priority Scheduling

What is an optimal algorithm in the sense of maximizing the number of jobs finished?
Basic scheduling algorithms
- FIFO (FCFS)
- Shortest job first
- Round Robin
- Priority Scheduling

What is an optimal algorithm in the sense of meeting the most deadlines (of real time tasks)?
Non-preemptive scheduling:
- The running process keeps the CPU until it voluntarily gives up the CPU
  - process exits
  - switches to blocked state
  - 1 and 4 only (no 3)

Preemptive scheduling:
- The running process can be interrupted and must release the CPU (can be forced to give up CPU)
• What is a signal in UNIX/Linux?
What is a signal in UNIX/Linux?
- A way for one process to send a notification to another
- A signal can be “caught”, “ignored”, or “blocked”
Signals

- What is a signal in UNIX/Linux?
  - A way for one process to send a notification to another
  - A signal can be “caught”, “ignored”, or “blocked”
- Signal is **generated** when the event that causes it occurs.
- Signal is **delivered** when a process receives it.
- The **lifetime** of a signal is the interval between its generation and delivery.
- Signal that is generated but not delivered is **pending**.
- Process **catches** signal if it executes a **signal handler** when the signal is delivered.
- Alternatively, a process can **ignore** a signal when it is delivered, that is to take no action.
- Process can temporarily prevent signal from being delivered by **blocking** it.
- **Signal Mask** contains the set of signals currently blocked.
### POSIX-required Signals*

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
<th>default action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>process abort</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>alarm clock</td>
<td>abnormal termination</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>access undefined part of memory object</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>child terminated, stopped or continued</td>
<td>ignore</td>
</tr>
<tr>
<td>SIGILL</td>
<td>invalid hardware instruction</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>SIGINT</td>
<td>interactive attention signal (usually ctrl-C)</td>
<td>abnormal termination</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>terminated (cannot be caught or ignored)</td>
<td>abnormal termination</td>
</tr>
</tbody>
</table>

*Not an exhaustive list*
### POSIX-required Signals*

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
<th>default action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGSEGV</td>
<td>Invalid memory reference</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>Execution stopped</td>
<td>stop</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>termination</td>
<td>Abnormal termination</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>Terminal stop</td>
<td>stop</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>Background process attempting read</td>
<td>stop</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>Background process attempting write</td>
<td>stop</td>
</tr>
<tr>
<td>SIGURG</td>
<td>High bandwidth data available on socket</td>
<td>ignore</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>User-defined signal 1</td>
<td>abnormal termination</td>
</tr>
</tbody>
</table>

* Not an exhaustive list
How can you send a signal to a process from the command line?
User-generated Signals

- How can you send a signal to a process from the command line?
- `kill 😢`
How can you send a signal to a process from the command line?

- **kill 🥺**

- `kill -l` will list the signals the system understands.

- `kill [-signal] pid` will send a signal to a process.
  - The optional argument may be a name or a number (default is SIGTERM).

- To unconditionally kill a process, use:
  - `kill -9 pid` which is `kill -SIGKILL pid`. 

User- generated Signals
A process can temporarily prevent a signal from being delivered by blocking it.

Signal Mask contains a set of signals currently blocked.

Important! Blocking a signal is different from ignoring signal. Why?
Signal Masks

- A process can temporarily prevent a signal from being delivered by **blocking** it.
- **Signal Mask** contains a set of signals currently blocked.
- **Important!** Blocking a signal is different from ignoring signal. Why?
- When a process blocks a signal, the OS does not deliver signal until the process unblocks the signal
  - A **blocked** signal is not delivered to a process until it is unblocked.
- When a process ignores signal, signal is delivered and the process handles it by throwing it away.
Deadlocks
When do deadlocks occur (hint: 4 preconditions)?
When do deadlocks occur (hint: 4 preconditions)?

- Mutual exclusion
- Hold and wait condition
- No preemption condition
- Circular wait condition
• resource R assigned to process A
• process B is requesting/waiting for resource S
• process C and D are in deadlock over resources T and U
Deadlocks

Strategies for Dealing with Deadlocks

- shouting
- detection and recovery
- dynamic avoidance (at run-time)
- prevention (by offline design)
  - by negating one of the four necessary conditions
Other Issues

- make sure you have a working understanding of other systems programming issues, e.g.:
  - dynamic avoidance (at run-time)
  - prevention (by offline design)
    - memory management
    - working with files
    - I/O