CSCC 69H3
Operating Systems
Winter 2013
Bianca Schroeder
U of T

Logistics
- Instructor: Bianca Schroeder
  - Email: bianca@cs.toronto.edu
  - Office: IC460
  - Office hours: Friday 3-4 pm
- Assistant instructor:
  - Ioan Stefanovic
- Webpage:
  - http://www.cs.toronto.edu/~bianca/cscc69w13/
  - See class information sheet

Course Objective
- To understand
  - the role of the OS
  - its major components
  - the design principles and implementations
  - Concurrency

Workload
- This course is very work-intensive
  - Why?
    - Lectures cover a lot of new concepts
    - Many of them very abstract
    - Learning by doing!
    - This course is much more like the real world:
      - Assignments build on realistic OS
        - Lot of existing code given to you
        - Don’t expect to understand all of it
        - You need to be able to work in a team
        - You need to be comfortable with the prereqs.

Linux kernel has more than 10 million lines of code!
There's tons of help …

- Instructor: Bianca Schroeder
- TAs: Ioan
- Discussion board:
  - URL will come soon (look for a link from course web page)
- Who to ask in case of questions?

<table>
<thead>
<tr>
<th>Question regarding lecture material or logistics?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me: <a href="mailto:bianca@cs.toronto.edu">bianca@cs.toronto.edu</a></td>
</tr>
<tr>
<td>Include &quot;C69&quot; in subject line</td>
</tr>
<tr>
<td>Office hours: Fri 3-4pm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question regarding assignment &amp; grading?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Tutorials: Thu 2-3, 3-4pm</td>
</tr>
<tr>
<td>2) Discussion board</td>
</tr>
<tr>
<td>3) TAs: <a href="mailto:c69tas@cs.toronto.edu">c69tas@cs.toronto.edu</a></td>
</tr>
</tbody>
</table>

Academic Dishonesty

- Plagiarism and cheating
- Serious academic offenses
- Can discuss tools and concepts with classmates
- Can discuss solutions to assignments with your partner only!
- All potential cases will be investigated fully

Grading

<table>
<thead>
<tr>
<th>Week</th>
<th>Notes</th>
<th>Weight</th>
<th>Due Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment 1</td>
<td>Implementing a Unix shell</td>
<td>10%</td>
<td>Tue Jan 29</td>
</tr>
<tr>
<td>Assignment 2</td>
<td>Synchronization and Concurrency</td>
<td>10%</td>
<td>Tue Feb 26</td>
</tr>
<tr>
<td>Midterm</td>
<td>In lecture</td>
<td>20%</td>
<td>Fri Feb 14</td>
</tr>
<tr>
<td>Assignment 3</td>
<td>Intro to OS 101 and System calls</td>
<td>8%</td>
<td>Tue Mar 19</td>
</tr>
<tr>
<td>Assignment 4</td>
<td>Virtual Memory</td>
<td>12%</td>
<td>Tue April 9</td>
</tr>
<tr>
<td>Final exam</td>
<td>You must receive at least 40% to pass</td>
<td>10%</td>
<td>See final exam schedule</td>
</tr>
</tbody>
</table>

Late policy

- Assignments will be submitted electronically
- Due at 10pm on due date
- You will have 4 grace day tokens per person
  - You may use up to two tokens per assignment.
  - Each team member has to pay 1 token for 1 grace day
  - Don’t use all tokens at once.
  - Don’t wait until end of semester …
- In the event of catastrophe get proper documentation.
What is an Operating System?

- The software layer between user applications and hardware.
- Turns ugly hardware into beautiful abstractions (provides services)
- Serves as a resource manager
  - allows proper use of resources (hardware, software, data)
- Serves as a control program (protection)
  - controls execution of user programs to prevent errors and improper use of the computer

Application programs
- Text editor
- Web browser
- Compiler

Operating system

Hardware
- CPU(s)
- Memory
- I/O devices

Part 1: The Process Concept

- process = job / unit of work
- process = a program in execution
- A process contains all state of program in execution
  - An address space
  - Set of OS resources
    - Open files network connections
    - Set of general-purpose registers with current values
    - Accounting info
  - A process is named by its process ID (PID)

Operations on Processes

- processes execute concurrently and must be created and deleted dynamically
- process creation: parent, child, tree of processes
- process termination:
  - when a process finishes executing last statement
  - when a parent causes the termination of a child
Process Creation

- A process is created by another process
  - Parent is creator, child is created
    - In Linux, the parent is the "PPID" field of "ps -f"
  - In some systems, the parent defines (or donates) resources and privileges for its children
    - Unix: Process User ID is inherited – children of your shell execute with your privileges
  - After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)

Process Creation: Unix

- In Unix, processes are created using fork()

  ```
  int fork()
  ```
  - Creates a new address space
  - Initializes the address space with a copy of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Fork returns twice
    - Returns the child’s PID to the parent, "0" to the child
    - Huh?

Example Output

```bash
skywolf% cc t.c
skywolf% ./a.out
My child is 486
Child of a.out is 486
```

fork()

```c
int main(int argc, char *argv[])
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s is %d\n", name, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

What does this program print?
Duplicating Address Spaces

child_pid = 486  
child_pid = 0

Parent  
Child

child_pid = fork();
if (child_pid == 0) {
    printf("child");
    printf("parent");
} else {
    printf("parent");
}

Why fork()?

- Very useful when the child...
- Is cooperating with the parent
- Relies upon the parent’s data to accomplish its task
- Example: Web server
  while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
      Handle client request
    } else {
      Close socket
    }
  }

Example for use of processes:
Concurrent Web Server

Parent  
Process  
(dispatcher)

Child 1

Child 2

Shared memory (Web cache)

User space

Kernel space
**Process Creation: Unix (2)**

- Wait a sec ... How do we actually start a new program?

- int execv(char *prog, char *argv[])
- execv() stops the current process
- Loads the program "prog" into the process' address space
- Initializes hardware context and args for the new program
- Places the PCB onto the ready queue
- Note: It does not create a new process
- What does it mean for exec to return?

**Unix Shells**

```c
while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        execv(cmd, NULL);
        panic("exec failed");
    } else {
        wait(child_pid);
    }
}
```

**How does the OS manage processes?**

- Each process is represented by a process control block (PCB), aka process descriptor
  - Program counter
  - CPU registers
  - CPU scheduling info
  - Memory management info
  - Accounting info
  - I/O status information
  - Program state
Linux PCB
- Called the task_struct in Linux
  - Defined in /include/linux/sched.h

Process Control Block
- Each process is represented by a process control block (aka process descriptor)
  - Program counter
  - CPU registers
  - CPU scheduling info
  - Memory management info
  - Accounting info
  - I/O status information
  - Program state

Process states
- The OS manages processes by keeping track of their state
  - Different events cause changes to a process state, which the OS must record/implement

State Queues
- How does the OS keep track of processes?
  - The OS maintains a collection of queues that represent the state of all processes in the system
  - Typically, the OS has one queue for each state
    - Ready, waiting, etc.
  - Each PCB is queued on a state queue according to its current state
  - As a process changes state, its PCB is unlinked from one queue and linked into another
State Queues

- Ready Queue
  - Head
  - PCB1
  - PCB3
  - PCB8

- Disk I/O Queue
  - Head
  - PCB4
  - PCB2

- Sleep Queue
  - ...
  - There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)

Process Scheduling

- Only one process can run at a time on a CPU
- Scheduler decides which process to run
- Goal of CPU scheduling:
  - Give illusion that processes are running concurrently
  - Maximize CPU utilization
- Will talk about CPU scheduling in more detail in later lecture ...

Processes - Recap

- A process includes many things
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)
- Often want multiple processes to work together to accomplish some task
- Creating a new process is costly due to the data structures that must be allocated and initialized

Parallel Programs

- Recall our Web server example that forks off copies of itself to handle multiple simultaneous requests
  - Or any parallel program that executes on a multiprocessor
- To execute these programs we need to:
  - Create several processes that execute in parallel
  - Create shared memory for processes to share data
  - Have the OS schedule these processes in parallel
- This situation is very inefficient
  - Space: Data structures, such as PCB, page tables, etc.
  - Time: create data structures, fork and copy addt space, etc.
Example for use of processes: Concurrent Web Server

Concurrent Web Server

Parent Process (dispatcher)  Child 1  Child 2

Stack  Stack  Stack

Heap  Heap  Heap

Data  Data  Data

Text  Text  Text

Shared memory (Web cache)

PCB0  PCB1  PCB2

User space

Kernel space

Rethinking Processes

What is similar in these cooperating processes?
- They all share the same code and data (address space)
- They all share the same privileges
- They all share the same resources (files, sockets, etc.)

What don’t they share?
- Each has its own execution state: PC, SP, and registers

Rethinking Processes

Key idea:
- Why don’t we separate the concept of a process from its execution state?
- Process: address space, privileges, resources, etc.
- Execution state: PC, SP, registers
- Exec state also called thread of control, or thread

Idea: Have threads in a Process!
Separate the concepts of processes and threads
- The thread defines a sequential execution stream within a process (PC, SP, registers)
- The process defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
  - Every process has at least one thread
- Processes are the containers in which threads execute
  - Processes become static, threads are the dynamic entities

Process/Thread Separation
- Separating threads and processes makes it easier to support parallel/concurrent applications
- Creating concurrency does not require creating new processes, just more threads
- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Writing parallel programs
- So multithreading is even useful on a uniprocessor

Recall our forking Web server:

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        Handle client request
    } else {
        Close socket
    }
}
```

Using fork() to create new processes to handle requests in parallel is overkill for such a simple task

Instead, we can create a new thread for each request

```c
web_server() {
    while (1) {
        int sock = accept();
        // Create a thread here that executes
        // handle_request(sock)
    }
}
handle_request(int sock) {
    Process request
    close(sock);
}
```
Thread Interface (Pthread API)

- `pthread_create(pthread_t *tid, pthread_attr_t attr, void *(*start_routine)(void *), void *arg)`
  - Create a new thread of control
  - New thread id returned in `tid`, new thread starts in `start_routine` with argument `arg`

- `pthread_join(pthread_t tid)`
  - Wait for `tid` to exit

- `pthread_cancel(pthread_t tid)`
  - Destroy `tid`

- `pthread_yield()`
  - Voluntarily give up the processor

- `pthread_exit()`
  - Terminate the calling thread

Will look at this in more detail in Assignment 2

Threads

- How do we implement threads?

Kernel-Level Threads

- We have taken the execution aspect of a process and separated it out into threads
  - To make concurrency cheaper

- As such, the OS now manages threads and processes
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system

- OS-managed threads are called kernel-level threads or lightweight processes

Kernel-level Threads

- Positive points?
- Negative points?
Kernel Thread Limitations
- Kernel-level threads make concurrency much cheaper than processes
  - Much less state to allocate and initialize
- However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead
  - Thread operations still require system calls
  - Ideally, want thread operations to be as fast as a procedure call
  - Kernel-level threads have to be general to support the needs of all programmers, languages, etc.
- For fine-grained concurrency, need even "cheaper" threads

User-Level Threads
- To make threads cheap and fast, they need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed entirely by the run-time system (user-level library)
- User-level threads are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call (no kernel involvement)
  - User-level thread operations are up to 100x faster than kernel threads
- But this depends on the quality of both implementations!

U/L Thread Limitations
- But, user-level threads are not a perfect solution
  - As with everything else, they are a tradeoff
- User-level threads are invisible to the OS
  - They are not well integrated with the OS

U/L Thread Limitations
- User-level threads are invisible to the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - Cannot make use of multiple processors
- Solving this requires communication between the kernel and the user-level thread manager
Kernel vs. User Threads

- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slow to create, manipulate, synchronize
- User-level threads
  - Fast to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel and user-level threads is important
  - For programming (correctness, performance)

Threads Summary

- Multithreading is very useful for applications
  - Efficient multithreading requires fast primitives
  - Processes are too heavyweight
- Solution is to separate threads from processes
  - Kernel-level threads better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Now, how do we get our threads to correctly cooperate with each other?
  - We will find out next week!