CS 153
Design of Operating Systems

Winter 2016

Lecture 5: Processes and Threads
Announcements

- Homework 1 out (to be posted on ilearn)
- Project 1
  - Make sure to go over it so that you can ask the TAs in lab if anything is unclear
  - Both design document and code due before class on Feb. 3rd
- Read scheduling and synchronization in textbook
  - Don’t wait for these topics to be covered in class
  - You especially need to understand priority donation in project
- Piazza enrollment
  - Some of you haven’t enrolled
  - You will miss announcements – especially about projects
- All set with project groups?
  - Email your TA today if you have not notified group or if you are looking for a partner
Process Creation: Unix

- In Unix, processes are created using `fork()`
  ```c
  int fork()
  ```
  Usually combined with `exec()`
  `fork() + exec() \sim= CreateProcess()`

- `fork()`
  - Creates and initializes a new PCB
  - 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)
  - Places the PCB on the ready queue

- Fork returns **twice**
  - Returns the child’s PID to the parent, “0” to the child
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?
Example Output

[well ~]$ gcc t.c
[well ~]$ ./a.out
My child is 486
Child of a.out is 486
Duplicating Address Spaces

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

Parent

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

Child

Child pid = 0

Child pid = 486
Divergence

child_pid = 486

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

child_pid = 0

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

[well ~]$ gcc t.c
[well ~]$ ./a.out
My child is 486
Child of a.out is 486
[well ~]$ ./a.out
Child of a.out is 498
My child is 498

Why is the output in a different order?
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
  ```c
  while (1) {
      int sock = accept();
      if ((child_pid = fork()) == 0) {
          Handle client request
      } else {
          Close socket
      }
  }
  ```
Process Creation: Unix (2)

- Wait a second. How do we actually start a new program?
  \[
  \text{int exec(char *prog, char *argv[])}
  \]

- `exec()`
  - 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
  - \text{Note: It} \text{ does not} create a new process

- What does it mean for `exec` to return?
- What does it mean for `exec` to return with an error?
Process Creation: Unix (3)

- `fork()` is used to create a new process, `exec` is used to load a program into the address space

- What happens if you run “exec sh” in your shell?

- What happens if you run “exec ls” in your shell? Try it.

- `fork()` can return an error. Why might this happen?
Process Termination

- All good processes must come to an end. But how?
  - Unix: `exit(int status)`, NT: `ExitProcess(int status)`
- Essentially, free resources and terminate
  - Terminate all threads (coming up)
  - Close open files, network connections
  - Allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures, delete
- Note that a process does not need to clean up itself
  - OS will handle this on its behalf
**wait() a second...**

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Use `wait()` (*WaitForSingleObject*)
  -Suspends the current process until a child process ends
  - `waitpid()` suspends until the specified child process ends
- **Wait has a return value...what is it?**
- Unix: Every process must be reaped by a parent
  - What happens if a parent process exits before a child?
  - What do you think is a “zombie” process?
while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
        exec(cmd);
        panic("exec failed");
    } else {
        if (!(run_in_background))
            waitpid(child_pid);
    }
}
Processes

- Recall that …
  - A process includes many things:
    - An address space (all code and data pages)
    - OS resources (e.g., open files) and accounting info
    - Execution state (PC, SP, regs, etc.)
  - Processes are completely isolated from each other

- Creating a new process is costly because of all of the data structures that must be allocated and initialized
  - Recall struct proc in Solaris
  - Expensive even with OS tricks

- Communicating between processes is costly because most communication goes through the OS
  - Overhead of system calls and copying data
Parallel Programs

- Also 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
  - Cause each to map to the same address space to share data
    » They are all part of the same computation
  - Have the OS schedule these processes in parallel

- This situation is very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.
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

- Key idea: Separate resources from execution state
- Exec state also called thread of control, or thread
Recap: Process Components

- A process is named using its process ID (PID)
- A process contains all of the state for a program in execution

<table>
<thead>
<tr>
<th>Per-Process State</th>
</tr>
</thead>
<tbody>
<tr>
<td>An address space</td>
</tr>
<tr>
<td>The code for the executing program</td>
</tr>
<tr>
<td>The data for the executing program</td>
</tr>
<tr>
<td>A set of operating system resources</td>
</tr>
<tr>
<td>» Open files, network connections, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Per-Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td>An execution stack encapsulating the state of procedure calls</td>
</tr>
<tr>
<td>The program counter (PC) indicating the next instruction</td>
</tr>
<tr>
<td>A set of general-purpose registers with current values</td>
</tr>
<tr>
<td>Current execution state (Ready/Running/Waiting)</td>
</tr>
</tbody>
</table>
Threads

- Modern OSes (Mac OS, Windows, Linux) 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
- Threads become the unit of scheduling
  - Processes are now the **containers** in which threads execute
  - Processes become static, threads are the dynamic entities
Recap: Process Address Space

- **Address Space**
  - **0x00000000**
  - **0xFFFFFFFF**
  - **Stack**
  - **Heap** (Dynamic Memory Alloc)
  - **Static Data** (Data Segment)
  - **Code** (Text Segment)

- **SP** (Stack Pointer)
- **PC** (Program Counter)
Threads in a Process

- Stack (T1)
- Stack (T2)
- Stack (T3)
- Heap
- Static Data
- Code

Thread 1
- PC (T1)

Thread 2
- PC (T2)

Thread 3
- PC (T3)
Thread Design Space

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Thread/Process</td>
<td>Many Address Spaces</td>
</tr>
<tr>
<td>One Address Space (MSDOS)</td>
<td>(Early Unix)</td>
</tr>
<tr>
<td>Many Threads/Process</td>
<td>Many Address Spaces</td>
</tr>
<tr>
<td>One Address Space (Pilot, Java)</td>
<td>(Mac OS, Unix, Windows)</td>
</tr>
</tbody>
</table>