Lecture 4: Processes(2)/Threads
How to support the process abstraction?

- First, we’ll look at what state a process encapsulates
  - State of the virtual processor we are giving to each program

- Next we will talk about process behavior/CPU time sharing
  - How to implement the process illusion

- Next, we discuss how the OS implements this abstraction
  - What data structures it keeps, and the role of the scheduler

- Finally, we see the process interface offered to programs
  - How to use this abstraction?
  - What system calls are needed?
Process system call API

- Process creation: how to create a new process?
- Process termination: how to terminate and clean up a process
- Coordination between processes
  - Wait, waitpid, signal, inter-process communication, synchronization
- Other
  - E.g., set quotas or priorities, examine usage, …
Process Creation

- A process is created by another process
  - Why is this the case?
  - Parent is creator, child is created (Unix: ps “PPID” field)
  - What creates the first process (Unix: init (PID 0 or 1))?

- 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)
The system call on Windows for creating a process is called, surprisingly enough, CreateProcess:

```c
BOOL CreateProcess(char *prog, char *args) (simplified)
```

CreateProcess

- Creates and initializes a new PCB
- Creates and initializes a new address space
- Loads the program specified by “prog” into the address space
- Copies “args” into memory allocated in address space
- Initializes the saved hardware context to start execution at main (or wherever specified in the file)
- Places the PCB on the ready queue
In Unix, processes are created using `fork()`

```c
int fork()
```

`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
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

Parent

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

Child

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

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

---

`child_pid = 486`

`child_pid = 0`
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
    }
  }
  ```
Wait a second. How do we actually start a new program?

```c
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
  - **Note:** It 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 csh” 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 (next lecture)
  - 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
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: check your understanding

- What are the units of execution?
  - Processes

- How are those units of execution represented?
  - Process Control Blocks (PCBs)

- How is work scheduled in the CPU?
  - Process states, process queues, context switches

- What are the possible execution states of a process?
  - Running, ready, waiting, …

- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination

- How are processes created?
  - CreateProcess (NT), fork/exec (Unix)
Break/fork examples
Recall that …

- A process includes:
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting info
  - Execution state (PC, SP, regs, etc.)
  - PCB to keep track of everything

Processes are completely isolated from each other

But…
Some issues with processes

- Creating a new process is costly because of new address space and data structures that must be allocated and initialized
  - Recall struct proc in xv6 or Solaris

- Communicating between processes is costly because most communication goes through the OS
  - Inter Process Communication (IPC) – we will discuss later
  - 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.

- 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 (CoW helps).
  - **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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• An address space</td>
</tr>
<tr>
<td></td>
<td>• The code for the executing program</td>
</tr>
<tr>
<td></td>
<td>• The data for the executing program</td>
</tr>
<tr>
<td></td>
<td>• A set of operating system resources</td>
</tr>
<tr>
<td></td>
<td>» Open files, network connections, etc.</td>
</tr>
</tbody>
</table>

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

- Separate execution and resource container roles
  - The **thread** defines a sequential execution stream within a process (PC, SP, registers)
  - The **process** defines the address space, resources, and general process attributes (everything but 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
Threads in a Process

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

Thread 1
Thread 2
Thread 3
PC (T1)
PC (T2)
PC (T3)