

*OS Structure,  
Processes & Process Management*

*Don Porter*

*Portions courtesy Emmett Witchel*

1

### What is a Process?

- ◆ A process is a program during execution.
  - Program = static file (image)
  - Process = executing program = program + execution state.
- ◆ A process is the basic unit of execution in an operating system
  - Each process has a number, its process identifier (pid).
- ◆ Different processes may run different instances of the same program
  - E.g., my javac and your javac process both run the Java compiler
- ◆ At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

2

### Program to Process

- ◆ We write a program in e.g., Java.
- ◆ A compiler turns that program into an instruction list.
- ◆ The CPU interprets the instruction list (which is more a graph of basic blocks).

```
void X (int b) {
    if(b == 1) {
    ...
int main() {
    int a = 2;
    X(a);
}
```

3

### Process in Memory

- ◆ Program to process.
- ◆ What you wrote
 

```
void X (int b) {
    if(b == 1) {
    ...
int main() {
    int a = 2;
    X(a);
}
```
- ◆ What is in memory.
 

main; a = 2 X; b = 2	<b>Stack</b>
↓ ↑	
<b>Heap</b>	
void X (int b) { if(b == 1) { ... int main() { int a = 2; X(a); }	<b>Code</b>
- ◆ What must the OS track for a process?

4

### Processes and Process Management

#### Details for running a program

- ◆ A program consists of code and data
- ◆ On running a program, the loader:
  - reads and interprets the executable file
  - sets up the process' s memory to contain the code & data from executable
  - pushes "argc", "argv" on the stack
  - sets the CPU registers properly & calls "\_start()"
- ◆ Program starts running at \_start()
 

```
_start(args) {
    initialize_java();
    ret = main(args);
    exit(ret)
}
```

we say "process" is now running, and no longer think of "program"
- ◆ When main() returns, OS calls "exit()" which destroys the process and returns all resources

5

### Keeping track of a process

- ◆ A process has code.
  - OS must track program counter (code location).
- ◆ A process has a stack.
  - OS must track stack pointer.
- ◆ OS stores state of processes' computation in a process control block (PCB).
  - E.g., each process has an identifier (process identifier, or PID)
- ◆ Data (program instructions, stack & heap) resides in memory, metadata is in PCB (which is a kernel data structure in memory)

6

### Context Switching

- ◆ The OS periodically switches execution from one process to another
- ◆ Called a **context switch**, because the OS saves one execution context and loads another

7

### What causes context switches?

- ◆ Waiting for I/O (disk, network, etc.)
  - Might as well use the CPU for something useful
  - Called a blocked state
- ◆ Timer interrupt (preemptive multitasking)
  - Even if a process is busy, we need to be fair to other programs
- ◆ Voluntary yielding (cooperative multitasking)
- ◆ A few others
  - Synchronization, IPC, etc.

8

### Process Life Cycle

- ◆ Processes are always either *executing*, *waiting to execute* or *blocked waiting for an event to occur*

```

    graph TD
      Start([Start]) --> Ready([Ready])
      Ready --> Running([Running])
      Running --> Done([Done])
      Running --> Blocked([Blocked])
      Blocked --> Ready
  
```

- ◆ A preemptive scheduler will force a transition from running to ready. A non-preemptive scheduler waits.

9

### Process Contexts

Example: Multiprogramming

10

### When a process is waiting for I/O what is its scheduling state?

1. Ready
2. Running
3. Blocked ☺
4. Zombie
5. Exited

11

### Scheduling Processes

- ◆ OS has PCBs for active processes.
- ◆ OS puts PCB on an appropriate queue.
  - Ready to run queue.
  - Blocked for IO queue (Queue per device).
  - Zombie queue.
- ◆ Stopping a process and starting another is called a context switch.
  - 100-10,000 per second, so must be fast.

12

Why Use Processes?
<p>Consider a Web server  get network message (URL) from client  fetch URL data from disk  compose response  send response</p> <p><b>How well does this web server perform?</b>  <b>With many incoming requests?</b>  <b>That access data all over the disk?</b></p>
11

Why Use Processes?
<p>Consider a Web server  get network message (URL) from client  create child process, send it URL</p> <p style="margin-left: 100px;">Child  fetch URL data from disk  compose response  send response</p> <ul style="list-style-type: none"> <li>◆ If server has configuration file open for writing <ul style="list-style-type: none"> <li>➢ Prevent child from overwriting configuration</li> </ul> </li> <li>◆ How does server know child serviced request? <ul style="list-style-type: none"> <li>➢ Need return code from child process</li> </ul> </li> </ul>
14

Where do new processes come from?
<ul style="list-style-type: none"> <li>◆ Parent/child model</li> <li>◆ An existing program has to spawn a new one <ul style="list-style-type: none"> <li>➢ Most OSes have a special 'init' program that launches system services, logon daemons, etc.</li> <li>➢ When you log in (via a terminal or ssh), the login program spawns your shell</li> </ul> </li> </ul>
15

Approach 1: Windows CreateProcess
<ul style="list-style-type: none"> <li>◆ In Windows, when you create a new process, you specify a new program <ul style="list-style-type: none"> <li>➢ And can optionally allow the child to inherit some resources (e.g., an open file handle)</li> </ul> </li> </ul>
16

Approach 2: Unix fork/exec()
<ul style="list-style-type: none"> <li>◆ In Unix, a parent makes a copy of itself using fork() <ul style="list-style-type: none"> <li>➢ Child inherits everything, runs same program</li> <li>➢ Only difference is the return value from fork()</li> </ul> </li> <li>◆ A separate exec() system call loads a new program</li> <li>◆ Major design trade-off: <ul style="list-style-type: none"> <li>➢ How easy to inherit</li> <li>➢ Vs. Security (accidentally inheriting something the parent didn't intend)</li> <li>➢ Note that security is a newer concern, and Windows is a newer design...</li> </ul> </li> </ul>
17

The Convenience of separating Fork/Exec
<ul style="list-style-type: none"> <li>◆ Life with <code>CreateProcess(filename);</code> <ul style="list-style-type: none"> <li>➢ But I want to close a file in the child.  <code>CreateProcess(filename, list of files);</code></li> <li>➢ And I want to change the child's environment.  <code>CreateProcess(filename, CLOSE_FD, new_envp);</code></li> <li>➢ Etc. (and a very ugly etc.)</li> </ul> </li> <li>◆ <code>fork()</code> = split this process into 2 (new PID) <ul style="list-style-type: none"> <li>➢ Returns 0 in child</li> <li>➢ Returns pid of child in parent</li> </ul> </li> <li>◆ <code>exec()</code> = overlay this process with new program (PID does not change)</li> </ul>
18

### The Convenience of Separating Fork/Exec

- Decoupling fork and exec lets you do anything to the child's process environment without adding it to the CreateProcess API.

```

int pid = fork();           // create a child
if(0 == pid) {            // child continues here
    // Do anything (unmap memory, close net connections...)
    exec("program", argc, argv0, argv1, ...);
}

```

- fork() creates a child process that inherits:
  - identical copy of all parent's variables & memory
  - identical copy of all parent's CPU registers (except one)
- Parent and child execute at the same point after fork() returns:
  - by convention, for the child, fork() returns 0
  - by convention, for the parent, fork() returns the process identifier of the child
  - fork() return code a convenience, could always use getpid()

19

### Program Loading: exec()

- The exec() call allows a process to "load" a different program and start execution at main (actually \_start).
- It allows a process to specify the number of arguments (argc) and the string argument array (argv).
- If the call is successful
  - it is the same process ...
  - but it runs a different program !!
- Code, stack & heap is overwritten
  - Sometimes memory mapped files are preserved.
- Exec does not return!**

20

### General Purpose Process Creation

In the parent process:

```

main()
...
int pid = fork();           // create a child
if(0 == pid) {            // child continues here
    exec_status = exec("calc", argc, argv0, argv1, ...);
    printf("Something is horribly wrong\n");
    exit(exec_status);
} else {                  // parent continues
    printf("Who's your daddy?");
    ...
    child_status = wait(pid);
}

```

Exec should not return

21

### A shell forks and then execs a calculator

```

int pid = fork();
if(pid == 0) {
    close(".history");
    exec("/bin/calc");
} else {
    wait(pid);
}

int pid = fork();
if(pid == 0) {
    close(".history");
    exec("calc");
} else {
    wait(pid);
}

```

USER

OS

Process Control Blocks (PCBs)

```

pid = 128
open files = ".history"
last_cpu = 0

pid = 128
open files =
last_cpu = 0

```

22

### A shell forks and then execs a calculator

main; a = 2	Stack		Stack
↓ ↑		↓ ↑	
0xFC0933CA	Heap	0x43178050	Heap
int shell_main() {		int calc_main() {	
int a = 2;		int q = 7;	
... }	Code	... }	Code

USER

OS

Process Control Blocks (PCBs)

```

pid = 128
open files = ".history"
last_cpu = 0

pid = 128
open files =
last_cpu = 0

```

23

### At what cost, fork()?

- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent's memory and CPU registers to child's
  - Expensive !!**
- In 99% of the time, we call exec() after calling fork()
  - the memory copying during fork() operation is useless
  - the child process will likely close the open files & connections
  - overhead is therefore high
- vfork()
  - a system call that creates a process "without" creating an identical memory image
  - child process should call exec() almost immediately
  - Unfortunate example of implementation influence on interface
    - Current Linux & BSD 4.4 have it for backwards compatibility
  - Copy-on-write to implement fork avoids need for vfork

24

Orderly Termination: exit()
<ul style="list-style-type: none"> <li>◆ After the program finishes execution, it calls <code>exit()</code></li> <li>◆ This system call: <ul style="list-style-type: none"> <li>➢ takes the "result" of the program as an argument</li> <li>➢ closes all open files, connections, etc.</li> <li>➢ deallocates memory</li> <li>➢ deallocates most of the OS structures supporting the process</li> <li>➢ checks if parent is alive: <ul style="list-style-type: none"> <li>◆ If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state</li> <li>◆ If not, it deallocates all data structures, the process is dead</li> </ul> </li> <li>➢ cleans up all waiting zombies</li> </ul> </li> <li>◆ Process termination is the ultimate garbage collection (resource reclamation).</li> </ul>
21

The wait() System Call
<ul style="list-style-type: none"> <li>◆ A child program returns a value to the parent, so the parent must arrange to receive that value</li> <li>◆ The <code>wait()</code> system call serves this purpose <ul style="list-style-type: none"> <li>➢ it puts the parent to sleep waiting for a child's result</li> <li>➢ when a child calls <code>exit()</code>, the OS unblocks the parent and returns the value passed by <code>exit()</code> as a result of the wait call (along with the pid of the child)</li> <li>➢ if there are no children alive, <code>wait()</code> returns immediately</li> <li>➢ also, if there are zombies waiting for their parents, <code>wait()</code> returns one of the values immediately (and deallocates the zombie)</li> </ul> </li> </ul>
26

Process Control
<p>OS must include calls to enable special control of a process:</p> <ul style="list-style-type: none"> <li>◆ Priority manipulation: <ul style="list-style-type: none"> <li>➢ <code>nice()</code>, which specifies base process priority (initial priority)</li> <li>➢ In UNIX, process priority decays as the process consumes CPU</li> </ul> </li> <li>◆ Debugging support: <ul style="list-style-type: none"> <li>➢ <code>ptrace()</code>, allows a process to be put under control of another process</li> <li>➢ The other process can set breakpoints, examine registers, etc.</li> </ul> </li> <li>◆ Alarms and time: <ul style="list-style-type: none"> <li>➢ Sleep puts a process on a timer queue waiting for some number of seconds, supporting an alarm functionality</li> </ul> </li> </ul>
27

Tying it All Together: The Unix Shell
<pre> while(! EOF) {   read input   handle regular expressions   int pid = fork();           // create a child   if(pid == 0) {             // child continues here     exec("program", argc, argv0, argv1, ...);   }   else {                     // parent continues here     ...   } } </pre> <ul style="list-style-type: none"> <li>◆ Translates &lt;CTRL-C&gt; to the <code>kill()</code> system call with <code>SIGKILL</code></li> <li>◆ Translates &lt;CTRL-Z&gt; to the <code>kill()</code> system call with <code>SIGSTOP</code></li> <li>◆ Allows input-output redirections, pipes, and a lot of other stuff that we will see later</li> </ul>
28

Summary
<ul style="list-style-type: none"> <li>◆ Understand what a process is</li> <li>◆ The high-level idea of context switching and process states</li> <li>◆ How a process is created</li> <li>◆ Pros and cons of different creation APIs <ul style="list-style-type: none"> <li>➢ Intuition of copy-on-write fork and <code>vfork</code></li> </ul> </li> </ul>
29