CS 31: Intro to Systems Processes

Martin Gagne
Swarthmore College
March 30, 2017
Anatomy of a Process

• Abstraction of a running program
  – a dynamic “program in execution”

• OS keeps track of process state
  – What each process is doing
  – Which one gets to run next

• Basic operations
  – Suspend/resume (context switch)
  – Start (spawn), terminate (kill)
Timesharing

- Multiple processes, single CPU (or small number)
- Conceptually, each process makes progress over time
- In reality, each periodically gets quantum of CPU time
- Illusion of parallel progress by rapidly switching CPU
How is Timesharing Implemented?

• Kernel keeps track of progress of each process
• Characterizes state of process’s progress
  – Running: actually making progress, using CPU
  – Ready: able to make progress, but not using CPU
  – Blocked: not able to make progress, can’t use CPU
• Kernel selects a ready process, lets it run
  – Eventually, the kernel gets back control
  – Selects another ready process to run, …
Process State Diagram

- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready
### Kernel Maintains Process Table

<table>
<thead>
<tr>
<th>Process ID (PID)</th>
<th>State</th>
<th>Other info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1534</td>
<td>Ready</td>
<td>Saved context, …</td>
</tr>
<tr>
<td>34</td>
<td>Running</td>
<td>Mem. areas used, …</td>
</tr>
<tr>
<td>487</td>
<td>Ready</td>
<td>Saved context, …</td>
</tr>
<tr>
<td>9</td>
<td>Blocked</td>
<td>Cond. to unblock, …</td>
</tr>
</tbody>
</table>

- **List of processes and their states**
  - Also sometimes called “process control block (PCB)”

- **Other state info includes**
  - contents of CPU context
  - areas of memory being used
  - other information

---

Values of registers in use by process
Multiprogramming

• Given a running process
  – At some point, it needs a resource, e.g., I/O device
  – If resource is busy (or slow), process can’t proceed
  – “Voluntarily” gives up CPU to another process

• Mechanism: Context switching
Context Switching

• Allocating CPU from one process to another
  – First, save context of currently running process
  – Next, load context of next process to run
Context Switching

• Allocating CPU from one process to another
  – First, save context of currently running process
  – Next, load context of next process to run

• Loading the context
  – Load general registers, stack pointer, etc.
  – Load program counter (must be last instruction!)
How a Context Switch Occurs

• Process makes system call or is interrupted (TRAP)
  – These are the only ways of entering the kernel
• In hardware
  – Switch from user to kernel mode: amplifies power
  – Go to fixed kernel location: interrupt/trap handler
• In software (in the kernel code)
  – Save context of last-running process
  – Conditionally
    • Select new process from those that are ready
    • Restore context of selected process
  – OS returns control to a process from interrupt/trap
Why shouldn’t processes control context switching?

A. It would cause too much overhead.

B. They could refuse to give up the CPU.

C. They don’t have enough information about other processes.

D. Some other reason(s).
Time Sharing / Multiprogramming

• Given a running process
  – At some point, it needs a resource, e.g., I/O device
  – If resource is busy (or slow), process can’t proceed
  – “Voluntarily” gives up CPU to another process

• Mechanism: Context switching
• Policy: CPU scheduling
The CPU Scheduling Problem

• Given multiple processes, but only one CPU
• How much CPU time does each process get?
• Which process do we run next?

• Possibilities
  – Keep CPU till done
  – Each process uses CPU a bit and passes it on
  – Each process gets proportional to what they pay
Which CPU scheduling policy is the best?

A. Processes keep CPU until done (maximize throughput)

B. Processes use a fraction of CPU and pass it on (ensure fairness)

C. Processes receive CPU in proportion to their priority or what they pay (prioritize importance)

D. Other (explain)
There is No Single Best Policy

• Depends on the goals of the system

• Different for…
  – Your personal computer
  – Large time-shared (super) computer
  – Computer controlling a nuclear power plant

• Often have multiple (conflicting) goals
Common Policies

- Details beyond scope of this course (Take OS)

- Different classes of processes
  - Those blessed by administrator (high/low priority)
  - Everything else
Common Policies

• Special class gets special treatment (varies)

• Everything else: *roughly* equal time quantum
  – “Round robin”
  – Give priority boost to processes that frequently perform I/O (Why?)

• “I/O bound” processes frequently block.
  – If we want them to get equal CPU time, we need to give them the CPU more often.
Linux’s Policy
(You’re not responsible for this.)

• Special “real time” process classes (high prio)

• Other processes:
  – Keep red-black BST of process, organized by how much CPU time they’ve received.
  – Pick the ready with process that has run for the shortest time thus far.
  – Run it, update it’s CPU usage time, add to tree.

• Interactive processes: Usually blocked, low total run time, high priority.
Where do processes come from?

The `fork()` system call creates a new process.

On boot, the kernel spawns the “init” process.

Init calls `fork()` to create child processes.

Those processes can also create children with `fork`. 
Managing Processes

• Given a process, how do we make it execute the program we want?

• Model: fork() a new process, execute program
fork()

- System call (function provided by OS kernel)
  - Creates a duplicate of the requesting process
    - Process is cloning itself:
      - CPU context
      - Memory “address space”

![Diagram of memory regions before and after fork()](image_url)
fork() return value

- The two processes are identical in every way, except for the return value of `fork()`.
  - The child gets a return value of 0.
  - The parent gets a return value of child’s PID.

```c
pid_t pid = fork(); // both continue after call
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Which process executes next? Child? Parent? Some other process?

Up to OS to decide. No guarantees. Don’t rely on particular behavior!
How many hello’s will be printed?

fork();
printf("hello");
if (fork()) {
    printf("hello");
}
fork();
printf("hello");
Common `fork()` usage: Shell

- A “shell” is the program controlling your terminal (e.g., bash).

- It `fork()`’s to create new processes, but we don’t want a clone (another shell).

- We want the child to execute some other program: `exec()` family of functions.
exec()

- Family of functions (execl, execvp, execv, …).

- Replace the current process with a new one.

- Loads program from disk:
  - Old process is overwritten in memory.
  - Does not return unless error.
Common `fork()` usage: Shell

1. `fork()` child process.

2. `exec()` desired program to replace child’s address space.

2. `wait()` for child process to terminate.

3. repeat...
Common `fork()` usage: Shell

1. `fork()` child process.
Common `fork()` usage: Shell

2. parent: `wait()` for child to finish
Common `fork()` usage: Shell

2. child: `exec()` user-requested program
Common `fork()` usage: Shell

2. child: `exec()` user-requested program
Common `fork()` usage: Shell

3. child program terminates, cycle repeats
Common `fork()` usage: Shell

3. child program terminates, cycle repeats
Process Termination

• When does a process die?
  – It calls `exit(int status);`
  – It returns (an int) from main
  – It receives a termination signal (from the OS or another process)

• Key observation: the dying process produces status information.

• Who looks at this?
• The parent process!
Reaping Children
(How’s THAT for a slide title!!)

• \texttt{wait()}: parents reap their dead children
  – Given info about why child died, exit status, etc.

• Two variants:
  – \texttt{wait()}: wait for and reap next child to exit
  – \texttt{waitpid()}: wait for and reap specific child

• This is how the shell determines whether or not the program you executed succeeded.
Common `fork()` usage: Shell

1. `fork()` child process.

2. `exec()` desired program to replace child’s address space.

3. `wait()` for child process to terminate.
   - Check child’s result, notify user of errors.

4. repeat...
What should happen if dead child processes are never reaped? (That is, the parent has not \texttt{wait()}ed on them?)

A. The OS should remove them from the process table (process control block / PCB).

B. The OS should leave them in the process table (process control block / PCB).
What should happen if dead child processes are never reaped? (That is, the parent has not \texttt{wait()}ed on them?)

A. The OS should remove them from the process table (process control block / PCB).

B. The OS should leave them in the process table (process control block / PCB).

C. The neglected processes seek revenge as undead in the afterlife.
“Zombie” Processes

- Zombie: A process that has terminated but not been reaped by parent. (AKA defunct process)

- Does not respond to signals (can’t be killed)

- OS keeps their entry in process table:
  - Parent may still reap them, want to know status
  - Don’t want to re-use the process ID yet

Basically, they’re kept around for bookkeeping purposes, but that’s much less exciting...
Summary: system calls for processes

• **fork**: spawns new process.
  • Called once, Returns twice (in parent and child process).

• **exit**: terminates own process.
  • Called once, never returns.
  • Puts it into “zombie” status.

• **wait** or **waitpid**: reap terminated children.

• **exec** family: runs new program in existing process.
  • Called once, (normally) never returns.
Signals

• How does a parent process know that a child has exited (and that it needs to call wait)?

• Signals: inter-process notification mechanism
  – Info that a process (or OS) can send to a process.
    • Please terminate yourself (SIGTERM)
    • Stop NOW (SIGKILL)
    • Your child has exited (SIGCHLD)
    • You’ve accessed an invalid memory address (SIGSEGV)
    • Many more (SIGWINCH, SIGUSR1, SIGPIPE, …)
Signals

**Signal**: a software interrupt: a small message to tell a process that some event has happened.

- OS sends a signal to a process
  - On behalf of another process that called the **kill** syscall
  - As the result of some event (NULL pointer dereference)

- A process **receives** a signal
  Asynchronous: signalee doesn’t know when it will get one
  A signal is **pending** before a process receives it

- A signal **interrupts** the receiving process, which then runs **signal handler** code
  - default handlers for each signal type in OS
  - programmer can also add signal handler code
Signals

OS identifies specific signal by its number, examples:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., ctrl-c from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate</td>
<td>Invalid memory reference (e.g. NULL ptr)</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>

Sending Signals:

Unix command:

$ kill -9 1234  # send SIGKILL signal to process 1234

System call:

kill(1234, SIGKILL);  // send SIGKILL to process 1234

Implicitly sent: side-effect of program doing something

(NULL ptr dereference causes SIGSEGV)
Receiving a Signal

• A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal.

• Three possible ways to react:
  • *Ignore* the signal (do nothing) not all signals can be ignored (e.g. SIGKILL)
  • *Terminate* the process on receipt of signal
  • *Catch* the signal by executing a user-level function called *signal handler*
Installing Signal Handlers

```c
signal(int signum, handler_t *handler);
```

- Modifies the default action associated with the receipt of a particular signal.

- `handler` is a *signal handler* function
  - When program receives signal, it jumps to start executing the `handler` function.
  - When the `handler` done executing, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.
Up Next

Virtual Memory

• How the OS gives every process the illusion they all have the same address space
• How the OS deals with memory allocation