# Processes

**CSE 351 Summer 2020**

**Instructor:** Porter Jones  
**Teaching Assistants:** Amy Xu, Callum Walker, Sam Wolfson, Tim Mandzyuk

<table>
<thead>
<tr>
<th>Refresh Type</th>
<th>Example Shortcuts</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft Refresh</td>
<td>Gmail Refresh button</td>
<td>Requests update within JavaScript</td>
</tr>
<tr>
<td>Normal Refresh</td>
<td>F5, Ctrl-R, ⌘R</td>
<td>Refreshes page</td>
</tr>
<tr>
<td>Hard Refresh</td>
<td>Ctrl-F5, Ctrl-⌘, ⌘R</td>
<td>Refreshes page including cached files</td>
</tr>
<tr>
<td>Harder Refresh</td>
<td>Ctrl-⌘-Hyper-ESC-R-F5</td>
<td>Remotely cycles power to datacenter</td>
</tr>
<tr>
<td>Hardest Refresh</td>
<td>Ctrl-⌘-⌘-R-F5-F-5-ESC-0-0-⌘-Scroll Lock</td>
<td>Internet starts over from ARPANET</td>
</tr>
</tbody>
</table>

[http://xkcd.com/1854/]
Administrivia

- Questions doc: https://tinyurl.com/CSE351-8-5

- hw17 due Friday (8/7) – 10:30am
- hw18 due Monday (8/10) – 10:30am

- Unit Summary 2 Due Tonight! (8/5) – 11:59pm

- Lab 4 due Wednesday (8/12) – 11:59pm
  - All about caches!
Roadmap

C:
```c
struct car {
    int miles;
    int gals;
};
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:
```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:
```assembly
get_mpg:
    pushq  %rbp
    movq   %rsp, %rbp
    ...
    popq   %rbp
    ret
```

Machine code:
```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110101000011111
```

Computer system:
```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110101000011111
```

OS:
```
Windows 10
OS X Yosemite
```

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Leading Up to Processes

❖ System Control Flow
  ▪ Control flow
  ▪ Exceptional control flow
  ▪ Asynchronous exceptions (interrupts)
  ▪ Synchronous exceptions (traps & faults)
Control Flow

❖ **So far:** we’ve seen how the flow of control changes as a *single program* executes

❖ **Reality:** multiple programs running *concurrently*
  ▪ How does control flow across the many components of the system?
  ▪ In particular: More programs running than CPUs

❖ *Exceptional control flow* is basic mechanism used for:
  ▪ Transferring control between *processes* and OS
  ▪ Handling *I/O* and *virtual memory* within the OS
  ▪ Implementing multi-process apps like shells and web servers
  ▪ Implementing concurrency
Control Flow

❖ Processors do only one thing:
  ▪ From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  ▪ This sequence is the CPU’s control flow (or flow of control)

Physical control flow

<startup>
instr_1
instr_2
instr_3
...
instr_n
<shutdown>

time
Altering the Control Flow

❖ Up to now, two ways to change control flow:
  ▪ Jumps (conditional and unconditional)
  ▪ Call and return
  ▪ Both react to changes in *program state*

❖ Processor also needs to react to changes in *system state*
  ▪ Unix/Linux user hits “Ctrl-C” at the keyboard
  ▪ User clicks on a different application’s window on the screen
  ▪ Data arrives from a disk or a network adapter
  ▪ Instruction divides by zero
  ▪ System timer expires

❖ Can jumps and procedure calls achieve this?
  ▪ No – the system needs mechanisms for “*exceptional*” control flow!
Java Digression

- Java has exceptions, but they’re *something different*
  - **Examples**: NullPointerException, MyBadThingHappenedException, ...
  - `throw` statements
  - `try/catch` statements (“throw to youngest matching catch on the call-stack, or exit-with-stack-trace if none”)

- Java exceptions are for reacting to (unexpected) program state
  - Can be implemented with stack operations and conditional jumps
  - A mechanism for “many call-stack returns at once”
  - Requires additions to the calling convention, but we already have the CPU features we need

- System-state changes on previous slide are mostly of a different sort (asynchronous/external except for divide-by-zero) and implemented very differently

This is extra (non-testable) material
Exceptional Control Flow

- Exists at all levels of a computer system

- Low level mechanisms
  - **Exceptions**
    - Change in processor’s control flow in response to a system event (i.e. change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software

- Higher level mechanisms
  - **Process context switch**
    - Implemented by OS software and hardware timer
  - **Signals**
    - Implemented by OS software
    - We won’t cover these – see CSE451 and CSE/EE474
Exceptions

- **An exception** is transfer of control to the operating system (OS) kernel in response to some **event** (i.e. change in processor state)
  - Kernel is the memory-resident part of the OS
  - **Examples**: division by 0, page fault, I/O request completes, Ctrl-C

How does the system know where to jump to in the OS?
Exception Table

- A jump table for exceptions (also called *Interrupt Vector Table*)
  - Each type of event has a unique exception number \( k \)
  - \( k \) = index into exception table (a.k.a interrupt vector)
  - Handler \( k \) is called each time exception \( k \) occurs

This is extra (non-testable) material
### Exception Table (Excerpt)

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Description</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>General protection fault</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>32-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>

This is extra (non-testable) material.
Leading Up to Processes

❖ System Control Flow
  ▪ Control flow
  ▪ Exceptional control flow
  ▪ Asynchronous exceptions (interrupts)
  ▪ Synchronous exceptions (traps & faults)
Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to “next” instruction

- **Examples:**
  - I/O interrupts
    - Hitting Ctrl-C on the keyboard
    - Clicking a mouse button or tapping a touchscreen
    - Arrival of a packet from a network
    - Arrival of data from a disk
  - Timer interrupt
    - Every few milliseconds, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - **Traps**
    - **Intentional**: transfer control to OS to perform some function
    - **Examples**: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - **Faults**
    - **Unintentional** but possibly recoverable
    - **Examples**: page faults, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - **Aborts**
    - **Unintentional** and unrecoverable
    - **Examples**: parity error, machine check (hardware failure detected)
    - Aborts current program
System Calls

❖ Each system call has a unique ID number
❖ Examples for Linux on x86-64:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
<td>Read file</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
<td>Write file</td>
</tr>
<tr>
<td>2</td>
<td>open</td>
<td>Open file</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
<td>Close file</td>
</tr>
<tr>
<td>4</td>
<td>stat</td>
<td>Get info about file</td>
</tr>
<tr>
<td>57</td>
<td>fork</td>
<td>Create process</td>
</tr>
<tr>
<td>59</td>
<td>execve</td>
<td>Execute a program</td>
</tr>
<tr>
<td>60</td>
<td>_exit</td>
<td>Terminate process</td>
</tr>
<tr>
<td>62</td>
<td>kill</td>
<td>Send signal to process</td>
</tr>
</tbody>
</table>
Traps Example: Opening File

- **User calls** `open(filename, options)`
- **Calls** `__open` function, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00    mov   $0x2,%eax  # open is syscall 2
e5d7e:   0f 05             syscall  # return value in %rax
e5d80:   48 3d 01 f0 ff ff  cmp   $0xffffffffffffff001,%rax
...
e5dfa:   c3                retq
```

- `%rax` contains syscall number
- Other arguments in `%rdi, %rsi, %rdx, %r10, %r8, %r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`
Fault Example: Page Fault w/Swapped Page

- User writes to memory location
- That portion (page) of user’s memory is currently swapped out (on disk)

```c
int a[1000];
int main () {
    a[500] = 13;
}
```

```
80483b7:  c7 05 10 9d 04 08 0d  movl  $0xd,0x8049d10
```

- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try
Fault Example: Invalid Memory Reference

```c
int a[1000];
int main() {
    a[5000] = 13;
}
```

```
80483b7:   c7 05 60 e3 04 08 0d  movl   $0xd,0x804e360
```

- Page fault handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with “segmentation fault”
Summary

❖ Exceptions

- Events that require non-standard control flow
- Generated externally (interrupts) or internally (traps and faults)
- After an exception is handled, one of three things may happen:
  - Re-execute the current instruction
  - Resume execution with the next instruction
  - Abort the process that caused the exception
Processes

❖ Processes and context switching
❖ Creating new processes
  ▪ `fork()`, `exec*()`, and `wait()`
❖ Zombies
What is a process? 

It’s an illusion!
What is a process?

❖ Another *abstraction* in our computer system
  ▪ Provided by the OS
  ▪ OS uses a data structure to represent each process
  ▪ Maintains the *interface* between the program and the underlying hardware (CPU + memory)

❖ What do *processes* have to do with *exceptional control flow*?
  ▪ Exceptional control flow is the *mechanism* the OS uses to enable *multiple processes* to run on the same system

❖ What is the difference between:
  ▪ A processor? A program? A process?
Processes

- A **process** is an instance of a running program
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:
  - **Logical control flow**
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called **context switching**
  - **Private address space**
    - Each program seems to have exclusive use of main memory
    - Provided by kernel mechanism called **virtual memory**
What is a process?

It’s an *illusion*!
What is a process?

It’s an illusion!

Operating System

CPU

Process 1

Process 2

Process 3

Process 4

Disk

/Applications/

Chrome.exe

Slack.exe

PowerPoint.exe
Multiprocessing: The Illusion

- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, ...
  - Background tasks
    - Monitoring network & I/O devices
Multiprocessing: The Reality

- Single processor executes multiple processes *concurrently*
  - Process executions interleaved, CPU runs *one at a time*
  - Address spaces managed by virtual memory system (later in course)
  - *Execution context* (register values, stack, ...) for other processes saved in memory
Multiprocessing

- **Context switch**
  1) **Save current registers in memory**
Context switch

1) Save current registers in memory
2) Schedule next process for execution
Multiprocessing

❖ Context switch

1) Save current registers in memory
2) Schedule next process for execution
3) Load saved registers and switch address space
Multiprocessing: The (Modern) Reality

- **Multicore processors**
  - Multiple CPUs (“cores”) on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Kernel schedules processes to cores
    - *Still constantly swapping processes*
Concurrent Processes

- Each process is a logical control flow
- Two processes run concurrently (are concurrent) if their instruction executions (flows) overlap in time
  - Otherwise, they are sequential
- Example: (running on single core)
  - Concurrent: A & B, A & C
  - Sequential: B & C

Assume only one CPU
User’s View of Concurrency

- Control flows for concurrent processes are physically disjoint in time
  - CPU only executes instructions for one process at a time
  - However, the user can think of concurrent processes as executing at the same time, in parallel
Context Switching

- Processes are managed by a *shared* chunk of OS code called the *kernel*
  - The kernel is not a separate process, but rather runs as part of a user process

- In x86-64 Linux:
  - Same address in each process refers to same shared memory location
Context Switching

- Processes are managed by a *shared* chunk of OS code called the kernel
  - The kernel is not a separate process, but rather runs as part of a user process
  - Context switch passes control flow from one process to another and is performed using kernel code

Assume only one CPU

![Diagram showing context switching between two processes](image-url)
Processes

❖ Processes and context switching

❖ Creating new processes
  ▪ `fork()`, `exec*()`, and `wait()`

❖ Zombies
Creating New Processes & Programs

Process 1

“Memory”
- Stack
- Heap
- Data
- Code

“CPU”
- Registers

Process 2

“Memory”
- Stack
- Heap
- Data
- Code

“CPU”
- Registers

fork()

exec*()

Chrome.exe
Creating New Processes & Programs

❖ fork-exec model (Linux):
  ▪ `fork()` creates a copy of the current process
  ▪ `exec*()` replaces the current process’ code and address space with the code for a different program
    • Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  ▪ `fork()` and `execve()` are system calls

❖ Other system calls for process management:
  ▪ `getpid()`
  ▪ `exit()`
  ▪ `wait()`, `waitpid()`
fork: Creating New Processes

- **pid_t fork(void)**
  - Creates a new “child” process that is *identical* to the calling “parent” process, including all state (memory, registers, etc.)
  - Returns 0 to the child process
  - Returns child’s process ID (PID) to the parent process

- Child is *almost* identical to parent:
  - Child gets an identical (but separate) copy of the parent’s virtual address space
  - Child has a different PID than the parent

- **fork is unique (and often confusing) because it is called once but returns “twice”**

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding `fork()`

**Process X  (parent; PID X)**

```c
pid_t fork_ret = fork();
if (fork_ret == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

**Process Y  (child; PID Y)**

```c
pid_t fork_ret = fork();
if (fork_ret == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding `fork()`

**Process X** (parent; PID X)

```c
pid_t fork_ret = fork();
if (fork_ret == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

**Process Y** (child; PID Y)

```c
fork_ret = Y
pid_t fork_ret = fork();
if (fork_ret == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

```c
fork_ret = 0
pid_t fork_ret = fork();
if (fork_ret == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding `fork()`

**Process X** (parent; PID X)

```c
pid_t fork_ret = fork();
if (fork_ret == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

**Process Y** (child; PID Y)

```c
fork_ret = Y
```

```c
pid_t fork_ret = fork();
if (fork_ret == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

**Which one appears first?**

```
hello from parent
```

```
fork_ret = 0
```

```
hello from child
```
Fork Example

```c
void fork1() {
    int x = 1;
    pid_t fork_ret = fork();
    if (fork_ret == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

- Both processes continue/start execution after `fork`
  - Child starts at instruction after the call to `fork` (storing into `pid`)
- Can’t predict execution order of parent and child
- Both processes start with `x = 1`
  - Subsequent changes to `x` are independent
- Shared open files: stdout is the same in both parent and child
Modeling `fork` with Process Graphs

- A process graph is a useful tool for capturing the partial ordering of statements in a concurrent program
  - Each vertex is the execution of a statement
  - $a \rightarrow b$ means $a$ happens before $b$
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inedges

- Any topological sort of the graph corresponds to a feasible total ordering
  - Total ordering of vertices where all edges point from left to right
Fork Example: Possible Output

```c
void fork1() {
    int x = 1;
    pid_t fork_ret = fork();
    if (fork_ret == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

```
x=1 fork --x printf printf
x=0 Parent Bye
x=2 ++x printf Bye
```

Summary

❖ Processes
  ▪ At any given time, system has multiple active processes
  ▪ On a one-CPU system, only one can execute at a time, but each process appears to have total control of the processor
  ▪ OS periodically “context switches” between active processes
    • Implemented using *exceptional control flow*

❖ Process management
  ▪ `fork`: one call, two returns
  ▪ `execve`: one call, usually no return
  ▪ `wait` or `waitpid`: synchronization
  ▪ `exit`: one call, no return