# Processes

CSE 351 Summer 2021

**Instructor:**  
Mara Kirdani-Ryan

**Teaching Assistants:**  
Kashish Aggarwal  
Nick Durand  
Colton Jobs  
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-⌥, ⌘KeyPress</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-⌘-Hyper-Esc-O-0-0-0-_SCROLL LOCK</td>
<td>Internet starts over from ARPANET</td>
</tr>
</tbody>
</table>

Gentle, Loving Reminders

- hw16 due Tonight! hw17 due Friday!
- Lab 4 due Monday (8/9!)
  - hw16 should be helpful preparation
  - Caches, caches, caches

- Final deadline for US#2 is tomorrow!
  - Today by 8pm for one late day
Learning Objectives

Understanding this lecture means you can:

- Explain the role of exceptions, and one way that they’re implemented (exception tables)
- Differentiate between synchronous and asynchronous exceptions, and explain how systems respond to both
- Explain how we can have multiple processes running on a single processor, and how we can create new processes
- Describe the first operating systems, in context with the first computers, and the first programmers
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*
Control Flow

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

\[
\begin{align*}
\text{Physical control flow} \\
\text{<startup>} \\
\text{instr}_1 \\
\text{instr}_2 \\
\text{instr}_3 \\
\vdots \\
\text{instr}_n \\
\text{<shutdown>}
\end{align*}
\]
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
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 the mechanism for:
  - Transferring control between *processes* and OS
  - Handling *I/O* and *virtual memory* within the OS
  - Implementing multi-process apps (shells, web servers)
  - Implementing concurrency
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 a 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

```
User Code

event
current_instr
next_instr

OS Kernel Code

exception

exception processing by exception handler, then:
• return to current_instr,
• return to next_instr, OR
• abort
```
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

**Diagram:**

- **User Code**
- **OS Kernel Code**

- **event** → **current_instr** → **exception** → **next_instr**

**exception processing by exception handler, then:**
- *return to current_instr*,
- *return to next_instr*, OR
- *abort*

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

- A jump table for exceptions (or, *Interrupt Vector Table*)
  - Each event type has an exception number $k$
  - $k$ indexes into the exception table
  - 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
How are you feeling about exceptions?
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`

```plaintext
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 $0xfffffffffffff001,%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
```

User code

OS Kernel code

- 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;
}
```

User Process

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

OS

- 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, 3 potential scenarios:
    - Re-execute the current instruction
    - Resume execution with the next instruction
    - Abort the process that caused the exception
Exception flow, feeling ok?
Processes

- Processes and context switching
- Creating new processes
  - `fork()`, `exec*()`, and `wait()`
What is a process? It’s an abstraction!

Process 1

Memory

- Stack
- Heap
- Data
- Code

CPU

- Registers
  - %rip

Disk

Chrome.exe
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 sole use of CPU
    - Provided via **context switching**
  - *Private address space*
    - Each program seems to have sole use of memory
    - Provided via **virtual memory**
What is a process?

It’s an abstraction!

Computer

Process 2

Process 1

Process 3

Process 4

Disk

/Applications/

Disk

Disk

Disk

Chrome.exe

Slack.exe

PowerPoint.exe
What is a process?

It’s an abstraction!

Computer

Process 2

“Memory”
- Stack
- Heap
- Data
- Code

“CPU”
- Registers

Operating System

CPU

Process 3

“Memory”
- Stack
- Heap
- Data
- Code

“CPU”
- Registers

Process 1

“Memory”
- Stack
- Heap
- Data
- Code

“CPU”
- Registers

Process 4

“Memory”
- Stack
- Heap
- Data
- Code

“CPU”
- Registers

Disk

/Applications/

Chrome.exe
Slack.exe
PowerPoint.exe

It’s an abstraction!
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 CPU executes multiple processes *concurrently*
  - Process executions interleaved, CPU runs *one at a time*
  - Address spaces managed by *virtual memory* system
  - *Execution context* (register values, stack, ...) for other processes saved in memory
Multiprocessing

- Context switch
  1) Save current registers in memory
Multiprocessing

- **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 1 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 one process at a time

- However, the user can think of concurrent processes as executing at the same time, in parallel

Assume only one CPU
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

Assume only one CPU
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

![Diagram of context switching]

Assume only one CPU
Context Switching, feeling ok?
Processes

- Processes and context switching
- **Creating new processes**
  - `fork()`, `exec*()`, and `wait()`
- Zombies

Take OS to learn more!
Creating New Processes & Programs

Process 1

```
<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
<th>Data</th>
<th>Code</th>
</tr>
</thead>
</table>
```

```
| CPU    | Registers |
```

```
| Memory |
```

Process 2

```
<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
<th>Data</th>
<th>Code</th>
</tr>
</thead>
</table>
```

```
| CPU    | Registers |
```

```
| Memory |
```

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

Which one appears first?
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);
}
```

Checking in!

- Are the following sequences of outputs possible?

```c
void nestedfork() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Seq 1:
- L0
- L1
- Bye
- Bye
- Bye
- L2

Seq 2:
- L0
- Bye
- Bye
- L1
- Bye
- L2
- Bye

- 🐶 No
- 🐕 No
- 🐐 Yes
- 🎀 Yes
- 🌬️ Help!
Summary

- Processes
  - At any time, system has multiple active processes
  - On a one-CPU system, only one can execute at a time, but each process appears to exclusively use the CPU
  - OS periodically “context switches” between processes
    - Implemented using *exceptional control flow*

- Process management
  - `fork`: one call, two returns
  - Take OS to learn more about `exec()` and `wait()`
The first operating systems
The first computers

- **Computer**: one who computes

The women of Bletchley Park, Credit: BBC
The first Computer

“Great” man

“Great” machine
Babbage, inspo by Gaspard De Prony

- Applied division of labor to produce logarithmic tables
- “…manufacture logarithms as one manufactures pins”
- 5 experts, 8 managers, 70 human computers
The first programmers

1940s

Jean Jennings (left), Marlyn Wescoff (center), and Ruth Lichterman program ENIAC at the University of Pennsylvania, circa 1946.
Photo: Corbis
What’s an operating system?

- Basically, a resource manager!
- Computers have all sorts of resources…
  - CPU, memory, disks, network cards, etc.
- Operating systems try to use those efficiently!
  - Ideally, the “user” only worries about their program

- Today: OS abstraction gives multi-process machines without any change from programmers
  - “Each program seems to have exclusive use of CPU/memory”
Why this abstraction?

Backwards compatibility! It’s where we started!
Tabulating Cards
A single instruction, via punch card
Programs, via punch card
Computer Operators
First operating systems

- Early computational resources:
  - A very, very expensive machine that could only run one program at a time
  - Basically just a big, programmable calculator

- Operating computational machines meant:
  1. Receive punch card programs from all sorts of people
  2. Prioritize and run programs
  3. Record results, and return to programmer

  - Also, manage the machine if something goes wrong!
    - If a punch card jams the machine
    - If a program doesn’t stop running
“Robot work” or “Human work”?
What happened?

- Computers slowly added more “features” that made the operators job easier
  - Security features: allow auditing of programs
  - Magnetic tape allows a digital “queue” that the computer could select from

- Slowly, operators jobs are automated away…
  - “optional” features become standard
  - “monitors” reassign HW resources as needed

- Good-paying job for women, gone
Summary

- Programs used to be physical stacks of cards that operators had to manage
- Operators maintained the OS abstraction
  - Potentially with more waiting time and a job queue
  - Viewed as “robot work” -- operating the machine
- Slowly, new computers can “operate themselves”
  - “Great! We won’t need to hire an operator!”
- We’ve seen this before, we’re seeing it now!
  - Computers, Programmers, Operating Systems