Processes
CSE 351 Winter 2024

Instructor:
Justin Hsia

Teaching Assistants:
Adithi Raghavan
Aman Mohammed
Connie Chen
Eyoel Gebre
Jiawei Huang
Malak Zaki
Naama Amiel
Nathan Khuat
Nikolas McNamee
Pedro Amarante
Will Robertson

https://ptbd.jwels.berlin/comic/20/
Relevant Course Information

❖ HW20 due tonight, HW21 due Friday, HW22 due Monday

❖ Lab 4 due Friday

❖ Lab 5 due next Friday, 3/8
  ▪ Section this week is to get your started with Lab 5
  ▪ Can use one late day; must be submitted by Sunday, 3/10

❖ Final March 11-13, regrade requests only Monday, March 18
Winter 2024 Crunch

- This quarter is unusually short
  - Winter is always the shortest of the year for MWF classes due to Monday holidays
  - This year, we also lost the first Monday due to New Year’s

- 29 → 26 lectures compared against 23au
  - Condensed “x86-64 Programming” from 4 lessons to 3
  - Cut “Exceptional Control Flow” (related to Processes)
  - Cutting “C and Java”

- Assignments compressed, too
  - Cut a number of homework questions throughout the quarter
  - Less time than usual to work on Lab 4 and 5
  - End topics (Processes, VM) will be stressed less than usual on Final
Processes
Lesson Summary (1/4)

❖ **A process** is an instance of a running program and provides two key abstractions: *logical control flow* and *private address space*.

❖ Multiple running processes can be run *concurrently* via **context switching**.
  - Parallelism only possible with multiple CPUs/cores.
Lesson Summary (2/4)

❖ The **fork-exec model**

- Every process is assigned a unique **process ID** (pid)
- Every process has a parent process except for init/system (pid 1)
- `fork()` returns 0 to child, child’s PID to parent
- `exec()` replaces the current process’ code and address space with the code for a different program

![Diagram showing fork-exec model](image)
Lesson Summary (3/4)

❖ Terminating a process
  ▪ Return from main() or explicit call to exit(status)
  ▪ Passes a **status code** (main’s return value or exit’s argument) to parent process
    • 0 for normal exit, nonzero for abnormal exit

❖ Processes and resources
  ▪ A terminated (**zombie**) process still consumes system resources until reaped
  ▪ Child is reaped when parent process terminates or explicitly calls wait/waitpid
  ▪ Orphaned children reaped by init/systemd
Lesson Summary (4/4)

❖ Concurrency and process diagrams

- Concurrently executing processes are scheduled non-deterministically by the operating system.
- A process graph is a useful tool for capturing the partial ordering of statements in a concurrent program.
  - Vertices are program statements, directed edges capture sequencing within a process.
  - Flexible visualization tool:

```
Parent

x=0  --x.printf  Bye
fork

Child

x=2  ++x.printf  Bye
fork

HP

x=1  x=0.printf  Bye
fork

HC

exit

HP

fork  printf  wait  printf

CT

Bye
```
Lesson Q&A

❖ Learning Objectives:
  ▪ Define exceptional control flow and explain its importance in enabling concurrency and error handling.
  ▪ Design process graphs to determine potential orderings of concurrent execution.

❖ What lingering questions do you have from the lesson?
  ▪ Chat with your neighbors about the lesson for a few minutes to come up with questions
Processes – Practice
Polling Questions (1/2)

❖ Are the following sequences of outputs possible?

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

<table>
<thead>
<tr>
<th>Seq 1:</th>
<th>Seq 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>L0</td>
</tr>
<tr>
<td>L1</td>
<td>Bye</td>
</tr>
<tr>
<td>Bye</td>
<td>L1</td>
</tr>
<tr>
<td>Bye</td>
<td>L2</td>
</tr>
<tr>
<td>Bye</td>
<td>Bye</td>
</tr>
<tr>
<td>L2</td>
<td>Bye</td>
</tr>
</tbody>
</table>

A. No  No
B. No  Yes
C. Yes No
D. Yes Yes
E. We’re lost...
Polling Questions (2/2)

❖ For the following scenarios, what will the outcome be for a child process that executes `exit(0)`: 

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Outcome for child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent is still executing:</td>
<td>Alive  Reaped     Zombie</td>
</tr>
<tr>
<td>Parent has called <code>wait()</code>:</td>
<td>Alive  Reaped     Zombie</td>
</tr>
<tr>
<td>Parent has terminated:</td>
<td>Alive  Reaped     Zombie</td>
</tr>
</tbody>
</table>
Processes – Context
Processes Demos

❖ How many processes are running on my computer right now?

❖ In Linux, the `ps` utility gives a snapshot of currently-running processes and `pstree` formats these as a tree
  - Can run `man ps` and `man pstree` for more info
  - Let’s see a simple `pstree`
  - Let’s check `attu` for some 351 zombie processes
The Hardware/Software Interface

- **Topic Group 3: Scale & Coherence**
  - Caches, Memory Allocation, Processes, Virtual Memory

- How do we maintain logical consistency in the face of more data and more processes?
  - How do we support control flow both within many processes and things external to the computer?
  - How do we support data access, including dynamic requests, across multiple processes?
The Operating System

❖ “The OS is everything you don’t need to write in order to run your application”
❖ This depiction invites you to think of the OS as a library
  ▪ In some ways, it is:
    • All operations on I/O devices require OS calls (syscalls – traps)
  ▪ In other ways, it isn't:
    • You use the CPU/memory without OS calls
    • It intervenes without having been explicitly called
Operating System Structure

❖ The OS sits between application programs (P for processes) and the hardware (D for devices)
  ▪ It mediates access (sharing and protection)
    • Programs request services via traps or exceptions; devices request attention via interrupts
  ▪ It abstracts away hardware into logical resources and well-defined interfaces to those resources (ease of use)
    • e.g., processes (CPU, memory), files (disk), programs (sequences of instructions), sockets (network)
OS Relevance in 351

❖ From programmer’s perspective, the application benefits include:
   - **Programming simplicity**
     - Can deal with high-level abstractions instead of low-level hardware details
     - Abstractions are *reusable* across many programs
   - **Portability** (across machine configurations or architectures)
     - Device independence: 3com card or Intel card?

❖ Want to learn more?
   - CSE 333 will cover the application interface with the OS via system calls
   - CSE 451 will have you implementing the complex details of an operating system
Group Work Time

❖ During this time, you are encouraged to work on the following:

1) If desired, continue your discussion
2) Work on the homework problems
3) Work on the lab (if applicable)

❖ Resources:

- You can revisit the lesson material
- Work together in groups and help each other out
- Course staff will circle around to provide support