CS 31: Intro to Systems C Programming
L20: Processes and Virtual Memory

Vasanta Chaganti & Kevin Webb
Swarthmore College
Nov 21, 2023
Reading Quiz
OS Big Picture Goals

• OS is a layer of code between user programs and hardware.

• Goal: Make life easier for users and programmers.

• How can the OS do that?
Key OS Responsibilities

1. Simplifying abstractions for programs
2. Resource allocation and/or sharing
3. Hardware gatekeeping and protection
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)
Managing Processes

• Processes created by calling `fork()`
  – “Spawning” a new process

• “Parent” process spawns “Child” process
  – Brutal relationship involving “zombies”, “killing” and “reaping”. (I’m not making this up!)

• Processes interact with one another by sending signals.
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”
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?

```c
fork();
printf("hello");
if (fork()) {
    printf("hello");
}
fork();
printf("hello");
```
How many hello’s will be printed?

```c
fork();
printf(“hello”);
if (fork()) {
    printf(“hello”);
}
fork();
printf(“hello”);
```

A. 6  
B. 8  
C. 12 
D. 16 
E. 18
fork();
printf("hello");
if (fork()) {
    printf("hello");
}
fork();
fork();
printf("hello");
How many hello’s will be printed?

```c
fork();
printf(“hello”);
if (fork()) {
    printf(“hello”);
}
fork();
printf(“hello”);
```
fork();
printf("hello");
if (fork()) {
    printf("hello");
}
fork();
printf("hello");
How many hello’s will be printed?

```c
fork();
printf("hello");
if (fork()) {
    printf("hello");
}
fork();
printf("hello");
```
fork();
printf("hello");
if (fork()) {
    printf("hello");
}
fork();
printf("hello");
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, execlp, 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.

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

The parent and child each do something different next.
Common `fork()` usage: Shell

1. `fork()` child process.
2. parent: \texttt{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

Diagram:
- Shell
  - `fork()`
  - `Shell (p)`
    - `wait()`
  - `Shell new prog`
    - `exec()`
      - Runs to completion
        - Child terminates
Common `fork()` usage: Shell

3. child program terminates, cycle repeats

![Diagram showing the lifecycle of a child program initiated by `fork()` and terminated by `wait()` or `exec()` in the context of Shell programming.](image-url)
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
(Bet you didn’t expect to see THAT title on a slide when you signed up for CS 31?)

• `wait()`: parents reap their dead child processes
  – Given info about why child terminated, exit status, etc.

• Two variants:
  – `wait()`: wait for and reap next child to exit
  – `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

`fork()` child process.

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

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

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

B. The OS should leave them in the process table

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...
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, …)
Signal Handlers

• By default, processes react to signals according to the signal type:
  – SIGKILL, SIGSEGV, (others): process terminates
  – SIGCHLD, SIGUSR1: process ignores signal

• You can define “signal handler” functions that execute upon receiving a signal.
  – Drop what program was doing, execute handler, go back to what it was doing.
  – Example: got a SIGCHLD? Enter handler, call \texttt{wait()}.
  – Example: got a SIGUSR1? Reopen log files.

• Some signals (e.g., SIGKILL) cannot be handled.
Process Management: Summary

• A process is the unit of execution.
• Processes are represented as Process Control Blocks in the OS
  – PCBs contain process state, scheduling and memory management information, etc
• A process is either New, Ready, Waiting, Running, or Terminated.
• On a uniprocessor, there is at most one running process at a time.
• The program currently executing on the CPU is changed by performing a context switch
• Processes communicate either with message passing or shared memory
Memory

• Abstraction goal: make every process think it has the same memory layout.
  – MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.

• Reality: there’s only so much memory to go around, and no two processes should use the same (physical) memory addresses.

OS (with help from hardware) will keep track of who’s using each memory region.
Memory Terminology

Virtual (logical) Memory: The abstract view of memory given to processes. Each process gets an independent view of the memory.

Physical Memory: The contents of the hardware (RAM) memory. Managed by OS. Only ONE of these for the entire machine!

Virtual address space (VAS): fixed size.

Address Space: Range of addresses for a region of memory.

The set of available storage locations.

OS

Process 1

Process 3

Process 2

Process 3

Process 1

0x0

0xFFFF_FFFFFFFF

0x0

0x...

(Determined by amount of installed RAM.)
Memory Terminology

Note: It is common for VAS to appear larger than physical memory.
32-bit (IA32): Can address up to 4 GB, might have less installed
64-bit (X86-64): Our lab machines have 48-bit VAS (256 TB), 39-bit PAS (512 GB)

Address Space: Range of addresses for a region of memory.
The set of available storage locations.

Virtual address space (VAS): fixed size.

0x0

0xFFFFF000

0x...

(Determined by amount of installed RAM.)
Cohabitating Physical Memory

• If process is given CPU, must also be in memory.

• Problem
  – Context-switching time (CST): 10 µsec
  – Loading from disk: 10 MB/s
  – To load 1 MB process: 100 msec = 10,000 x CST
  – Too much overhead! Breaks illusion of simultaneity

• Solution: keep multiple processes in memory
  – Context switch only between processes in memory
Cohabitating Physical Memory

• If process is to be given CPU, it must also be in memory.
• Problem
  – Context-switching time (CST): 10 µsec
  – Loading from disk: 10 MB/s
  – To load 1 MB process: 100 msec = 10,000 x CST
  – Too much overhead! Breaks illusion of simultaneity
• Solution: keep multiple processes in memory
  – Context switch only between processes already in memory
Memory Issues and Topics

• Where should process memories be placed?
  – Topic: “Classic” memory management
• How does the compiler model memory?
  – Topic: Logical memory model
• How to deal with limited physical memory?
  – Topics: Virtual memory, paging

Plan: Start with the basics (very old, classic problems) to motivate why we need the complex machinery of virtual memory and paging.
Problem: Placement

• Where should process memories be placed?
  – Topic: “Classic” memory management

• How does the compiler model memory?
  – Topic: Logical memory model

• How to deal with limited physical memory?
  – Topics: Virtual memory, paging
Memory Management

- Physical memory starts as one big empty space.
Memory Management

- Physical memory starts as one big empty space.
- Processes need to be in memory to execute.
Memory Management

• Physical memory starts as one big empty space.
• When creating process, allocate memory
  – Find space that can contain process
  – Allocate region within that gap
  – Typically, leaves a (smaller) free space
Memory Management

• Physical memory starts as one big empty space.
• When creating process, allocate memory
  – Find space that can contain process
  – Allocate region within that gap
  – Typically, leaves a (smaller) free space
• When process exits, free its memory
  – Creates a gap in the physical address space.
  – If next to another gap, coalesce.
Fragmentation

- Eventually, memory becomes fragmented
  - After repeated allocations/de-allocations

- Internal fragmentation
  - Unused space within process
  - Cannot be allocated to others
  - Can come in handy for growth

- External fragmentation
  - Unused space outside any process (gaps)
  - Can be allocated (too small/not useful?)
Which form of fragmentation is easiest for the OS to reduce/eliminate? Why?

A. Internal fragmentation

B. External fragmentation

C. Neither
Placing Memory

• When searching for space, what if there are multiple options?
• Algorithms
  – *First (or next) fit*
  – Best fit
  – Worst fit
Placing Memory

• When searching for space, what if there are multiple options?
• Algorithms
  – First (or next) fit
  – Best fit
  – Worst fit
Placing Memory

• When searching for space, what if there are multiple options?
• Algorithms
  – First (or next) fit
  – Best fit
  – Worst fit
Which memory allocation algorithm would you choose? Why?

A. first-fit

B. worst-fit

C. best-fit

Is leaving small fragments a good thing or a bad thing?
Placing Memory

• When searching for space, what if there are multiple options?
• Algorithms
  – *First (or next) fit*: fast
  – Best fit
  – Worst fit
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - Best fit: leaves small fragments
  - Worst fit
Placing Memory

• When searching for space, what if there are multiple options?
• Algorithms
  – First (or next) fit
  – Best fit
  – *Worst fit: leaves large fragments*
What if it doesn’t fit?

• There may still be significant unused space
  – External fragments
  – Internal fragments
• Approaches
What if it doesn’t fit?

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - *Compaction*
What if it doesn’t fit?

• There may still be significant unused space
  – External fragments
  – Internal fragments

• Approaches
  – Compaction
    – *Break process memory into pieces*
      • Easier to fit.
      • More state to keep track of.
Problem Summary: Placement

• When placing a process, it may be difficult to find a large enough free region in physical memory.

• Fragmentation makes this harder over time (free pieces get smaller, spread out).

• General solution: don’t require all of a process’s memory to be in one piece!
Problem Summary: Placement

• General solution: don’t require all of a process’s memory to be in one piece!
Problem Summary: Placement

• General solution: don’t require all of a process’s memory to be in one piece!
Problem Summary: Placement

• General solution: don’t require all of a process’s memory to be in one piece!
**Problem Summary: Placement**

- General solution: don’t require all of a process’s memory to be in one piece!
Problem: Addressing

• Where should process memories be placed?
  – Topic: “Classic” memory management

• How does the compiler model memory?
  – Topic: Logical memory model

• How to deal with limited physical memory?
  – Topics: Virtual memory, paging
(More) Problems with Memory Cohabitation

- **Addressing:**
  - Compiler generates memory references
  - Unknown where process will be located

- **Protection:**
  - Modifying another process’s memory

- **Sharing Space:**
  - The more processes there are, the less memory each individually can have
Compiler’s View of Memory

• Compiler generates memory addresses
  – Needs empty region for text, data, stack
  – Ideally, very large to allow data and stack to grow

• Without abstractions compiler would need to know...
  – Physical memory size
  – Where to place data (e.g., stack at high end)
    • Must avoid allocated regions in memory
Address Spaces

• Address space
  – Set of addresses for memory

• Usually linear: 0 to $N-1$ (size $N$)

• Physical Address Space
  – 0 to $N-1$, $N = \text{size}$
  – Kernel occupies lowest addresses
Virtual vs. Physical Addressing

- **Virtual addresses**
  - Assumes separate memory starting at 0
  - Compiler generated
  - Independent of location in physical memory

- **OS**: Map virtual to physical
When should we perform the mapping from virtual to physical address? Why?

A. When the process is initially loaded: convert all the addresses to physical addresses

B. When the process is running: map the addresses as they’re used.

C. Perform the mapping at some other time. When?
Hardware for Virtual Addressing

- Base register filled with start address
- To translate address, add base
- Achieves “relocation”: process’s physical memory location could be moved.
- To move process: change base

Note: This is a simpler model than what we do in real systems today. We’re still working toward the real thing.
Hardware for Virtual Addressing

• Base register filled with start address
• To translate address, add base
• Achieves “relocation”: process’s physical memory location could be moved.
• To move process: change base
• Protection?
Protection

• Bound register works with base register
• Is address < bound
  – Yes: add to base
  – No: invalid address, invoke OS
• Achieves protection

When would we need to update these base & bound registers?
Memory Registers Part of Context

• On Every Context Switch
  – Load base/bound registers for selected process
  – Only kernel does loading of these registers
  – Kernel must be protected from all processes

• Benefit
  – Allows each process to be separately located
  – Protects each process from all others
Problem Summary: Addressing

• Compiler has no idea where, in physical memory, the process’s data will be.
• Compiler generates instructions to access VAS.

• General solution: OS must translate process’s VAS accesses to the corresponding physical memory location.
Problem Summary: Addressing

• General solution: OS must translate process’s VAS accesses to the corresponding physical memory location.

When the process tries to access a virtual address, the OS translates it to the corresponding physical address.

### ldr x0, [address]

Physical Memory

<table>
<thead>
<tr>
<th></th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Process 1</td>
</tr>
<tr>
<td></td>
<td>Process 2</td>
</tr>
<tr>
<td></td>
<td>Process 3</td>
</tr>
<tr>
<td></td>
<td>Process 1</td>
</tr>
<tr>
<td></td>
<td>Process 3</td>
</tr>
<tr>
<td></td>
<td>Process 2</td>
</tr>
</tbody>
</table>
Problem Summary: Addressing

• General solution: OS must translate process’s VAS accesses to the corresponding physical memory location.

When the process tries to access a virtual address, the OS translates it to the corresponding physical address.

```
ldr x0, [address]
```
Let’s combine these ideas:

1. Allow process memory to be divided up into multiple pieces.

2. Keep state in OS (+ hardware/registers) to map from virtual addresses to physical addresses.

Result: Keep a table to store the mapping of each region.
Problem Summary: Addressing

• General solution: OS must translate process’s VAS accesses to the corresponding physical memory location.

When the process tries to access a virtual address, the OS translates it to the corresponding physical address.

OS must keep a table, for each process, to map VAS to PAS. One entry per divided region.

movl (address 0x74), %eax
Two (Real) Approaches

• Segmented address space/memory
• Partition address space and memory into segments
• Segments are generally different sizes

• Paged address space/memory
• Partition address space and memory into pages
• All pages are the same size
Two (Real) Approaches

- Segmented address space/memory
- Partition address space and memory into segments
  - Segments are generally different sizes

- Paged address space/memory
- Partition address space and memory into pages
  - All pages are the same size

In this class, we’re only going to look at paging, the most common method today.