Virtual Memory I
CSE 351 Spring 2019

Instructor:
Ruth Anderson

Teaching Assistants:
Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh

http://rebrn.com/re/bad-chrome-1162082/
Administrivia

- Homework 4, due Wed (5/22) (Structs, Caches)
- Lab 4, due Fri (5/24)
Processes

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

- A terminated process still consumes system resources
  - Various tables maintained by OS
  - Called a “zombie” (a living corpse, half alive and half dead)
- Reaping is performed by parent on terminated child
  - Parent is given exit status information and kernel then deletes zombie child process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by \texttt{init} process (\texttt{pid} == 1)
    - \textbf{Note:} on recent Linux systems, \texttt{init} has been renamed \texttt{systemd}
  - In long-running processes (e.g. shells, servers) we need \textit{explicit} reaping
wait: Synchronizing with Children

- **int wait (int *child_status)**
  - Suspends current process (i.e. the parent) until one of its children terminates
  - Return value is the PID of the child process that terminated
    - *On successful return, the child process is reaped*
  - If child_status != NULL, then the *child_status value indicates why the child process terminated*
    - Special macros for interpreting this status – see *man wait(2)*

- **Note:** If parent process has multiple children, wait will return when *any* of the children terminates
  - `waitpid` can be used to wait on a specific child process
wait: Synchronizing with Children

```c
void fork_wait() {
    int child_status;

    if (fork() == 0) { // child
        printf("HC: hello from child\n");
        exit(0);
    } else { // parent
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

Feasible output: HCHPCTBye

Infeasible output: HPCTByeHC
Example: Zombie

```c
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1); /* Infinite loop */
    }
}
```

- `ps` shows child process as "defunct"
- Killing parent allows child to be reaped by `init`
Example: Non-terminating Child

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

```c
void fork8() {
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1); /* Infinite loop */
    } else { /* child persists */
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```bash
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
    PID  TTY         TIME CMD
  6585  ttyp9    00:00:00  tcsh
  6676  ttyp9    00:00:06 forks
  6677  ttyp9    00:00:00  ps
linux> kill 6676
linux> ps
    PID  TTY         TIME CMD
  6585  ttyp9    00:00:00  tcsh
  6678  ttyp9    00:00:00  ps
```
Process Management Summary

- **fork** makes two copies of the same process (parent & child)
  - Returns different values to the two processes
- **exec*** replaces current process from file (new program)
  - Two-process program:
    - First **fork()**
    - **if** (pid == 0) { /* child code */ } **else** { /* parent code */ }
  - Two different programs:
    - First **fork()**
    - **if** (pid == 0) { execv(...) } **else** { /* parent code */ }

- **wait** or **waitpid** used to synchronize parent/child execution and to reap child process
Roadmap

C:
```c
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:
```
get_mpg:
    pushq  %rbp
    movq  %rsp, %rbp
    ...
    popq  %rbp
    ret
```

Machine code:
```
0111010000011000
100011010000010000000010
1000100111000010
1100000111111101000011111
```

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Virtual Memory (VM*)

- Overview and motivation
- VM as a tool for caching
- Address translation
- VM as a tool for memory management
- VM as a tool for memory protection

**Warning:** Virtual memory is pretty complex, but crucial for understanding how processes work and for debugging performance.

*Not to be confused with “Virtual Machine” which is a whole other thing.*
Memory as we know it so far... is *virtual*!

- Programs refer to virtual memory addresses
  - `movq (%rdi), %rax`
  - Conceptually memory is just a very large array of bytes
  - System provides private address space to each process

- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space

- But...
  - We *probably* don’t have $2^w$ bytes of physical memory
  - We *certainly* don’t have $2^w$ bytes of physical memory *for every process*
  - Processes should not interfere with one another
    - Except in certain cases where they want to share code or data
Problem 1: How Does Everything Fit?

64-bit virtual addresses can address several exabytes
(18,446,744,073,709,551,616 bytes)

Physical main memory offers a few gigabytes
(e.g. 8,589,934,592 bytes)

(Not to scale; physical memory would be smaller than the period at the end of this sentence compared to the virtual address space.)

1 virtual address space per process, with many processes...
Problem 2: Memory Management

We have multiple processes:
- Process 1
- Process 2
- Process 3
- ...
- Process n

Each process has...
- stack
- heap
- .text
- .data
- ...

What goes where?

Physical main memory
Problem 3: How To Protect

Physical main memory

Process i

Process j

Problem 4: How To Share?

Physical main memory

Process i

Process j
How can we solve these problems?

- “Any problem in computer science can be solved by adding another level of **indirection.**” – David Wheeler, inventor of the subroutine

- Without Indirection

- With Indirection

*What if I want to move Thing?*
Indirection

- *Indirection*: The ability to reference something using a name, reference, or container instead of the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.
  - Adds some work (now have to look up 2 things instead of 1)
  - But don’t have to track all uses of name/address (single source!)

- **Examples:**
  - **Phone system**: cell phone number portability
  - **Domain Name Service (DNS)**: translation from name to IP address
  - **Call centers**: route calls to available operators, etc.
  - **Dynamic Host Configuration Protocol (DHCP)**: local network address assignment
Indirection in Virtual Memory

- Each process gets its own private virtual address space
- Solves the previous problems!
Address Spaces

- **Virtual address space**: Set of \( N = 2^n \) virtual addr
  - \( \{0, 1, 2, 3, \ldots, N-1\} \)

- **Physical address space**: Set of \( M = 2^m \) physical addr
  - \( \{0, 1, 2, 3, \ldots, M-1\} \)

- Every byte in main memory has:
  - one physical address (PA)
  - zero, one, or more virtual addresses (VAs)
Mapping

- A virtual address (VA) can be mapped to either **physical memory** or disk
  - Unused VAs may not have a mapping
  - VAs from _different_ processes may map to the same location in memory/disk
A System Using Physical Addressing

- Used in “simple” systems with (usually) just one process:
  - Embedded microcontrollers in devices like cars, elevators, and digital picture frames
A System Using Virtual Addressing

✿ Physical addresses are *completely invisible to programs*
  - Used in all modern desktops, laptops, servers, smartphones...
  - One of the great ideas in computer science
Why Virtual Memory (VM)?

- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - Some non-cached parts stored on disk
    - Some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - Transfer data back and forth as needed

- **Simplifies memory management for programmers**
  - Each process “gets” the same full, private linear address space

- **Isolates address spaces (protection)**
  - One process can’t interfere with another’s memory
    - They operate in *different address spaces*
  - User process cannot access privileged information
    - Different sections of address spaces have different permissions
VM and the Memory Hierarchy

- Think of virtual memory as an array of \( N = 2^n \) contiguous bytes.
- *Pages* of virtual memory are usually stored in physical memory, but sometimes spill to disk:
  - Pages are another unit of aligned memory (size is \( P = 2^p \) bytes).
  - Each virtual page can be stored in *any* physical page (no fragmentation!).

![VM and Memory Hierarchy Diagram]

**Virtual memory**:
- VP 0
- VP 1
- VP 2^{n-p}-1

**Physical memory**:
- PP 0
- PP 1
- PP 2^{m-p-1}

**Disk**:
- "Swap Space"
or: Virtual Memory as DRAM Cache for Disk

- Think of virtual memory as an array of $N = 2^n$ contiguous bytes stored on a disk.
- Then physical main memory is used as a cache for the virtual memory array.
  - These “cache blocks” are called *pages* (size is $P = 2^p$ bytes).

![Diagram showing virtual and physical memory relationship]
Memory Hierarchy: Core 2 Duo

SRAM
Static Random Access Memory

L1 I-cache
32 KB
Throughput: 16 B/cycle
Latency: 3 cycles

Miss Penalty (latency) 33x

L1 D-cache
8 B/cycle
14 cycles

L2 unified cache
~4 MB
2 B/cycle
100 cycles

Main Memory
~8 GB
1 B/30 cycles
millions

Miss Penalty (latency) 10,000x

DRAM
Dynamic Random Access Memory

Disk
~500 GB

Not drawn to scale
Virtual Memory Design Consequences

- Large page size: typically 4-8 KiB or 2-4 MiB
  - Can be up to 1 GiB (for “Big Data” apps on big computers)
  - Compared with 64-byte cache blocks

- Fully associative (physical memory is single set)
  - Any virtual page can be placed in any physical page
  - Requires a “large” mapping function – different from CPU caches

- Highly sophisticated, expensive replacement algorithms in OS
  - Too complicated and open-ended to be implemented in hardware

- Write-back rather than write-through (track dirty pages)
  - Really don’t want to write to disk every time we modify something in memory
  - Some things may never end up on disk (e.g. stack for short-lived process)
Why does VM work on RAM/disk?

- Avoids disk accesses because of *locality*
  - Same reason that L1 / L2 / L3 caches work

- The set of virtual pages that a program is “actively” accessing at any point in time is called its *working set*
  - If *(working set of one process ≤ physical memory)*:
    - Good performance for one process (after compulsory misses)
  - If *(working sets of all processes > physical memory)*:
    - *Thrashing*: Performance meltdown where pages are swapped between memory and disk continuously (CPU always waiting or paging)
    - This is why your computer can feel faster when you add RAM
Virtual Memory (VM)

- Overview and motivation
- VM as a tool for caching
- Address translation
- VM as a tool for memory management
- VM as a tool for memory protection
Address Translation

How do we perform the virtual → physical address translation?

CPU Chip

Virtual address (VA) 0x4100

MMU

Physical address (PA) 0x4

Main memory

0:
1:
2:
3:
4:
5:
6:
7:
8:

... 

M-1:

Data (int/float)
Address Translation: Page Tables

- CPU-generated address can be split into:
  - Request is Virtual Address (VA), want Physical Address (PA)
  - Note that Physical Offset = Virtual Offset (page-aligned)

- Use lookup table that we call the page table (PT)
  - Replace Virtual Page Number (VPN) for Physical Page Number (PPN) to generate Physical Address
  - Index PT using VPN: page table entry (PTE) stores the PPN plus management bits (e.g. Valid, Dirty, access rights)
  - Has an entry for every virtual page – why?
Page Table Diagram

- Page tables stored in physical memory
  - Too big to fit elsewhere – managed by MMU & OS
- How many page tables in the system?
  - One per process
Page Table Address Translation

In most cases, the MMU can perform this translation without software assistance.
Page Hit

- **Page hit**: VM reference is in physical memory

**Example**: Page size = 4 KiB = $2^{12}$ B $\iff$ p=12 bits = 3 hex digits

**Virtual Addr**: 0x00740b

**Physical Addr**: 0x240b

1. **VPN**: 7
2. **PPN**: 2
Summary

- Virtual memory provides:
  - Ability to use limited memory (RAM) across multiple processes
  - Illusion of contiguous virtual address space for each process
  - Protection and sharing amongst processes

- Indirection via address mapping by page tables
  - Part of memory management unit and stored in memory
  - Use virtual page number as index into lookup table that holds physical page number, disk address, or NULL (unallocated page)
  - On page fault, throw exception and move page from swap space (disk) to main memory
Detailed examples:
- `wait()` example
- `waitpid()` example
**wait() Example**

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```
waitpid(): Waiting for a Specific Process

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```