Processes II, Virtual Memory I
CSE 351 Summer 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

Well, I'm having trouble opening new tabs. And the others are having problems too. The IDE, Skype, everyone really...

Hmm, well I guess it would be nice to boost the IDE a bit...

OK, here's an extra 4 gigs. Make sure you share it around, there aren't any more slots left!

Yeah, OK, that'll do.

Nom nom nom

So, what did he say? Will he give us some more RAM?

Burp! Um...

He told you to get lost

Yeah, what a...

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

- Questions doc: [https://tinyurl.com/CSE351-8-7](https://tinyurl.com/CSE351-8-7)

- hw18 due Monday (8/10) – 10:30am
- hw19 is optional
  - Can complete it at any point before the quarter ends
  - Practice with virtual memory concepts

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

*Note: hw18 is not for credit.*
Fork Example

```c
void fork1() {
    int x = 1;
    pid_t fork_ret = fork();
    if (fork_ret == 0) // child
        printf("Child has x = %d\n", ++x);
    else // parent
        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);
}
```
Polling Question [Proc II]

Are the following sequences of outputs possible?

Vote at [http://pollev.com/pbjones](http://pollev.com/pbjones)

```
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
- L2
- Bye

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

A. No No
B. No Yes
C. Yes No
D. Yes Yes
E. We’re lost...
Fork-Exec

- fork-exec model:
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process’ code and address space with the code for a different program
    - Whole family of `exec` calls – see `exec(3)` and `execve(2)`

```c
// Example arguments: path="/usr/bin/ls",
void fork_exec(char *path, char *argv[]) {
    pid_t fork_ret = fork();
    if (fork_ret != 0) { // parent
        printf("Parent: created a child %d\n", fork_ret);
    } else { // child
        printf("Child: about to exec a new program\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```

Note: the return values of `fork` and `exec*` should be checked for errors.
### Exec-ing a new program

Very high-level diagram of what happens when you run the command "ls" in a Linux shell:

- This is the loading part of CALL!

```
$ ls
```

```bash
while (user has not quit) {
    get input ();
    if (child) execChild();
}
```

```
$ ls
```

---

**Diagram:**

- **Parent**
  - Stack
  - Heap
  - Data
  - Code: /usr/bin/bash

- **Child (only one child executes)**
  - Stack
  - Heap
  - Data
  - Code: /usr/bin/bash
  - Exec*()
**execve Example**

Execute "\texttt{/usr/bin/ls -l lab4}" in child process using current environment:

<table>
<thead>
<tr>
<th>myargv[0]</th>
<th>&quot;/usr/bin/ls&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>myargv[1]</td>
<td>&quot;-l&quot;</td>
</tr>
<tr>
<td>myargv[2]</td>
<td>&quot;lab4&quot;</td>
</tr>
</tbody>
</table>

\textit{myargv} arguments:

<table>
<thead>
<tr>
<th>argc argv[0] = NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv[1] = &quot;-l&quot;</td>
</tr>
<tr>
<td>argv[2] = &quot;lab4&quot;</td>
</tr>
</tbody>
</table>

\textit{envp} environment:

<table>
<thead>
<tr>
<th>envp[n] = NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>envp[n-1]</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>envp[0]</td>
</tr>
</tbody>
</table>

\texttt{envp[0]} = "PWD=/homes/iws/rea"

\texttt{envp[n]} = "USER=rea"

\texttt{argv[0]} = "/usr/bin/ls"

\texttt{argv[1]} = "-l"

\texttt{argv[2]} = "lab4"

\texttt{argc} = 3

\texttt{execve("/usr/bin/ls", myargv, environ")} < 0

\texttt{printf("%s: Command not found.\n", myargv[0]);}

\texttt{exit(1);}

\texttt{if ((pid = fork()) == 0) { /* Child runs program */)
if (execve(myargv[0], myargv, environ) < 0) {
    printf("%s: Command not found.\n", myargv[0]);
    exit(1);
}
}

Run the \texttt{printenv} command in a Linux shell to see your own environment variables.
Stack Structure on a New Program Start

Null-terminated environment variable strings

Null-terminated command-line arg strings

argv[argc] = NULL
argv[argc-1]
...
argv[0]

envp[n] == NULL
envp[n-1]
...
envp[0]

argv (in %rsi)
argc (in %rdi)

Stack frame for libc_start_main

Future stack frame for main

environ (global var)
envp (in %rdx)

This is extra (non-testable) material
exit: Ending a process

- **void exit(int status)**
  - Explicitly exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit

- The *return statement from main() also ends a process in C*
  - The return value is the status code

```c
main() {  
    exit(0);  
    return 0;  
}
```
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 *init* process (pid of 1)
    - **Note:** on recent Linux systems, *init* has been renamed *systemd*
  - In long-running processes (*e.g.* shells, servers) we need *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) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

---

**Feasible output:**

- HC
- HP
- CT
- Bye

**Infeasible output:**

- HP
- CT
- Bye
- HC
Example: Zombie

*ps* shows a list of current processes

```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 */
    }
}
```

`forks.c`

*ps* shows child process as “defunct” *Zombie*

Killing parent allows child to be reaped by *init*
Example:
Non-terminating Child

```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);
    }
}
```

`forks.c`

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely
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
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
class Car {
    int miles;
    int gals;

    public Car() {
        // constructor
    }

    public void setMiles(int miles) {
        this.miles = miles;
    }

    public void setGals(int gals) {
        this.gals = gals;
    }

    public float getMPG() {
        // calculate MPG
        return mpg;
    }
}
```

Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000001111
```

Computer system:

- Windows 10
- OS X Yosemite

OS:

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)

1 virtual address space per process, with many processes...

(Not to scale; physical memory would be smaller than the period at the end of this sentence compared to the virtual address space.)
Problem 2: Memory Management

We have multiple processes:

- Process 1
- Process 2
- Process 3
- ... 
- Process n

Each process has...

Physical main memory

What goes where?
Problem 3: How To Protect

Problem 4: How To Share?
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
  - $\{0, 1, 2, 3, ..., N-1\}$

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

- 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 same location in memory/disk

![Diagram of mapping virtual addresses to physical memory and disk]

- Process 1’s Virtual Address Space
- Process 2’s Virtual Address Space
- Physical Memory
- Disk
- “Swap Space”
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!)

![Diagram of virtual memory and physical memory]

Virtual memory

- VP 0
- VP 1
- VP \(2^n-1\)
- VP \(2^{n-p-1}\)

Physical memory

- Empty
- Empty
- Empty
- 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)
**Memory Hierarchy: Core 2 Duo**

- **SRAM**
  - Static Random Access Memory
  - L1 I-cache: 32 KB (16 B/cycle, 3 cycles)
  - L1 D-cache: ~4 MB (8 B/cycle, 14 cycles)
  - L2 unified cache: 2 B/cycle, 100 cycles

- **DRAM**
  - Dynamic Random Access Memory
  - Main Memory: ~8 GB (1 B/30 cycles, millions)

- **Disk**
  - ~500 GB

**Throughput:**
- CPU: 16 B/cycle
- Main Memory: 8 B/cycle
- DRAM: 2 B/cycle
- Disk: 1 B/30 cycles

**Latency:**
- CPU: 3 cycles
- Main Memory: 14 cycles
- DRAM: 100 cycles
- Disk: millions

**Miss Penalty (latency):**
- L1 I-cache: 33x
- L1 D-cache: 10,000x

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**
  - 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**
  - *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 \textit{locality}
  - Same reason that L1 / L2 / L3 caches work

- The set of virtual pages that a program is \textit{“actively”} accessing at any point in time is called its \textit{working set}
  - If (working set of one process \leq physical memory):
    - Good performance for one process (after compulsory misses)
      - \(8 \text{ GB} \approx 100-200 \text{ hard working Chrome tabs}\)
  - If (working sets of all processes > physical memory):
    - \textit{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
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
Detailed examples:
- Consecutive forks
- `wait()` example
- `waitpid()` example
Example: Two consecutive `forks`

```c
void fork2() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Feasible output:
- L0
- L1
- Bye
- Bye
- L1
- Bye

Infeasible output:
- L0
- Bye
- L1
- Bye
- L1
- Bye
Example: Three consecutive forks

- Both parent and child can continue forking

```c
void fork3() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```
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

\textbf{pid\_t waitpid(pid\_t pid, int \&status, int options)}

- suspends current process until specific process terminates
- various options (that we won’t talk about)

```c
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);
    }
}
```