CSE 351 Section V – Processes, Virtual Memory

Linux Process-Related System Calls

- `fork()` – Clones the currently running process. Returns `0` to the child and the child's PID to the parent.
- `exec*()` – Family of operations to replace the current process image with a new process image.
- `getpid()` – Returns the process ID of the calling process.
- `exit()` – Causes normal process termination.
- `wait()`, `waitpid()` – Wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

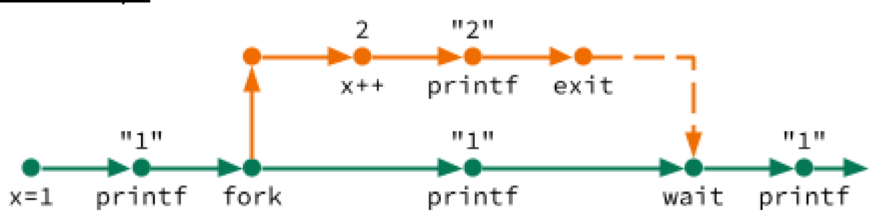
<u>Example</u>: Process Graph

The operating system switches between running processes. Each process runs its instructions in sequential order, but users cannot predict the interleaving of instructions *between* different processes. **Process graphs** can help us analyze possible interleavings:

- <u>Vertices</u> indicate statements/instructions of note from a process and are labeled with their effect.
- <u>Edges</u> impose sequential ordering between vertices (also branching and merging of processes.

```
int x = 1;
printf("%d", x);
if ( !fork() ) {
  x++;
  printf("%d", x);
  exit();
} else {
  printf("%d", x);
  wait();
  printf(“%d”, x);
}
```

<u>Process Graph</u>:



<u>Possible outputs</u>:

- **1211**
- **1121**

<u>Exercise 1</u>: Fork and Concurrency

Consider this code using Linux's `fork`. Draw out a process graph and write all **four** different possible outputs (*i.e.*, order of things printed) for this code.

```
int x = 7;
if ( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

<u>Possible outputs</u>:

-
-
-
-

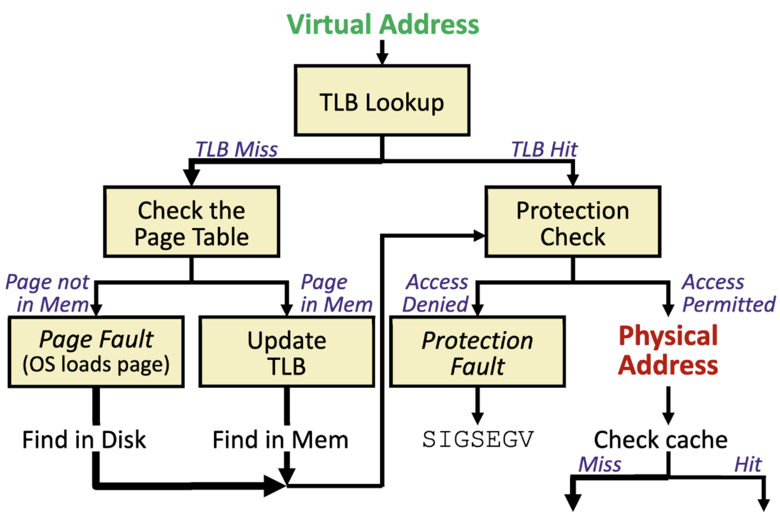
Virtual Memory

Physical and virtual memory are broken up into fixed-size **pages**.

- The most recently used virtual pages will map to a physical page.
- Allocated pages that “spill out” of physical memory end up in swap space on the disk.
- Unused virtual pages (white) don’t have physical space allocated for them!

Benefits of virtual memory (not every system utilizes VM):

- Simplifies memory management for programmers/applications (*i.e.*, consistent view of memory).
- Enforces protection and enables sharing between processes via access rights.
- Bridges memory and disk in the memory hierarchy.



Potential outcomes:

1a) TLB Hit 👍👍

1b) TLB Miss

  2a) Page Table Hit 👍

  2b) Page Fault 😫

3a) Data Fetch ✔️

3b) Segmentation Fault ❌ (protection or otherwise)

<u>Exercise 2</u>: Memory Access Questions

A. Why does it make sense to have one page table per process?

B. What should be done with our TLB when we change processes? Why?

C. What two situations could cause a Page Table Entry to be invalid? What must we do in these situations? (Hint: where could a virtual page be?)

D. On a TLB hit, we can also have a ____________. (Circle all that apply below)

Page Table Hit   Page Fault   Protection Fault   Cache Miss

Example: Virtual Memory Simulator Memory Accesses

Set up the virtual memory simulator (https://courses.cs.washington.edu/courses/cse351/vmsim/)as follows to get started:

- Generate the system with the following parameters: Virtual Address Width: **10** bits, Page Size: **32** bytes, TLB Size: **4** entries, TLB Associativity: **2** way, Physical Memory Size: **128** bytes.
- Allocate the following virtual pages *in order* by clicking: `0x03`, `0x05`, `0x01`.

What will happen if we **write** the byte **`0xAD`** to address **`0x28F`**?

```
Address: 0b 10 100|0 1111
PO   = 0b01111
VPN  = 0b1010|0 = 0x14
TLBI = 0b0 = 0x0
TLBT = 0b1010 = 0xA
```

TLB: **TLB Miss** (cold)

Page Table: **Page Hit** (`0x14` is valid)

Outcome: **Data fetch/write**

System state: Pull PTE 0x14 into TLB, set dirty bit (write), change byte in PM

Exercise 3: TLB Hit

What is the **largest/highest address** we can **read** that will result in a **TLB Hit**?

Exercise 4: Page Fault

What is the **smallest/lowest address** we can **read** that will result in a **Page Fault** (but not a Segfault)?

Exercise 5: Write to `0x036`

What will happen if we **write** the byte `0xEF` to address **`0x036`**?

- TLB Access: TLB Hit or TLB Miss?
- Page Table Access: Page Hit or Page Fault?
- Outcome: Data fetch or Segfault?
- What system state changes need to happen?

Exercise 6: Read from `0x2E0`

What will happen if we **read** from address **`0x2E0`**?

- TLB Access: TLB Hit or TLB Miss?
- Page Table Access: Page Hit or Page Fault?
- Outcome: Data fetch or Segfault?
- What system state changes need to happen?