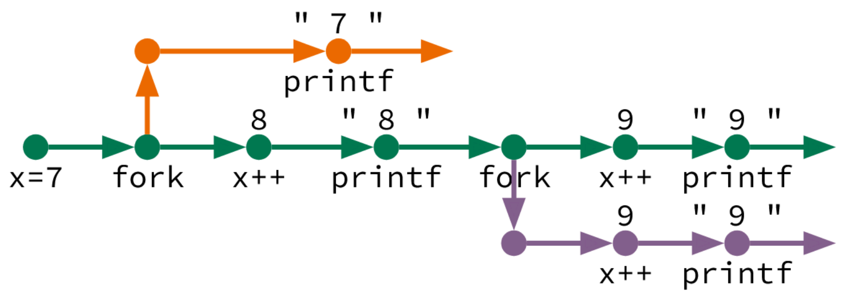
CSE 351 Section V – Processes, Virtual Memory

Exercise 1: 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);
}
```



- 7 8 9 9
- 8 7 9 9
- 8 9 7 9
- 8 9 9 7

From our first fork, we know Child 1 will print **" 7 "**, but since this print statement is not dependent on any other code (besides the initial fork), it could be printed at any time.

We also know the Parent will have to print **" 8 "** before the second fork, meaning that the **" 8 "** is printed before the **" 9 "**s. Since the Parent and Child 2 both print out **" 9 "**, the output remains the same even if the ordering of their prints changes.

Exercise 2: Memory Access Questions

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

Each process has its own virtual space and mappings – different processes might map the same virtual address to a different physical address – so separate page tables are needed.

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

We must invalidate the TLB when we context switch. The PTEs in the TLB will correspond to the mappings of the old process and, thus, are not valid for the new process we are context switching to.

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?)

(1) The page is on disk, not in main memory. (2) The page does not exist/is unallocated.

First, raise a page fault, which will pull the requested page to memory (if it exists). We then restart the VPN to PPN translation process and get a guaranteed page hit.

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

Page Table Hit   Page Fault   **Protection Fault**   **Cache Miss**

If we get a TLB hit, we don't have to check the page table!

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: `0x0`**`3`**, `0x0`**`5`**, `0x0`**`1`**.
- Write the byte `0xAD` to address `0x28F`.

Exercise 3: TLB Hit

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

- Only 1 valid entry in the TLB, so must access that page.
- TLB Index is `0b0`, TLB Tag is `0x06` → VPN `0x14` (already known from access to `0x28F`).
- Page offset should be the maximum = `0b11111`.
- Putting the address together: `0b10100`|`11111` = **0x29F**.

Exercise 4: Page Fault

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

- Need an allocated page on disk: smallest VPN on disk is `0x01`.
- Page offset should be the minimum = `0b00000`.
- Putting the address together: `0b00001`|`00000` = **`0x020`**.

Exercise 5: Write to `0x036`

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

- Address: `0b00001`|`10110` → VPN = `0b0000`|`1` = `0x0C` → TLBI = 0b1 = `0x1`, TLBT = `0b0000` = `0x0`
- TLB Access: **TLB Hit**
- Page Table Access: **Never accessed**
- Outcome: **Segfault** (Protection Fault – no write access!)
- What system state changes need to happen? **No change** – process will react to signal

Exercise 6: Read from `0x2E0`

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

- Address: `0b10111`|`00000` → VPN = `0b1011`|`1` = `0x1E` → TLBI = `0b1` = `0x1`, TLBT = `0b1011` = `0xB`
- TLB Access: **TLB Miss**
- Page Table Access: **Page Fault**
- Outcome: **Segfault** (unallocated page)
- What system state changes need to happen? **No change** – process will react to signal