# Allocating Memory

Where does **`malloc`** get memory?

See **`mmap.c`**

# Picking Virtual Addresses

See **`mmap2.c`** and **`mmap3.c`**

# Freeing Pages

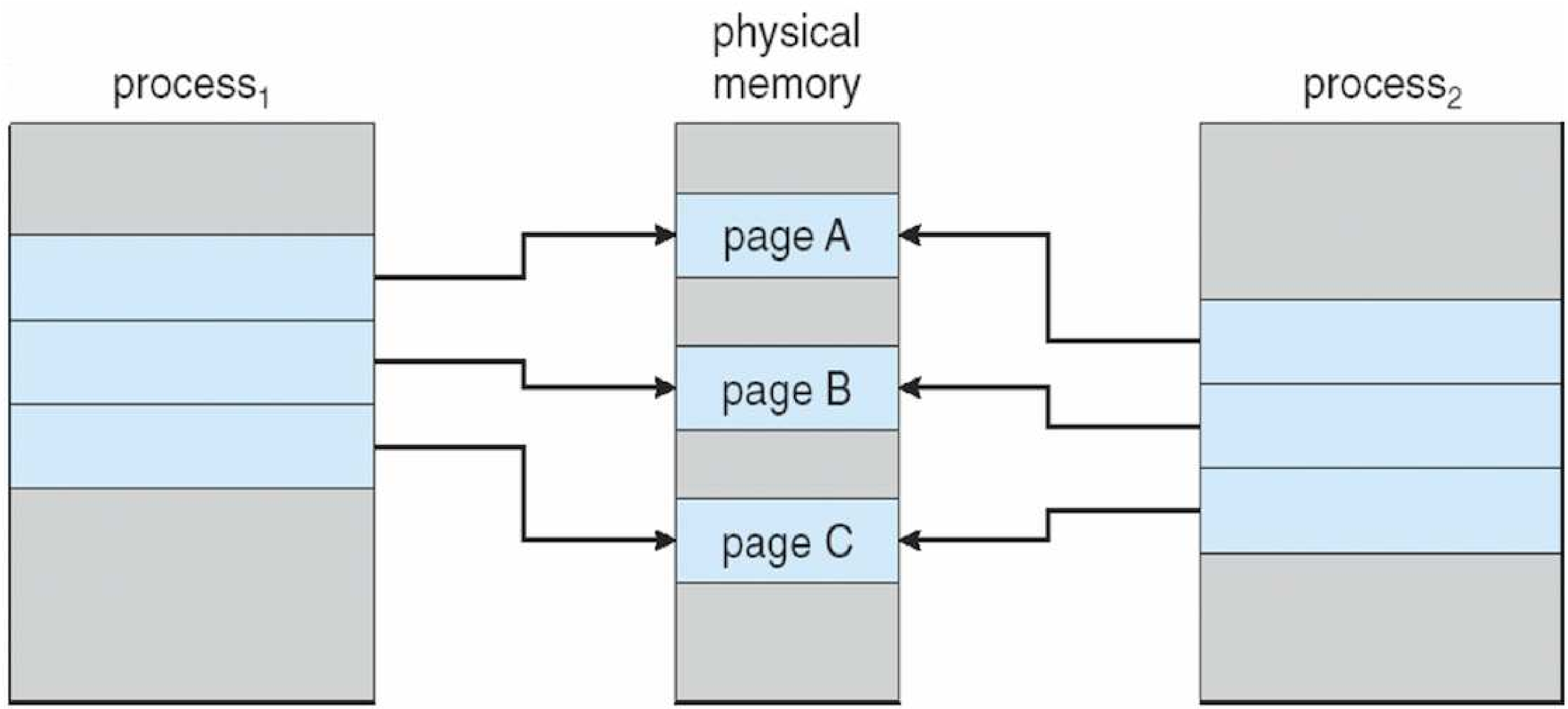
See **`munmap.c`**

# Pages and Processes

See **`mmap+fork.c`** and **`mmap+fork2.c`**

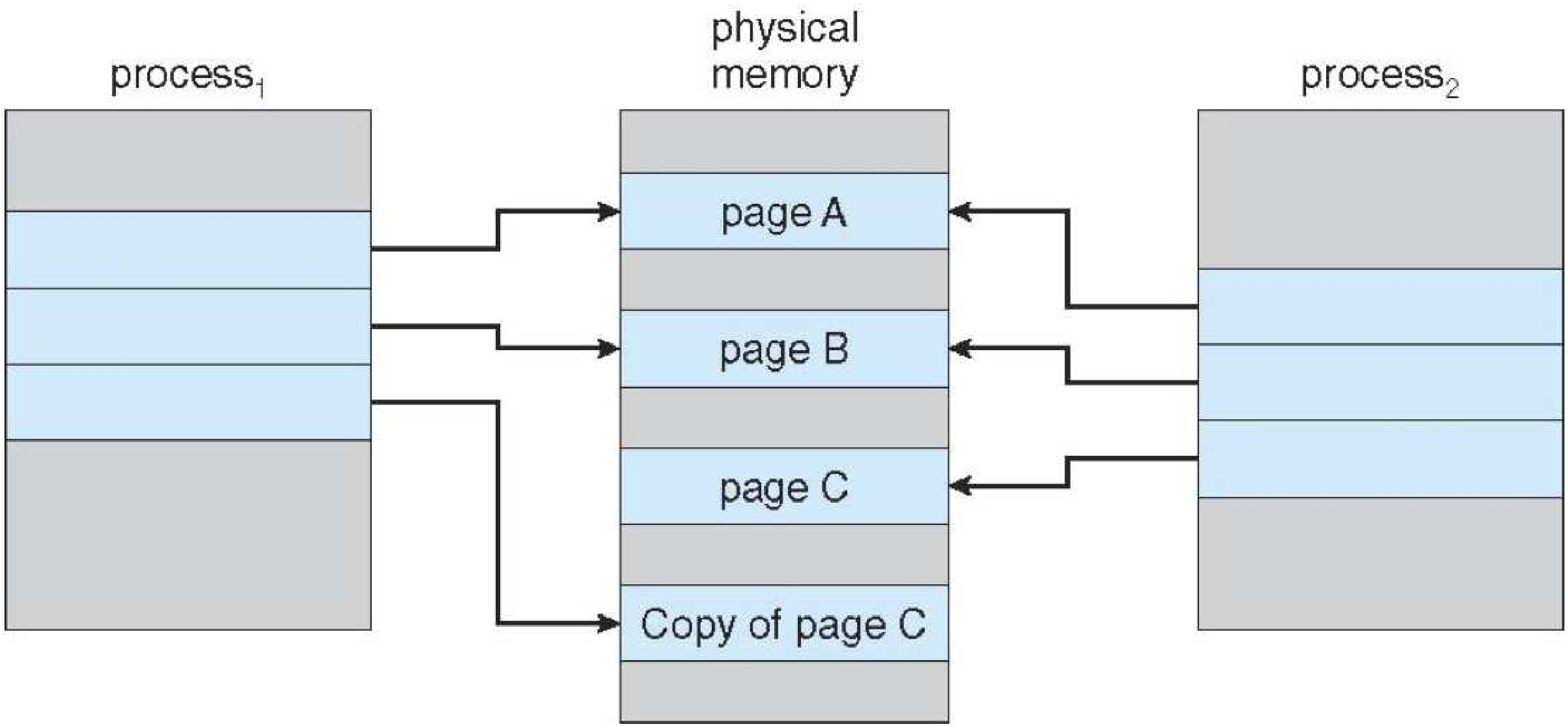
# Copy-on-Write

Before write:



# Copy-on-Write

After write:



# Pages and Protection

See **`mprotect.c`**

# Windows Notes

- **`mmap()`** ⇒ **`VirtualAlloc()`**
  - but allocation granularity can be more than a page
- **`munmap()`** ⇒ **`VirtualFree()`**
  - but only pages allocated by a single **`VirtualAlloc()`** call
- **`mprotect()`** ⇒ **`VirtualProtect()`**

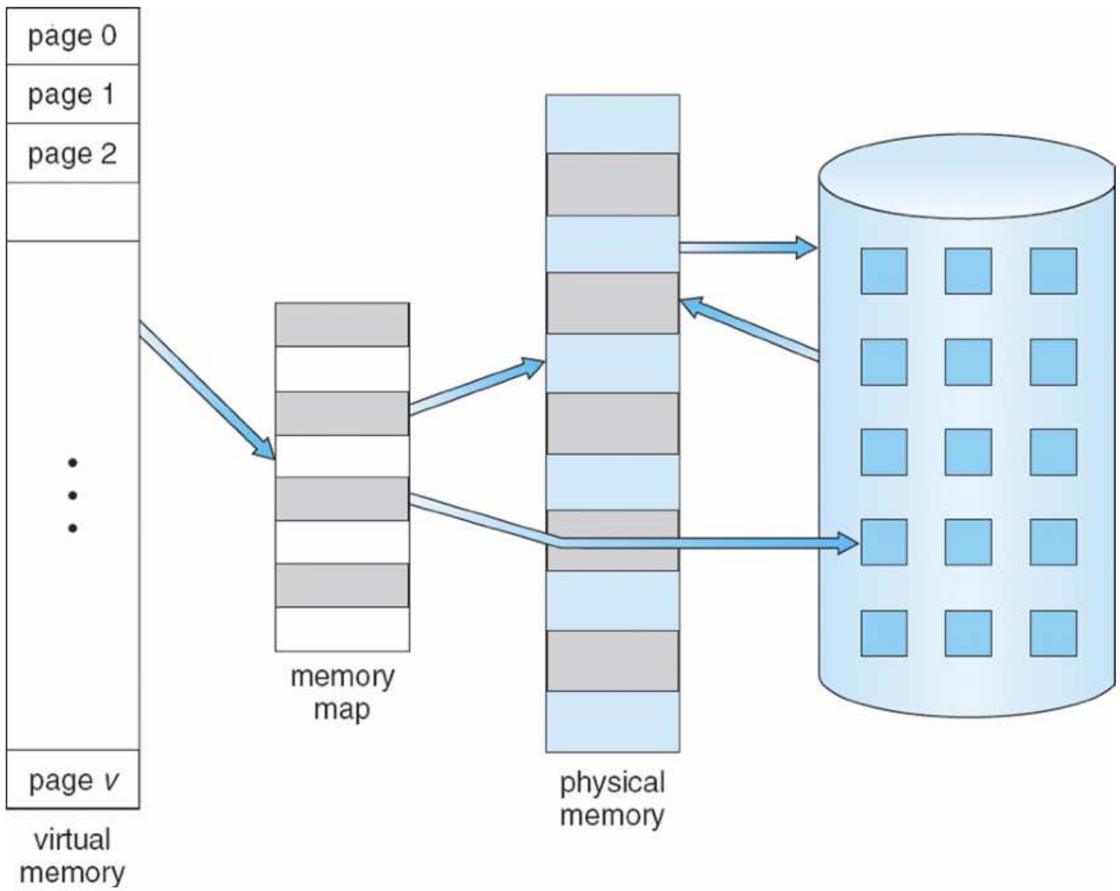
# Paging

Try this at home:

```
#include <stdlib.h>
#include <assert.h>
#define MB 512 /* adjust to match your machine */
#define SIZE (1024*1024*MB)

int main (void) {
  int i;
  char *c = (char *) malloc (SIZE);
  assert (c);

  for (i=0; i<SIZE; i++) c[i] = 0;
  for (i=0; i<SIZE; i++) c[i] = 0;
  for (i=0; i<SIZE; i++) c[i] = 0;

  return 0;
}
```

# Paging

***Paging*** means moving the data in virtual pages to secondary storage (to the physical frames can be reused)



# Loading Pages

- **When the process starts:** The virtual address space must no larger than the physical memory
- **Demand paging:** OS loads a page the first time it is referenced, and may remove a page from memory to make room for the new page.
- **Overlays:** Application programmer indicates when to load and remove pages (painful and error-prone)
- **Pre-paging:** OS guesses which pages the process will need and pre-loads them into memory; corect guesses allow more overlap of CPU and I/O (but difficult to get right due to branches in code)

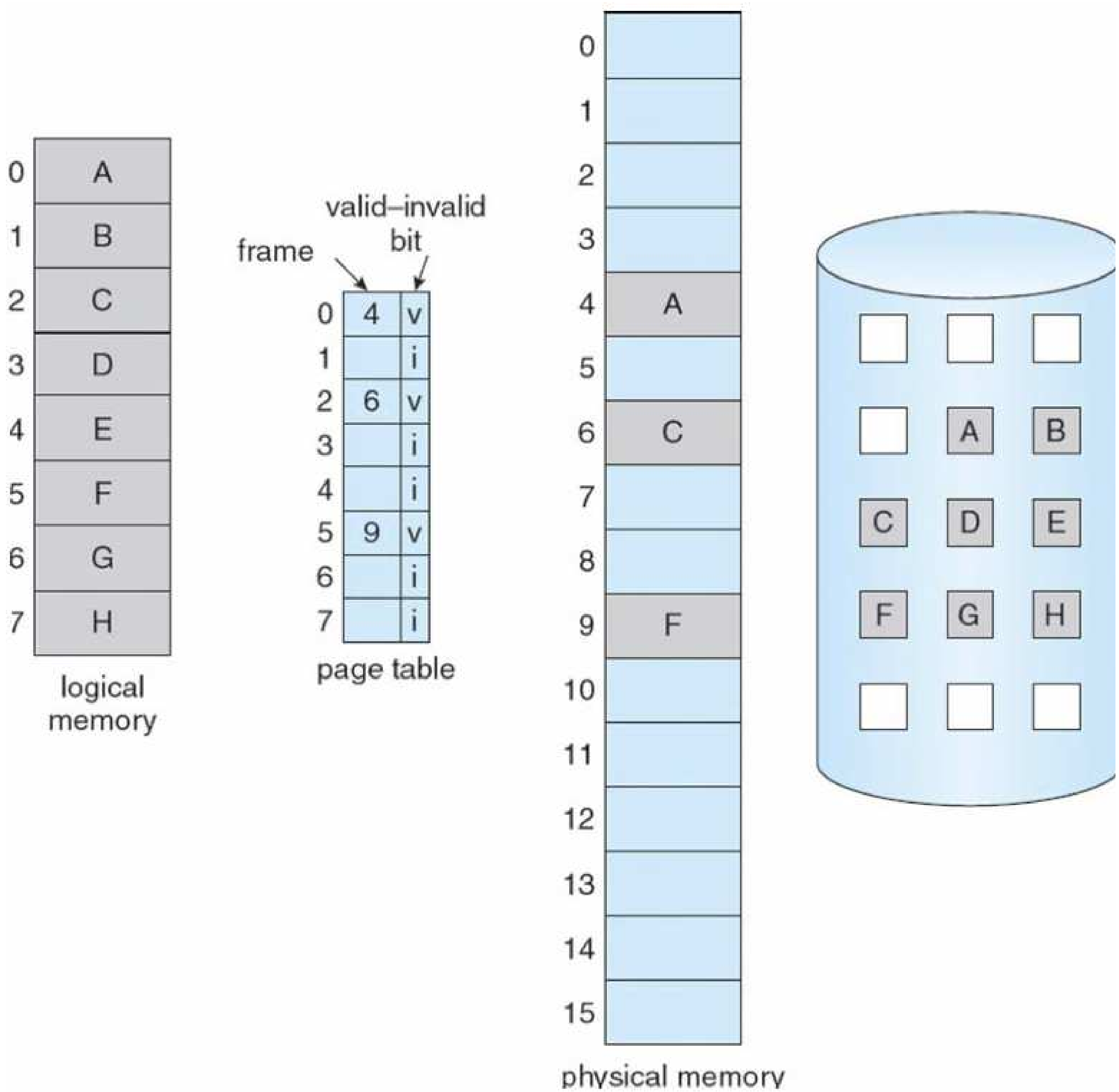
# Demand Paging

For each page, the page table either says:

- Page is in memory, and here is the frame number
- Page is on disk, and here is the block number

The *valid* bit is used to distinguish between these cases

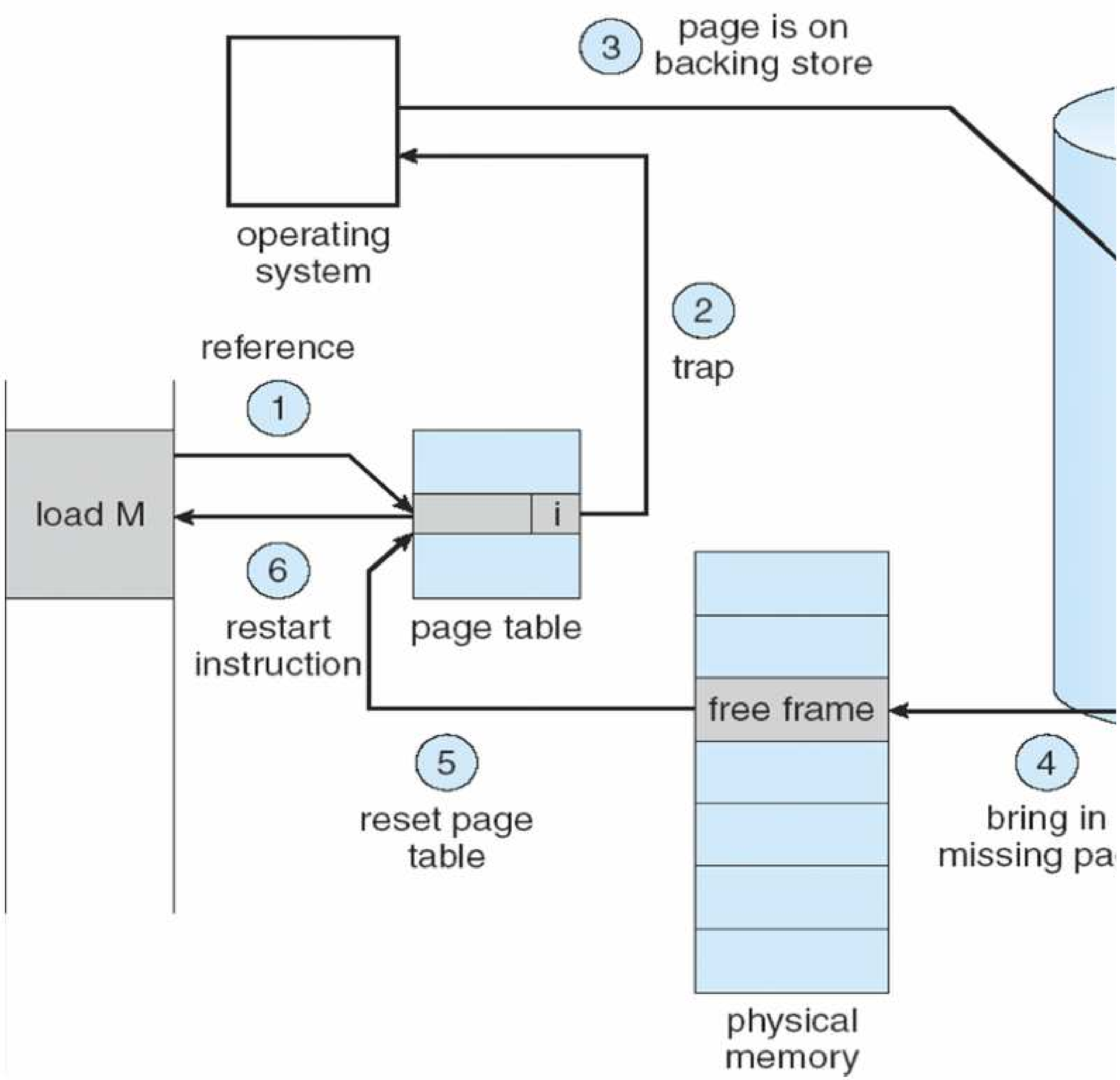
# Demand Paging



# Page Faults

A ***page fault*** is a virtual address is referenced and its data is on disk instead of memory

# Handling Page Faults



# Handling Page Faults

Are we in an interrupt handler?

- **Yes** — panic!
- **No** — Faulting address in current process space?
  - **Yes** — Access type matches permissions?
    - **Yes** — Demand-paging stuff: allocate a new frame, add it to the page table, ...
    - **No** — **`SIGSEGV`**
  - **No** — In user mode?
    - **Yes** — **`SIGSEGV`**
    - **No** — Address allowed to fault?
      - **Yes** — error code or **`SIGSEGV`**
      - **No** — panic!

Adapted from *Understanding the Linux Kernel* by Bovet and Cesati. Real code is more complicated; this is one of the grungier parts of an OS.

# Handling Page Faults

Hardware helps by saving the faulting instruction & CPU state

What about instructions with side-effects? (CISC)

```
mov a, (r10)+
```

which moves **a** into the address contained in register 10 and increments register 10

Solution: unwind side effects

Watch out for block-transfer instructions where the source and destination overlap

# Page Faults and TLB Miss

- ***Page fault***: page not in memory
- ***TLB miss***: virtual → physical mapping not cached
  - TLB hit ⇒ no page fault
  - TLB miss ⇒ maybe a page fault, maybe not

Hardware may or may not update TLB automatically

# Making Demand Paging Efficient

***Working set***: the set of pages a process will access in the near future

To work well, the working set of a process must fit in memory and must stay there

# Locality

Theoretically, a process could access a new page (or more!) of memory with each instruction

Fortunately, processes typically exhibit locality of reference

- ***Spatial locality***: when data is accessed, nearby data is likely to be accessed
- ***Temporal locality***: when data is accessed, it is likely to be accessed again

The 90/10 rule: a program spends 90% of its time using 10% of its data

# Performance

- *mem* is cost of accessing memory
- *pf* is the cost of handling a page fault
- $p$ is the probability of a page fault ($0 \leq p \leq 1$)

Assume no cache: every instruction accesses memory

$$\text{Effective access time} = (1-p) \times mem + p \times pf$$

If memory access time is 60 nanoseconds while it takes 6 milliseconds to handle a page fault:

$$\text{Effective access time} = (1-p) \times 60 + p \times 6{,}000{,}000$$

If we want the effective access time to be only 10% slower than memory access time, what value must $p$ have?

# Swap Space

Where do evicted pages go?

- If page has code, forget it and re-load from program image
- Otherwise, write the page to designated ***swap space*** on the disk

So, a page can be

- in memory
- on disk
- in swap space

# Summary

Benefits of demand paging:

- Virtual address space >> physical address space
- Processes can run without being fully loaded into memory
- Processes start faster, because they only need to load a few pages (for code and data) to start running
- Processes share memory more effectively, reducing the cost of context switches

***Virtual memory*** is

- Separation of virtual and physical address spaces—commonly implemented with pages
- Decoupling of size of virtual address space from size of physical address space—commonly implemented using demand paging



See the book for information on ***segmentation***