COMP1521 24T2 — Virtual Memory
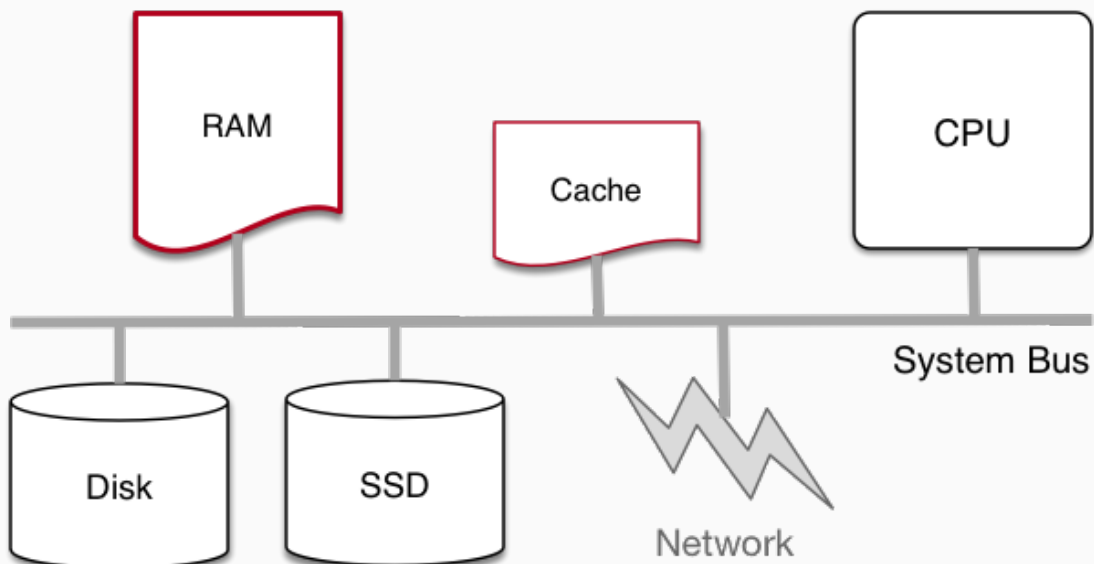
https://www.cse.unsw.edu.au/~cs1521/24T2/

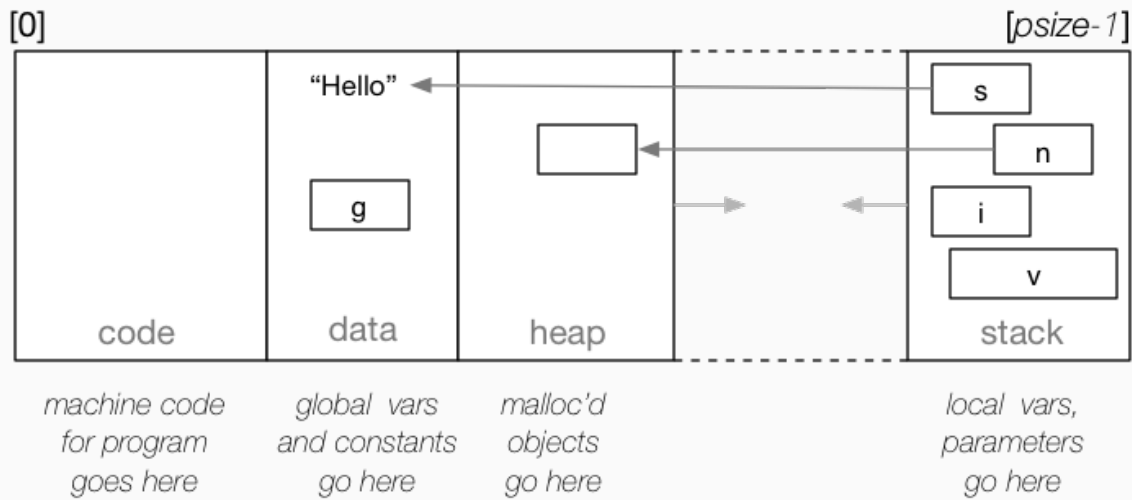## Introduction to Virtual Memory and Caching

- Short introduction to virtual memory and caching.

Memory

General purpose computers typically contain 4-128GB of volatile Random Access Memory (RAM)

Memory Regions

A view of memory for individual processes

Single Process Resident in RAM without Operating System

- Many small embedded systems run without operating system.

- Single program running, typically written in C, perhaps with some assembler.

- Devices (sensors, switches, ...) often wired at particular address.

- E.g motor speed can be set by storing byte at 0x100400.

- Program accesses (any) RAM directly.

- Development and debugging tricky.
    - might be done by sending ascii values bit by bit on a single wire

- Widely used for simple micro-controllers.

- Parallelism and exploiting multiple-core CPUs problematic

Single Process Resident in RAM with Operating System

- Operating systems need (simple) hardware support.

- Part of RAM (kernel space) must be accessible only in a privileged mode.

- System call enables privileged mode and passes execution to operating system code in kernel space.

- Privileged mode disabled when system call returns.

- Privileged mode could be implemented by a bit in a special register

- If only one process resident in RAM at any time - switching between processes is slow .

- Operating system must write out all RAM used by old process to disk (or flash) and read all memory of new process from disk.

- OK for some uses, but inefficient in general.

- Little used in modern computing.

## Multi Processes Resident in RAM without Virtual Memory

- If multiple processes to be resident in RAM operating system can swap execution between them quickly.

- RAM belonging to other processes & operating system operating system must be protected

- Hardware support can limit process accesses to particular **segment** (region) of RAM.

- BUT program may be loaded anywhere in RAM to run

- Breaks instructions which use absolute addresses, e.g.: `lw`, `sw`, `jr`

- Either programs can't use absolute memory addresses (relocatable code)

- Or code has to be modified (relocated) before it is run - not possible for all code!

- Major limitation - much better if programs can assume always have same address space

- Little used in modern computing.

Virtual Memory

- Big idea - disconnect address processes use from actual RAM address.

- Operating system translates (virtual) address a process uses to an physical (actual) RAM address.

- Convenient for programming/compilers - each process has same virtual view of RAM.

- Can have multiple processes be in RAM, allowing fast switching

- Can load part of processes into RAM on demand.

- Provides a mechanism to share memory betwen processes.

- Address to fetch every instruction to be executed must be translated.

- Address for load/store instructions (e.g. `lw`, `sw`) must be translated .

- Translation needs to be really fast - needs to be largely implemented in hardware (silicon).

## Virtual Memory with One Memory Segment Per Process

Consider a scenario with multiple processes loaded in memory:
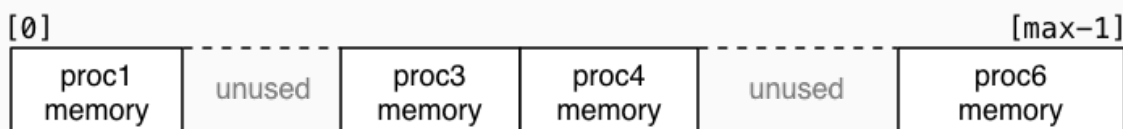
- Every process is in a contiguous section of RAM, starting at address **base** finishing at address **limit**.

- Each process sees its own address space as [0 .. size - 1]

- Process can be loaded anywhere in memory without change.

- Process accessing memory address **a** is translated to **a + base**

- and checked that **a + base** is **< limit** to ensure process only access its memory

- Easy to implement in hardware.

Virtual Memory with One Memory Segment Per Process

Consider the same scenario, but now we want to add a new process
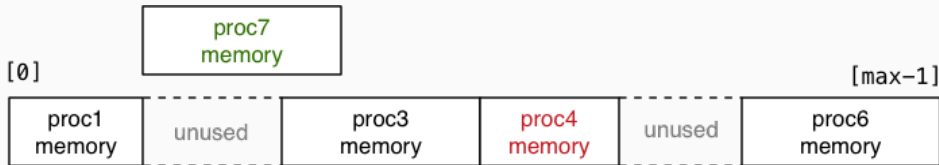
- The new process doesn't fit in any of the unused slots (fragmentation).
  - Need to move other processes to make a single large slot
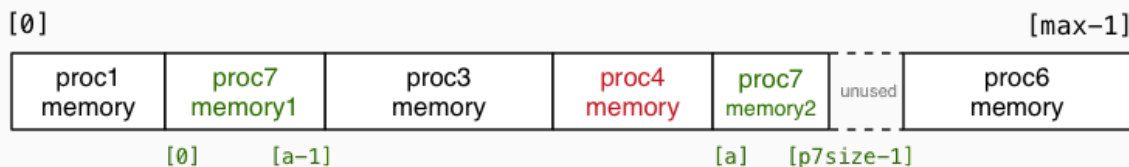

- Slow if RAM heavily used.
- Does not allow sharing or loading on demand.
- Limits process address space to size of RAM.
- Little used in modern computing.

Virtual Memory with Multiple Memory Segments Per Process

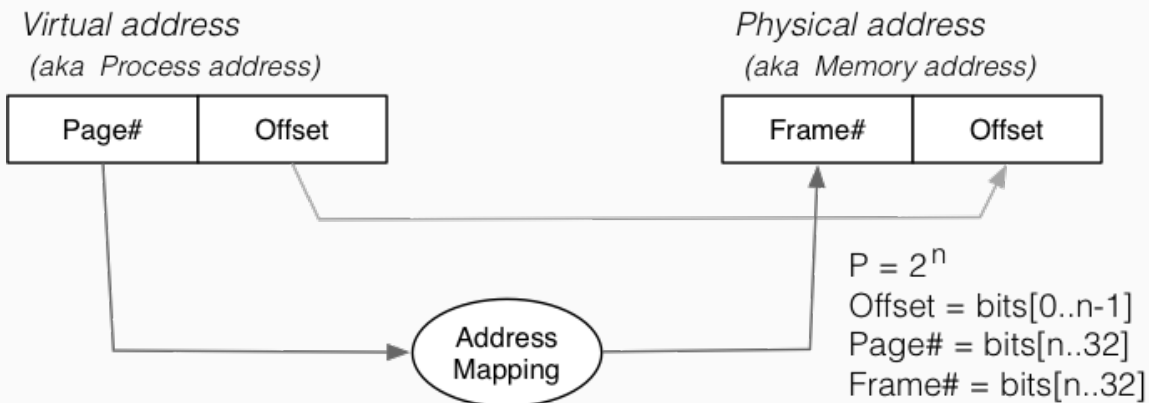Idea: split process memory over multiple parts of physical memory.
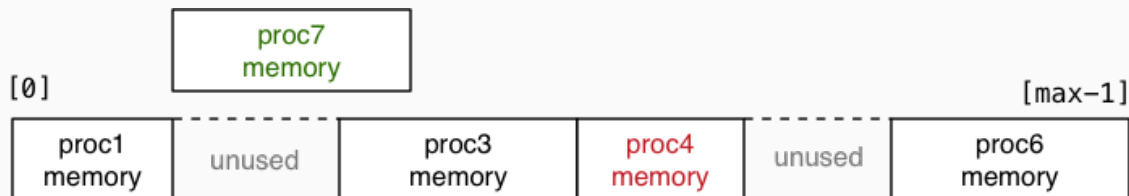
becomes

Virtual Memory with Pages

Big idea: make all segments same size, and make size power of 2

- call each segment of address space a *page* and make all pages the same size *P*

- translation of addresses can be implemented with an array

- each process has an array called the *page table*

- each array element contains the physical address in RAM of that page

- for virtual address *V*, *page_table[V / P]* contains physical address of page

- physical pages called **frames**

- the address will at be at offset *V % P* in both

- so physical address for *V* is: *page_table[V / P] + V % P*

- calculation can be faster/simpler bit operations if $P == 2^n$, e.g. 4096, 8192, 16384

- this is simple enough to implement in hardware (silicon)

Address Mapping

If $P == 2^n$, then some bits (*offset*) are the same in virtual and physical address

Virtual address
(aka Process address)

Page#	Offset

Physical address
(aka Memory address)

Frame#	Offset

Address
Mapping

$P = 2^n$
Offset = bits[0..n-1]
Page# = bits[n..32]
Frame# = bits[n..32]

Virtual Memory with pages - Lazy Loading

A side-effect of this type of virtual $\rightarrow$ physical address mapping

- don't need to load all of process's pages up-front

- start with a small memory "footprint" (e.g. `main` + stack top)

- load new process address pages into memory *as needed*

- grow up to the size of the (available) physical memory

The strategy of ...

- dividing process memory space into fixed-size pages

- on-demand loading of process pages into physical memory

is what is generally meant by *virtual memory*

Virtual Memory

4096 bytes is a common pages/frame size, but sizes 512 to 262144 bytes used

With 4GB memory, would have $\approx$ 1 million $\times$ 4KB frames

Each frame can hold one page of process address space

Leads to a memory layout like this (with *L* total pages of physical memory):

[0]	[1]	[2]	[3]				...	[L−1]	
proc1 page5	proc7 page1	proc1 page0	proc1 page1	*free*	proc4 page1	proc7 page3		proc7 page0	proc4 page3

Total L frames

When a process completes, all of its frames are released for re-use

Virtual Memory - Loading Pages

Consider a new process commencing execution ...

- initially has zero pages loaded

- load page containing code for `main()`

- load page for `main()`'s stack frame

- load other pages when process references address within page

Do we ever need to load all process pages at once?

Virtual Memory - Working Sets

From observations of running programs ...

- in any given window of time, process typically access only a small subset of their pages

- often called *locality of reference*

- subset of pages called the *working set*

Implications:

- if each process has a relatively small working set,
  can hold pages for many active processes in memory at same time

- if only need to hold some of process's pages in memory,
  process address space can be larger than physical memory

Virtual Memory - Loading Pages

We say that we "load" pages into physical memory

But where are they loaded from?

- code is loaded from the executable file stored on disk into read-only pages

- some data (e.g. C strings) also loaded into read-only pages

- initialised data (C global/static variables) also loaded from executable file

- pages for uninitialised data (heap, stack) are zero-ed

    - prevents information leaking from other processes

    - results in uninitialised local (stack) variables often containing 0

Virtual Memory - Loading Pages

We can imagine that a process's address space …

- exists on disk for the duration of the process's execution

- and only some parts of it are in memory at any given time
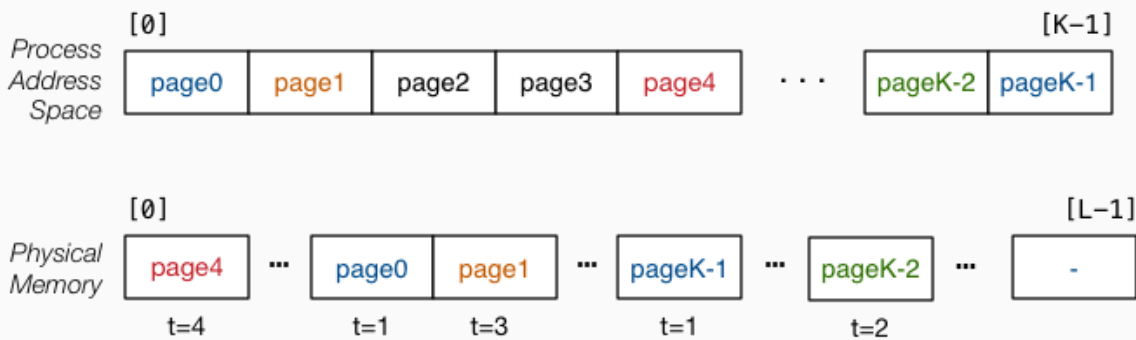
Transferring pages between disk↔memory is **very** expensive

- need to ensure minimal reading from / writing to disk

Virtual Memory - Handling Page Faults

An access to a page which is not-loaded in RAM is called a *page fault*.

Where do we load it in RAM?

First need to check for a free frame

- need a way of quickly identifying free frames

- commonly handled via a free list

What if there are currently no free page frames, possibilities:

- *suspend* the requesting process until a page is freed

- *replace* one of the currently loaded/used pages

Suspending requires the operating system to

- mark the process as unable to run until page available

- switch to running another process

Page Replacement

If no free pages we need to choose a page to evict:

- best page is one that won't be used again by its process

- prefer pages that are read-only (no need to write to disk)

- prefer pages that are unmodified (no need to write to disk)

- prefer pages that are used by only one process (see later)

OS can't predict whether a page will be required again by its process

But we do know whether it has been used recently (if we record this)
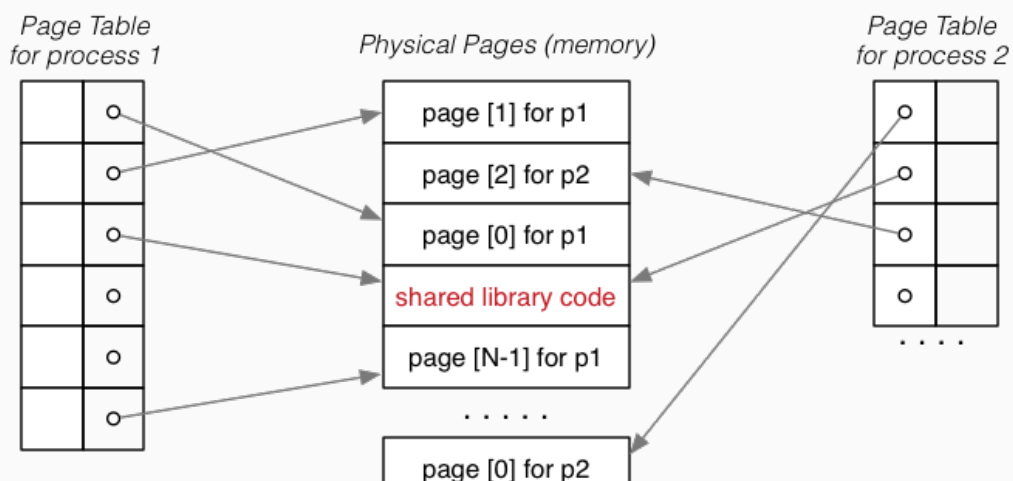
One good heuristic - replace Least Recently Used (LRU) page.

- page not used recently probably not needed again soon

Virtual Memory - Read-only Pages

Virtual memory allows sharing of read-only pages (e.g. for library code)

- several processes include same frame in virtual address space

- allows all running programs to use same pages for e.g. C library code (printf)

Page Table for process 1 | Physical Pages (memory) | Page Table for process 2

page [1] for p1
page [2] for p2
page [0] for p1
shared library code
page [N-1] for p1
. . . . .
page [0] for p2

Memory Management Hardware

Address translation is very important/frequent

- provide specialised hardware (MMU) to do it efficiently

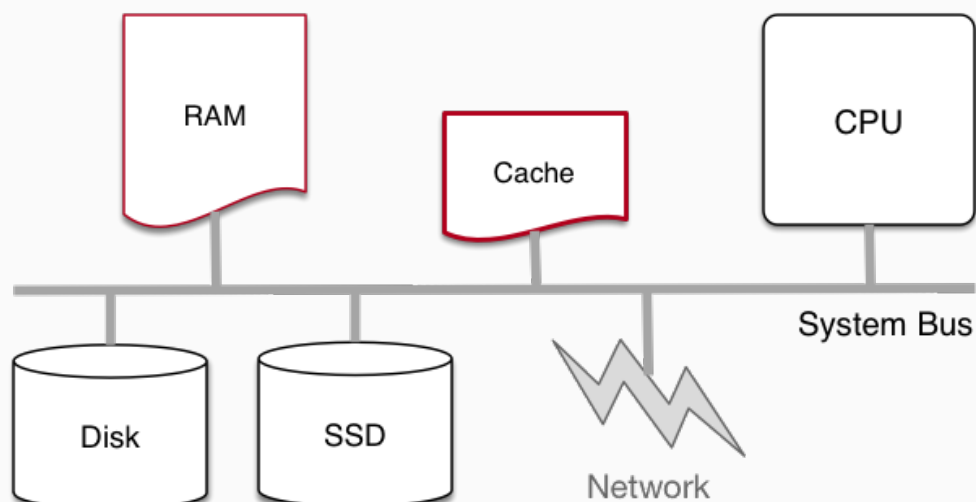- sometimes located on CPU chip, sometimes separate

RAM · Cache · CPU · MMU · System Bus · Disk · SSD · Network
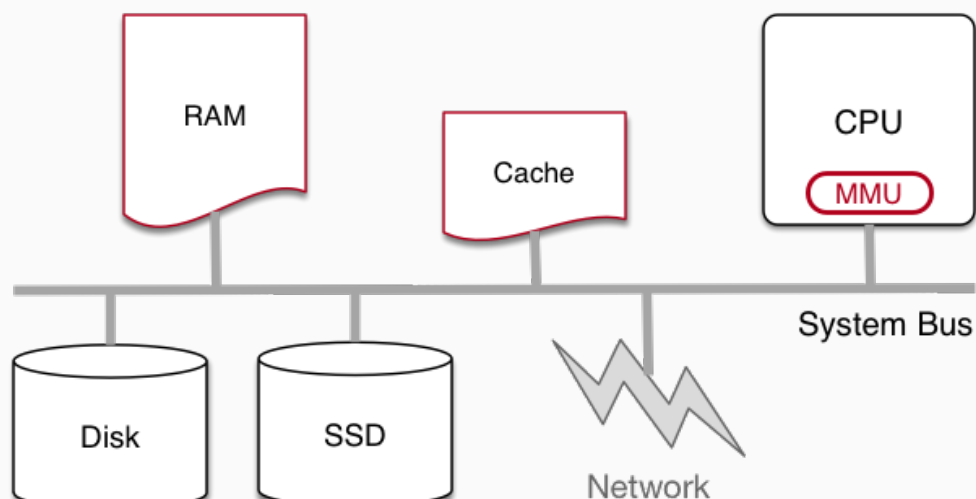
Cache Memory

Cache memory = small*, fast memory* close to CPU

RAM · Cache · CPU · System Bus · Disk · SSD · Network

Small = MB, Fast = $5 \times RAM$

Cache Memory

- cache memory makes memory accesses (e.g. **lw**, **sw**) faster
- cache memory implemented entirely in silicon typically on same chip as CPU
- independent of virtual memory (works with physical address)
- holds small blocks of RAM that are have been recently used
    - cache blocks also called cache lines
- typical size of cache blocks (line) 64 bytes
- CPU hardware (silicon) when loading or storing adddress first looks in cache
    - if block containing address is there, cache is used
        - for load operations value in cache is used
        - for store operations value in cache is changed
        - in both cases, much faster than access RAM
    - if not, block containing address is fetched from RAM into cache
    - possibly evicting an existing cache block
        - which may require writing (flushing) its contents to RAM
- cache replacement strategies have similar issues to virtual memory
- modern CPU may have multiple (3+) levels of caching