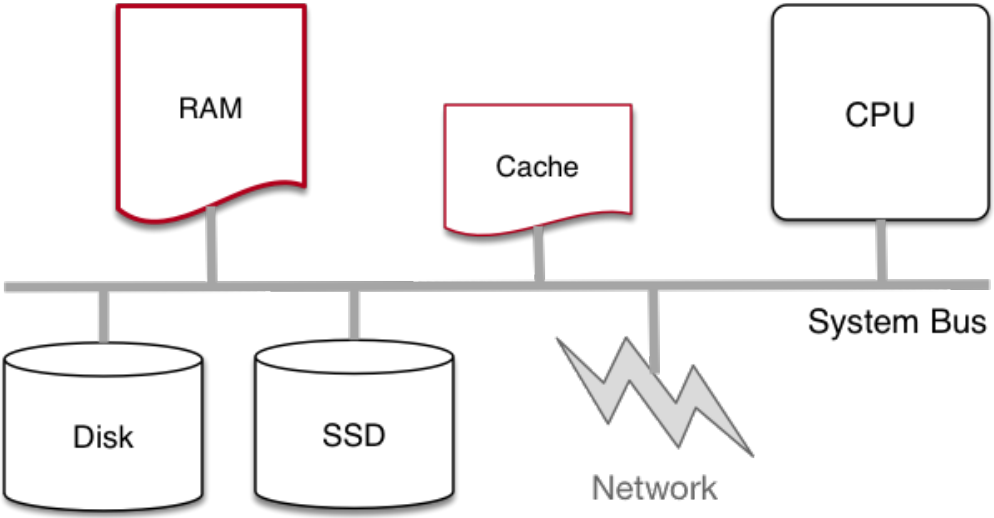
# COMP1521 21T3 — Virtual Memory

https://www.cse.unsw.edu.au/~cs1521/21T3/

# Memory

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



# Single Process Resident in RAM without Operating System

- Many small embedded system 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

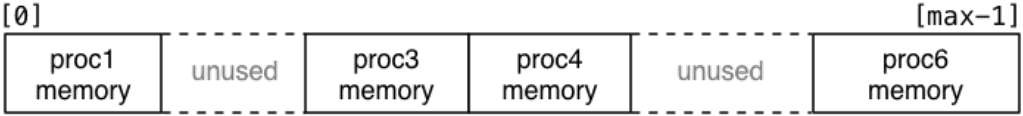
- If multiple processes to be resident in RAM O/S can swap execution between them quickly.
- RAM belonging to other processes & kernel must be protected
- Hardware support can limit process accesses to particular **segment** (region) of RAM.
- BUT program may be loaded anywhere in RAM to run
- Breaks 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

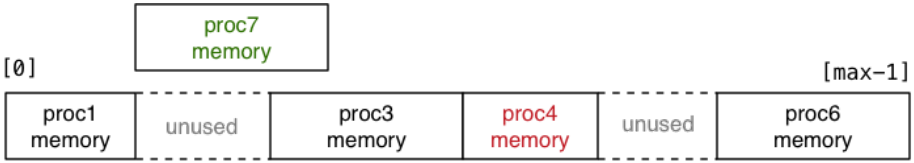
Consider a scenario with multiple processes loaded in memory:



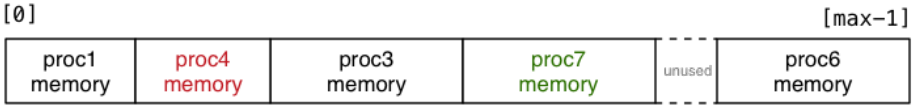

- Every process is in a contiguous section of RAM, starting at address **base** finishing at address **limit**.
- Each process sees its own address space as [0 .. size - 1]
- Process can be loaded anywhere in memory without change.
- Process accessing memory address **a** is translated to **a + base**
- 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

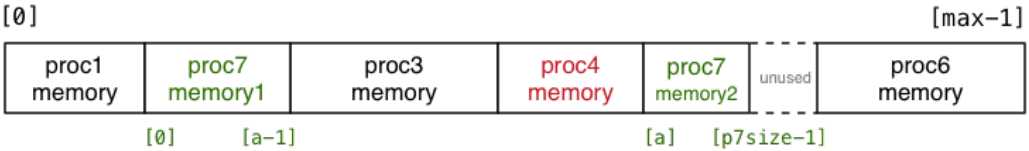



- Slow if RAM heavily used.
- Does not allow sharing or loading on demand.
- Limits process address space to size of RAM.
- Little used in modern computing.

# Virtual Memory with Multiple Memory Segments Per Process

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


becomes



# Virtual Memory with Multiple Memory Segments Per Process

With arbitrary sized memory segments, translating virtual to physical address is complicated making hardware support difficult:

```
// translate virtual_address to physical RAM address
uint32_t translate(uint32_t process_id, uint32_t virtual_addr) {
  uint32_t n_segments;
  Segment *segments = get_segments(process_id, &n_segments);
  for (int i = 0; i < n_segments; i++) {
    Segment *c = &segments[i];
    if (virtual_addr >= c->base &&
        virtual_addr <  c->base + c->size) {
      uint32_t offset = virtual_addr - c->base;
      return c->mem + offset;
    }
  }
  // handle illegal memory access
}
```

# Virtual Memory with Pages

Address mapping would be simpler if all segments were same size

- call each segment of address space a ***page***
- make all pages the same size ***P***
- page ***I*** holds addresses: ***I*P ... (I+1)*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
- the address will at be at offset ***V % P*** in both pages
- so physical address for ***V*** is: ***page_table[V / P] + V % P***

# Virtual Memory with Pages

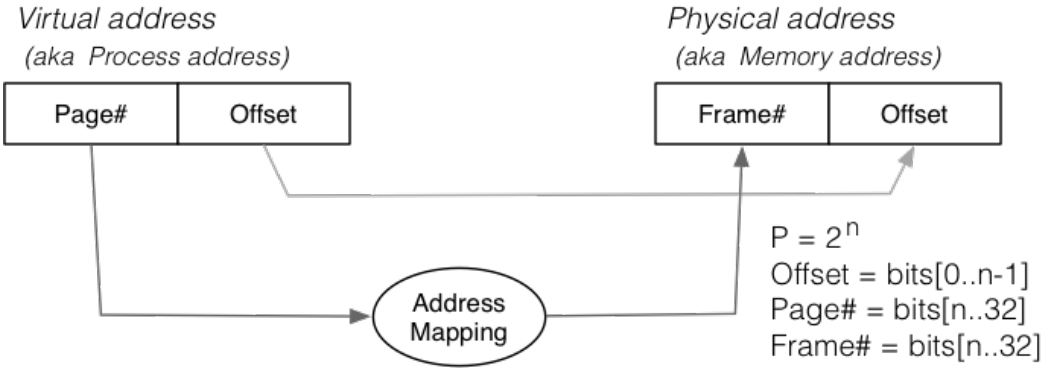
With pages, translating virtual to physical address is simpler making hardware support difficult:

```
// translate virtual_address to physical RAM address
uint32_t translate(uint32_t process_id, uint32_t virtual_addr) {
  uint32_t pt_size;
  PageInfo *page_table = get_page_table(process_id, &pt_size);
  page_number = virtual_addr / PAGE_SIZE;
  if (page_number < pt_size) {
      uint32_t offset = virtual_addr % PAGE_SIZE;
      return PAGE_SIZE * page_table[page_number].frame + offset;
  }
  // handle illegal memory access
}
```

- Calculation of *page_number* and *offset* can be faster/simpler bit operations if *PAGE_SIZE* $== 2^n$, e.g. 4096, 8192, 16384
- Note **PageInfo** entries will have more information about the page ...

# Address Mapping

If $P == 2^n$, then address mapping becomes



# 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

Pages/frames are typically 4KB .. 256KB in size

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):


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

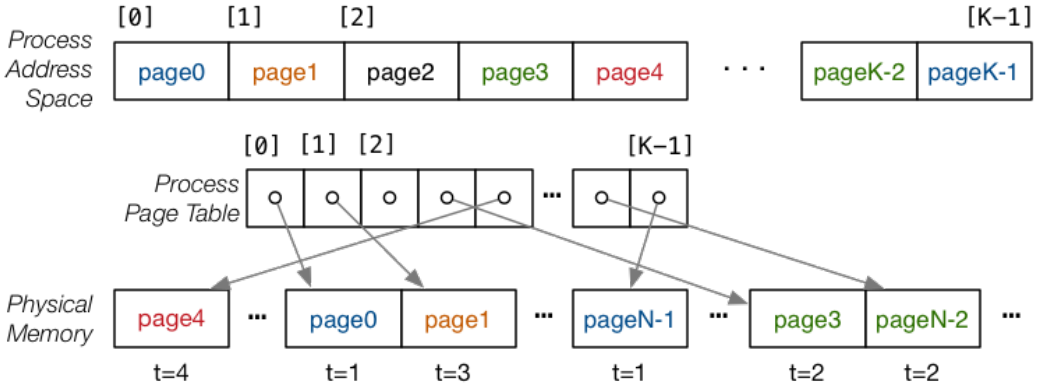
# Page Tables

Consider a possible per-process page table, e.g.

- each page table entry (PTE) might contain
  - page status ... *not_loaded*, *loaded*, *modified*
  - frame number of page (if *loaded*)
  - ... maybe others ... (e.g. last accessed time)
- we need $\lceil ProcSize / PageSize \rceil$ entries in this table

# Example Page Table

Example of page table for one process:




Timestamps show when page was loaded.

# Virtual Memory - Loading Pages

```
typedef struct {int status, int frame, ...} PageInfo;

uint32_t translate(uint32_t process_id, uint32_t virtual_addr) {
  uint32_t pt_size;
  PageInfo *page_table = get_page_table(process_id, &pt_size);
  page_number = virtual_addr / PAGE_SIZE;
  if (page_number < pt_size) {
      if (page_table[page_number].status != LOADED) {
          // page fault - need to load page into free frame
          page_table[page_number].frame = ???
          page_table[page_number].status = LOADED;
      }
      uint32_t offset = virtual_addr % PAGE_SIZE;
      return PAGE_SIZE * page_table[page_number].frame + offset;
  }
  // handle illegal memory access
}
```

# 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?

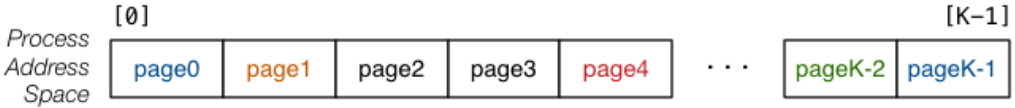
- code is loaded from the executable file stored on disk into read-only pages
- some data (e.g. C strings) also loaded into read-only pages
- initialised data (C global/static variables) also loaded from executable file
- pages for uninitialised data (heap, stack) are zero-ed
  - prevents information leaking from other processes
  - results in uninitialised local (stack) variables often containing 0

Consider a process whose address space exceeds physical memory

# Virtual Memory - Loading Pages

We can imagine that a process's address space ...

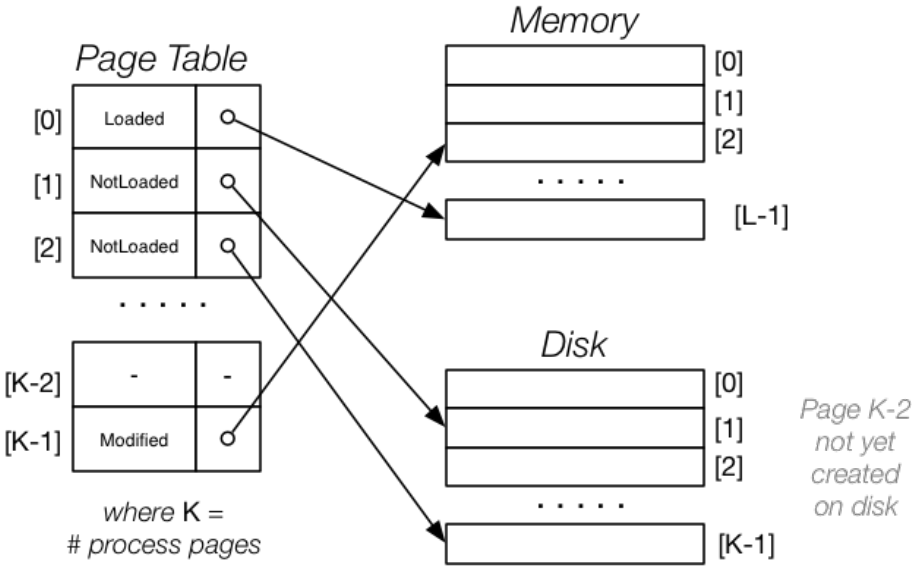
- exists on disk for the duration of the process's execution
- and only some parts of it are in memory at any given time




Transferring pages between disk↔memory is **very** expensive

- need to ensure minimal reading from / writing to disk

# Page table with some pages not loaded



# 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
- mark the process as able to run when page available

# 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 Ued (LRU) page.

- page not used recently probably not needed again soon

# Exercise: Page Replacement

Show how the page frames and page tables change when

- there are 4 page frames in memory
- the process has 6 pages in its virtual address space
- a LRU page replacement strategy is used

For each of the following sequences of virtual page accesses

```
0, 5, 0, 0, 5, 1, 5, 1, 2, 4, 3, 3, 4, 2, 5, 3, 2
```

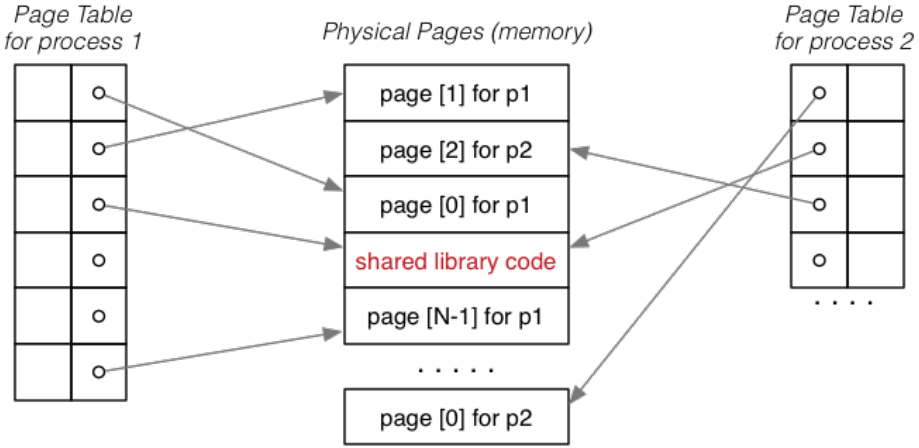
```
5, 0, 0, 0, 5, 1, 1, 5, 1, 5, 2, 2, 3, 0, 0, 5
```

Assume that all PTEs and frames are initially empty/unused

# Virtual Memory - Read-only Pages

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

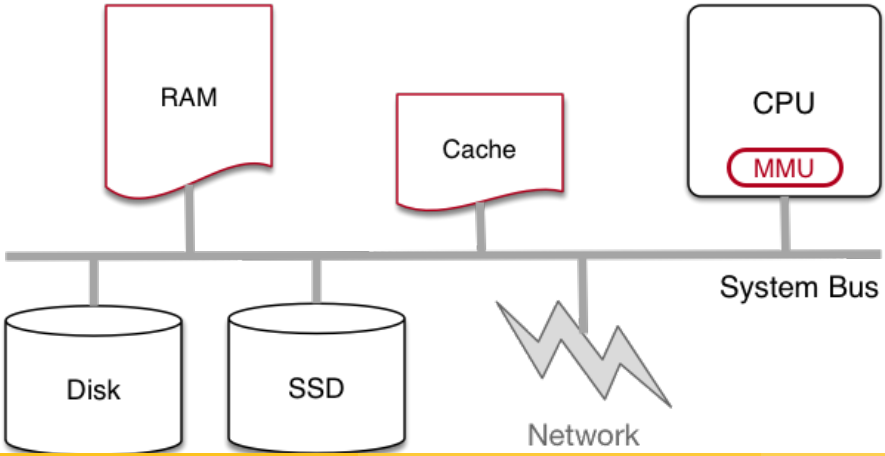
- several processes include same frame in virtual address space



# Memory Management Hardware

Address translation is very important/frequent

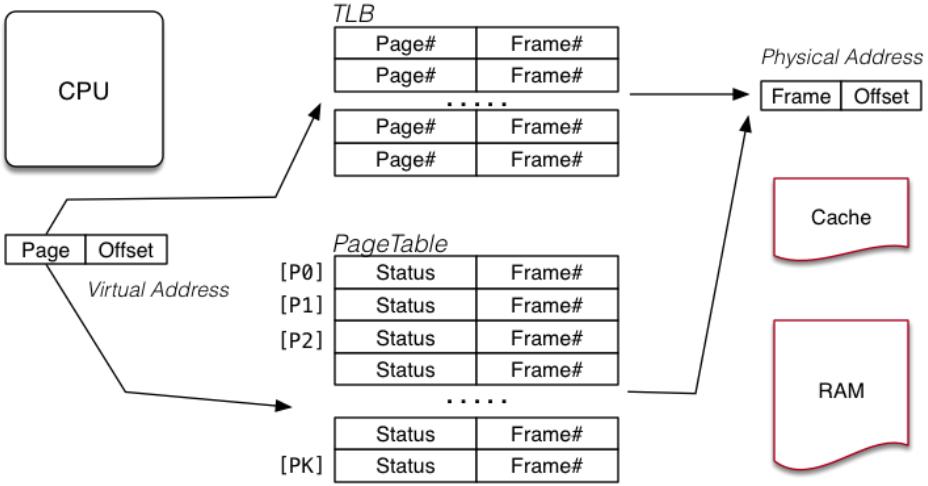
- provide specialised hardware (MMU) to do it efficiently
- sometimes located on CPU chip, sometimes separate



# Memory Management Hardware

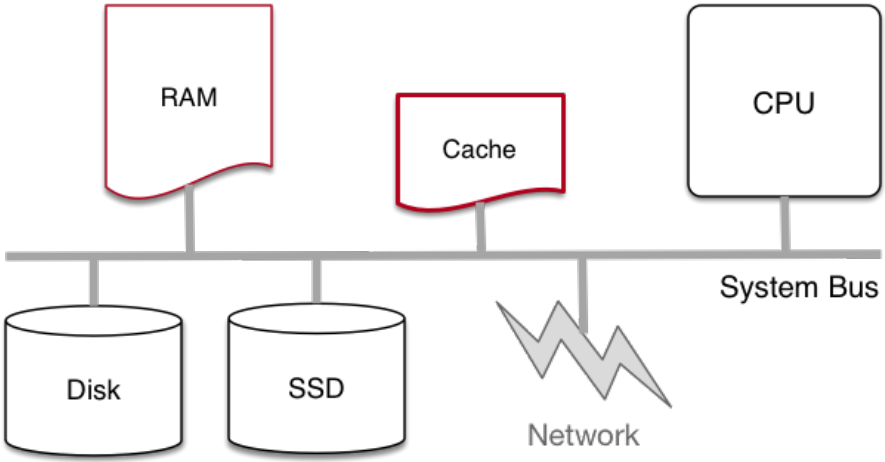
TLB = translation lookaside buffer

- lookup table containing (virtual,physical) address pairs



# Cache Memory

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




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