MONASH University

Information Technology

FIT2100 Operating Systems

MEMORY MANAGEMENT - PART 1

Week 9
Semester 2 2024
(Reading: Tanenbaum: Chapter 3 and Stallings: Chapter 7, 8)

Dr Charith Jayasekara
Faculty of Information Technology

OUTLINE

- Swapping
- Shared Memory
- Virtual Memory
- Paged Virtual Memory
- Page Replacement Algorithms
- Segmented Virtual Memory
- Segment Placement Algorithms

LEARNING OUTCOMES

- Upon the completion of week 9, you should be able to:
  - Understand why **virtual memory** is useful
  - Understand the differences between **paging** and **segmentation**
  - Understand how to translate a **logical** memory address into a **physical** address
  - Understand the most common **page and segment replacement** algorithms

WHY STUDY MEMORY MANAGEMENT?

- *Mass storage and VM organisation and design strongly impact performance and functionality of all modern computer systems.*

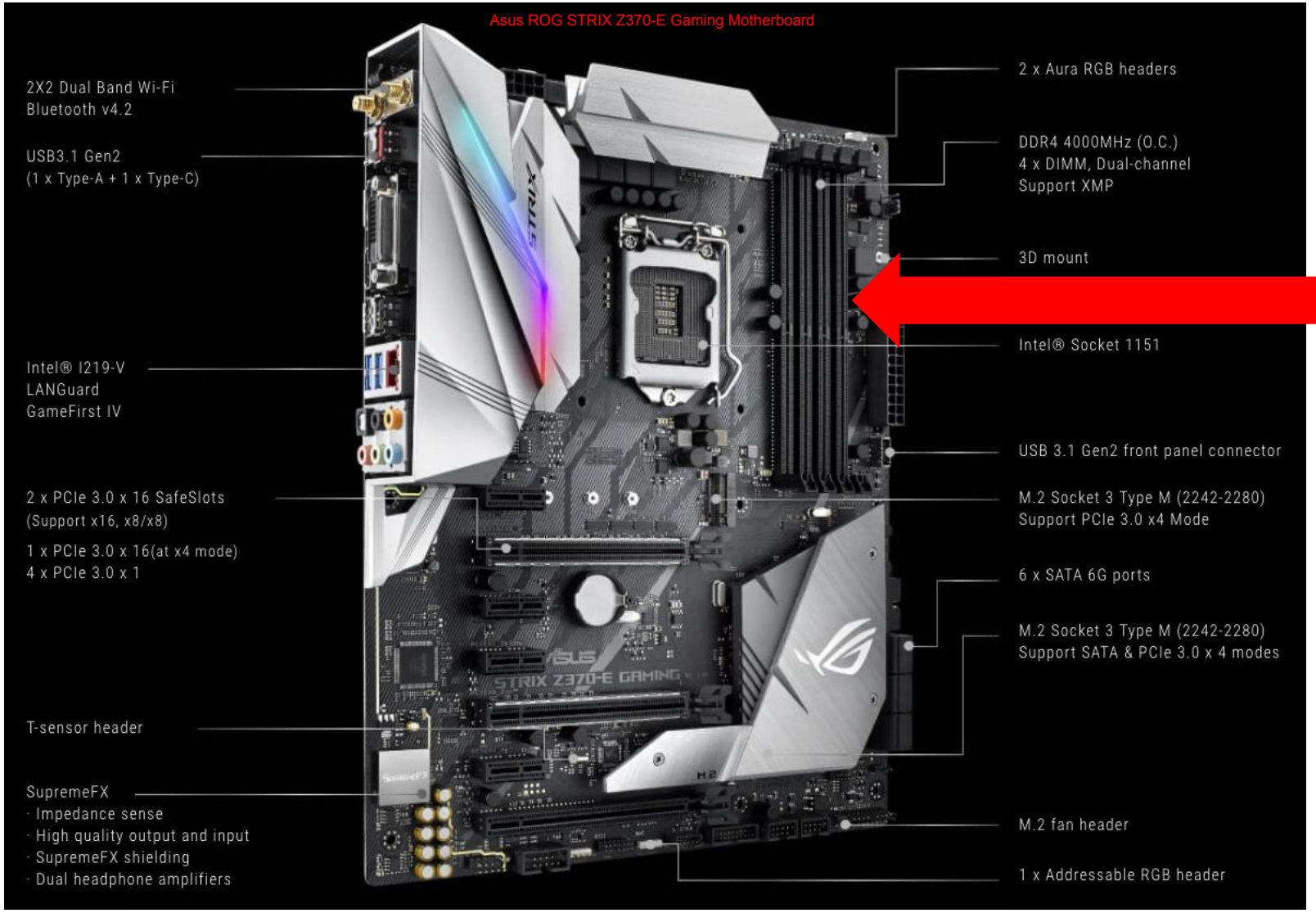
- **Foundation knowledge**: Mass storage and VM are critical to system performance and a major area in contemporary machine design.

- **Practical skills**: Ability to optimise application code design to best take advantage of memory architecture and thus maximise application performance.

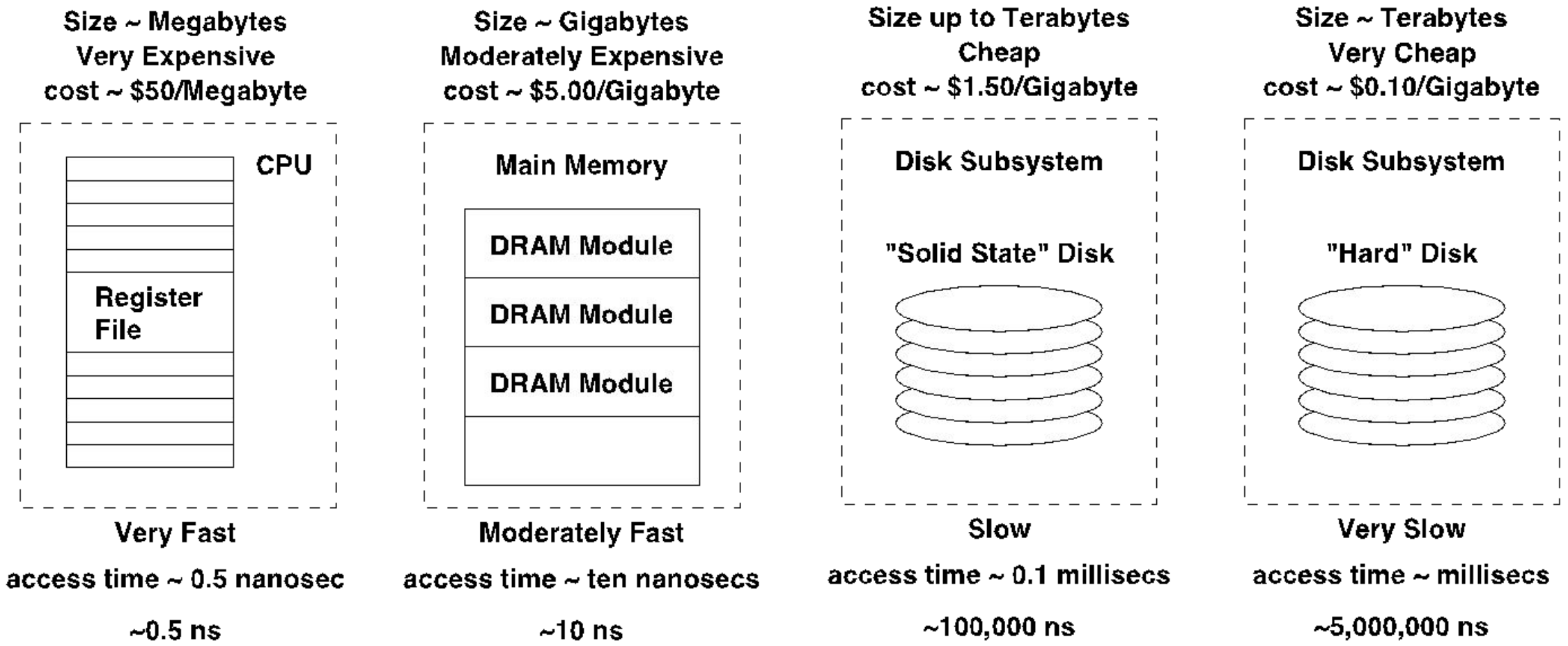
- **Practical skills**: Ability to recognise and differentiate host performance by understanding memory management limitations of the OS.

- **Practical skills**: Ability to anticipate likely application performance problems by relating application design to memory management design in target platforms for the application.

MEMORY FUNDAMENTALS



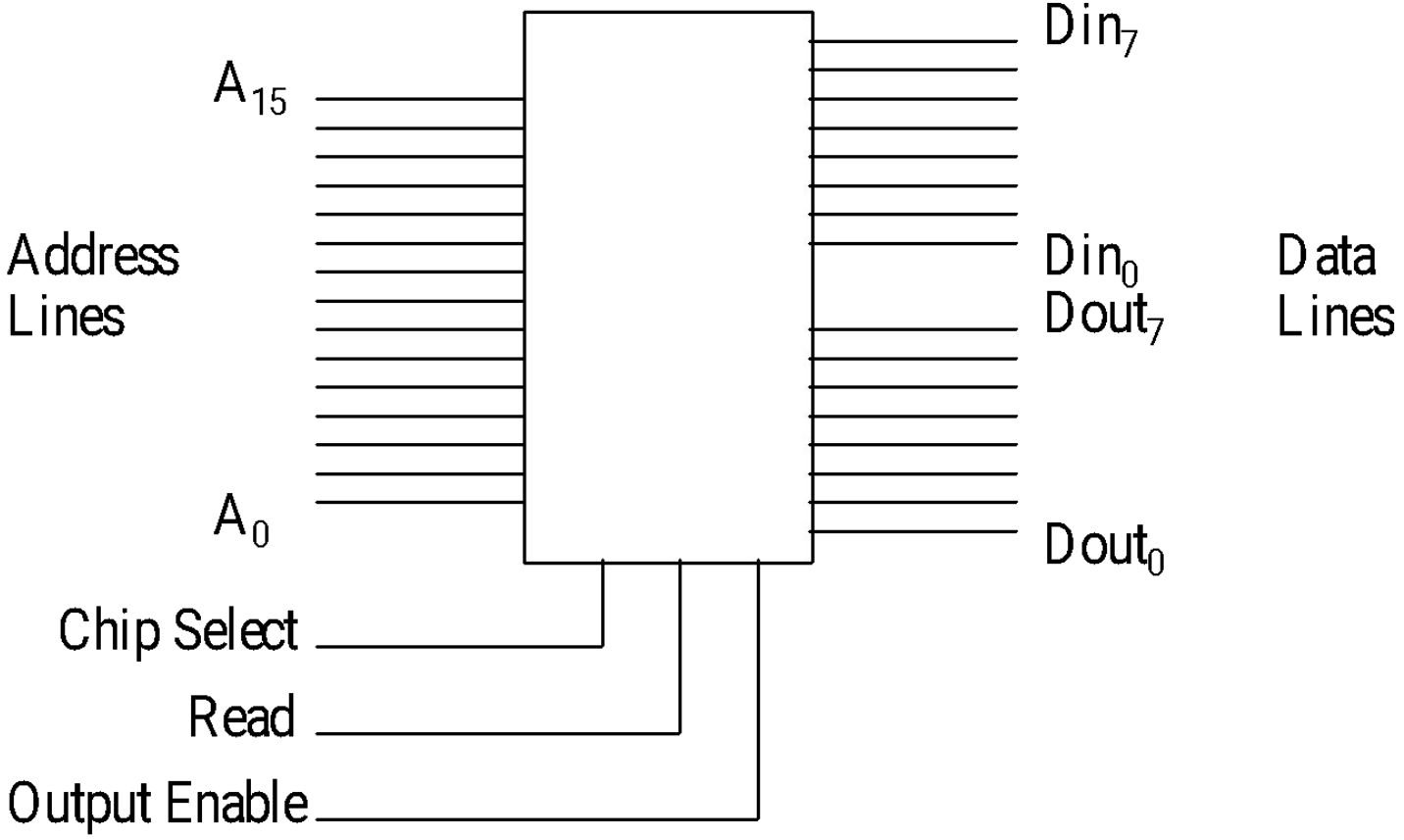
Storage Hierarchies



Note the trade between capacity and cost per capacity;
This tradeoff evolves due to Moore's and Kryder's Laws

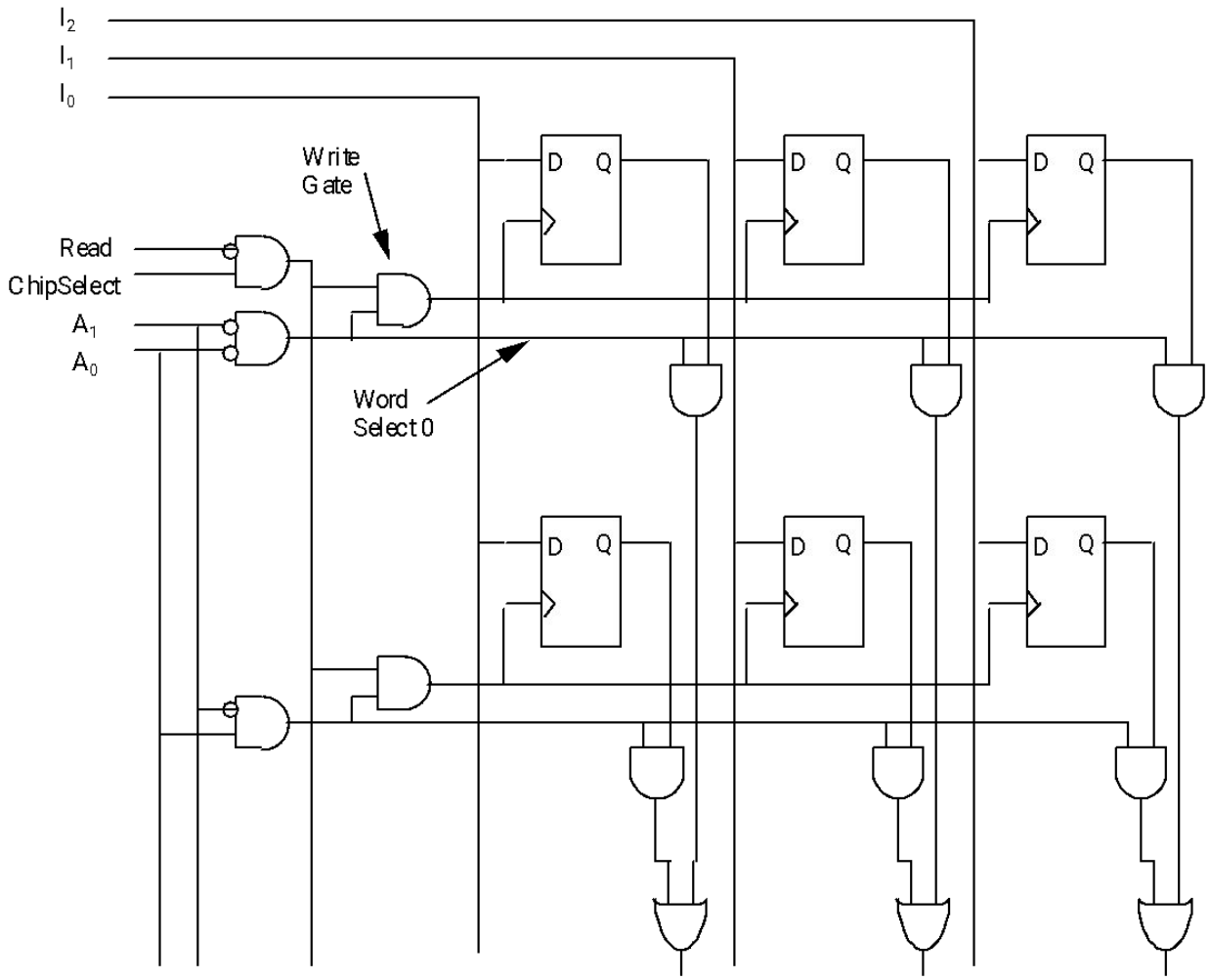
Memory (Revision)

- Flip flops are the building blocks for memory units.
- Memory chips need:
  - A data interface
  - An address interface
  - A control interface



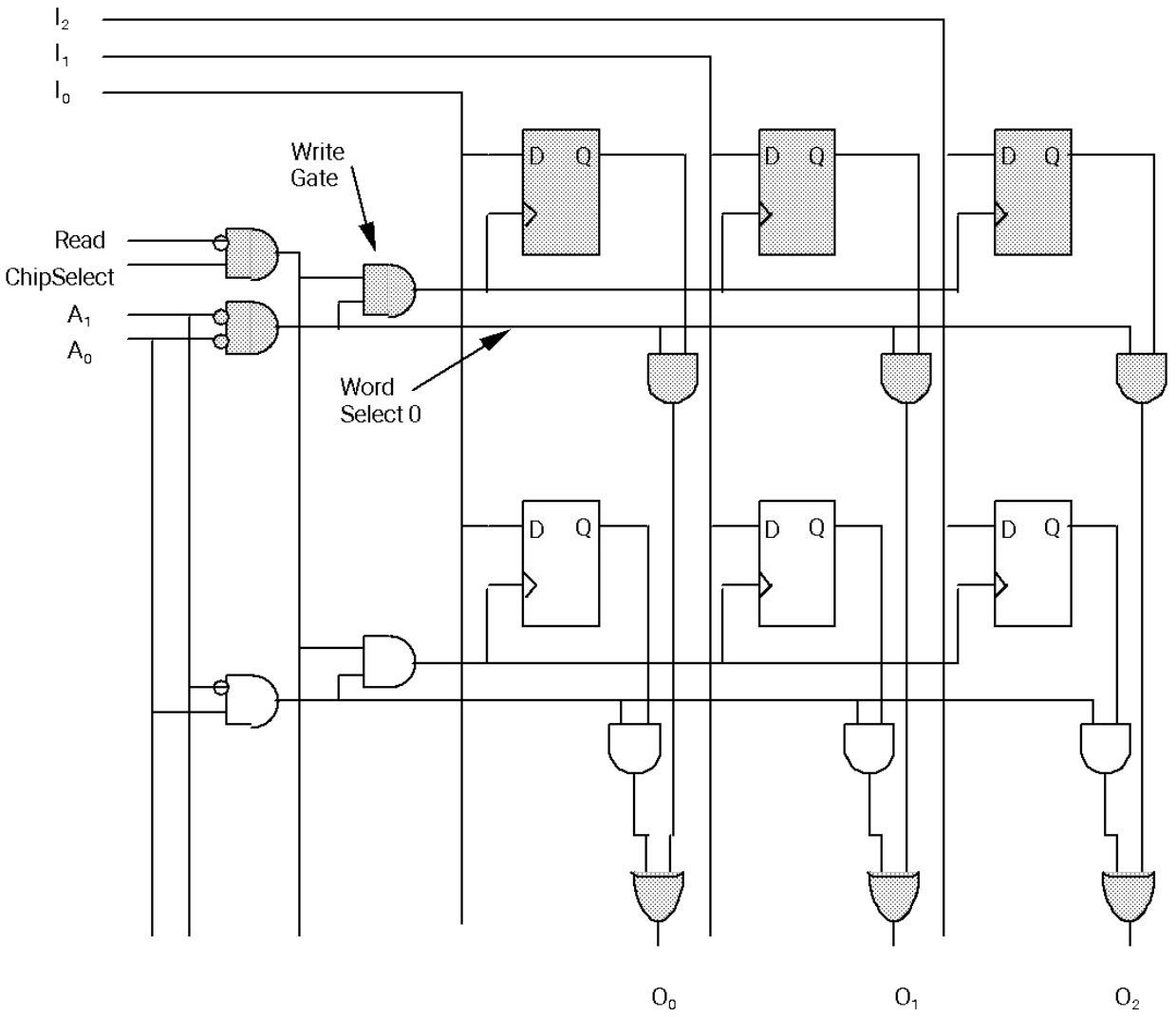
Memory (Revision)

- Address is decoded to produce a word select signal
- A write gate controls the clock signals
- Word select controls the output from each word
- Example is 2 words of a 4 word memory. Word size is 3 bits

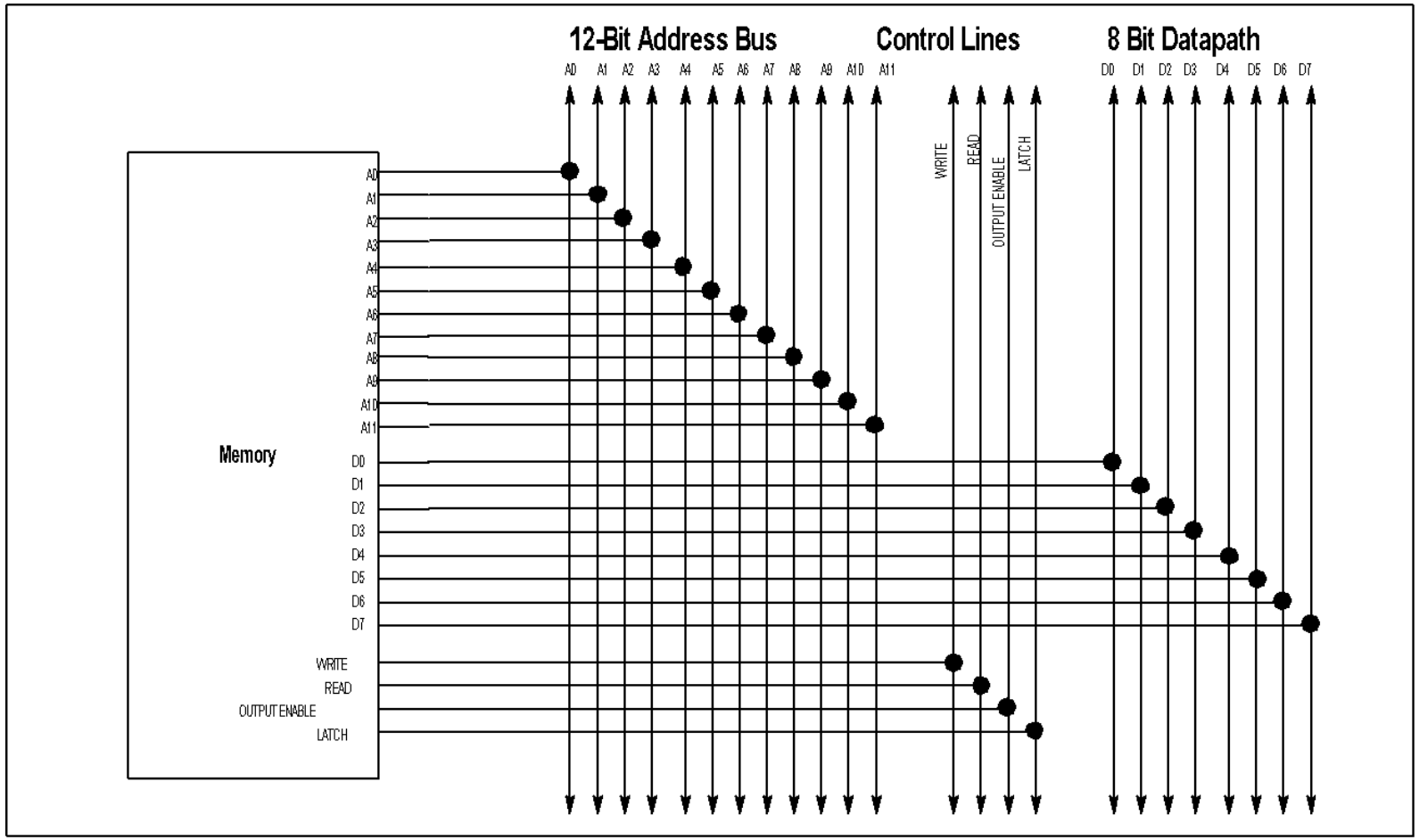


Memory (Revision)

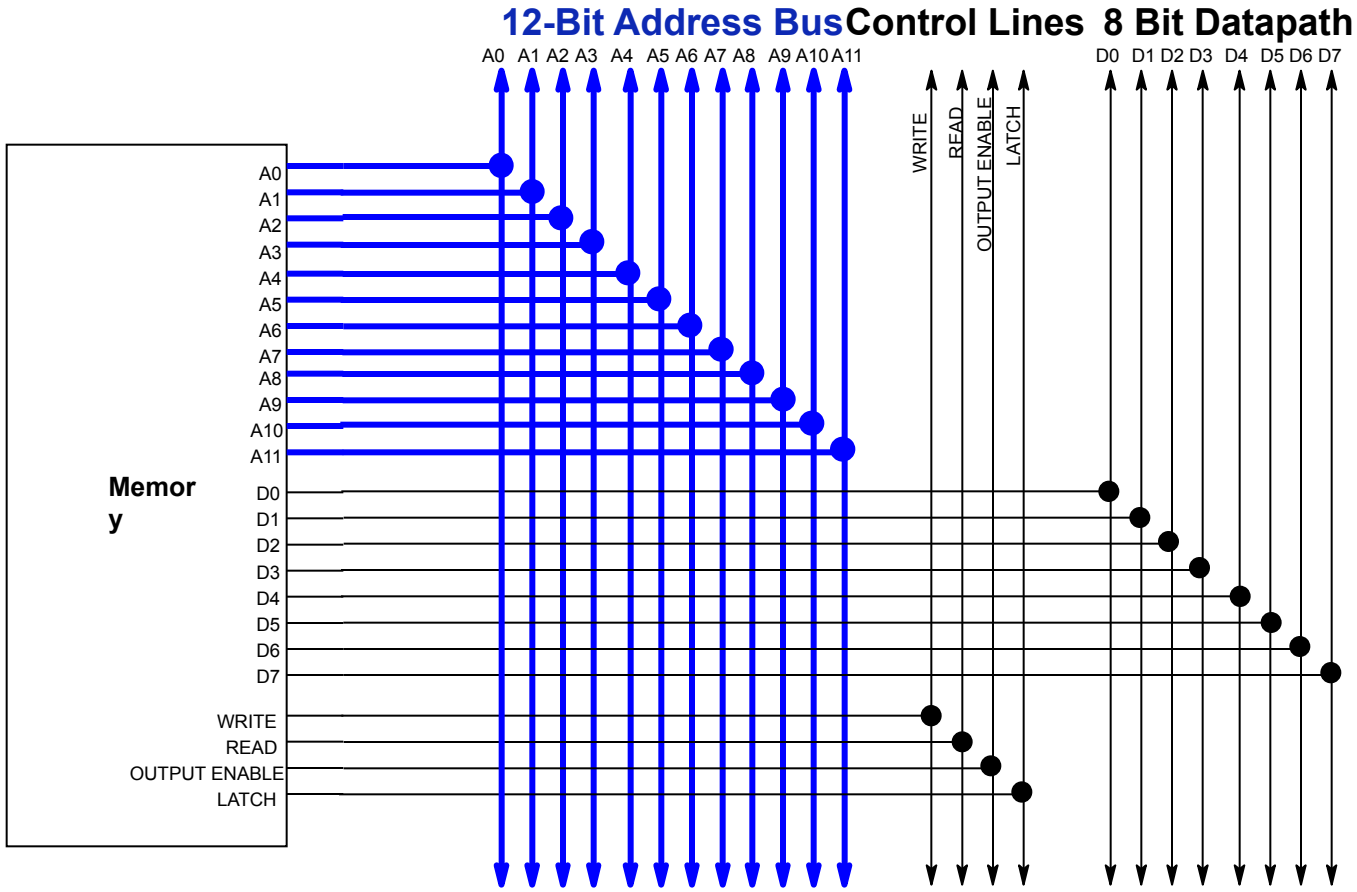
- Word 0 is enabled by address 00
- This allows data from the flip flops to flow to the output OR gates
- If a chip select is done whilst the read line is 0 then a clock is generated



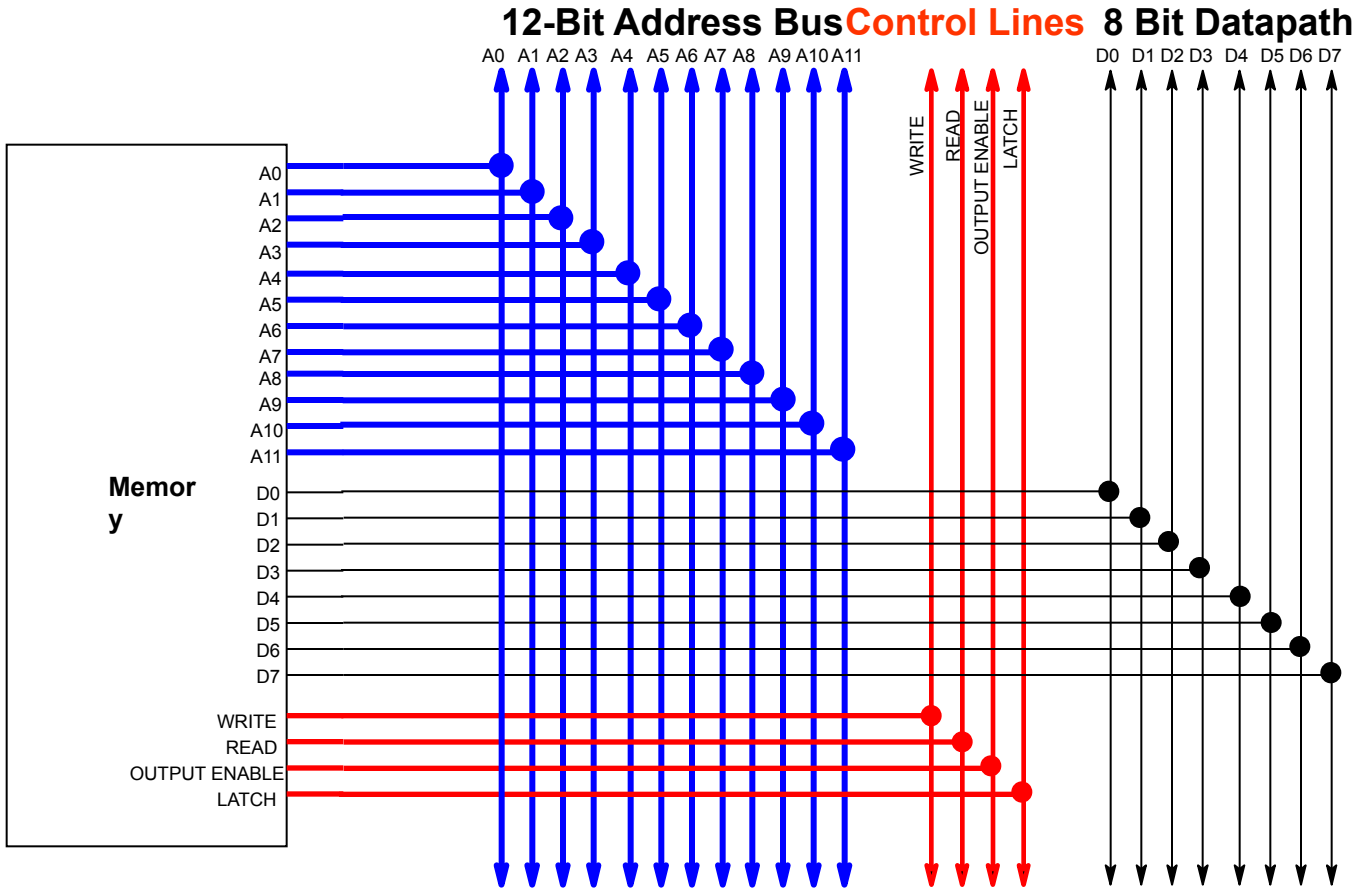
Memory Addressing (Revision)



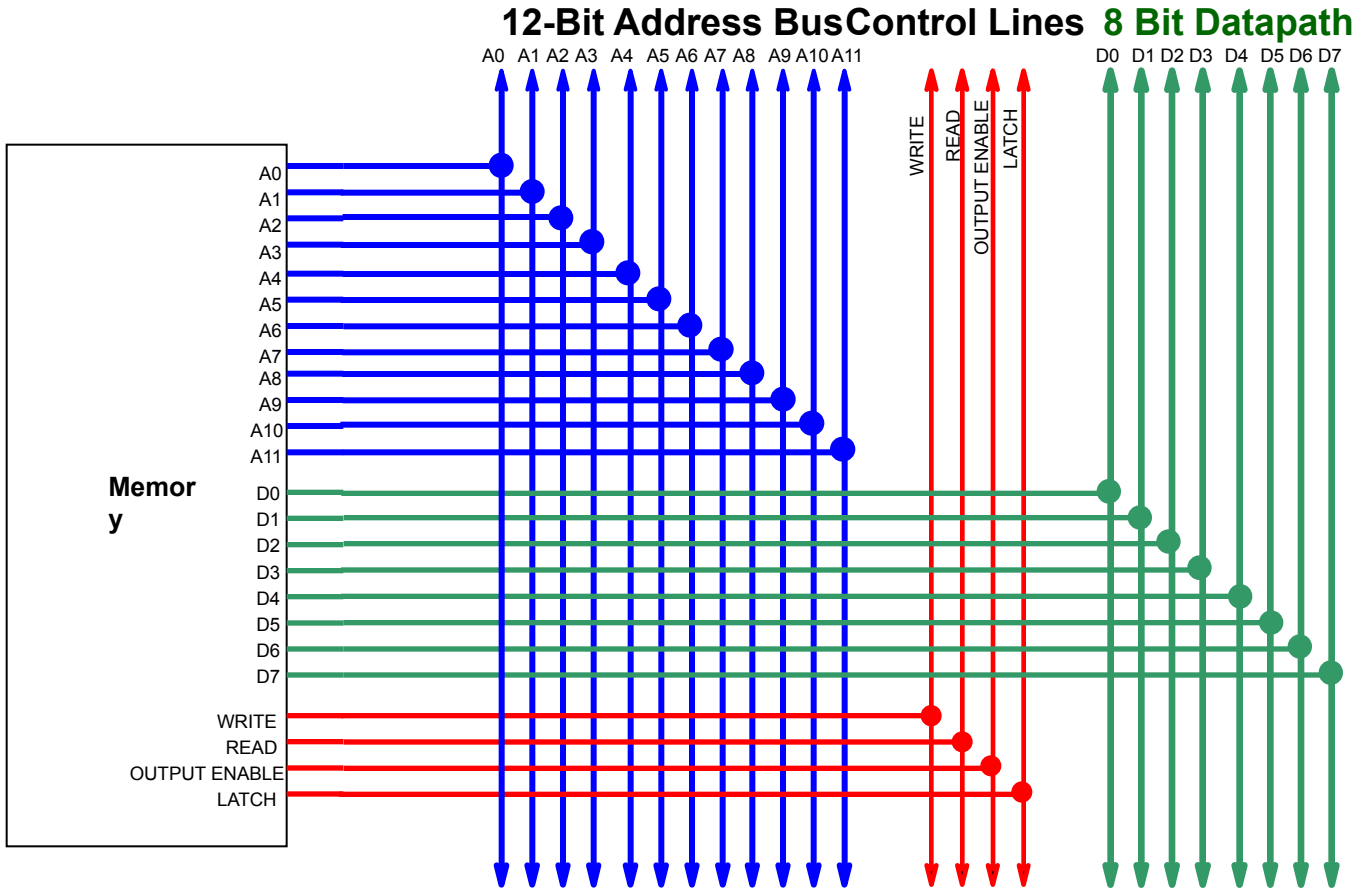
Memory Addressing (Revision)



Click to edit Master title style



Click to edit Master title style




KINGSTON
FURY
DDR4
BEAST

SWAPPING


FIRECUDA
540 Series
SEAGATE
PCIe Gen5 SSD

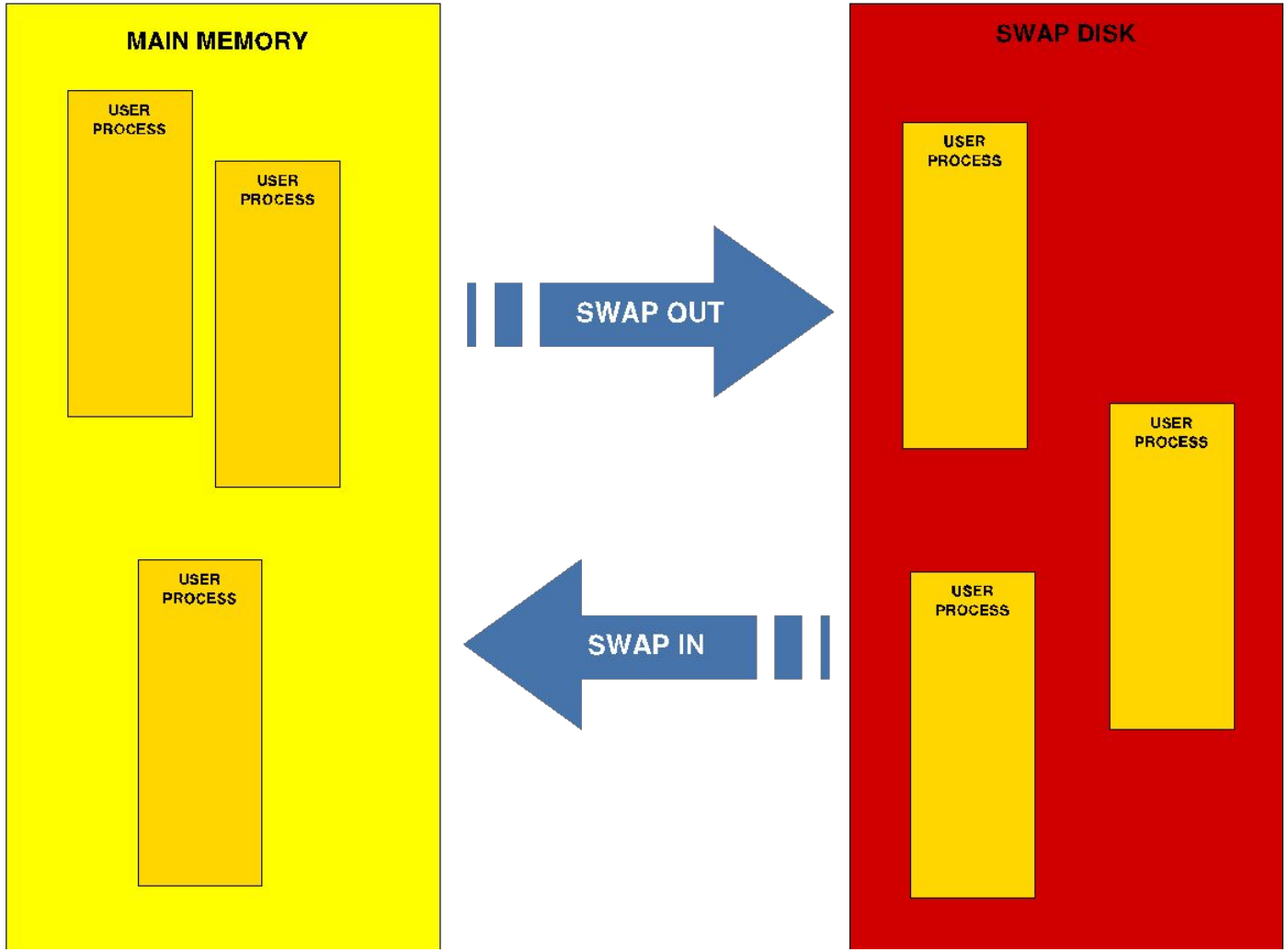
SWAPPING

MAIN MEMORY ↔ SECONDARY STORAGE

- **WHAT IS SWAPPING?**
  - Moving chunks of data between main memory and secondary (disk) storage.
  - Process data which is not currently needed can be moved out to the disk
  - Moved back into physical memory when needed
  - Often whole idle processes are swapped to disk storage – dedicated swap drives or partitions are used for this purpose
- **IS IT USEFUL?**
  - Allows more complete utilisation of main memory
  - But secondary storage is very **S L O W**
  - Need to avoid swapping too often.
  - **Thrashing:** a condition where the time spent of swapping between memory and disk exceeds time spent executing instructions.

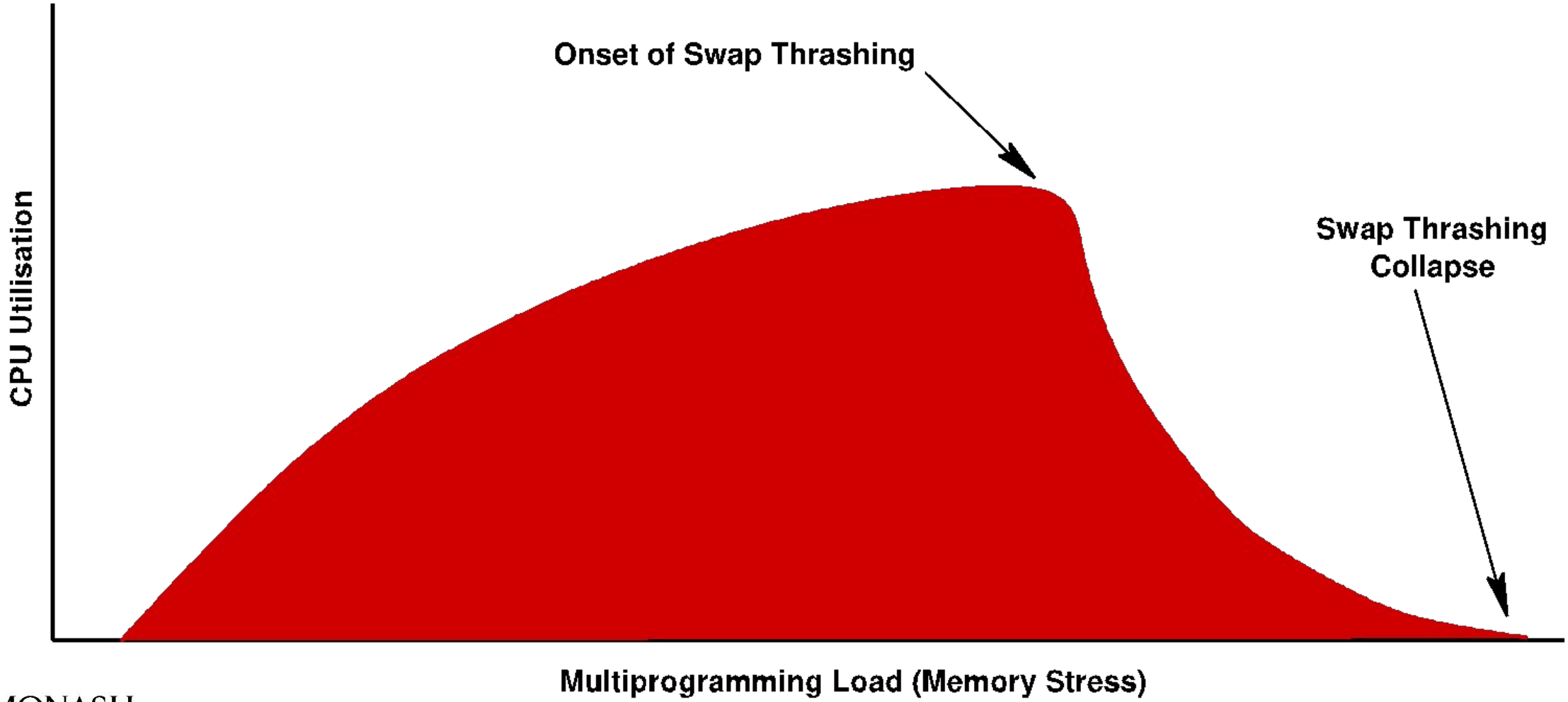
SWAPPING

MAIN MEMORY ↔ SECONDARY STORAGE



SWAP THRASHING

CPU CYCLES CONSUMED WITH SWAPPING EXCEED USEFUL WORK



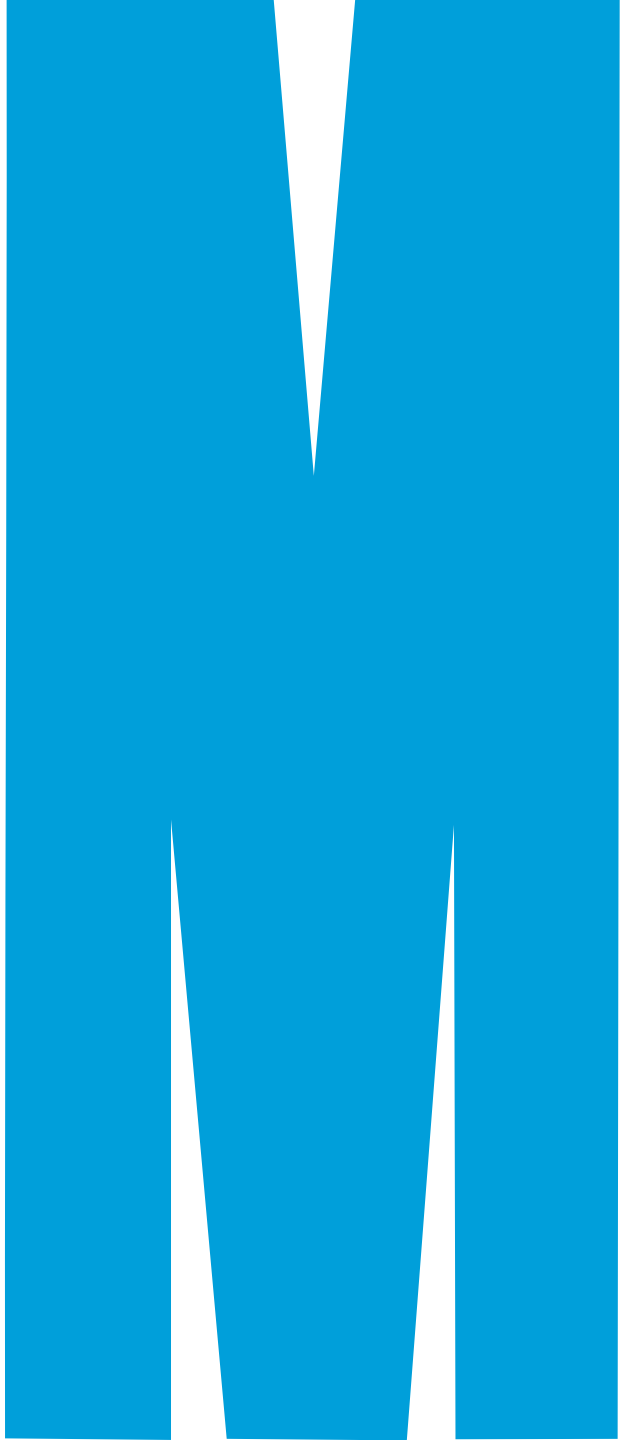
SWAPPING

MAIN MEMORY ↔ SECONDARY STORAGE

- **HOW BIG SHOULD A SWAP DRIVE BE?**
  - Decades ago, when DRAM was extremely expensive, the “rule of thumb” for sizing swap drives was to make them four or more times larger than the main memory in the host system.
  - This allowed fours times the number of resident processes to be held on the swap drive
  - Faster drives were favoured for use as swap drives due to the performance impact of swapping.
  - Contemporary practice is less disciplined as with ample main memory, swap is seldom used, especially with desktop systems

- **OTHER USES OF THE SWAP DRIVE?**
  - Applications that may need more space than available memory sometimes use swap space instead with performance penalties
  - Many *nix systems will use swap space to save a memory image during a crash

SHARED MEMORY

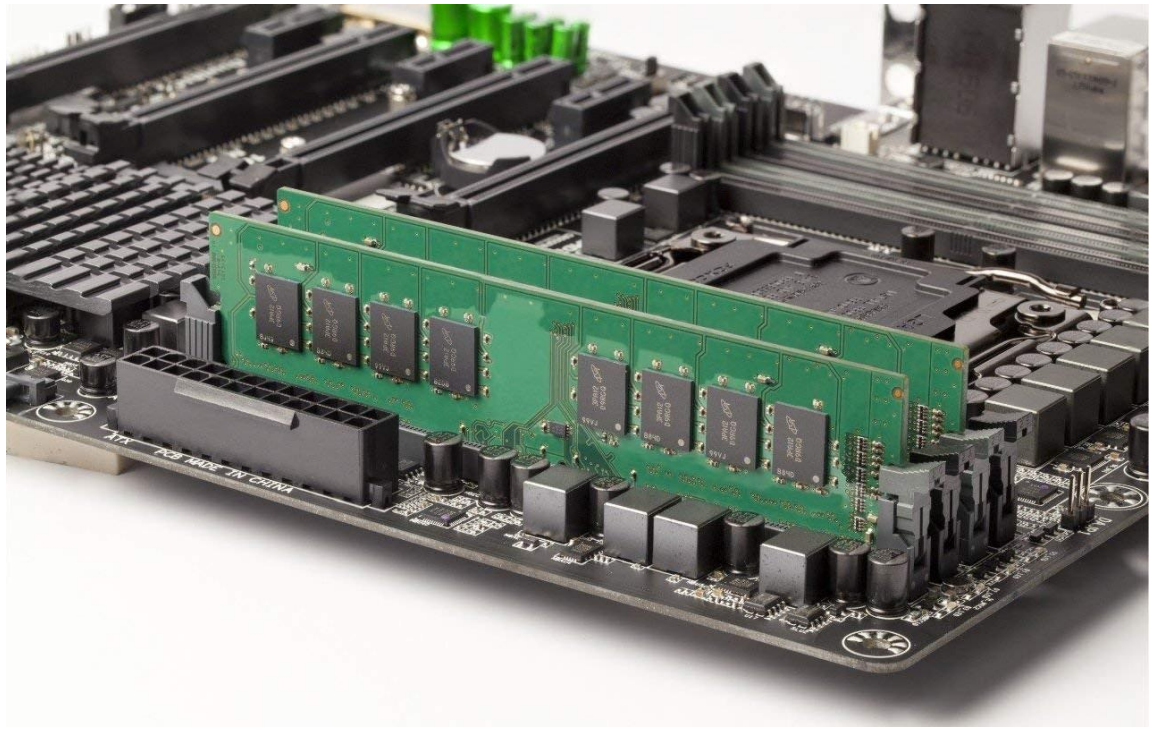


SHARED MEMORY

WHAT IS SHARED MEMORY

- **A segment in memory that is allocated to multiple processes**
  - A memory segment or group of consecutive pages may be shared between multiple processes
  - Like with multi-threading, multiple processes can then access each other's memory
  - Unlike with multi-threading, it does not happen automatically.
    - User must manually request a new section of shared memory to be allocated.
    - The required number of bytes must be specified.
    - User must manually assign a data structure to the shared block of bytes.
  - There is no concurrency protection offered. Mutual exclusion mechanisms like semaphores must be used.
  - Shared memory is an important tool for inter-process communication. Even unrelated processes write and read to and from share memory.
  - Compared to other mechanisms for inter-process communication, shared memory is very fast and has no further overhead once set up and running.

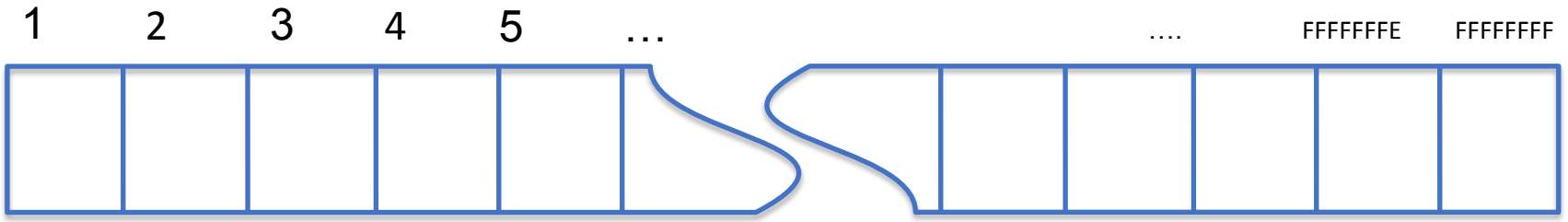
VIRTUAL MEMORY



VIRTUAL MEMORY

WHAT IS PHYSICAL MEMORY?

HOW DOES MAIN MEMORY APPEAR TO A PROGRAMMER?



A LONG, LINEAR LIST OF ADDRESSABLE WORDS OR BYTES.

VIRTUAL MEMORY

ABSTRACTION OF THE PHYSICAL MEMORY

- **LOGICAL (VIRTUAL) ADDRESSING**
  - Instead of dealing with physical memory addresses directly, the OS gives programs **logical addresses** to work with.
  - Logical memory addresses are then translated into **physical addresses**.

- **THE 'REAL' MEMORY ADDRESS IS HIDDEN FROM THE PROGRAM.**
  - The logical address is also known as a **virtual address**.
  - All memory address pointers you've worked with in C have been virtual addresses.

VIRTUAL MEMORY

ABSTRACTION OF THE PHYSICAL MEMORY

Nomenclature:

- *"Physical Memory"* is a term used to describe the memory hardware in the machine: for instance, 4096 Megabytes or 4 Gigabytes of DDR4 SDRAM chips.
- *"Virtual Memory"* (VM) is a term used to describe the memory model seen by the programmer, and also the hardware which implements it.
- It is important to recognise different usage of "Memory" between physical and virtual.
- A VM System is a combination of hardware and software which make the machine's physical memory and disk subsystems appear as the VM model.
- *In effect, the VM System emulates the behaviour of a cache, in the sense of hiding the large/slow disk behind the smaller/faster main memory, yet it also hides the complex behaviour of the disk hardware from the programmer.*

VIRTUAL MEMORY

WHY VIRTUALISE THE ADDRESS SPACE? (1/3)

EFFICIENCY, SECURITY, CONCURRENCY, FLEXIBILITY, …

- Efficient allocation of resources:
  - Main memory is a limited, expensive resource.
  - Buy more memory? But most of it will be unused most of the time…
  - Why not swap parts of memory out to secondary storage when not required?
    - Divide memory into either **pages** or **segments** that can be copied in and out…

    (Will be discussed later in this lecture)
- Make better use of available memory:
- More processes can be run at a time
- A process can be run even if larger than **all** of main memory.

VIRTUAL MEMORY

WHY VIRTUALISE THE ADDRESS SPACE? (2/3)

EFFICIENCY, SECURITY, CONCURRENCY, FLEXIBILITY, …

- Enables protection against unauthorised access to another process's memory space
- Enables **shared memory** among related processes
  - A segment of memory can be 'mapped' into the memory space of multiple related processes
  - e.g. multiple processes can share certain variables
  - Multiple instances of the same program can share code.

- e.g. When **forking** or running multiple instances of the **same program**, the program code only needs to be loaded once.
- Each process sees its own **logical** copy of the program code, but the logical segments of memory are translated to a common physical location.
- This is why you can't modify hard-coded string literals in C. They are part of the program code itself, which may be mapped into multiple instances' memory spaces.

```
char* str = "Hi!";
str[0] = 'h';
//segmentation fault!
```

VIRTUAL MEMORY

WHY VIRTUALISE THE ADDRESS SPACE? (3/3)

EFFICIENCY, SECURITY, CONCURRENCY, FLEXIBILITY, …

- **Relocation:**
  - If a program must be copied out to secondary storage, and is copied back into main memory later on…
  - The same location in main memory might not be available.
    - Might be relocated to a different part of physical memory.
  - Relocating a program means the physical memory addresses are all different.
  - All the program's instructions and variables now have different physical memory addresses.
  - Dealing with logical addresses rather than physical addresses means the program's pointers remain valid
  - The program can continue to run normally regardless of where it is located in physical memory.
    - Program does not depend on being loaded into a particular location.

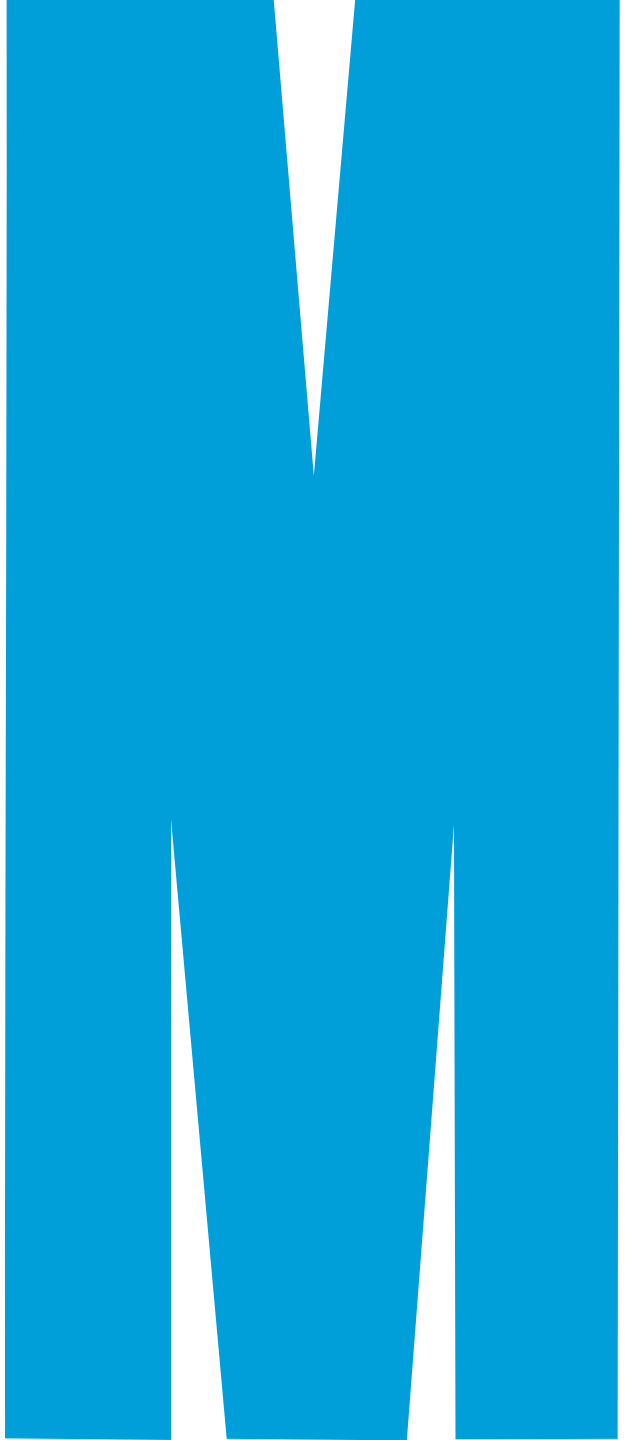
VIRTUAL MEMORY

WHY VIRTUALISE THE ADDRESS SPACE?

Key Points:

1. *Provide a large address space for programmer, hiding the limitations of the machine's main memory.*
2. *Provide a caching mechanism to hide the performance limitations of the disk hardware.*
3. *Provide a means of hiding the complexity of the disk and I/O hardware behaviour from the programmer.*

VIRTUAL MEMORY ALLOCATION



VIRTUAL MEMORY

IMPLEMENTING VM

- In a VM System we have two types of addresses.
- A *Physical Address* is the binary address which is used to directly access Main Memory and I/O Controllers.
- A *Virtual Address* is the address seen by the CPU, and thus the programmer.
- For the CPU to access memory and I/O, the *Virtual Address* must be *translated* into a *Physical Address*.
- Hardware, specifically a Memory Management Unit (MMU) usually inside the CPU is used to translate a *Virtual Address* into a *Physical Address*, and manage this mapping information

VIRTUAL MEMORY ALLOCATION

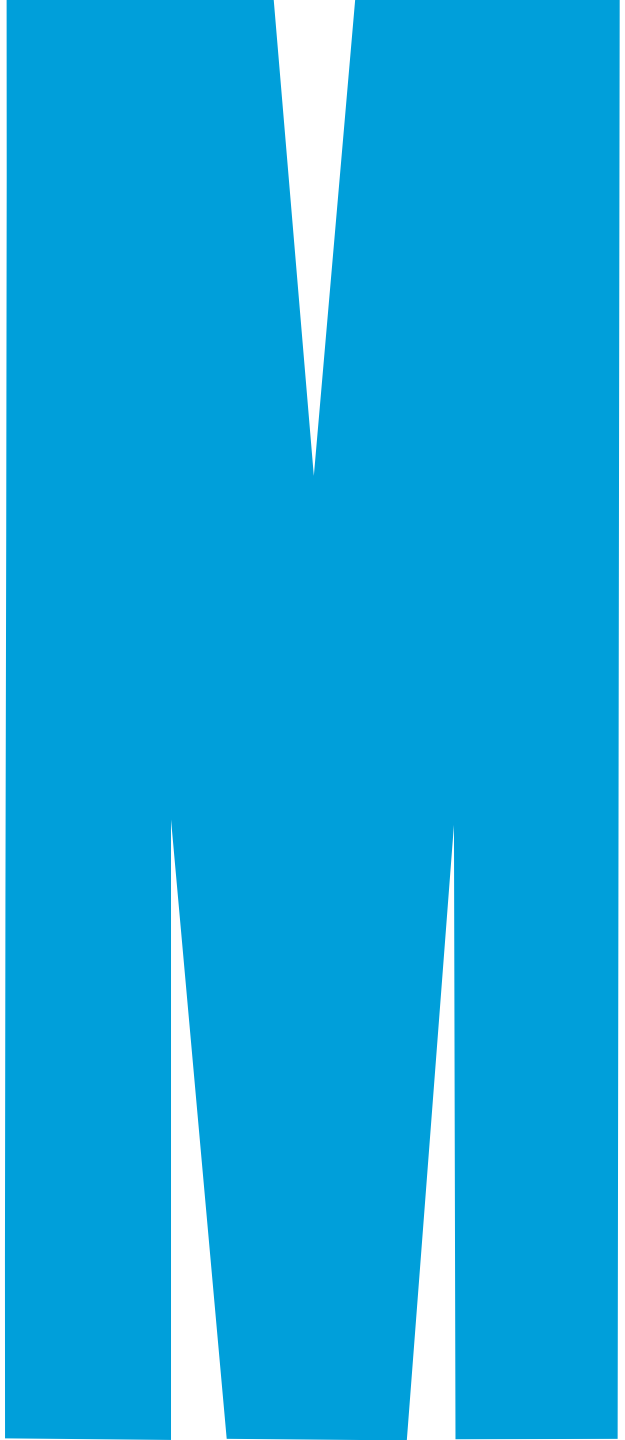
TWO APPROACHES

- **PAGING or PAGED VIRTUAL MEMORY**
  - Memory is **partitioned** into **fixed-sized** chunks

- **SEGMENTATION or SEGMENTED VIRTUAL MEMORY**
  - Memory is managed as multiple **segments** of **different sizes**

- **HYBRID VIRTUAL MEMORY**
  - Combining both PAGED and SEGMENTED VIRTUAL MEMORY is also possible. Many Intel CPUs employ this approach

BUFFER CACHES



VIRTUAL MEMORY

BUFFER CACHES

- Our discussion of VM has so far been focused on dealing with processes and managing movement of whole processes, or parts of processes, between main memory and disks
- Executable code and shared libraries are frequently used by multiple processes, often using the shared memory functionality to make them simultaneously accessible to many processes - disk blocks that store such items are therefore frequently accessed
- Many operating systems include what is termed a "*buffer cache*" to keep copies of such disk blocks resident in memory long term as a cache to improve speed
- A *buffer cache* may be a reserved area of physical memory used to hold these blocks, or as in BSD based systems, a scheme to use all free memory not being used for processes to buffer frequently accessed disk blocks
- *A well designed buffer cache can significantly increase disk I/O performance*

PAGED VIRTUAL MEMORY

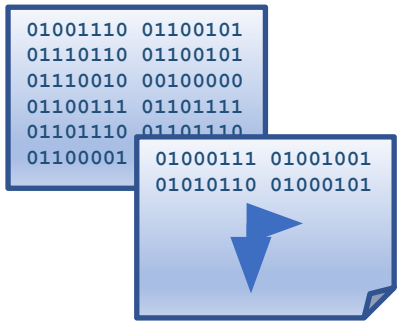
PAGED VIRTUAL MEMORY

FRAMES AND PAGES

- **PAGES FIT INTO FRAMES**
  - **Physical memory** is partitioned into equal sized **frames.**
  - A **page** is a fixed size chunk of data that can fit into a frame
  - All frames are of equal size, and a page is the same size as a frame.
  - Some pages might be in physical memory (loaded into frames)
  - Other pages might be **paged out** to secondary storage (taken out of frames)
    - To make way for other pages to be paged in.
  - The OS kernel maintains a **page table** to keep track of which pages are in which frames.

PAGED VIRTUAL MEMORY

FRAGMENTATION



❒ INTERNAL FRAGMENTATION

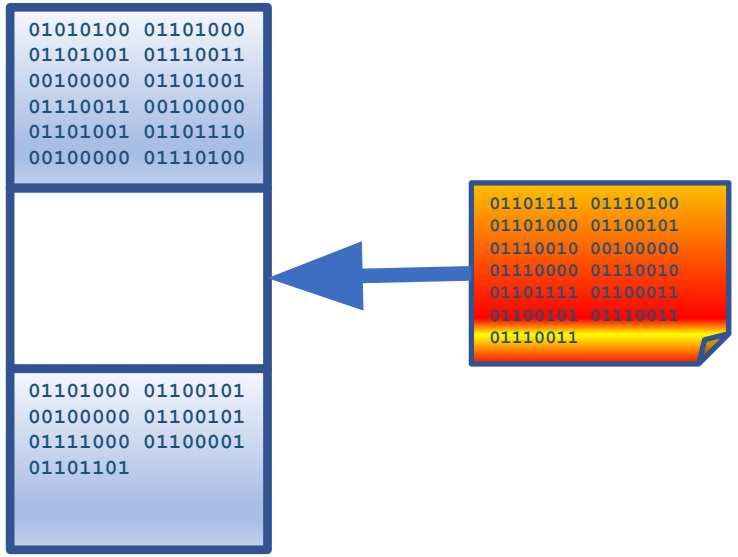
- A process may occupy **one or more** frames in memory.
- If an entire page is not needed, the space cannot be used by another process
- Since all pages are fixed size, any excess space is wasted.

❒ NO EXTERNAL FRAGMENTATION

- Can space **between** page frames be wasted?
- No. All pages are equally sized, so any empty frame in memory can **always** be allocated a whole page.

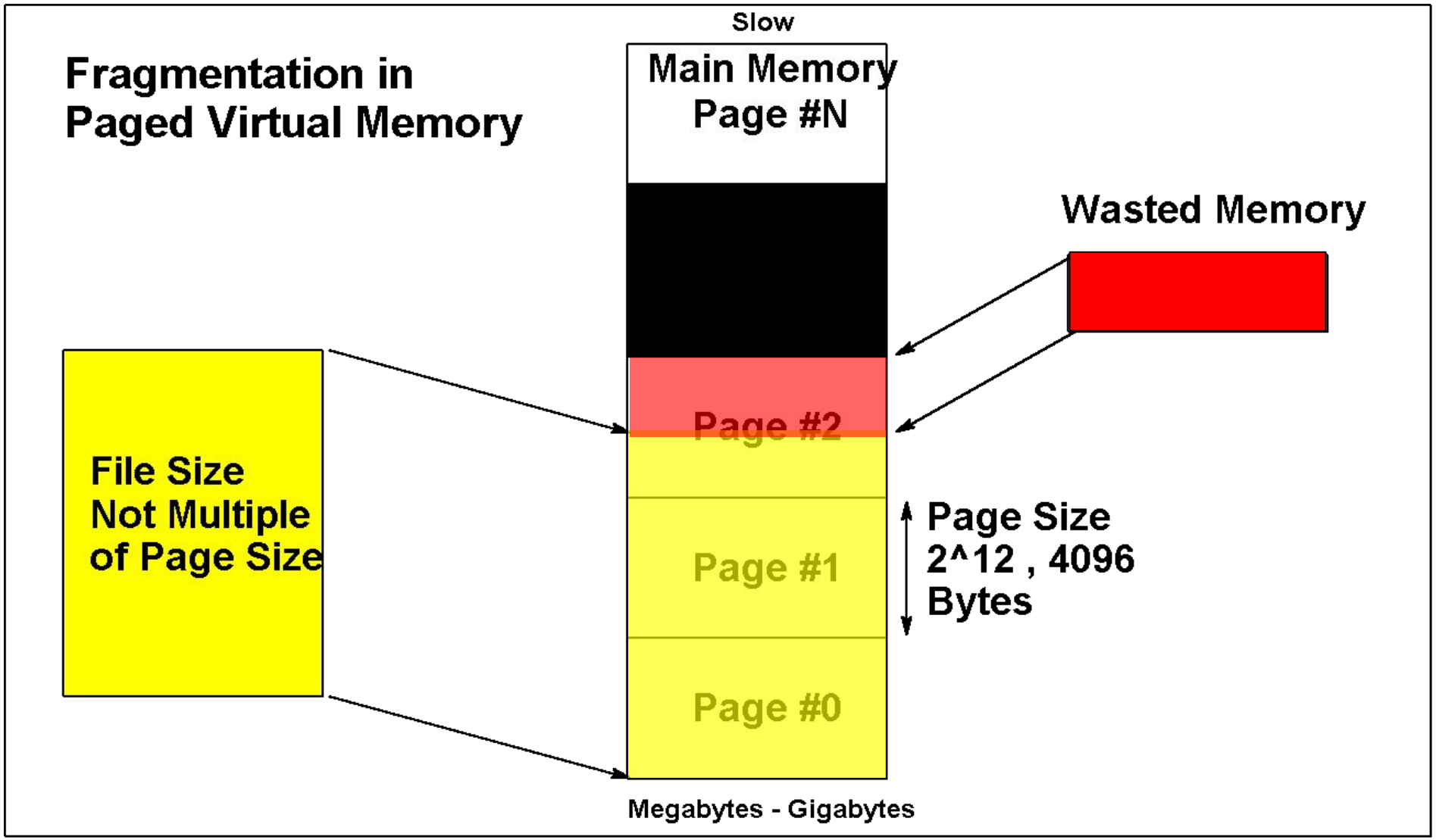
  ⇒ no external fragmentation

❒ *…but this is not true for virtual memory systems that use **segmentation** (see later)*



MONASH University

PAGED VIRTUAL MEMORY



PAGED VIRTUAL MEMORY

FRAGMENTATION

- **INTERNAL FRAGMENTATION**
  - Consider what happens when you page into memory a whole program or a data file.
  - It is unlikely that either will have a size which is an exact multiple of the size of a page.
  - Therefore only part of the last page will be filled with bytes of program or data. The remaining memory space is wasted.
  - Consider a situation where every file paged into memory is hypothetically smaller than a page in size.
  - How much space is wasted on average?

- On average, 50% of memory is wasted in pages where files do not end on a page boundary.
- This effect is termed “internal fragmentation”.

PAGED VIRTUAL MEMORY

FRAGMENTATION

- Fragmentation reduces the efficiency of memory usage on a VM system.
- This problem will arise in any paged VM system.
- *Are there alternatives?*
- *Can we share pages between programs ?*
- *No - because the VM system is usually employed to improve security by segregating the VA spaces of different users on the system. Sharing pages defeats the security mechanism.*
- *Can we make the pages smaller ?*
- *In principle yes, but the smaller the page size, the bigger the required Page Table sizes, and MMU complexity. In practice the trend is toward bigger page sizes since MMUs are very expensive.*

PAGED VIRTUAL MEMORY

LOGICAL ADDRESSES

- The memory address contains a **page number** and an **offset**

Example:

```
00000000001010000000000000010000
Page number 10    Offset 16 bytes
```

- The most significant bits in the **logical address** are used for the page number. The remaining bits are an **offset** from the start of the page (i.e. position inside the page)
- The address is structured so the **number of offset bits** matches the addressable range of the **page size**. Different systems may use a different page size.
- It is not possible to specify an offset that goes ‘out of bounds’ of the specified page.

PAGED VIRTUAL MEMORY

LOGICAL ADDRESSES

- The simplest way to map a VA into a PA is to use a table, in which every entry represents a VA->PA mapping.
- Each table entry includes a bit which indicates whether the location is in main memory or on the disk.
- We can then use the VA to find the PA, and determine whether it is in memory or on disk.
- **Question**: *What is wrong with this scheme?*

- Consider that the table entry to address one word in main memory must be big enough to hold at least that address.
- Since the address is similar in size to a word (e.g. 32-bit word/24-bit address), the table must be similar in size to the memory itself!
- Is it practical to make a translation table of similar size to the main memory? The answer is obviously - NO.
- **Question**: *how can we organise the table of mappings to be of a reasonable size?*

PAGED VIRTUAL MEMORY

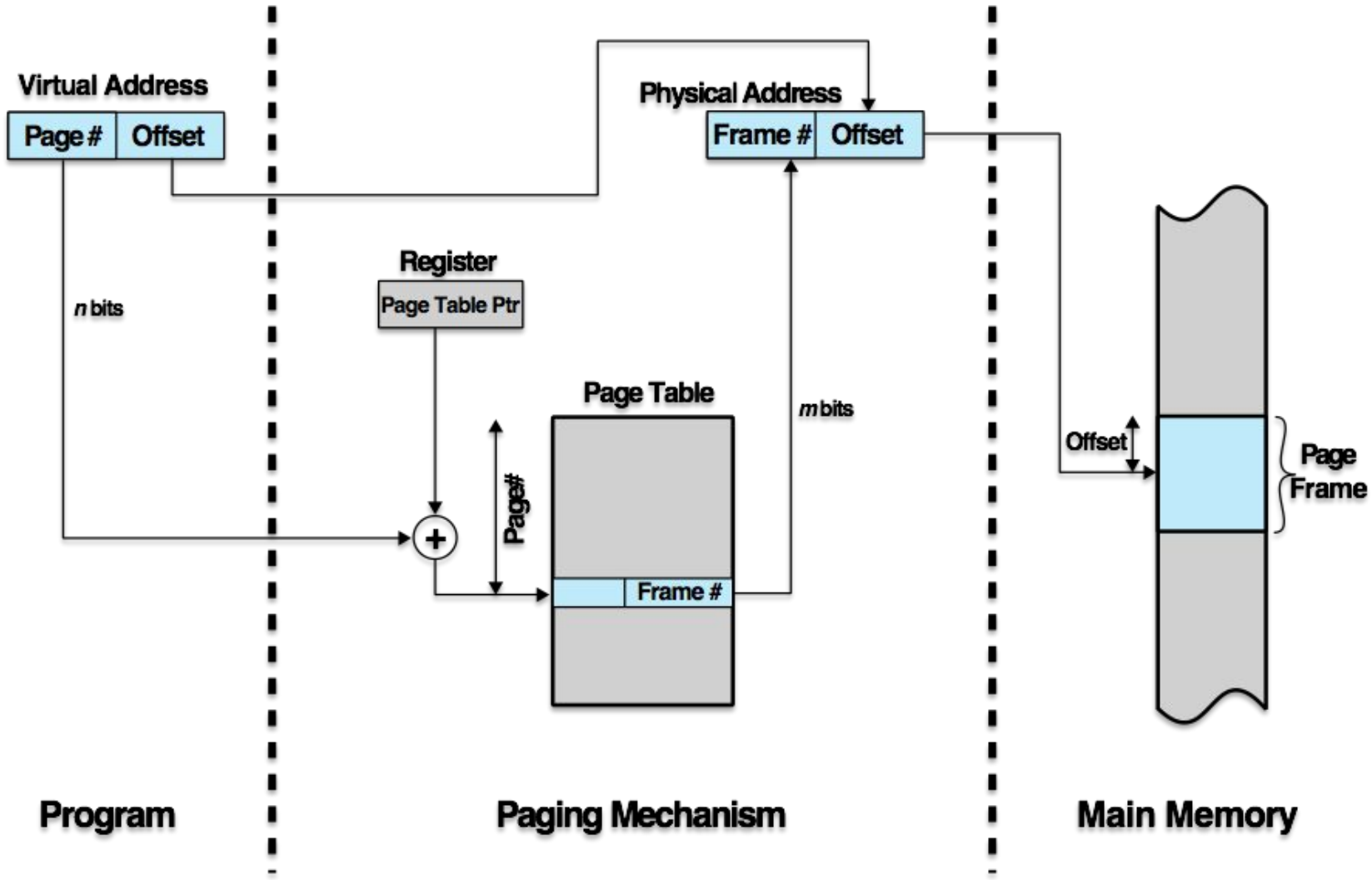
LOGICAL ADDRESSES

- What happens if we divide the main memory into equally sized blocks (termed “pages”, and usually the same size as a disk block) ?
- The physical address can then be divided into two parts - the upper bits which address the page, and the lower bits which address the word inside the page.
- This means every word in a particular page shares the same upper address bits, and therefore the same VA->PA mapping.
- The size of the table which contains the mappings need only be large enough to contain one entry for every page in the main memory.
- A table which maps the upper address bits of the VA into the address which finds a page in main memory is termed a page table.
- A page table is still too large to fit into a register file - therefore it must reside in some reserved part of the main memory.
- **Question**: *What problem does this cause?*
- The machine must perform at least one extra access to memory, for every intended access to memory => VERY SLOW.
- *In practice, MMUs will cache the mapping of the Virtual Address to Physical Address for speed*

PAGED VIRTUAL MEMORY

ADDRESS TRANSLATION (PAGING)

Change (logical) page number to (physical) frame number **physical address.**



PAGED VIRTUAL MEMORY

WHAT IS PAGE FAULT?

- **WHAT HAPPENS IF A PAGE DOES NOT HAVE A FRAME IN THE PAGE TABLE?**
  - It means the physical address of page is not resident in main memory

    'PAGE FAULT'!

  - The requested **logical** memory address is **valid**, but the page is in **secondary storage**. The page must be loaded into a frame in main memory.

- **WHAT PROBLEMS DOES A PAGE FAULT CAUSE?**
  - If all the frames are already full, another page needs to be paged out of main memory to make room for the needed one.
  - We need a 'page replacement algorithm' to determine which page should be paged out to make way for the page that is needed in main memory.
  - **NOTE**: even if there **is** an empty frame to load the needed page into, a **page fault** still occurs if the page was not found in physical memory.

PAGE REPLACEMENT ALGORITHMS

PAGE REPLACEMENT ALGORITHMS

WHY NEEDED?

❒ **PAGE FAULT == PAGE NOT FOUND IN PHYSICAL MEMORY**

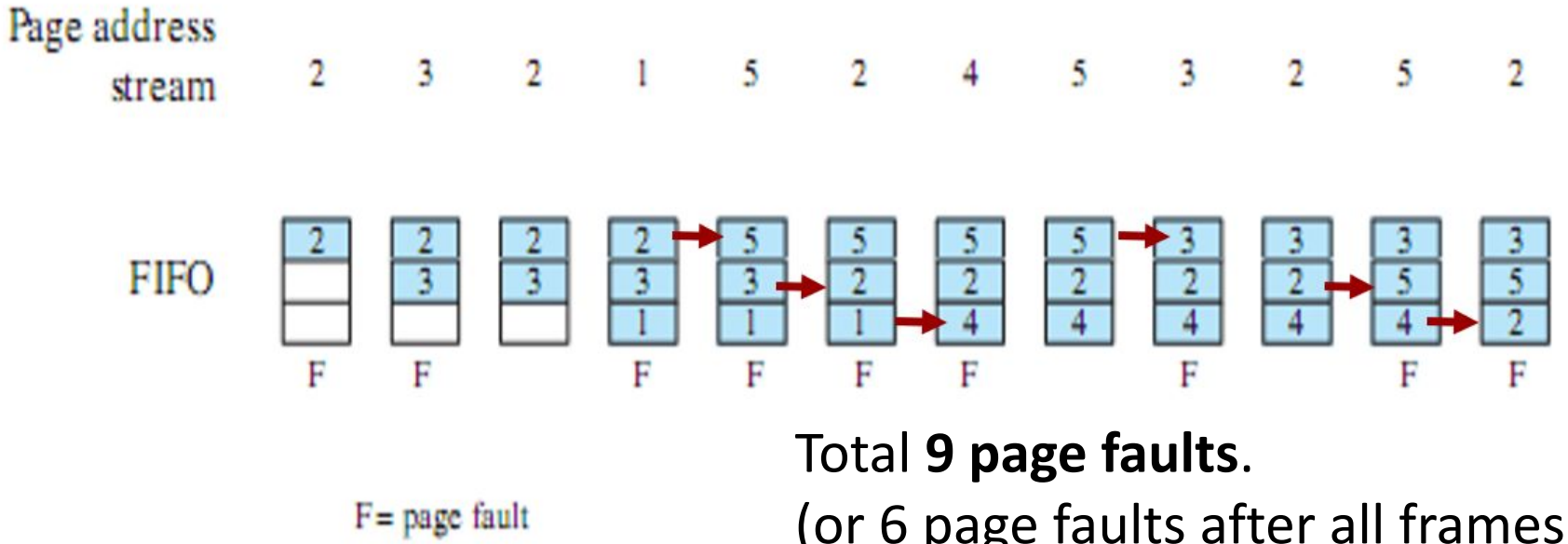
- If we get a page fault, we need to load a page from disk into a frame in main memory.
- What if all the frames are already full?
- Need a strategy for knowing which page to page out.
- **INTENT**:
  - Page out a page that won't be needed again for a long time
    - ➢ PREVENT **THRASHING**
  - Try to make a 'fast guess'. Don't try to be 'too clever'.
    - ➢ MORE ACCURATE PREDICTION → MORE COMPUTATION TIME NEEDED
    - ➢ LESS ACCURATE PREDICTION → MORE SLOW PAGING OPERATIONS NEEDED
    - ➢ TRY TO FIND A BALANCE.

PAGE REPLACEMENT ALGORITHMS

FIRST IN FIRST OUT (FIFO)

- Simplest page replacement policy to implement
- Page that has been in memory the longest is replaced
- Treat frames allocated to processes as a ‘queue’ for replacement
- Pages are removed in round-robin style

EXAMPLE



Total **9 page faults**.
(or 6 page faults after all frames filled)

PAGE REPLACEMENT ALGORITHMS

SECOND CHANCE (MODIFIED FIRST IN FIRST OUT) ALGORITHM

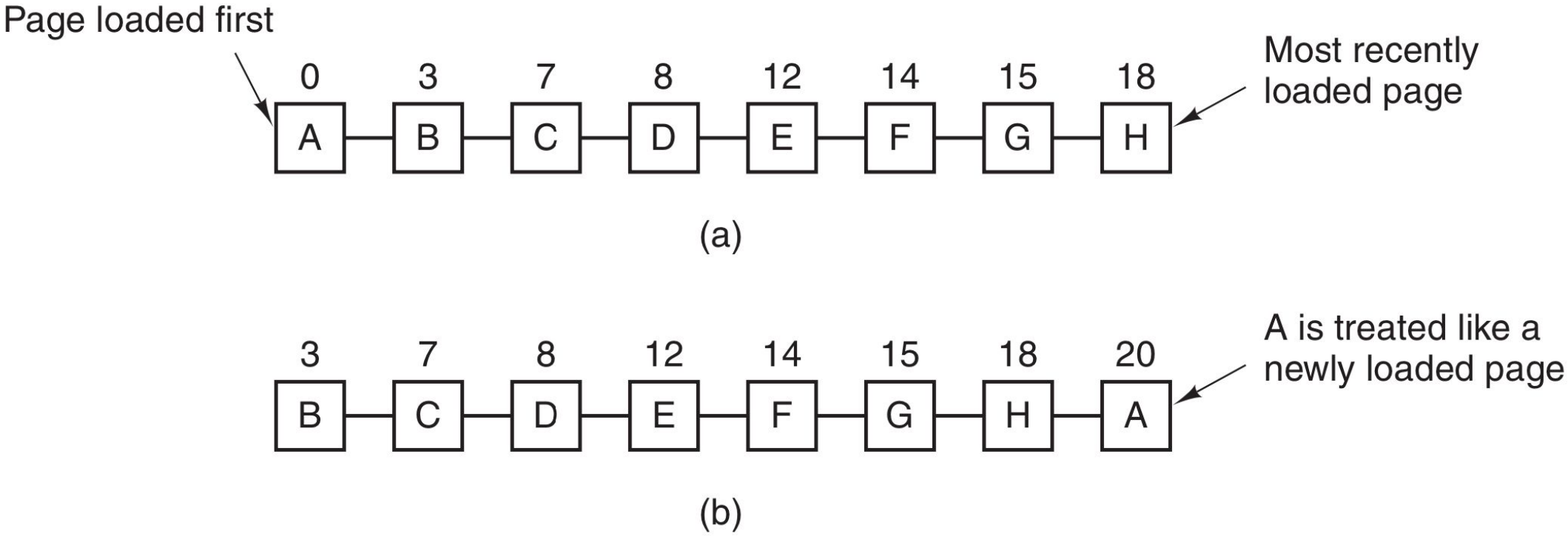


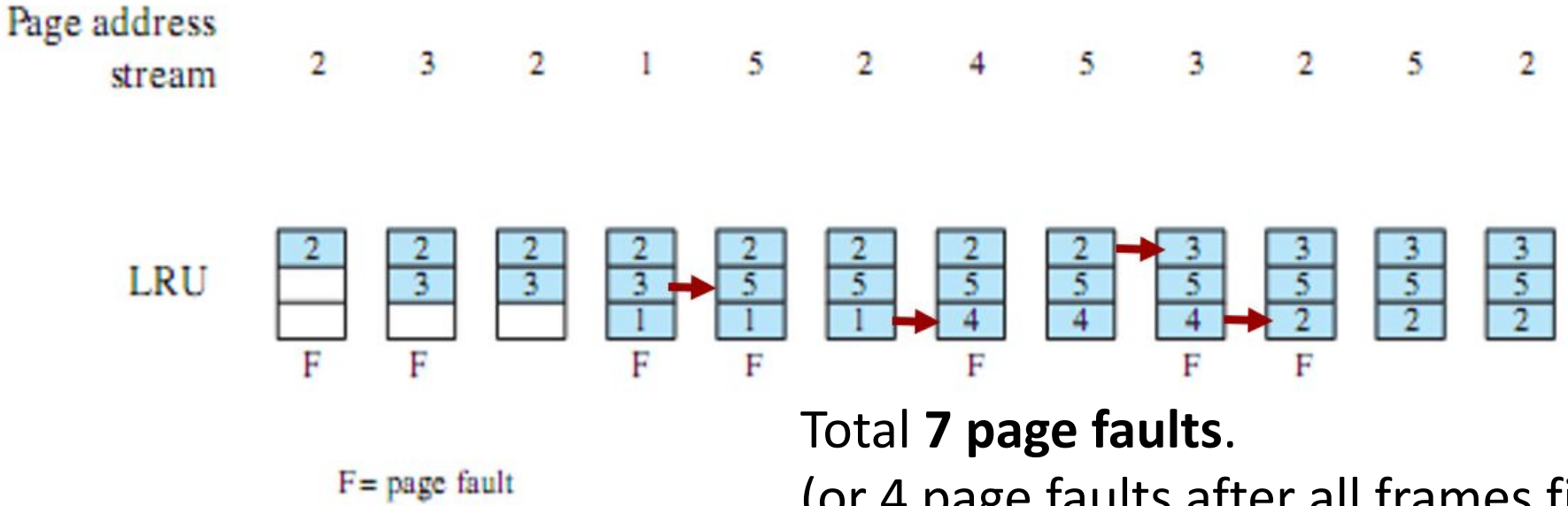
Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

PAGE REPLACEMENT ALGORITHMS

LEAST RECENTLY USED (LRU)

- Difficult to implement
- Replace the page that has **not been referenced** for the **longest time** by the principle of locality, pages that have been referenced recently are likely to be referenced again in near future.
- A lot of overhead to maintain/check the time since last reference for every page.

EXAMPLE



Total **7 page faults**.
(or 4 page faults after all frames filled)

PAGE REPLACEMENT ALGORITHMS

LEAST RECENTLY USED (LRU)

- LRU can be implemented with special hardware.
- Let us consider the simplest way first.
- This method requires equipping the hardware with a 64-bit counter, C, that is automatically incremented after each instruction.
- Furthermore, each page table entry must also have a field large enough to contain the counter.
- After each memory reference, the current value of C is stored in the page table entry for the page just referenced.
- When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one.
- That page is the least recently used.
- Linux employs a variant of the NFU algorithm, intended to approximate the LRU algorithm

PAGE REPLACEMENT ALGORITHMS

NOT FREQUENTLY USED (NFU)

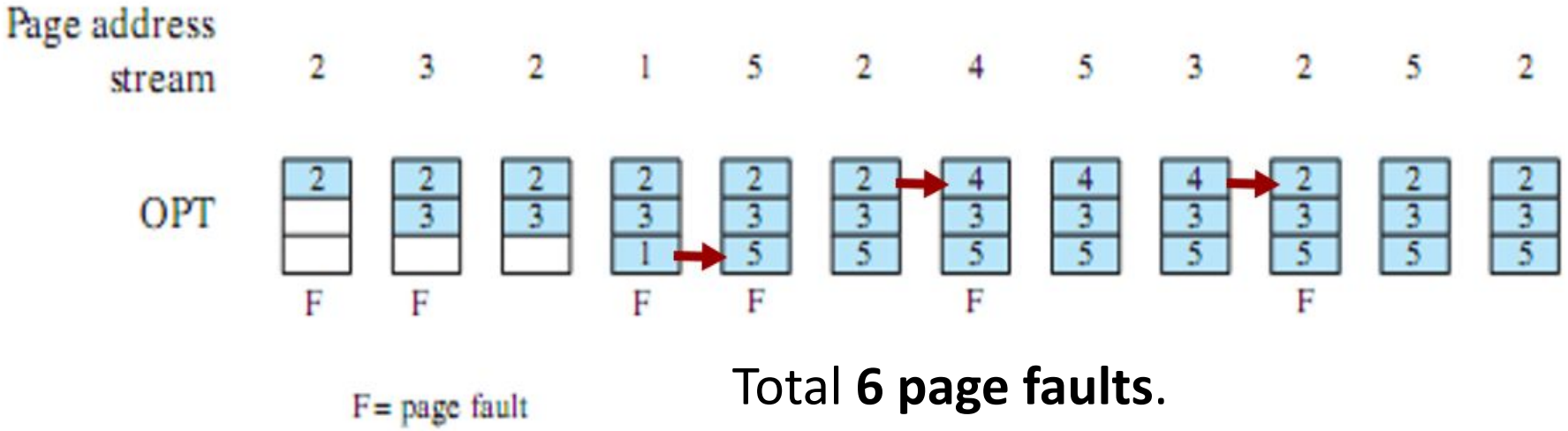
- NFU requires a software counter associated with each page, initially zero.
- At each clock interrupt, the operating system scans all the pages in memory.
- For each page, the R bit, which is 0 or 1, is added to the counter. The counters roughly keep track of how often each page has been referenced.
- When a page fault occurs, the page with the lowest counter is chosen for replacement.
- The main problem with NFU is that it never forgets anything.
- For example, in a multipass compiler, pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass-1 pages.
- Consequently, the operating system will remove useful pages instead of pages no longer in use.

PAGE REPLACEMENT ALGORITHMS

OPTIMAL POLICY (OPT)

- Theoretical **best case**
- Replace the page that **will not be referenced** for **the longest time**
- Produces the **smallest** possible number of page faults.
- **Impossible** to implement in practice: the operating system cannot see the future. If we knew everything the process would do in the **future**, we would not need to run the program!

EXAMPLE



Total **6 page faults**.
(or 3 page faults after all frames filled)

MONASH University

PAGE REPLACEMENT ALGORITHMS

SUMMARY

PERFORMANCE

	PAGE FAULTS	COMPLEXITY
FIFO	9	Simplest
LRU	7	Complex
OPT	6	'Impossible'

PAGE REPLACEMENT ALGORITHMS ARE NOT PERFECT

- Page replacement is always a trade-off between **minimising** page faults and **minimising** computational complexity in executing the algorithm.
- Page replacement can happen **very often** (whenever data is paged in). Identifying a page to replace must be done in **as few instructions as possible** while still giving a 'good enough' result.
    - So that the program's instructions can be executed instead!

PAGE REPLACEMENT ALGORITHMS

ALTERNATE ALGORITHMS (TANENBAUM)

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Page replacement algorithms vary widely in performance and computational complexity. Good choices are essential where memory is stressed in a host system.

Summary

- So far we have discussed
  - Paged VM.
  - Four different page-replacement algorithms.
  - Swapping
  - Shared memory
  - Why virtualised memory is useful
  - Paging and segmentation
  - Buffer caches
- Next week
  - More on Memory Management
  - Inter Process Communication
- Reading
  - Stallings, Chapter 7 & 8 (7th Edition)
  - Tanenbaum, Chapter 3