# Week 6
# Memory Management

IN 1011 Operating Systems

## Memory Management

- Background and Introduction
- Memory Abstraction
- Address Space
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement

## Memory Management

- Background and Introduction
- Memory Abstraction
- Address Space
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement

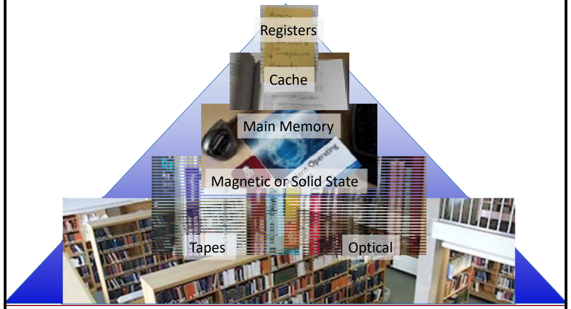
## Memory Wish List

- Memory that is:
  - Private
  - Infinitely large
  - Infinitely fast
  - Non-volatile
  - Inexpensive

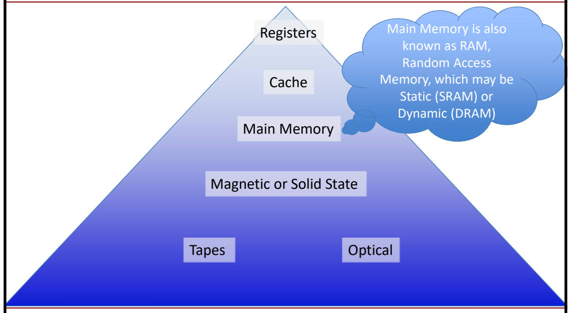
Not really possible …

## Think of memory as of information (in a library)

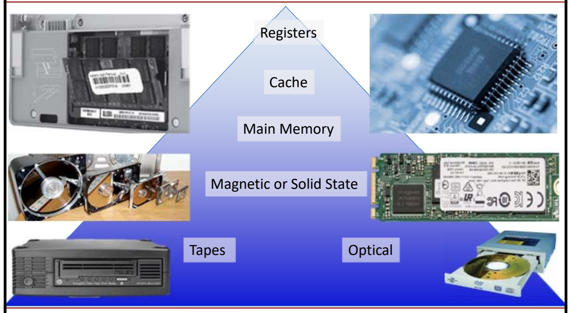
## Think of memory as of information (in a library)



## Memory Hierarchy


Registers
Cache
Main Memory
Magnetic or Solid State
Tapes
Optical
Main Memory is also known as RAM, Random Access Memory, which may be Static (SRAM) or Dynamic (DRAM)

## Memory Hierarchy



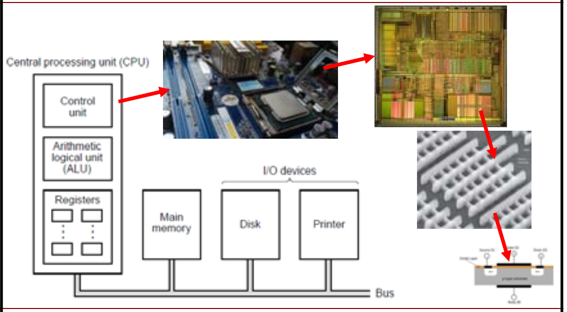
## Memory Hierarchy

- Programme must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers

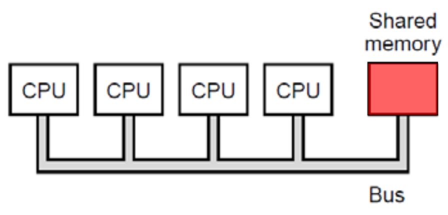
## SRAM / DRAM / Disks

| Memory technology | Typical access time [ns] | $ per GB in 2004 Why is this so expensive??? |
|---|---|---|
| SRAM | 0.5-5 | $4000-$10,000 |
| DRAM | 50-70 | $100-$ 200 |
| Magnetic disk | 5,000,000-20,000,000 | $0.50-$2 |

## Processors



## Multiprocessors



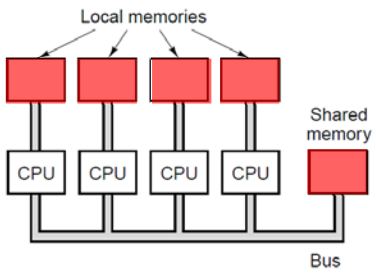
## Principle of locality

- The principle of locality states that programs access *a relatively small portion* of their address space at any instant of time, (like a library's collection).
- There are two different types of locality:
  - **Temporal locality** (locality in time): If an item is referenced, it will tend to be referenced again soon
  - **Spatial locality** (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon. (books on the same shelf).

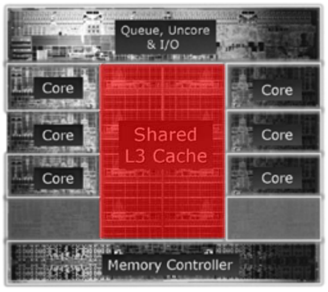
## Taking Advantage of Locality

- Store **everything** somewhere where it is cheap although slow to access, e.g. a **disk**
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory (**Main memory**)
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory (**Cache memory attached to CPU**)

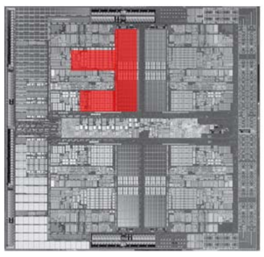
## Multiprocessors

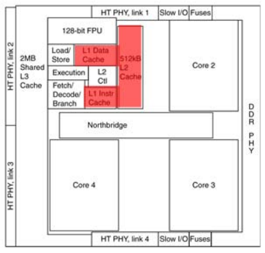


## x86 Architecture

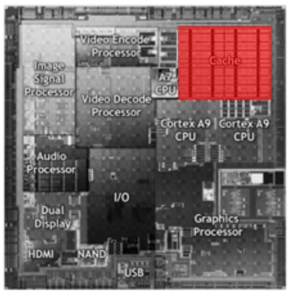


## AMD Barcelona: 4 processor cores

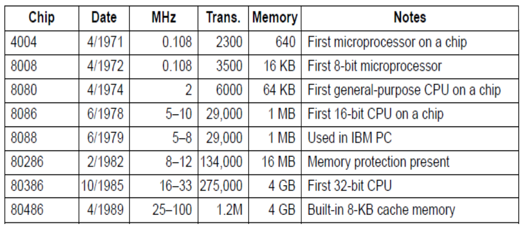




## ARM Architecture



## x86 Architecture

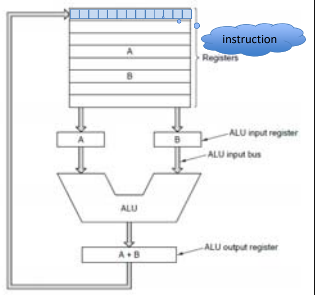


| Chip | Date | MHz | Trans. | Memory | Notes |
|---|---|---|---|---|---|
| 4004 | 4/1971 | 0.108 | 2300 | 640 | First microprocessor on a chip |
| 8008 | 4/1972 | 0.108 | 3500 | 16 KB | First 8-bit microprocessor |
| 8080 | 4/1974 | 2 | 6000 | 64 KB | First general-purpose CPU on a chip |
| 8086 | 6/1978 | 5–10 | 29,000 | 1 MB | First 16-bit CPU on a chip |
| 8088 | 6/1979 | 5–8 | 29,000 | 1 MB | Used in IBM PC |
| 80286 | 2/1982 | 8–12 | 134,000 | 16 MB | Memory protection present |
| 80386 | 10/1985 | 16–33 | 275,000 | 4 GB | First 32-bit CPU |
| 80486 | 4/1989 | 25–100 | 1.2M | 4 GB | Built-in 8-KB cache memory |

Key members of the Intel CPU family.

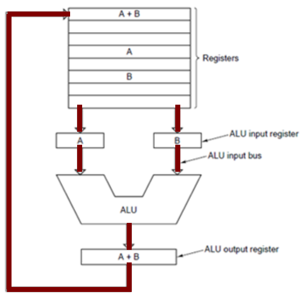
## CPU Organization

- Processor
- Memory
- Data Path



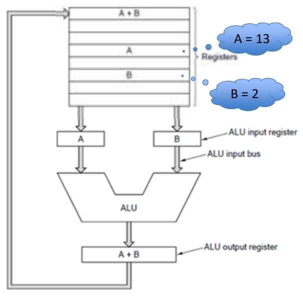
## CPU Organization

- Processor
- Memory
- Data Path
  - Bus



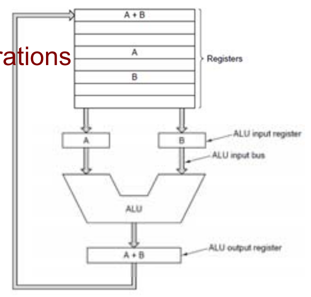
## CPU Organization

- Processor
- Memory
  - Registers (data
- Data Path

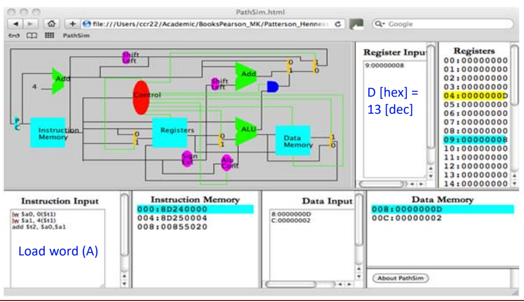


## CPU Organization

- Processor
  - Arithmetic Operations
- Memory
- Data Path



## Example: add 13 + 2



Register Input

```
9:00000008
```

D [hex] =
13 [dec]

Registers

```
00:00000000
01:00000000
02:00000000
03:00000000
04:0000000D
05:00000000
06:00000000
07:00000000
08:00000000
09:00000008
10:00000000
11:00000000
12:00000000
13:00000000
14:00000000
```

Instruction Input

```
lw $a0, 0($t1)
lw $a1, 4($t1)
add $t2, $a0,$a1
```

Load word (A)

Instruction Memory

```
000:8D240000
004:8D250004
008:00855020
```

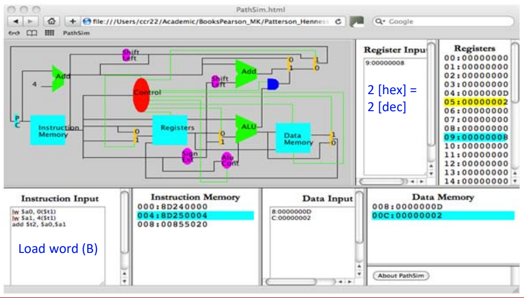
Data Input

```
8:0000000D
C:00000002
```

Data Memory

```
008:0000000D
00C:00000002
```

## Example: add 13 + 2



Register Input
```
9:00000008
```
2 [hex] =
2 [dec]

Registers
```
00:00000000
01:00000000
02:00000000
03:00000000
04:0000000D
05:00000002
06:00000000
07:00000000
08:00000000
09:00000008
10:00000000
11:00000000
12:00000000
13:00000000
14:00000000
```

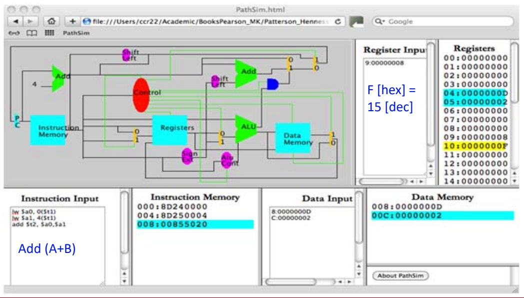
Instruction Input
```
lw $a0, 0($t1)
lw $a1, 4($t1)
add $t2, $a0,$a1
```
Load word (B)

Instruction Memory
```
000:8D240000
004:8D250004
008:00855020
```

Data Input
```
8:0000000D
C:00000002
```

Data Memory
```
008:0000000D
00C:00000002
```

## Example: add 13 + 2



Register Input
```
9:00000008
```
F [hex] =
15 [dec]

Registers
```
00:00000000
01:00000000
02:00000000
03:00000000
04:0000000D
05:00000002
06:00000000
07:00000000
08:00000000
09:00000008
10:0000000F
11:00000000
12:00000000
13:00000000
14:00000000
```

Instruction Input
```
lw $a0, 0($t1)
lw $a1, 4($t1)
add $t2, $a0,$a1
```
Add (A+B)

Instruction Memory
```
000:8D240000
004:8D250004
008:00855020
```

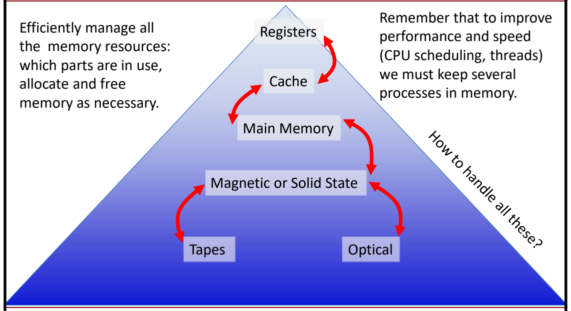
Data Input
```
8:0000000D
C:00000002
```

Data Memory
```
008:0000000D
00C:00000002
```

## Memory manager

Efficiently manage all the memory resources: which parts are in use, allocate and free memory as necessary.

Remember that to improve performance and speed (CPU scheduling, threads) we must keep several processes in memory.

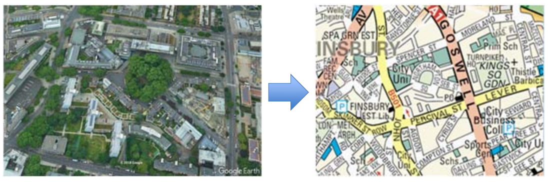


## Memory Management

- Background and Introduction
- **Memory Abstraction**
- Address Space
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement

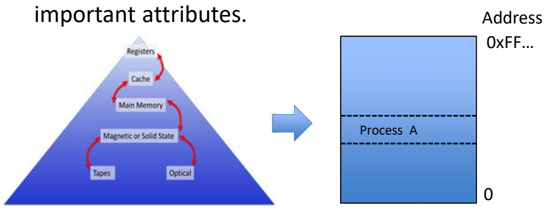
## Memory abstraction

- To better understand a complex process or mechanism, we can simplify and remove as many unnecessary details and keep only the most important attributes.

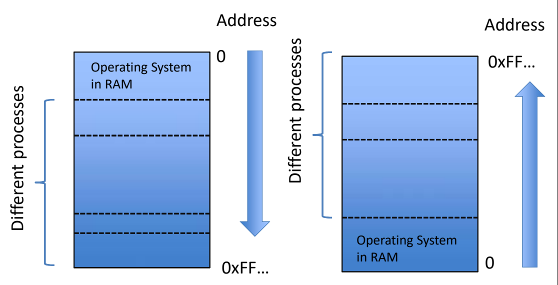


## Memory abstraction

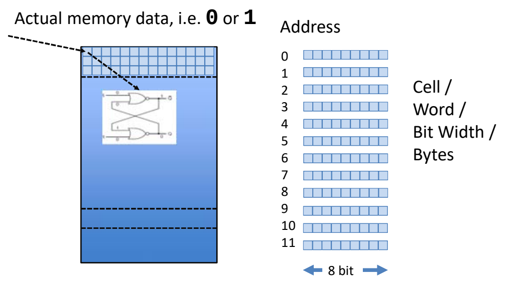
- To better understand a complex process or mechanism, we can simplify and remove as many unnecessary details and keep only the most important attributes.



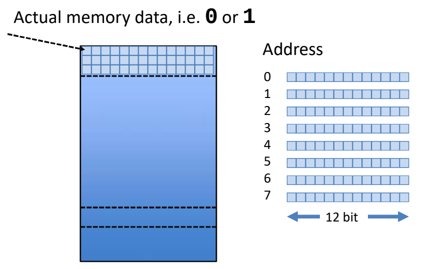
## Memory abstraction(s)



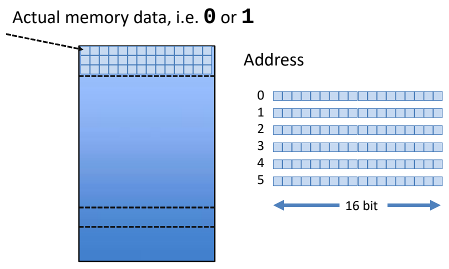
## Memory organisation



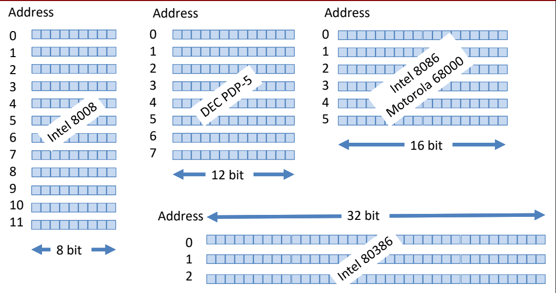
## Memory organisation



## Memory organisation



## Ways to organise memory (96 bits)



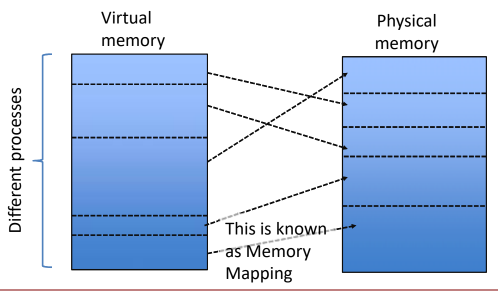
## Memory Management

- Background and Introduction
- Memory Abstraction
- **Address Space**
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement

## Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

## Virtual Memory



## Context Switching

- Remember that switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.
- This task is known as **a context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB (Process control block) and loads the saved context of the new process scheduled to run.

## Memory Management

- Background and Introduction
- Memory Abstraction
- Address Space
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement

## Memory-Management Unit (MMU)

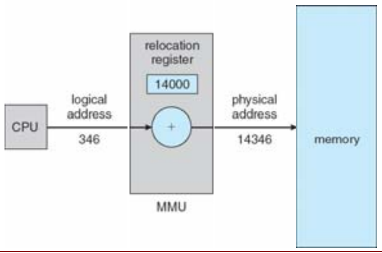
- Hardware device that at run time maps virtual to physical address, there are many methods possible
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

## Memory-Management Unit (MMU)

- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
- The value in the relocation register is *added* to every address generated by a user process at the time the address is sent to memory

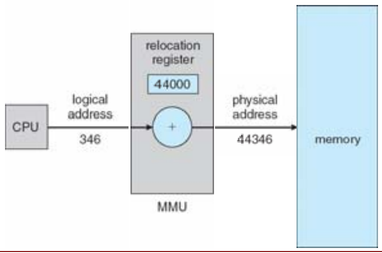
## Dynamic relocation using a relocation register

MS-DOS on Intel 80x86 used 4 relocation registers



## Dynamic relocation using a relocation register

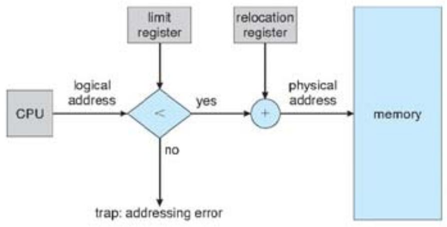
When a context switch happens, the register is updated and the new address is generated.



## Protection

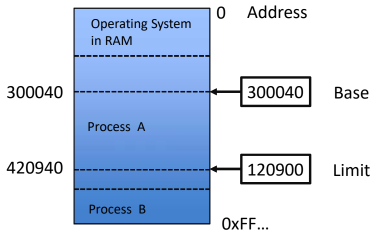
- Recall that the OS needs to prevent processes from accessing one another's memory
  - using a system call for each memory access would be hopelessly slow
  - the MMU needs to play a role
- Simple dynamic relocation does not provide memory protection
  - Why?
  - need to add a limit register

## Dynamic relocation using a relocation and limit register



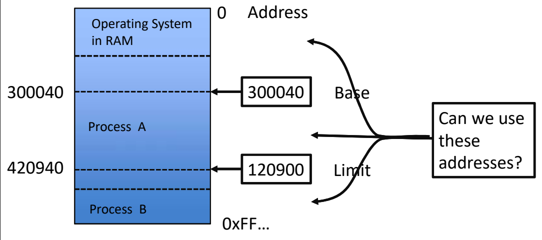
## Logical Address Space

- A pair of **base** and **limit registers** define the logical address space



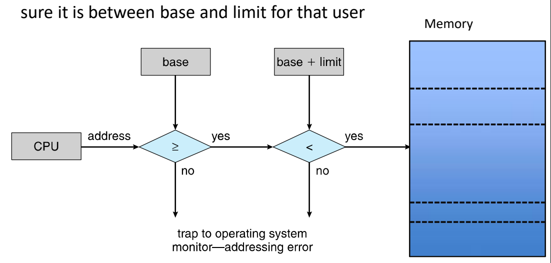
## Logical Address Space

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



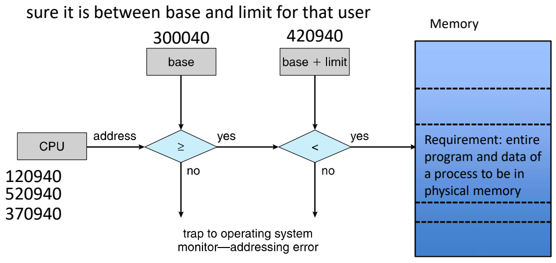
## Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



## Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user


Memory
300040
420940
base
base + limit
CPU
address
≥
yes
<
yes
no
no
120940
520940
370940
Requirement: entire program and data of a process to be in physical memory
trap to operating system monitor—addressing error

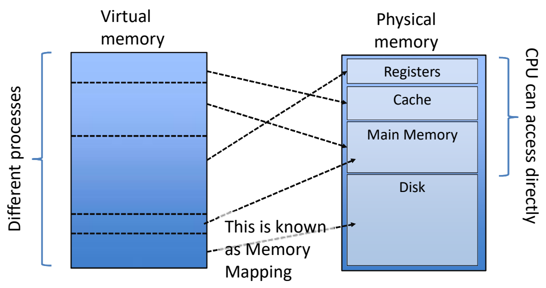
## Memory Management

- Background and Introduction
- Memory Abstraction
- Address Space
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement

## Larger Processes...

- A process needs to be loaded in memory to be executed.
- If the physical memory is large enough, then things are ok.
- But if not ...

## Virtual Memory

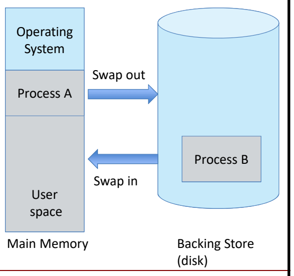


## Swapping

- May not be enough physical memory to hold all current processes at the same time, but only one process is actually on the CPU at a time
- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- The largest part of swap time is **disk read/write** time; the total transfer time is proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (incl. UNIX, Linux, and Windows)

## Swapping

- Bring in each entire process, run for a while, then putting it back on the disk.
- Idle processes are mostly stored on disk, so they do not take up any memory when they are not running.

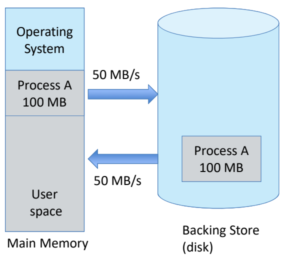


## Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- **Exercise**: For a process of 100MB swapping to hard disk with transfer rate of 50MB/sec, calculate:
  - Swap out time of ???? ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of ???? ms

## Swapping

- The largest part of **swap time is disk read/write** time. Less significant is **latency**.
- The total transfer time is proportional to the amount of memory swapped.
- If latency = 8μs, what is the total swapping time?



## *Slowdown* factor

- Sometimes processes run as fast as possible, but not always …
- Slowdown factor calculates the ratio of the slow case over the fast case:

$$Slowdown = \frac{t_{slow}}{t_{fast}}$$

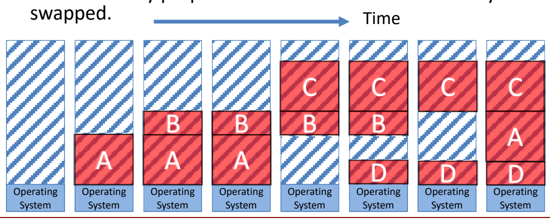
## *Slowdown* factor

- **Exercise**: Assume a process of **5 MB**, which runs normally in **10 ms**. The average time between context switching **50 μs**, and **500 MB/s** for transfer to from disk. Calculate the slowdown factor if the process needs to be swapped to/from disk (*careful with the units*).

$$Slowdown = \frac{t_{slow} = \text{with swap}}{t_{fast} = \text{without swap}}$$

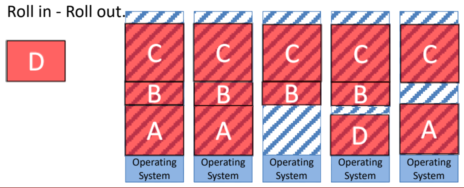
## Swapping

- Processes are created or swapped from disk. The location depends on the free locations of the memory.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.



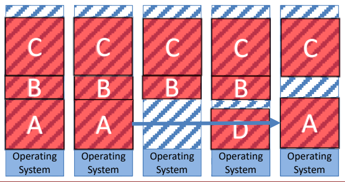
## Swapping

- If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process.
- When the higher-priority process finishes, the lower-priority process can be swapped back. This variant of swapping is sometimes called Roll in - Roll out.



## Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space



## Context Switch Time including Swapping

- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

## Context Switch Time and Swapping (Cont.)

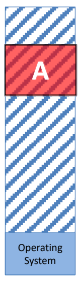
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead

## Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

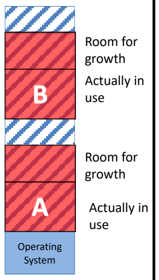
## Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory



## Room for growth

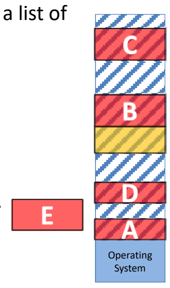
- How much memory should be allocated for a process when it is created or swapped in?
- If the process is fixed -> easy.
- If the process can grow …
  - If there is a hole adjacent, allocate and grow in the hole,
  - If there is no memory to grow, suspend and wait, or kill it
- Or allocate some space so that the process can grow.



## Dynamic Storage-Allocation Problem

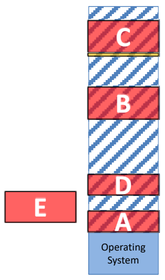
How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough
- Find a hole that is big enough for the process
- Hole is divided into one section for the process and the free space, if any.
- Fast algorithm as it searches as little as possible.



## Dynamic Storage-Allocation Problem

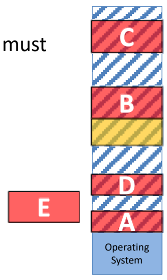
- **Best-fit**: Allocate the ***smallest*** hole that is big enough
- Must search entire list, unless ordered by size and use the smallest hole that is adequate.
- Rather than breaking up a big hole that may be needed later.
- Best fit is slower than first fit as it needs to search. It also produces the smallest leftover hole, which, being tiny, tends to be useless!



## Dynamic Storage-Allocation Problem

- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit are better than worst-fit in terms of speed and storage utilization



## External Fragmentation

- Both the first-fit and best-fit strategies for memory allocation will suffer from **external fragmentation**.
- As each processes is loaded/removed from memory, free memory space is divided into little pieces.
- External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes

## Internal Fragmentation

- Example: a partition allocation with a hole of 18,464 bytes. Next process requests 18,462 bytes -> a hole of 2 bytes would be useless, and the overhead to keep record of the hole is larger than the 2 bytes.
- Use fixed size blocks instead, which may have some unused space

## Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

## Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
- Now consider that backing store has same fragmentation problems

## Memory Management

- Background and Introduction
- Memory Abstraction
- Address Space
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement

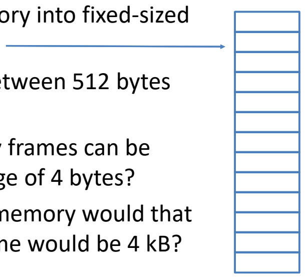
## Paging or Paged Memory

- In a modern system, virtual memory is managed in *chunks* of contiguous addresses known as *pages (virtual memory) or frames (in physical memory)*

  - Each page of virtual memory is mapped to a same-sized frame of contiguous *physical* memory
  - Page sizes typically range between 4 kB and 4 MB
  - Standard Linux page size is 4 kB (4096 bytes)

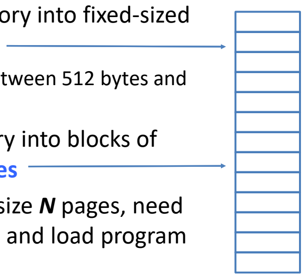
## Paging or Paged Memory

- Divide physical memory into fixed-sized blocks called **frames**
- Size is power of 2, between 512 bytes and 16 Mbytes.
- **Question**: how many frames can be pointed to with a page of 4 bytes?
- How much physical memory would that comprise if each frame would be 4 kB?



## Paging or Paged Memory

- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program

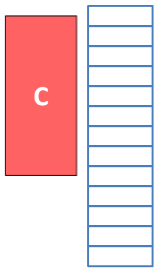


## Paging or Paged Memory

- Keep track of all free frames
- Set up a **page table** to translate logical to physical addresses
- External Fragmentation > no
- Internal Fragmentation > yes
- Small page sizes seem desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases.

## Paging Internal Fragmentation

- **Calculate** internal fragmentation if
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - How many pages are required?
  - What is the internal fragmentation?
  - What would be the best/worst cases?



## Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit

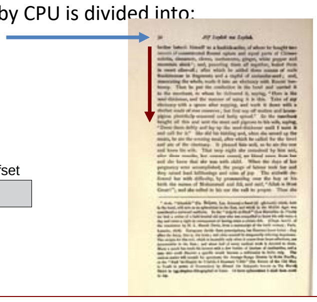
| page number | page offset |
|---|---|
| p | d |
| m - n bits | n bits |

  - For given logical address space $2^m$ and page size $2^n$

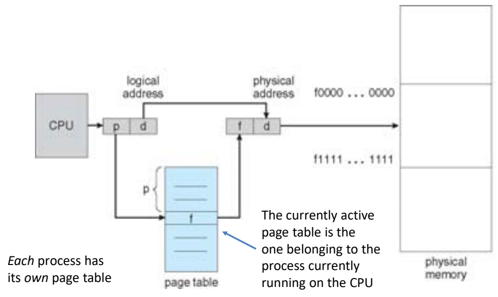
## Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (*p*) –
  - **Page offset** (*d*) –

| page number | page offset |
|---|---|
| p | d |
| m - n bits | n bits |


[illegible]

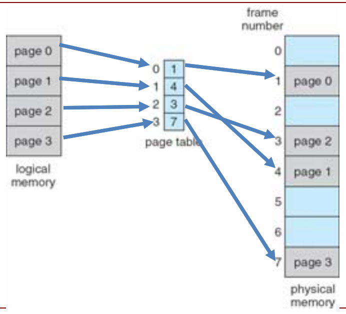
## Paging Hardware



The currently active page table is the one belonging to the process currently running on the CPU

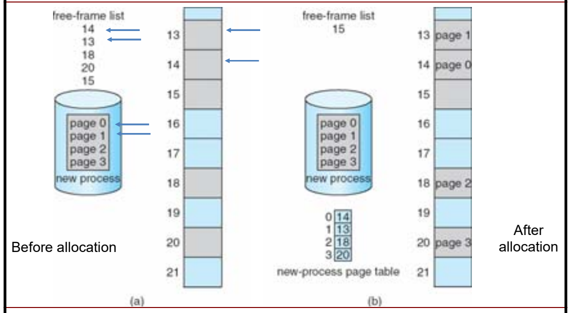
*Each* process has its *own* page table

## Paging Model of Logical and Physical Memory



In this case, there are fewer pages than frames, this is not always the case

## Free Frames



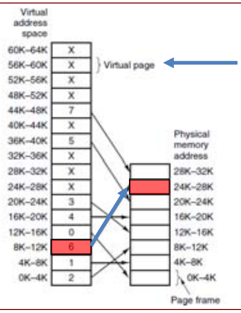
## "Impossibly" large Virtual Memory

- The virtual memory address space for a process can be bigger than the total amount of physical memory
  - Possible because the mapping is *dynamic*
  - *Scenario 1: process never actually uses all of the virtual memory it owns* (some pages never need to be mapped at all)
  - *Scenario 2: process uses different parts of its virtual address space at different times.* The OS can "page out" (copy to disk) frames not currently in use and remap new pages to those frames

## Demand Paging

- Bring a page into memory ("page in") only when it is needed
  - Page is needed -> process tries to access an address in that page
    - illegal reference (virtual address not allocated to process) -> abort
    - not-in-memory ("page-fault") -> bring to memory
- A pager is a **lazy swapper** – never swaps a page into memory unless the page is used (**demanded**)

## Paging Model of Logical and Physical Memory



Some pages do not correspond to a physical memory address.

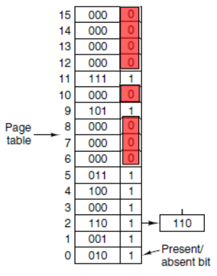
```
MOV REG,8192
```

```
MOV REG,24576
```

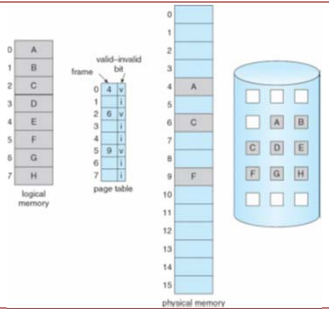
In hardware a bit indicates presence or absence.

## Present / Absent pages

- Tag each page table entry
- with a **present / absent** or **valid / invalid** bit
  - 1, v in-memory
  - 0, i not-in-memory
- Initially valid–invalid bit is set to 0,i on all entries
- If a page absent page is requested, it creates a page `fault`

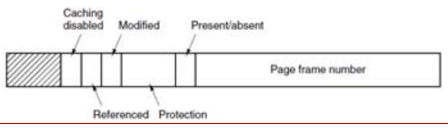


## Page Table When Some Pages Are Not in Main Memory



## Structure of a Page Table Entry

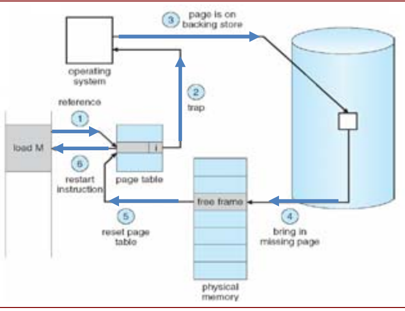
- Present/absent (valid/invalid) if 1 can be accessed, if 0 is not available.
- Protection: kind of access permitted (read, write, read/write).
- Modified, Referenced, keep track of the page usage, when modified is called “dirty”.
- Caching, sometimes it is important to recall from devices (I/O) and not just a cached value.



## Page Fault

- A page-fault occurs if the requested entry in the page table is marked 0,***i*** (invalid)
  - Causes a trap (interrupt)
- OS (interrupt handler) checks page number:
  - If *illegal* (outside process address space) *abort* (protection violation)
  - Otherwise (legal but not currently in memory):
  1. Get empty frame
  2. Swap page into frame
  3. Change validation bit from ***i*** to **v**
  4. Restart the instruction that caused the page-fault

## Steps in Page Fault



## Performance of Demand Paging

- Page-Fault Rate 0 < *p* < 1.0
  - if *p* = 0 no page-faults
  - if *p* = 1, every memory reference causes a page-fault
- Effective Access Time (EAT)
  - the *average* time taken to service a memory reference
  - if there were never any page-faults (p = 0) this would just be the base hardware memory access time (actual RAM access time)
- When a page-fault happens, the following *extra* time is spent:
  - time taken to execute page-fault trap + time taken to swap pages in and out from disk + time taken to return from trap and restart the process which made the memory reference
- We call this combined extra time the *page-fault service time*

## Effective Access Time Exercise

- Suppose base memory access time = 100 ns and average page-fault service time = 1 ms
- Suppose one memory reference in every 1000 results in a page-fault (so p = 1/1000)
- **Estimate** the EAT
- What is the slowdown factor?

## Effective Access Time Exercise

- Let *p* be the page-fault rate ($0 \le p \le 1$)
- Let *a* be the hardware memory access time
- Let *S* be the page-fault service time

  $EAT = (1-p)a + p(a + S) =$

  $EAT = a - pa + pa + pS$

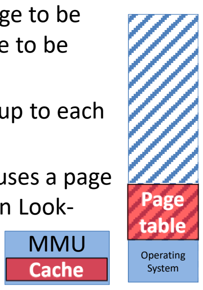
  $EAT = a + pS$
- **Calculate EAT.**

## Effective Access Time

- Adding *RAM* tends to *decrease* p
- Adding *processes* tends to *increase* p
- Increasing *disk speed* tends to *decrease* S

## Translation Look-aside Buffer

- Page tables are generally too large to be implemented in registers so have to be stored in main memory
- This adds an extra memory lookup to each memory access
- To reduce this overhead, MMU uses a page table *cache* called the Translation Look-aside Buffer (TLB)

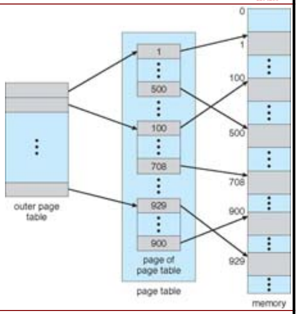


## Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB ($2^{12}$)
  - Page table would have 1 million entries ($2^{32}$ / $2^{12}$)
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

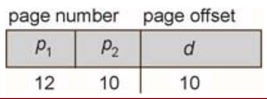
## Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table



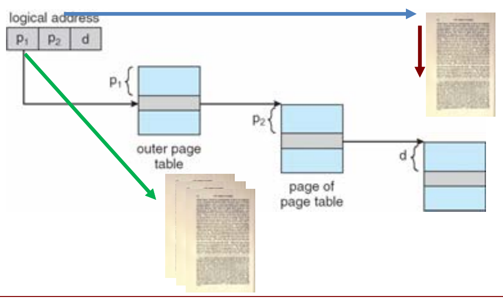
## Hierarchical Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



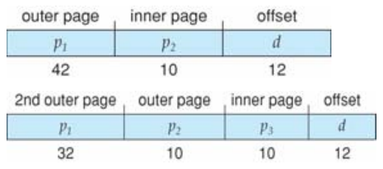
| page number | | page offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 12 | 10 | 10 |

## Address-Translation Scheme



## Two-Level Paging Example

- $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the inner page table (**forward-mapped page table**)
- It could be extended to three-levels:



## Memory Management

- Background and Introduction
- Memory Abstraction
- Address Space
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement

## Page replacement

- Suppose we need to bring a page into memory but there is no free frame
- **Page replacement**: find some page which is in memory, but not currently being used, and swap it out.
- Page replacement algorithms decide which memory pages to swap out when a page of memory needs to be allocated.

## Page replacement

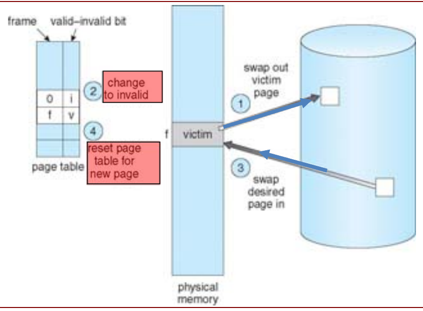
- Page-fault service routine is modified to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes the separation between logical memory and physical memory: large virtual memory can be provided on a smaller physical memory

## Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty (i.e. has been modified)
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

## Page Replacement



## Page and Frame Replacement Algorithms

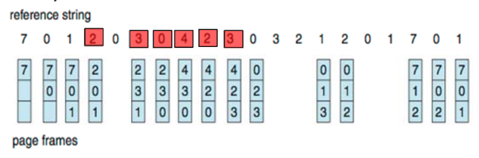
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available

## Many Page Replacement Algorithms

- First-in, first-out (FIFO) algorithm
- Optimal algorithm
- Second-chance algorithm
- Clock Algorithm
- Least recently used (LRU) algorithm
- *and many more*

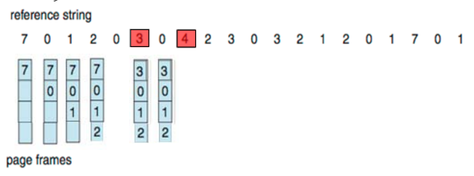
## First-In-First-Out (FIFO) Algorithm

- Order of pages: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

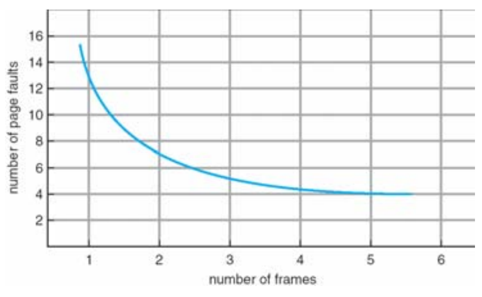


## First-In-First-Out (FIFO) Algorithm

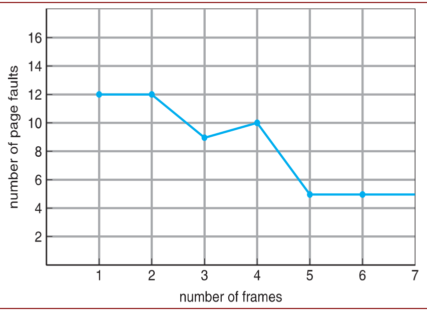
- Order of pages: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 4 frames (4 pages can be in memory at a time per process)



## Graph of Page Faults Versus The Number of Frames

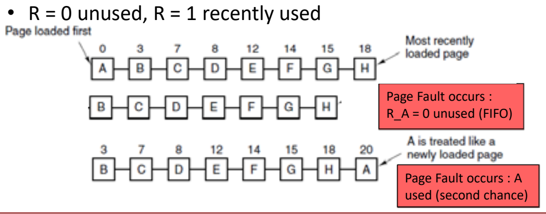


## FIFO Illustrating Belady's Anomaly

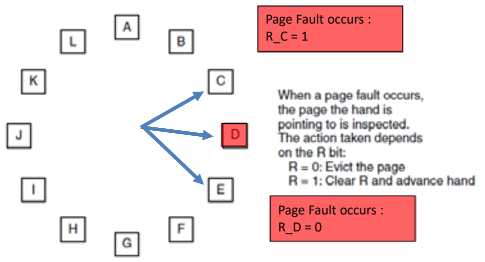


## Second-Chance Algorithm

- FIFO does not discriminate if a page is useful or not, and the oldest may still be useful
- Use status bits R (read/write), M (modified)
- R = 0 unused, R = 1 recently used

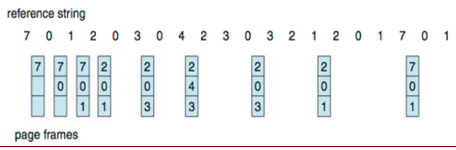

Page loaded first
0 3 7 8 12 14 15 18
A B C D E F G H
Most recently loaded page
B C D E F G H
Page Fault occurs : R_A = 0 unused (FIFO)
3 7 8 12 14 15 18 20
B C D E F G H A
A is treated like a newly loaded page
Page Fault occurs : A used (second chance)

## Clock Page Replacement Algorithm


A
B
C
D
E
F
G
H
I
J
K
L
Page Fault occurs : R_C = 1
When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand
Page Fault occurs : R_D = 0

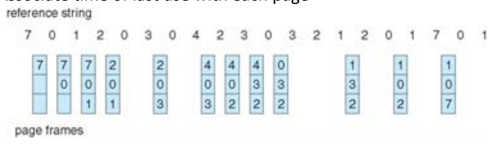
## Optimal Algorithm

- Replace page that will not be used for longest period of time (in the future, that is)
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



## Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

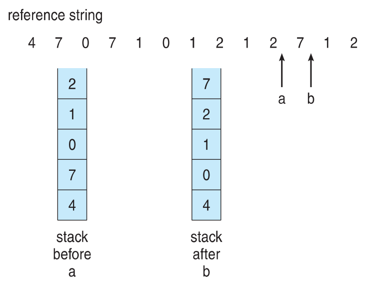


- Better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- Theoretically possible but difficult to implement

## LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

## Use Of A Stack to Record Most Recent Page References



## LRU Approximation Algorithms

- LRU needs special hardware and still slow
- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- Second-chance algorithm
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

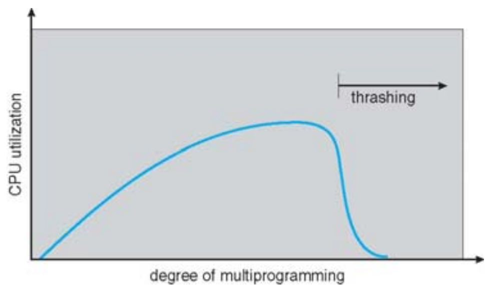
## Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

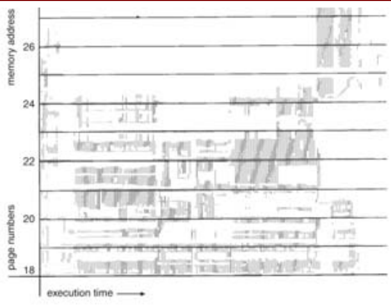
## Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to Low CPU utilization
- Thrashing ≡ a process is busy swapping pages in and out

## Thrashing (Cont.)

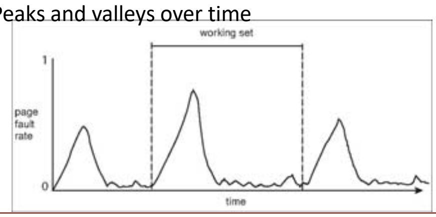


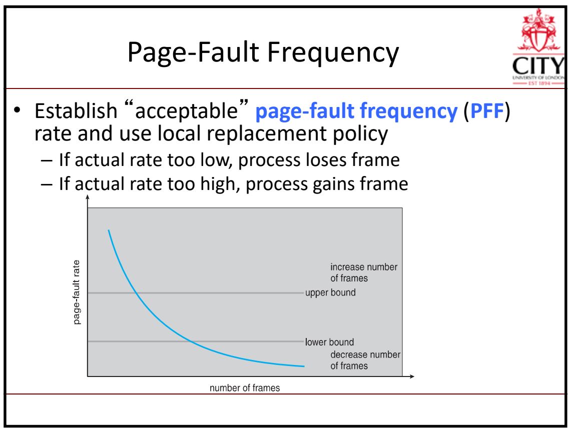
## Locality In A Memory-Reference Pattern



## Working Sets and Page Fault Rates

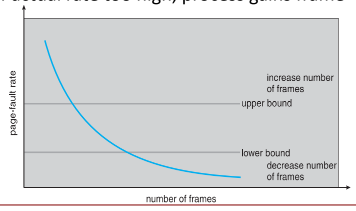
- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time

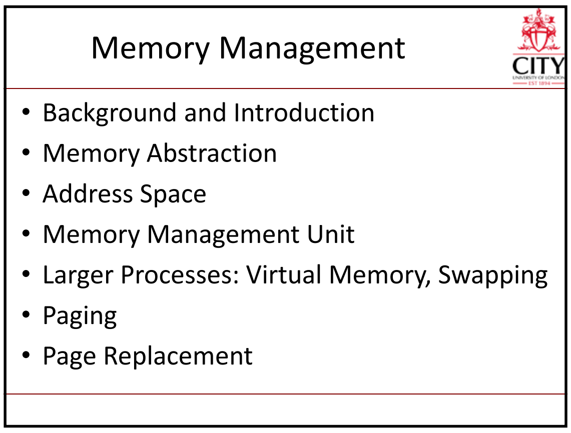




## Page-Fault Frequency

- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame





## Memory Management

- Background and Introduction
- Memory Abstraction
- Address Space
- Memory Management Unit
- Larger Processes: Virtual Memory, Swapping
- Paging
- Page Replacement