Department of Electrical Engineering, Feng-Chia University

Chapter 9 ~ 10:
Memory Management

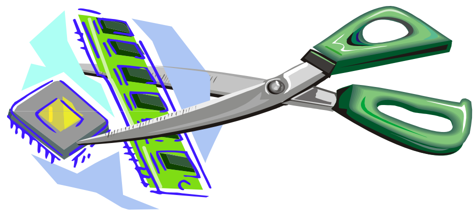
王振傑 (Chen-Chieh Wang)
ccwang@mail.ee.ncku.edu.tw

Department of Electrical Engineering, Feng-Chia University

Outline

- Background (address translation)
- Segmentation
- Paging
- Virtual Memory
- Page Replacement

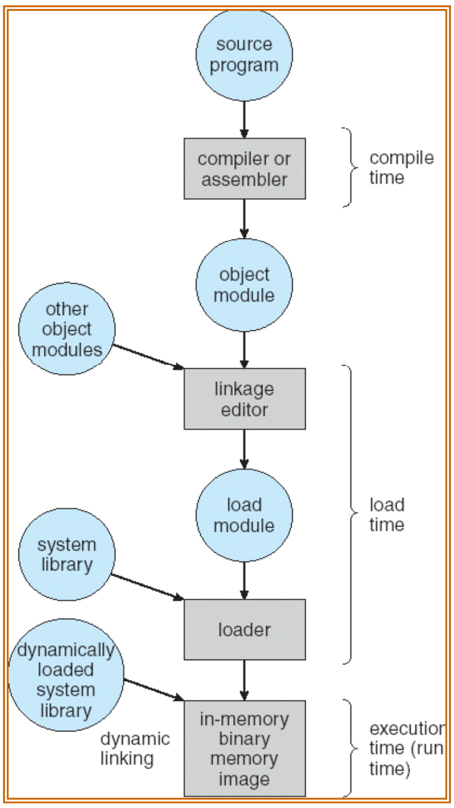
Virtualizing Resources



- Physical Reality: Different Processes/Threads share the same hardware
  - Need to multiplex CPU (CPU scheduling)
  - Need to multiplex use of Memory (Today)
  - Need to multiplex disk and devices
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
  - Probably don't want different threads to even have access to each other's memory (protection)

Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at:
  - Compile time (i.e. "gcc")
  - Link/Load time (unix "ld" does link)
  - Execution time (e.g. dynamic libs)
- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system
- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Department of Electrical Engineering, Feng-Chia University

Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

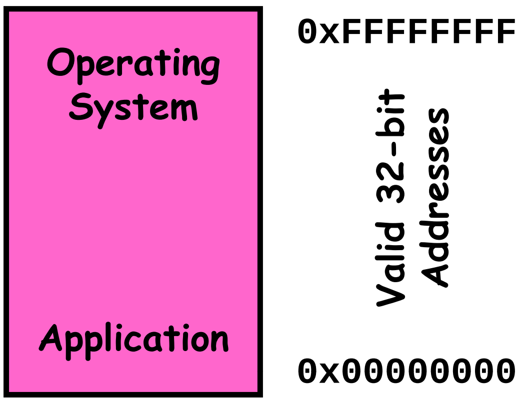
Department of Electrical Engineering, Feng-Chia University

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address

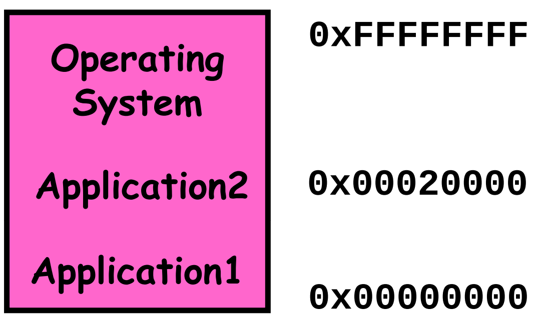


  - Application given illusion of dedicated machine by giving it reality of a dedicated machine
- Of course, this doesn't help us with multithreading

Department of Electrical Engineering, Feng-Chia University

Multiprogramming (First Version)

- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads

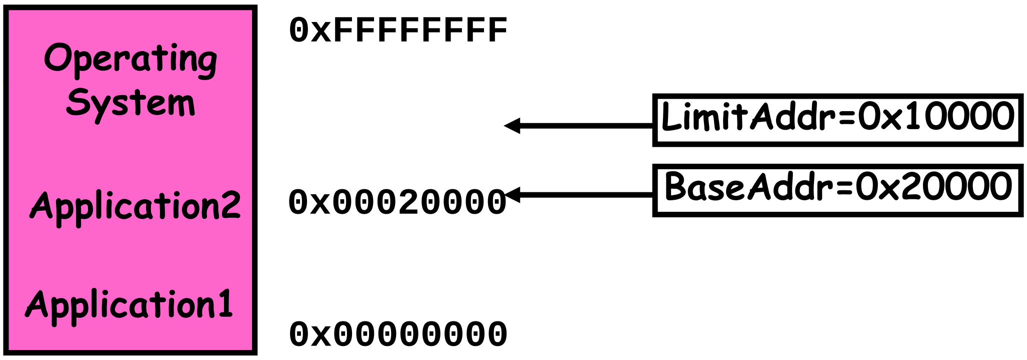


  - Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - Everything adjusted to memory location of program
    - Translation done by a linker-loader
    - Was pretty common in early days
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

Department of Electrical Engineering, Feng-Chia University

Multiprogramming (Version with Protection)

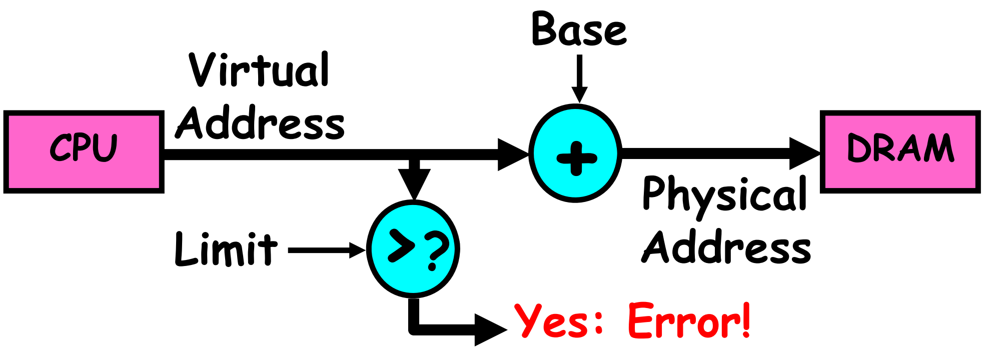
- Can we protect programs from each other without translation?



  - Yes: use two special registers BaseAddr and LimitAddr to prevent user from straying outside designated area
    - If user tries to access an illegal address, cause an error
  - During switch, kernel loads new base/limit from TCB
    - User not allowed to change base/limit registers

Department of Electrical Engineering, Feng-Chia University

Simple Segmentation: Base and Bounds

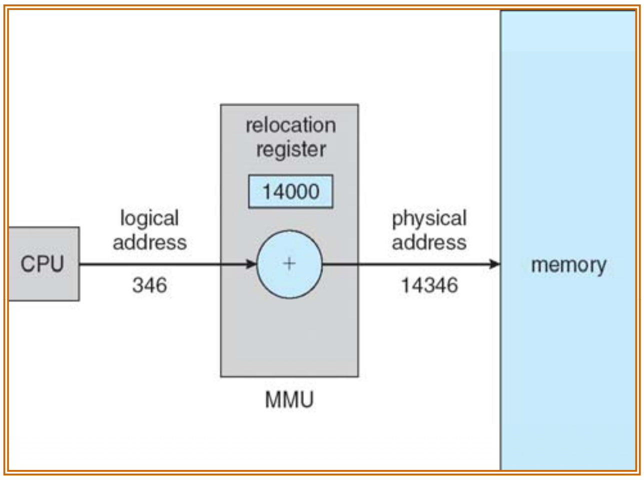


- Can use base & bounds/limit for dynamic address translation (Simple form of "segmentation"):
  - Alter every address by adding "base"
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Department of Electrical Engineering, Feng-Chia University

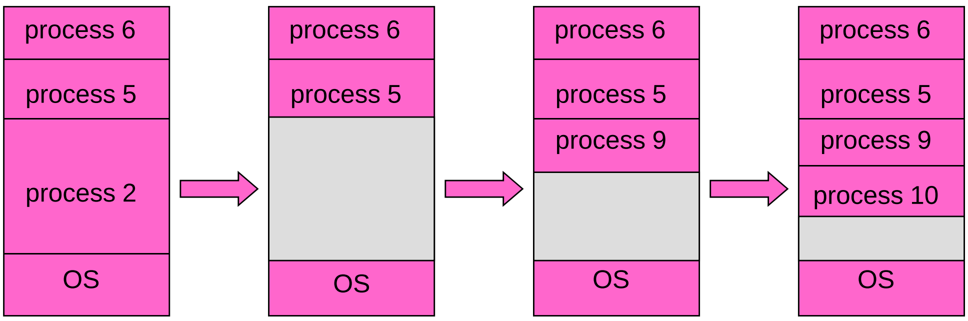
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses



Department of Electrical Engineering, Feng-Chia University

Issues with simple segmentation method



- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process
- Need enough physical memory for every process

Department of Electrical Engineering, Feng-Chia University

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
  - **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the *largest* hole; must also search entire list
    - Produces the largest leftover hole

Department of Electrical Engineering, Feng-Chia University

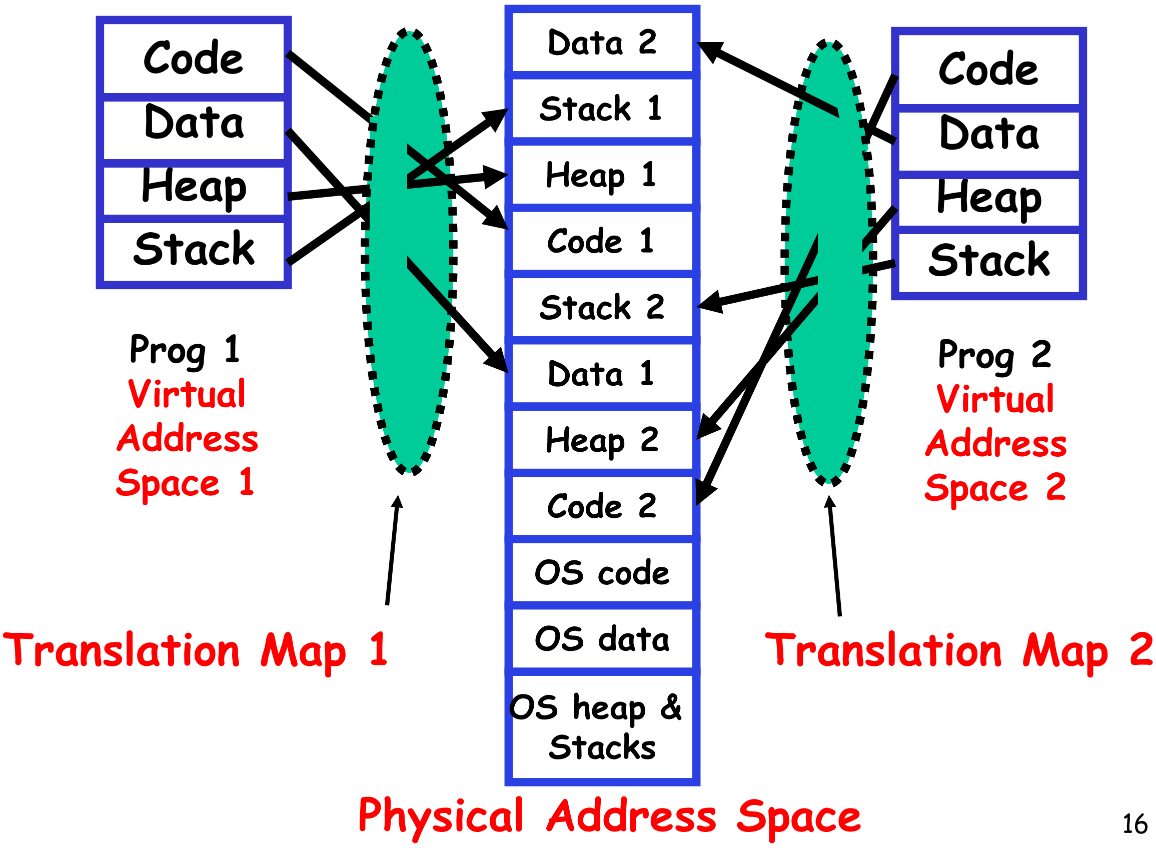
Fragmentation

- 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
- 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
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Multiprogramming (Translation and Protection version 2)

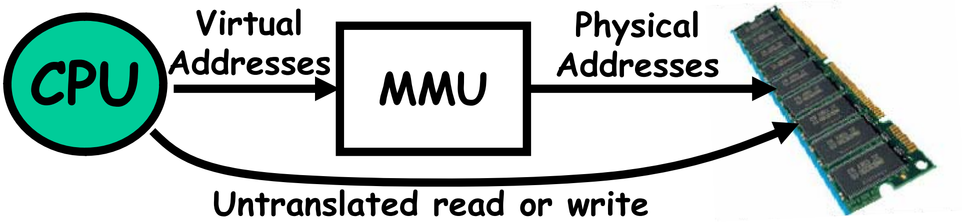
- Problem: Run multiple applications in such a way that they are protected from one another
- Goals:
  - Isolate processes and kernel from one another
  - Allow flexible translation that:
    - Doesn't lead to fragmentation
    - Allows easy sharing between processes
    - Allows only part of process to be resident in physical memory
- (Some of the required) Hardware Mechanisms:
  - General Address Translation
    - Flexible: Can fit physical chunks of memory into arbitrary places in users address space
    - Not limited to small number of segments
    - Think of this as providing a large number (thousands) of fixed-sized segments (called "pages")
  - Dual Mode Operation
    - Protection base involving kernel/user distinction

Example of General Address Translation



Department of Electrical Engineering, Feng-Chia University

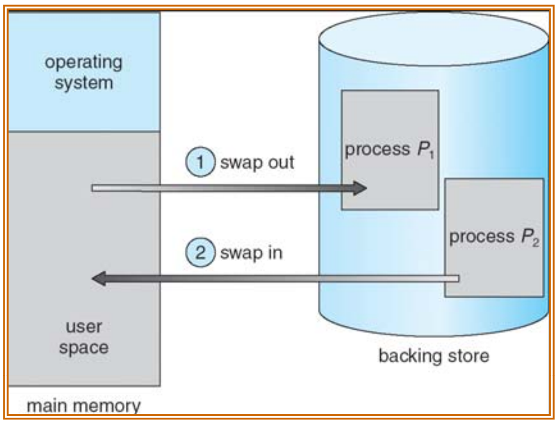
Two Views of Memory



- Recall: Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently: two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box converts between the two views
- Translation helps to implement protection
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space
  - Overlap avoided through translation, not relocation

Department of Electrical Engineering, Feng-Chia University

Schematic View of Swapping



- Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
    - Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- Desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory

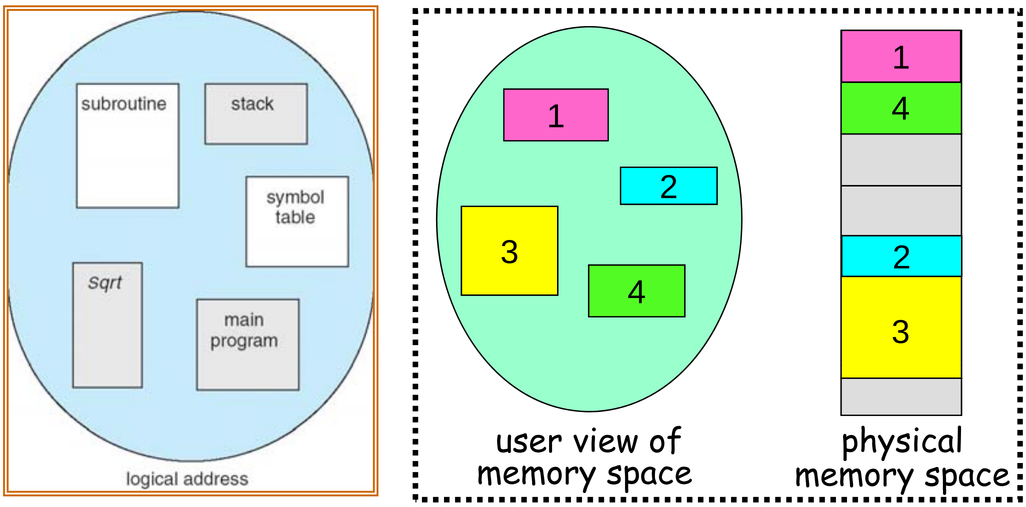
Department of Electrical Engineering, Feng-Chia University

Outline

- Background (address translation)
- Segmentation
- Paging
- Virtual Memory
- Page Replacement

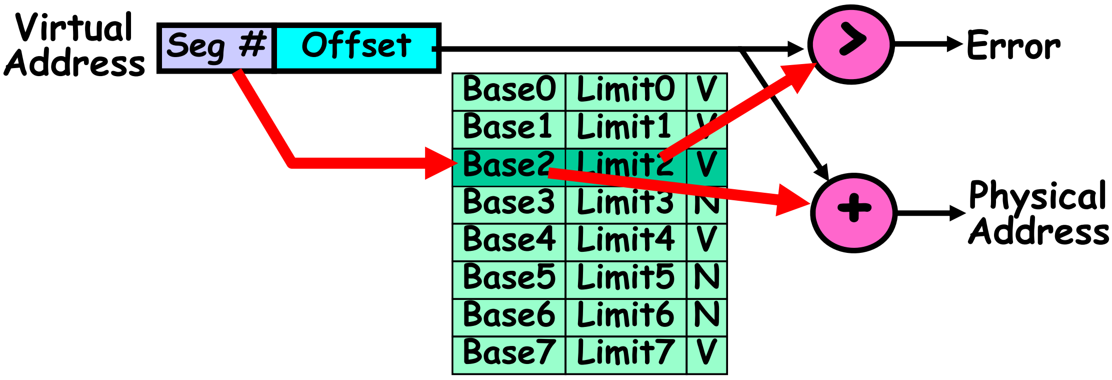
Department of Electrical Engineering, Feng-Chia University

More Flexible Segmentation



- Logical View: multiple separate segments
  - Typical: Code, Data, Heap, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

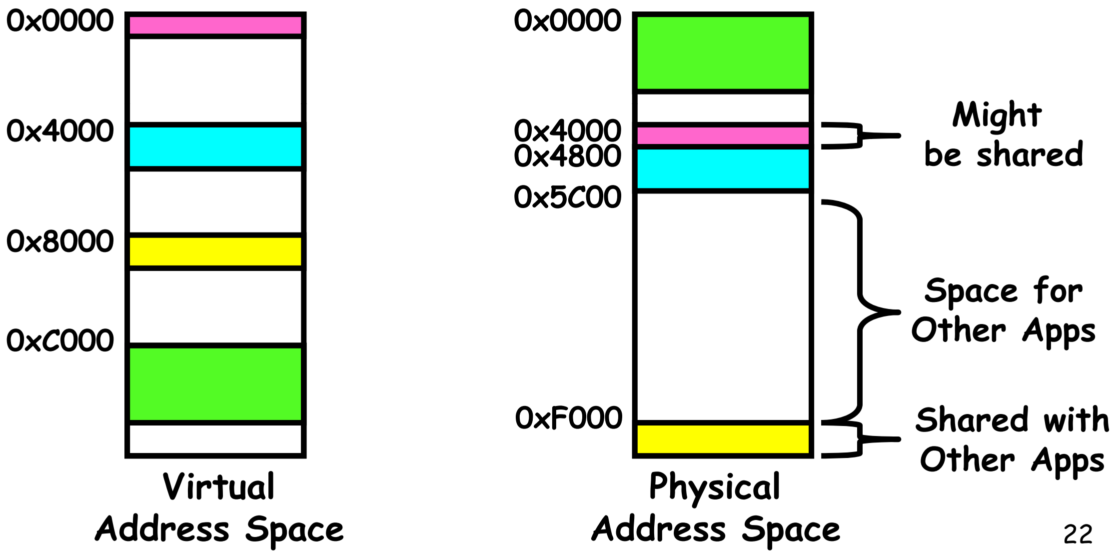
Implementation of Multi-Segment



- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: `mov [es:bx],ax`.
- What is "V/N"?
  - Can mark segments as invalid; requires check as well

Example: Four Segments (16 bit addresses)

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Example of segment translation

```
0x240     main:      la $a0, varx
0x244                jal strlen
  …                    …
0x360     strlen:    li   $v0, 0  ;count
0x364     loop:      lb   $t0, ($a0)
0x368                beq  $r0,$t1, done
  …                    …
0x4050    varx       dw   0x314159
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 → $a0, Move PC+4→PC
2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get "li $v0,0"
   Move 0x0000 → $v0, Move PC+4→PC
4. Fetch 0x364. Translated to Physical=0x4364. Get "lb $t0,($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

Observations about Segmentation

- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    - If it does, trap to kernel and dump core
- When it is OK to address outside valid range:
  - This is how the stack and heap are allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
  - Shared segment could be read-only or read-write
- What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called "swapping")

Department of Electrical Engineering, Feng-Chia University

Outline

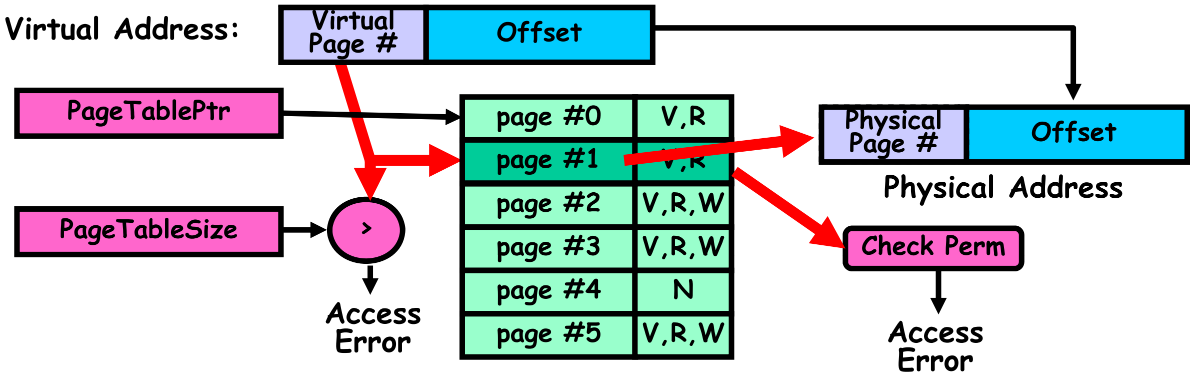
- Background (address translation)
- Segmentation
- Paging
- Virtual Memory
- Page Replacement

Department of Electrical Engineering, Feng-Chia University

Paging: Physical Memory in Fixed Size Chunks

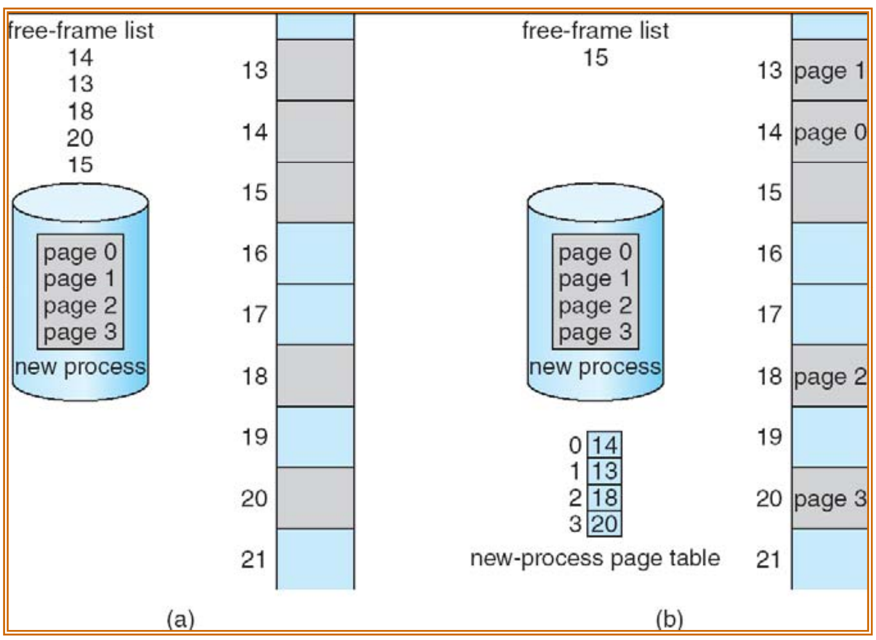
- Problems with segmentation?
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks
- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - Can use simple vector of bits to handle allocation:
      00110001110001101 … 110010
    - Each bit represents page of physical memory
      1⇒allocated, 0⇒free
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

How to Implement Paging?



- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

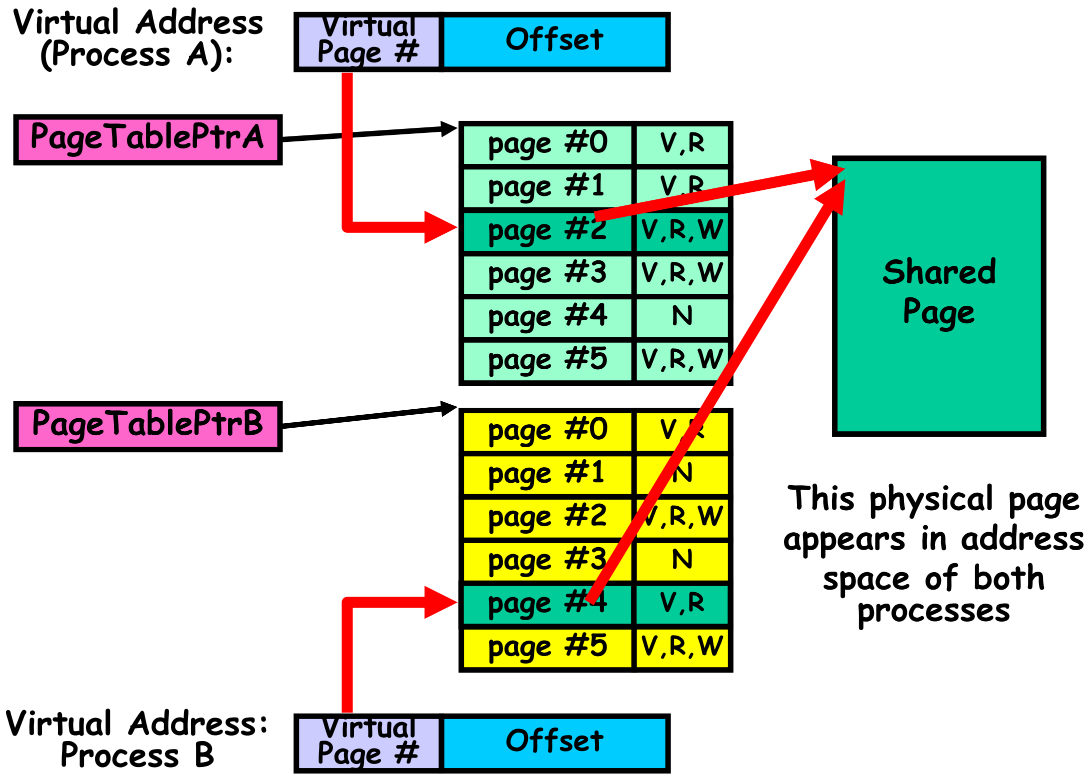
Free Frames (Physical Pages)



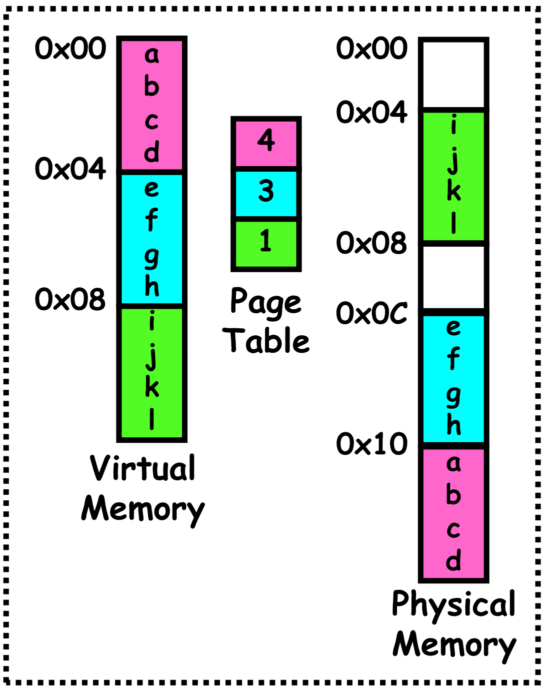
Before allocation

After allocation

What about Sharing?



Simple Page Table Discussion

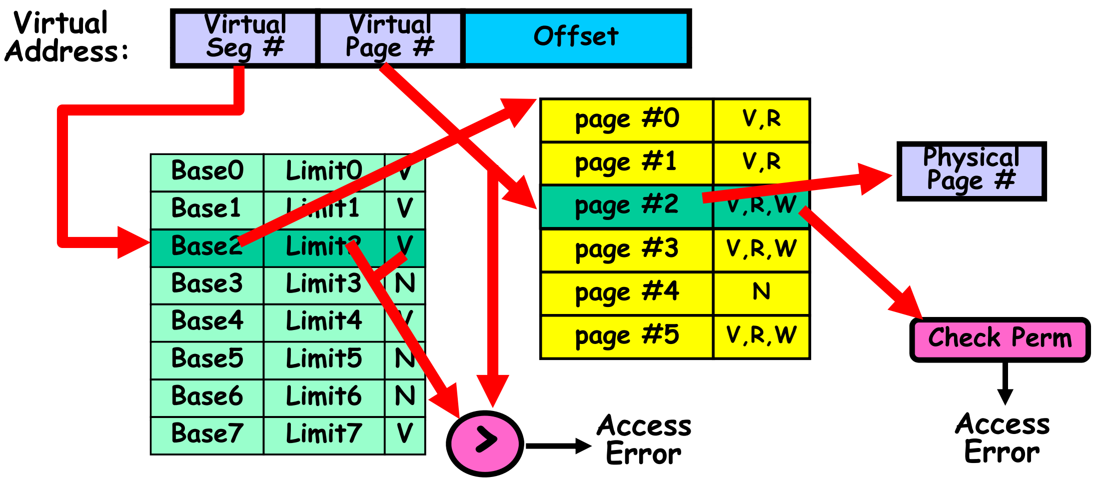


Example (4 byte pages)

- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - Simple memory allocation
    - Easy to Share
  - Con: What if address space is sparse?
    - E.g. on UNIX, code starts at 0, stack starts at (2^{31}-1).
    - With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    - Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

Multi-level Translation

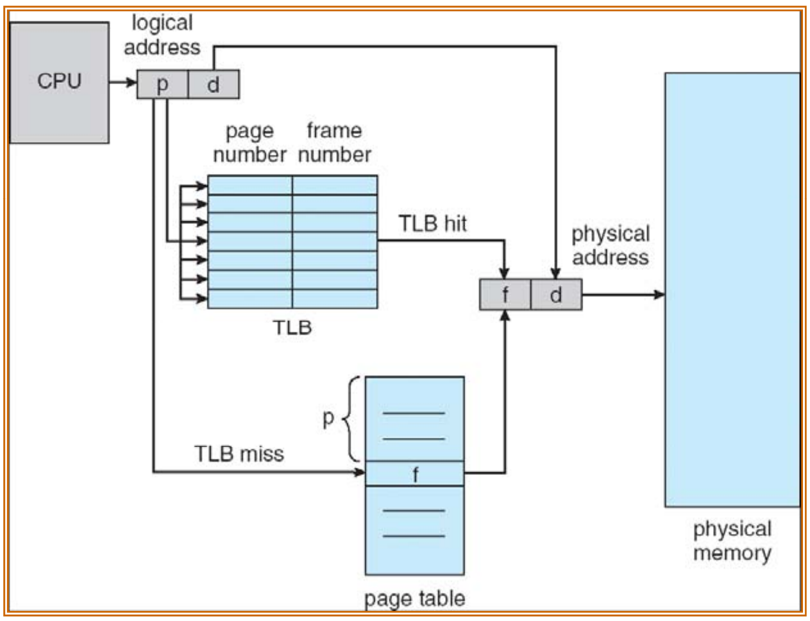
- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):



- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

Paging Hardware With TLB

- Making Address Translation Fast
- A cache for address translations: Translation Lookaside Buffer



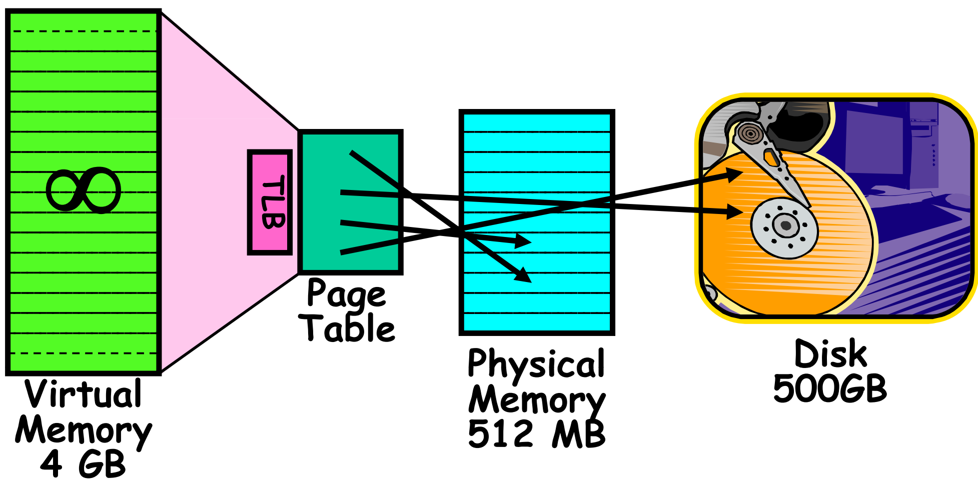
Department of Electrical Engineering, Feng-Chia University

Outline

- Background (address translation)
- Segmentation
- Paging
- Virtual Memory
- Page Replacement

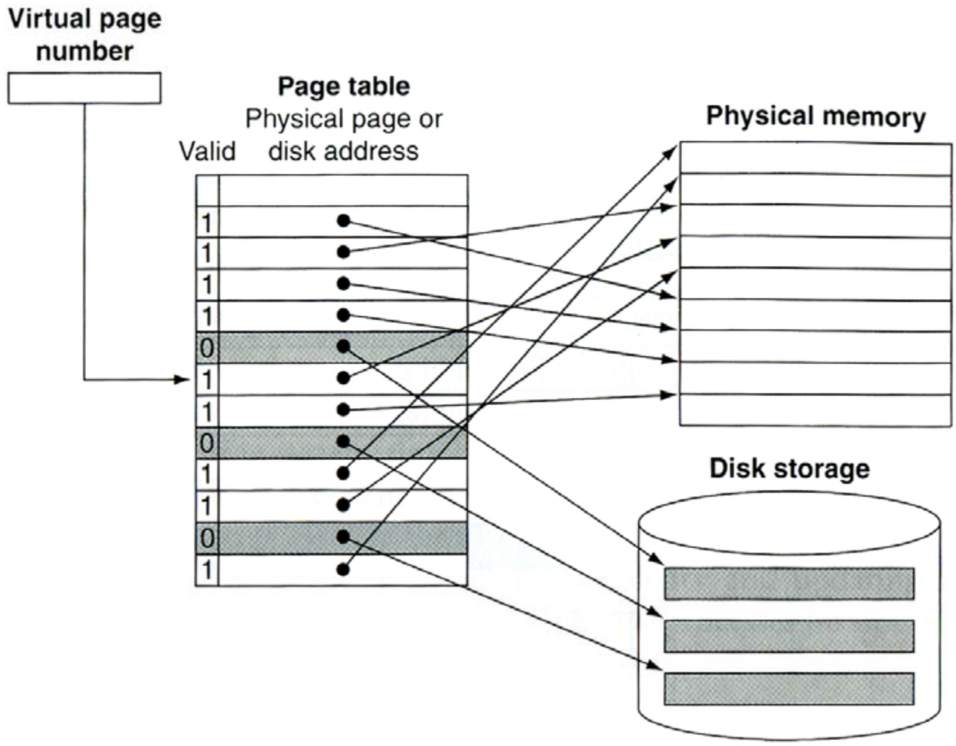
Department of Electrical Engineering, Feng-Chia University

Virtual Memory



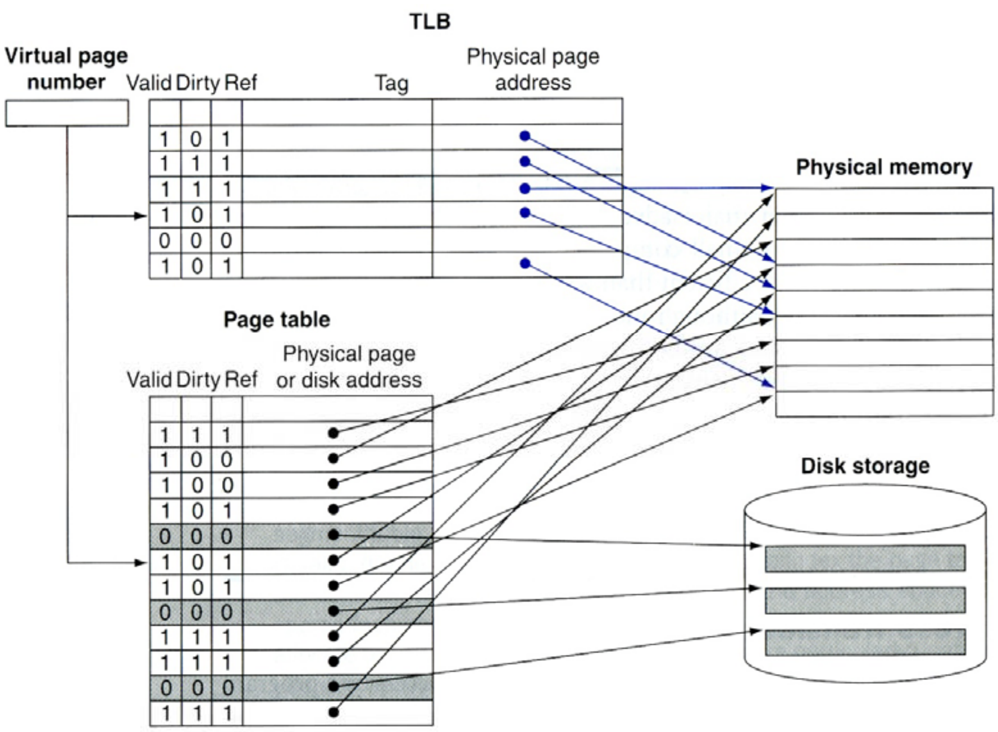
- Illusion of Infinite Memory
- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory

Page Tables



Department of Electrical Engineering, Feng-Chia University

Translation Lookaside Buffer (TLB)



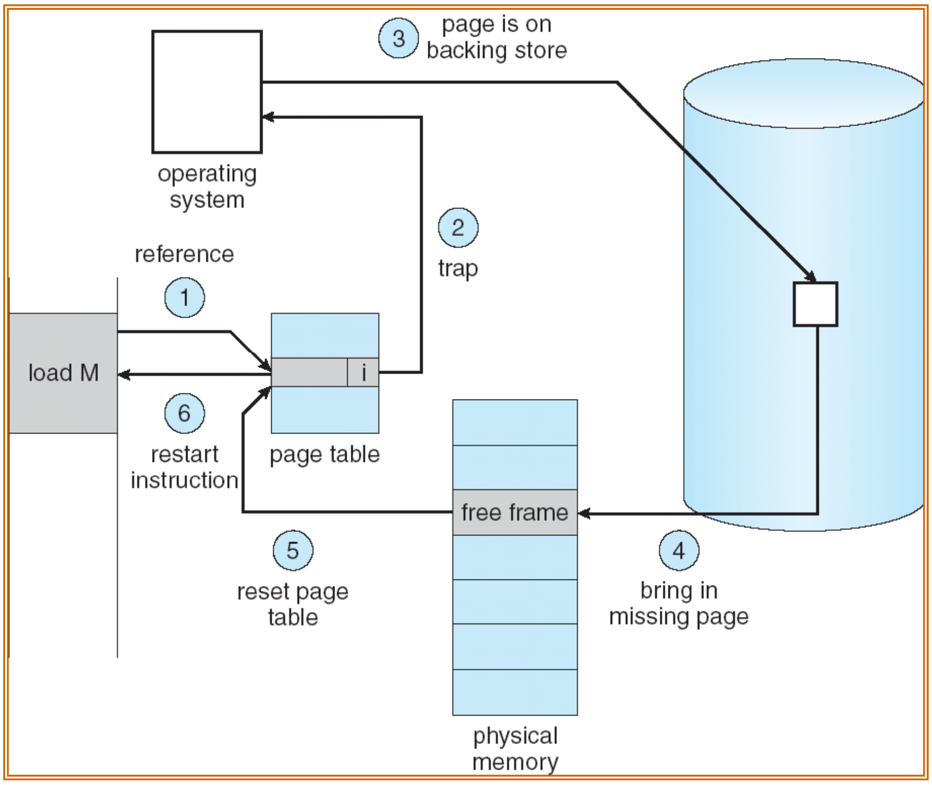
Department of Electrical Engineering, Feng-Chia University

TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

Department of Electrical Engineering, Feng-Chia University

Steps in Handling a Page Fault



Outline

- Background (address translation)
- Segmentation
- Paging
- Virtual Memory
- Page Replacement

What happens if there is no free frame?

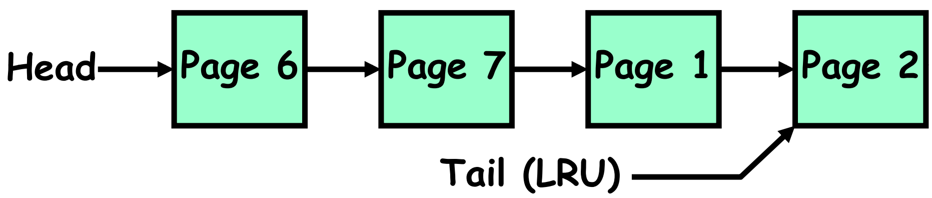
- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum, Optimal):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list!



  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Department of Electrical Engineering, Feng-Chia University

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

A	B	C	A	B	D	A	D	B	C	B
A	A	A	A	A	D	D	D	D	C	C
	B	B	B	B	B	A	A	A	A	A
		C	C	C	C	C	C	B	B	B

  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Department of Electrical Engineering, Feng-Chia University

Example: MIN (Optimal)

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

A	B	C	A	B	D	A	D	B	C	B
A	A	A	A	A	A	A	A	A	C	C
	B	B	B	B	B	B	B	B	B	B
		C	C	C	D	D	D	D	D	D

  - MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future.
- What will LRU do?
  - Same decisions as MIN here, but won't always be true!

Department of Electrical Engineering, Feng-Chia University

When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

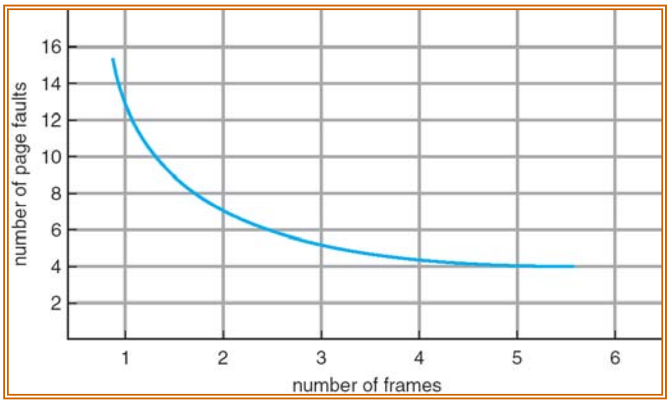
A	B	C	D	A	B	C	D	A	B	C	D
A	A	A	D	D	D	C	C	C	B	B	B
	B	B	B	A	A	A	D	D	D	C	C
		C	C	C	B	B	B	A	A	A	D

  - Every reference is a page fault!
- MIN Does much better:

A	B	C	D	A	B	C	D	A	B	C	D
A	A	A	A	A	A	A	A	A	B	B	B
	B	B	B	B	B	C	C	C	C	C	C
		C	D	D	D	D	D	D	D	D	D

Department of Electrical Engineering, Feng-Chia University

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?
- No: BeLady’s anomaly
  - Certain replacement algorithms (FIFO) don’t have this obvious property!

Department of Electrical Engineering, Feng-Chia University

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady's anomaly)

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

9 page faults

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3

10 page faults

Department of Electrical Engineering, Feng-Chia University

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- Thrashing ≡ a process is busy swapping pages in and out