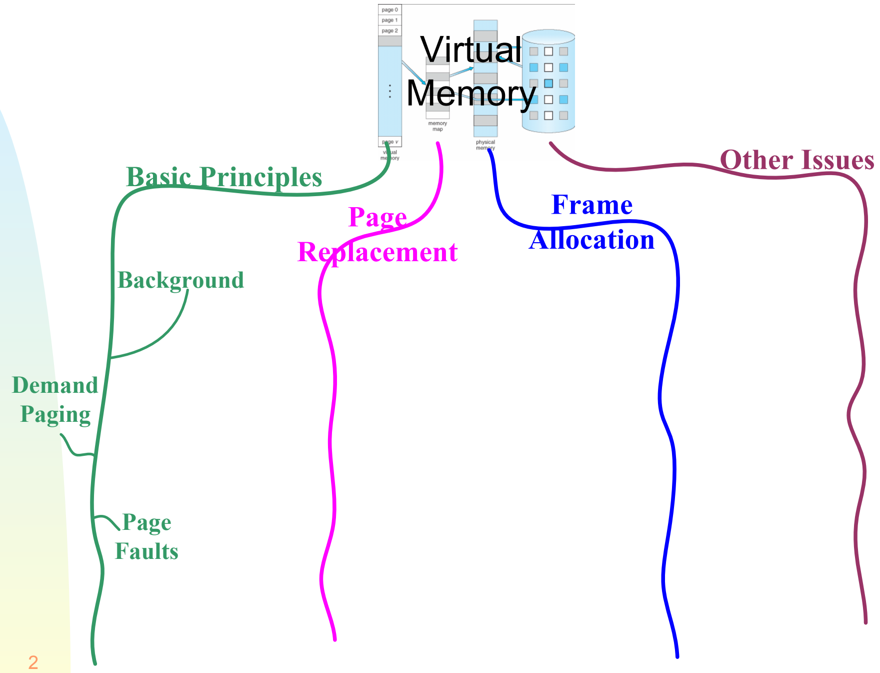
Module 8: Virtual Memory

Reading: Chapter 9

Objectives:

- **To describe the benefits of a virtual memory system.**
- **To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.**
- **To discuss the principles of working-set model.**



Virtual Memory

Basic Principles

Background

Demand Paging

Page Faults

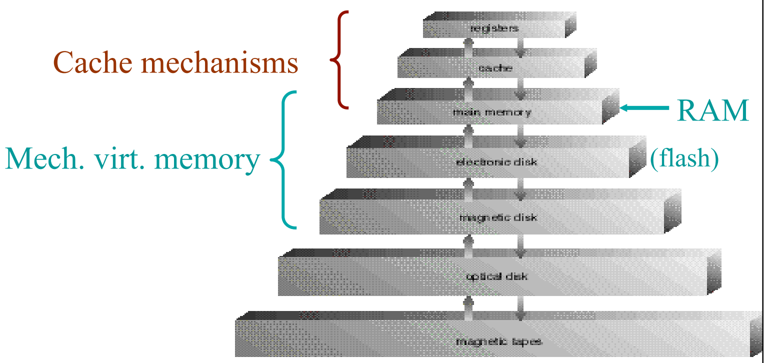
Page Replacement

Frame Allocation

Other Issues

Virtual Memory and Memory Hierarchy

- Mechanisms used with virtual memory are similar to those used for cache memory
  - But for cache memory – implemented in hardware



From Paging and Segmentation to Virtual Memory

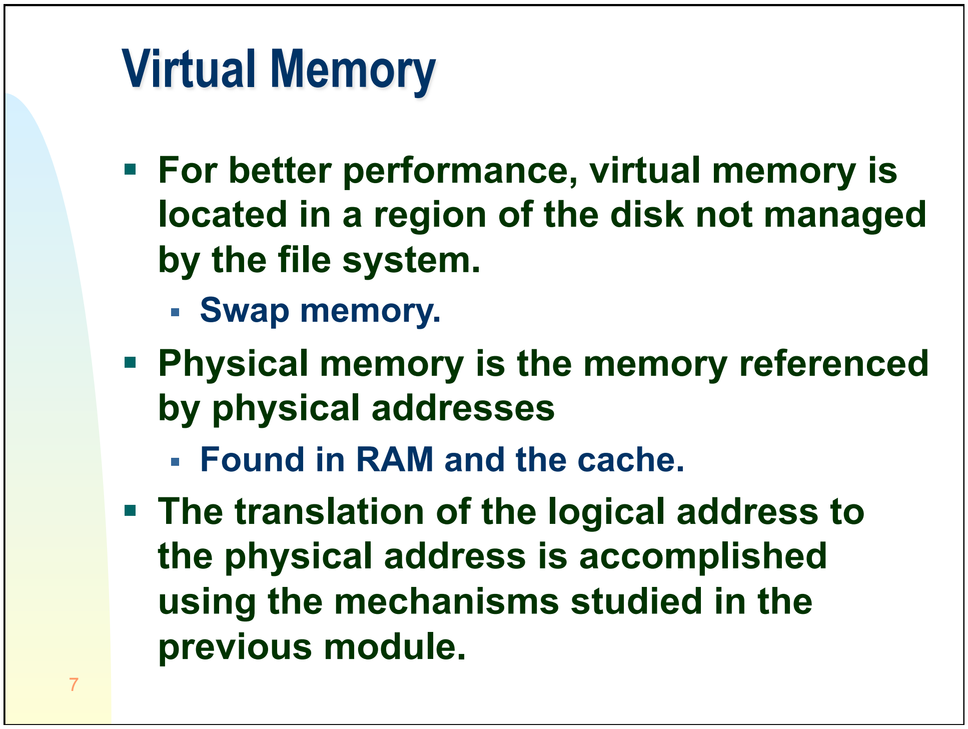
- A process is made up of pieces (pages or segments) that do not need to occupy continuous region in main memory.
- References to memory are translated to physical addresses during execution.
  - A process can be moved to different regions in memory, and in secondary memory!
- all pieces of a process need not be in main memory during execution.
  - Execution can continue under the condition that the next instruction (or data) is in a piece found in main memory.
- The sum of logical memory in processes can exceed the physical memory available
  - This is a basic concept of virtual memory.
- An image of all address space of the process is kept in secondary memory (normally the disk) where missing pages can be accessed when needed
  - Swapping mechanism.

Advantages of Partial Loading

- More processes can be maintained in memory for execution
  - Only a few pieces are loaded for each process.
  - The user can be happy, since he/she can execute many processes and reference large data structures without worrying about filling main memory.
  - With many processes in memory, more probable to have a process in the ready state, which increases utilization of the CPU.
- Many pages or segments rarely used need not be loaded at all into memory.
- It is now possible to execute a set of processes even if their sizes exceed the main memory
  - It is possible to use more bits in the logical address than the number of bits used for addressing main memory.
  - Logical address space > > physical address space.

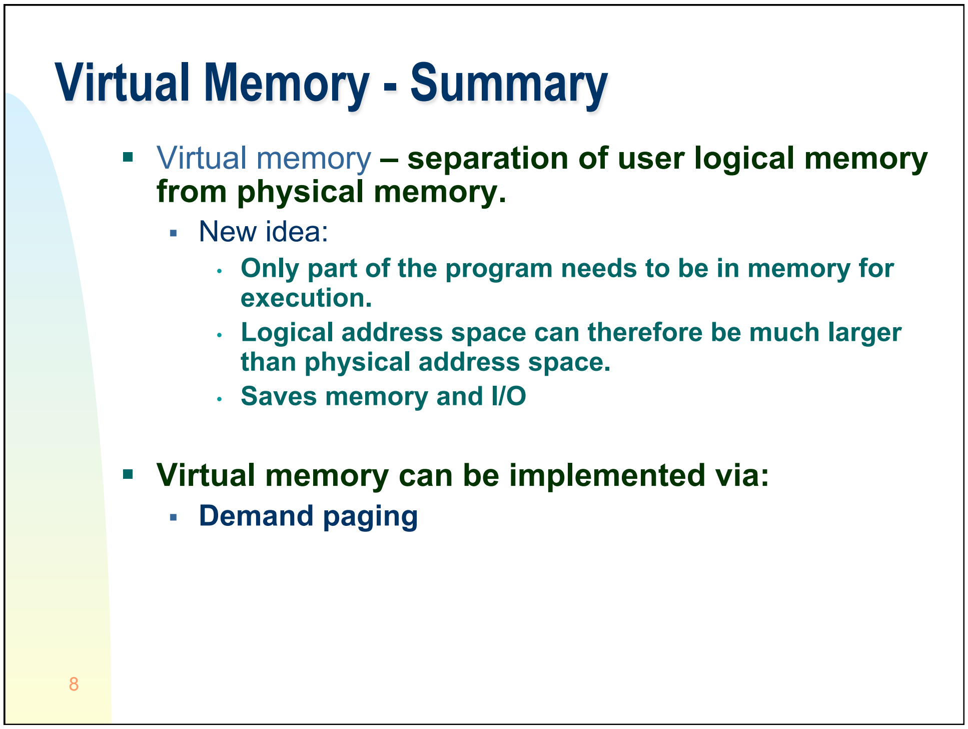
Virtual Memory: Can Be LARGE!

- Ex: 16 bits are required to address physical memory of size 64KB
- Using 1KB pages, 10 bits are required for offset.
- For the *page number* of the logical address, it is possible to use more than 6 bits, since not all pages need be in memory.
- Thus the limit of virtual memory is defined by the number of bits that can be reserved for the address
  - In some architectures, these bits can be included in registers.
- Logical memory is thus called *virtual memory.*
  - Is maintained in secondary memory
  - Pieces are brought into main memory only when necessary, on demand.



Virtual Memory

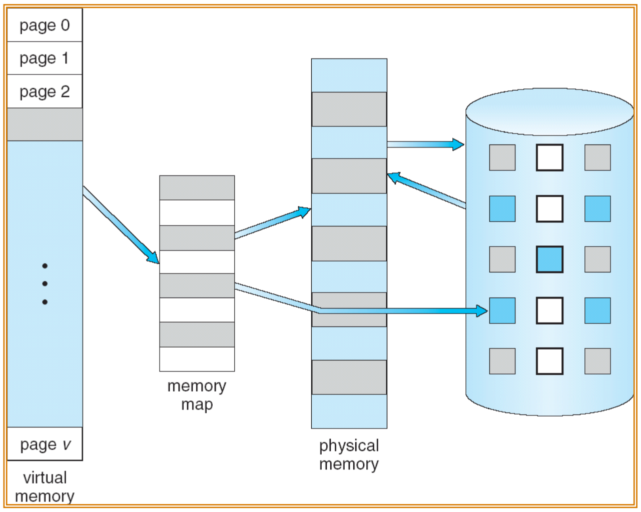
- For better performance, virtual memory is located in a region of the disk not managed by the file system.
  - Swap memory.
- Physical memory is the memory referenced by physical addresses
  - Found in RAM and the cache.
- The translation of the logical address to the physical address is accomplished using the mechanisms studied in the previous module.



Virtual Memory - Summary

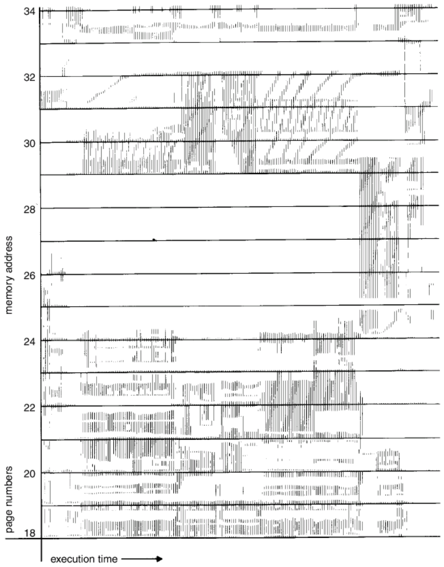
- Virtual memory – separation of user logical memory from physical memory.
  - New idea:
    - Only part of the program needs to be in memory for execution.
    - Logical address space can therefore be much larger than physical address space.
    - Saves memory and I/O
- Virtual memory can be implemented via:
  - Demand paging

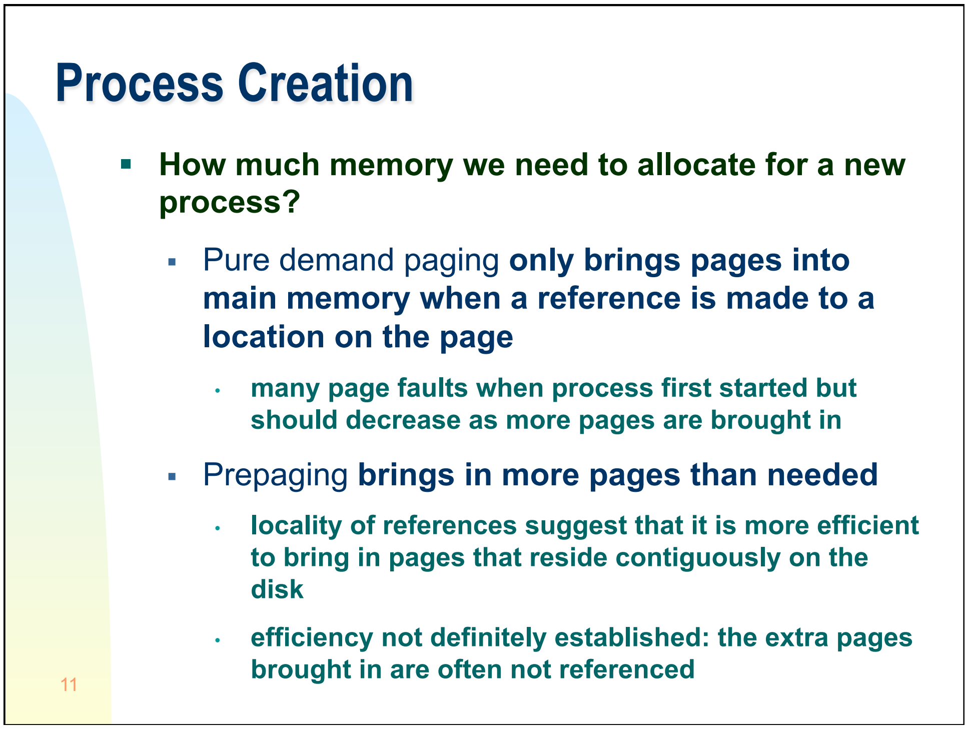
Virtual Memory That is Larger Than Physical Memory



Locality and Virtual Memory

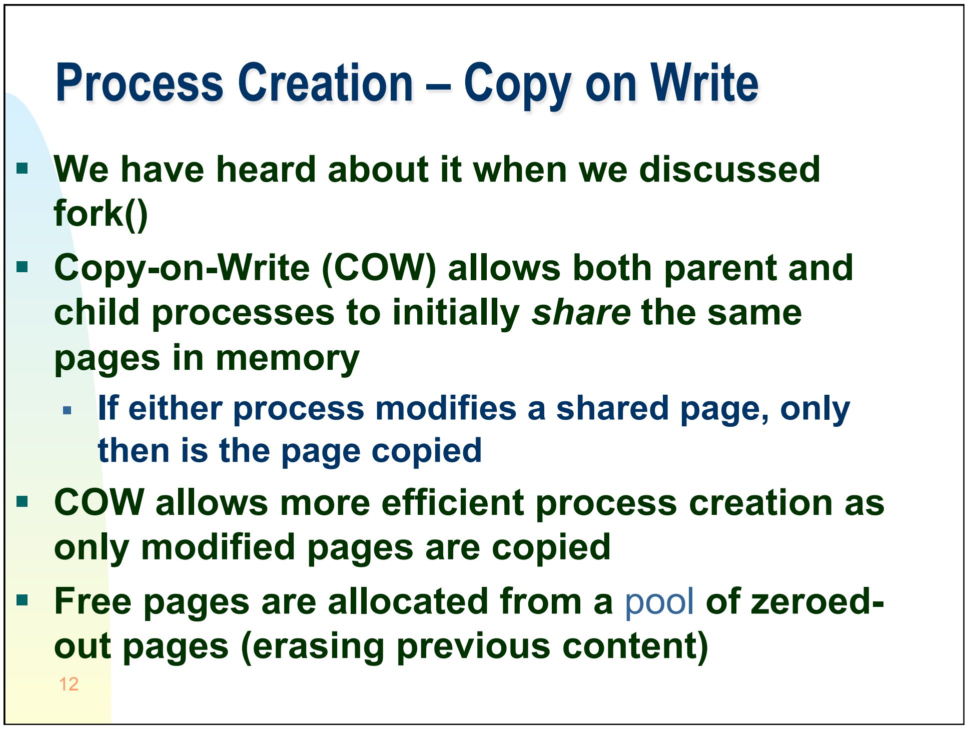
- The locality of references: references to memory by a process tend to be grouped.
- Thus: only a few pieces of the process are used during a small period of time (pieces: pages or segments).





Process Creation

- **How much memory we need to allocate for a new process?**
  - Pure demand paging **only brings pages into main memory when a reference is made to a location on the page**
    - **many page faults when process first started but should decrease as more pages are brought in**
  - Prepaging **brings in more pages than needed**
    - **locality of references suggest that it is more efficient to bring in pages that reside contiguously on the disk**
    - **efficiency not definitely established: the extra pages brought in are often not referenced**



Process Creation – Copy on Write

- **We have heard about it when we discussed fork()**
- **Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory**
  - **If either process modifies a shared page, only then is the page copied**
- **COW allows more efficient process creation as only modified pages are copied**
- **Free pages are allocated from a** pool **of zeroed-out pages (erasing previous content)**

How do we know a page is in memory?

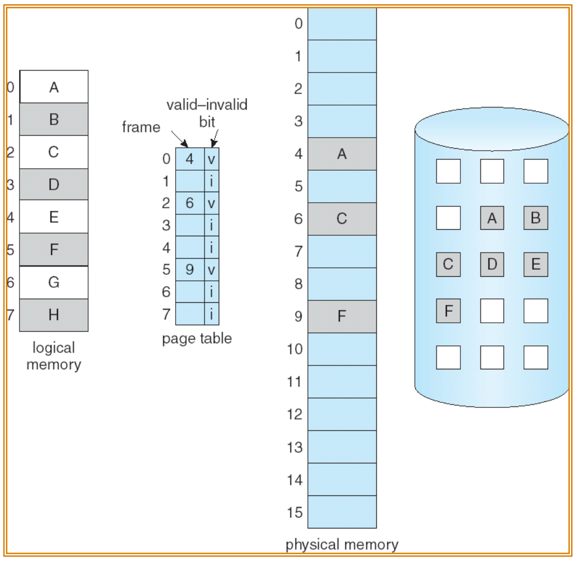
- **With each page table entry a** valid/invalid **(also** present/not-present**) bit is associated**
  - **1 ⇒ in-memory, 0 ⇒ not-in-memory**
- **Initially valid–invalid bit is set to 0 on all entries**
- **Example of a page table snapshot:**

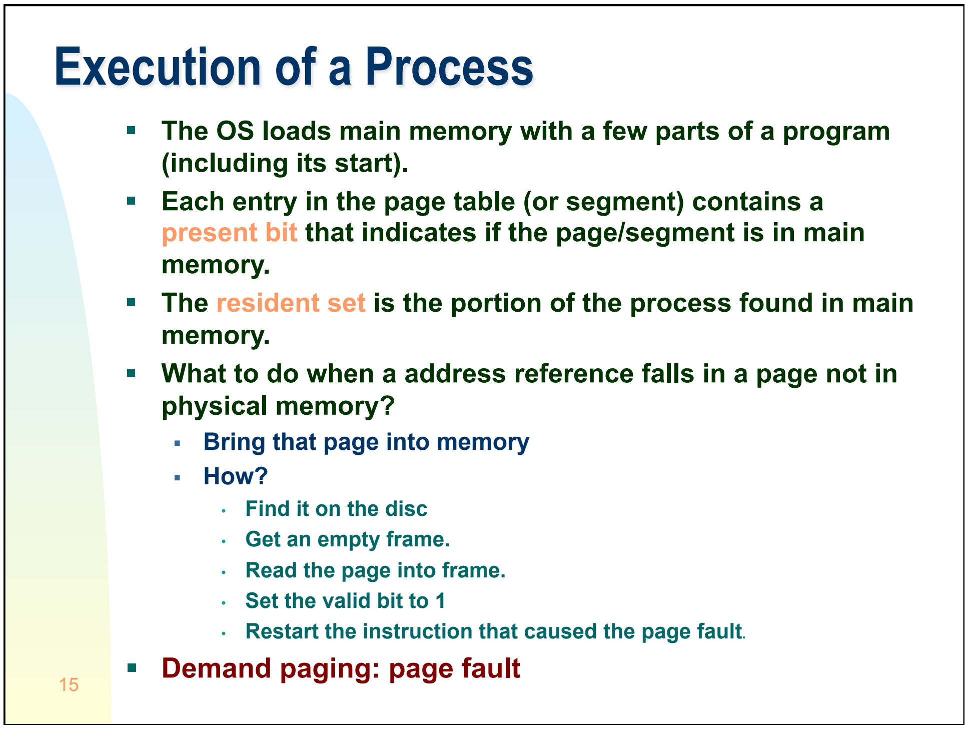
Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

- **During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault**

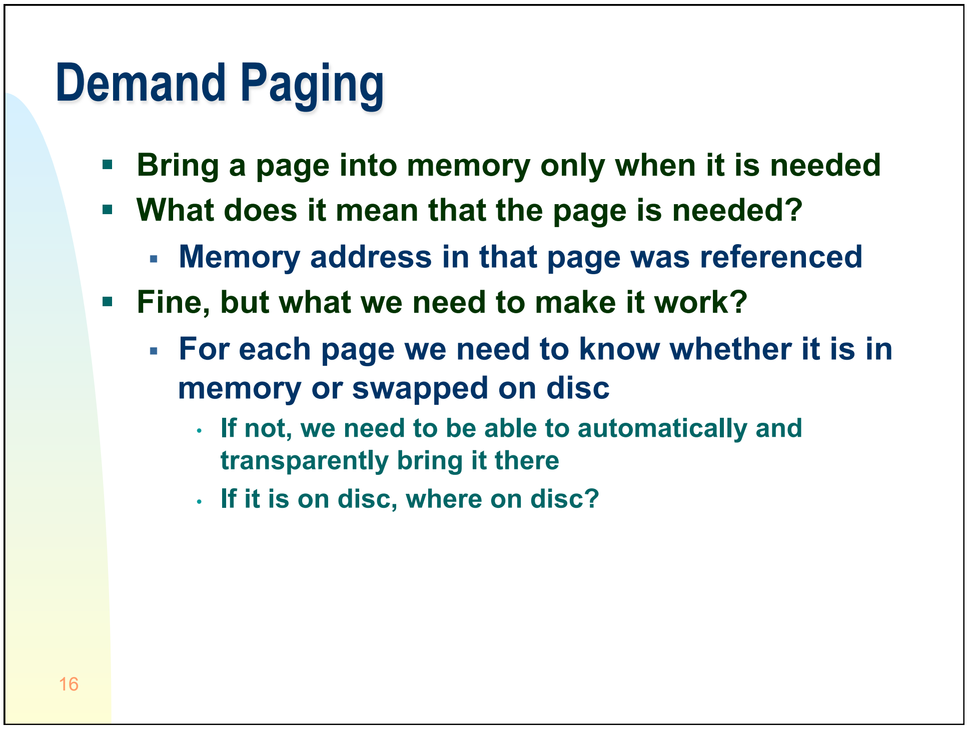
Page Table When Some Pages Are Not in Main Memory





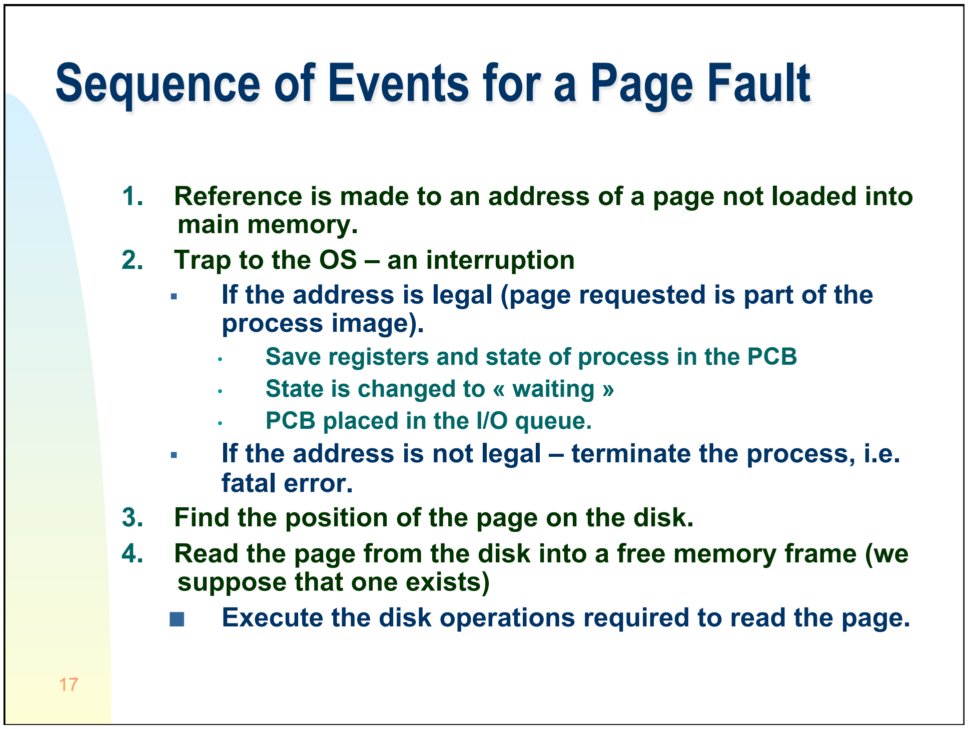
Execution of a Process

- The OS loads main memory with a few parts of a program (including its start).
- Each entry in the page table (or segment) contains a present bit that indicates if the page/segment is in main memory.
- The resident set is the portion of the process found in main memory.
- What to do when a address reference falls in a page not in physical memory?
  - Bring that page into memory
  - How?
    - Find it on the disc
    - Get an empty frame.
    - Read the page into frame.
    - Set the valid bit to 1
    - Restart the instruction that caused the page fault.
- Demand paging: page fault



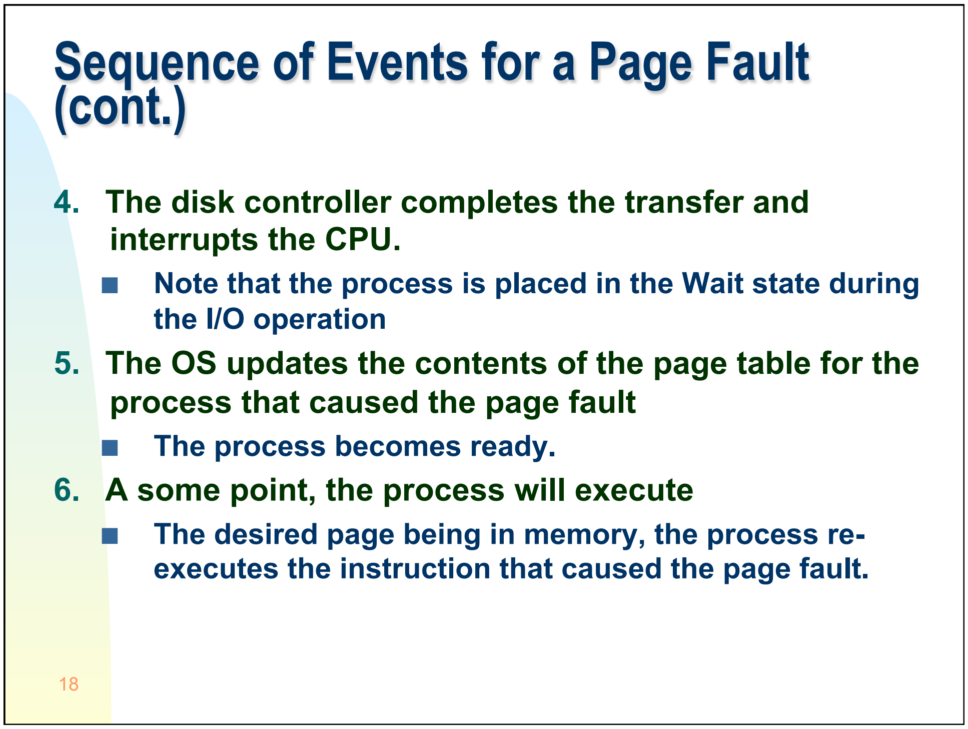
Demand Paging

- Bring a page into memory only when it is needed
- What does it mean that the page is needed?
  - Memory address in that page was referenced
- Fine, but what we need to make it work?
  - For each page we need to know whether it is in memory or swapped on disc
    - If not, we need to be able to automatically and transparently bring it there
    - If it is on disc, where on disc?



Sequence of Events for a Page Fault

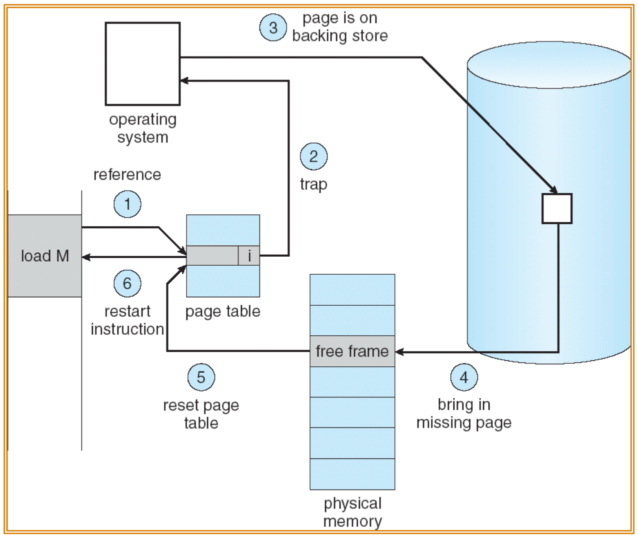
1. Reference is made to an address of a page not loaded into main memory.
2. Trap to the OS – an interruption
   - If the address is legal (page requested is part of the process image).
     - Save registers and state of process in the PCB
     - State is changed to « waiting »
     - PCB placed in the I/O queue.
   - If the address is not legal – terminate the process, i.e. fatal error.
3. Find the position of the page on the disk.
4. Read the page from the disk into a free memory frame (we suppose that one exists)
   - Execute the disk operations required to read the page.



Sequence of Events for a Page Fault (cont.)

4. The disk controller completes the transfer and interrupts the CPU.
   - Note that the process is placed in the Wait state during the I/O operation
5. The OS updates the contents of the page table for the process that caused the page fault
   - The process becomes ready.
6. A some point, the process will execute
   - The desired page being in memory, the process re-executes the instruction that caused the page fault.

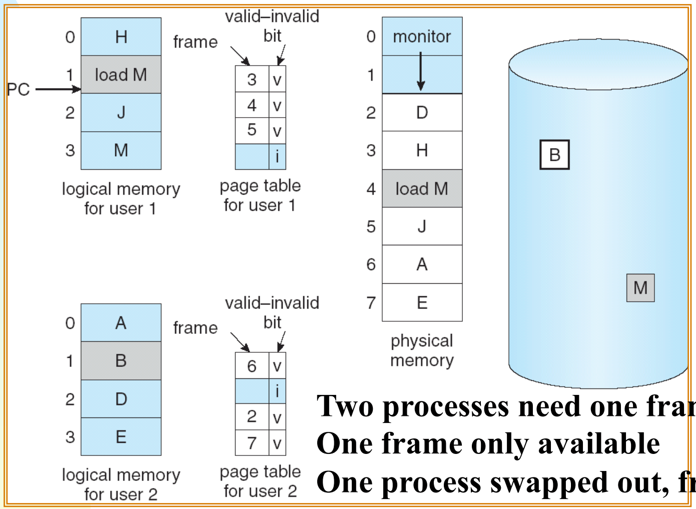
Steps in Handling a Page Fault



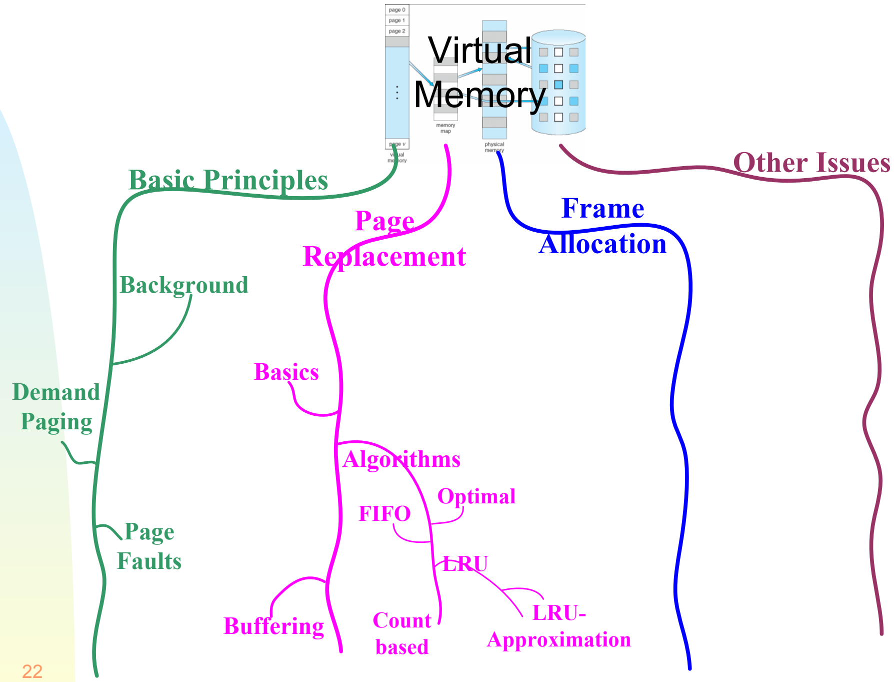
What happens if there is no free frame?

- Find a page to replace (*page replacement*)
  - Swap out the replaced page
- How to chose which page to replace?
  - We will see several algorithms
- Does is matter which algorithm we choose?
  - Very much so…
    - With memory access time cca 200ns
    - Page replacement time is determined by swapping, around 8 ms
    - Page fault rate = p (proportional)
    - Effective access time = (1-p)x200+px8000000

Need For Page Replacement



Two processes need one frame each
One frame only available
One process swapped out, freeing all its frames



Virtual Memory

- Basic Principles
  - Background
  - Demand Paging
  - Page Faults
- Page Replacement
  - Basics
  - Algorithms
    - FIFO
    - Optimal
    - LRU
    - LRU-Approximation
    - Count based
  - Buffering
- Frame Allocation
- Other Issues

Basic Page Replacement

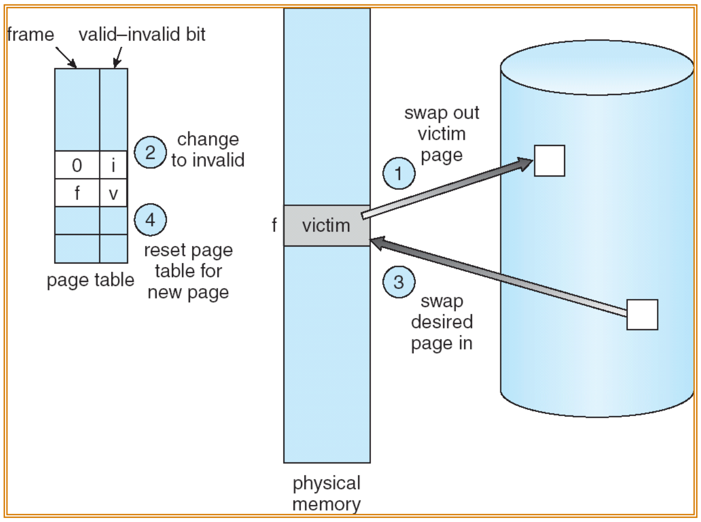
1. Find the location of the desired page on disk

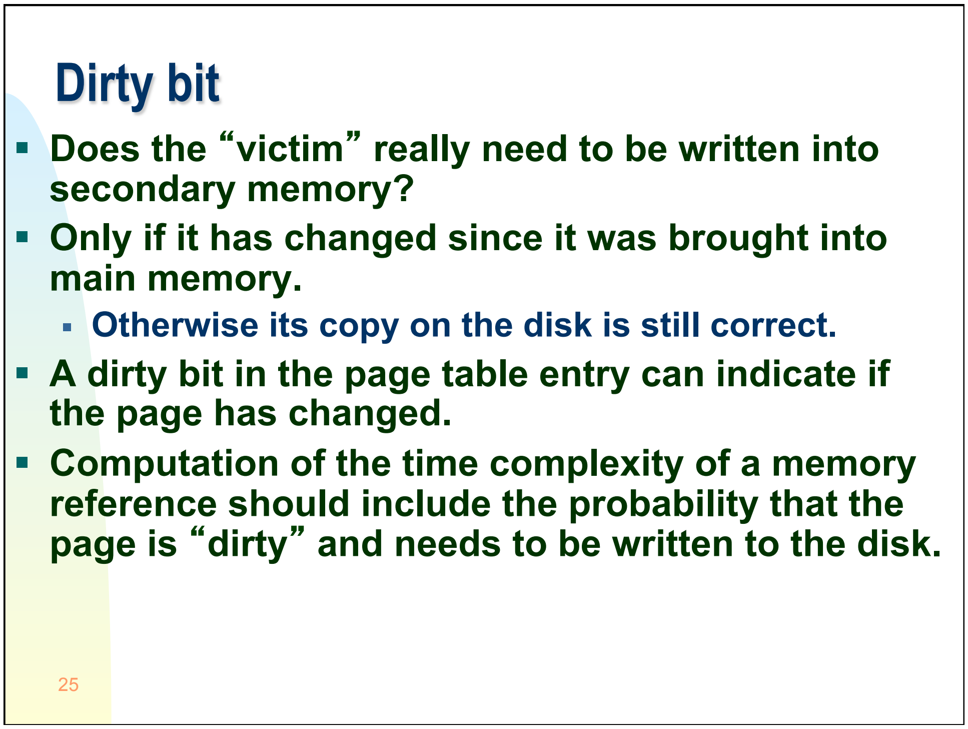
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process

Page Replacement

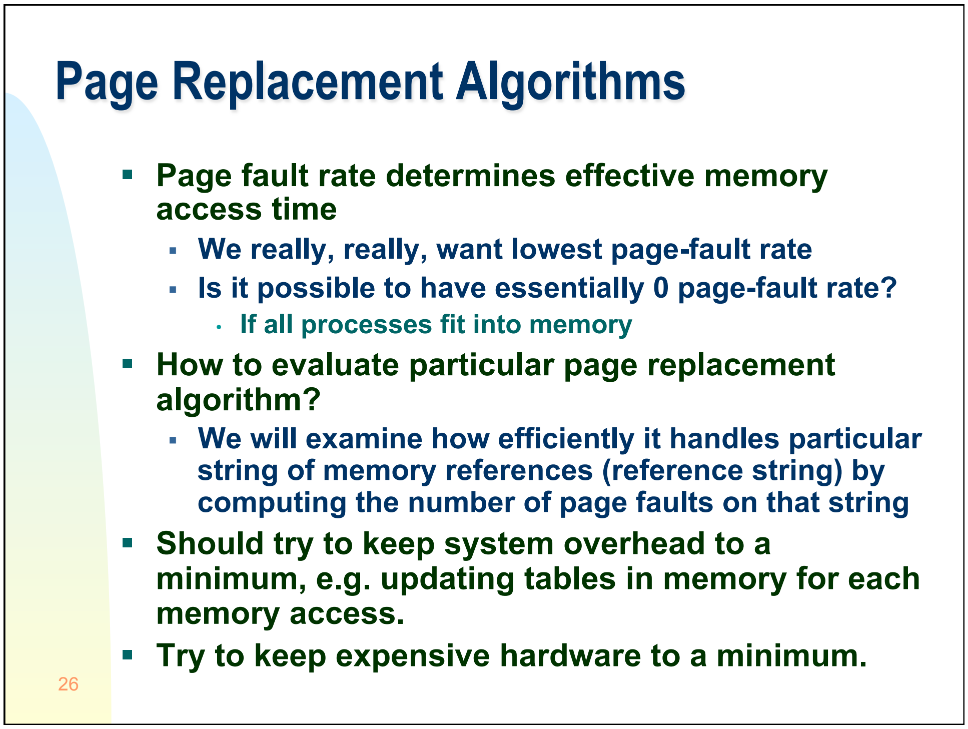




Dirty bit

- Does the "victim" really need to be written into secondary memory?
- Only if it has changed since it was brought into main memory.
  - Otherwise its copy on the disk is still correct.
- A dirty bit in the page table entry can indicate if the page has changed.
- Computation of the time complexity of a memory reference should include the probability that the page is "dirty" and needs to be written to the disk.

25



Page Replacement Algorithms

- Page fault rate determines effective memory access time
  - We really, really, want lowest page-fault rate
  - Is it possible to have essentially 0 page-fault rate?
    - If all processes fit into memory
- How to evaluate particular page replacement algorithm?
  - We will examine how efficiently it handles particular string of memory references (reference string) by computing the number of page faults on that string
- Should try to keep system overhead to a minimum, e.g. updating tables in memory for each memory access.
- Try to keep expensive hardware to a minimum.

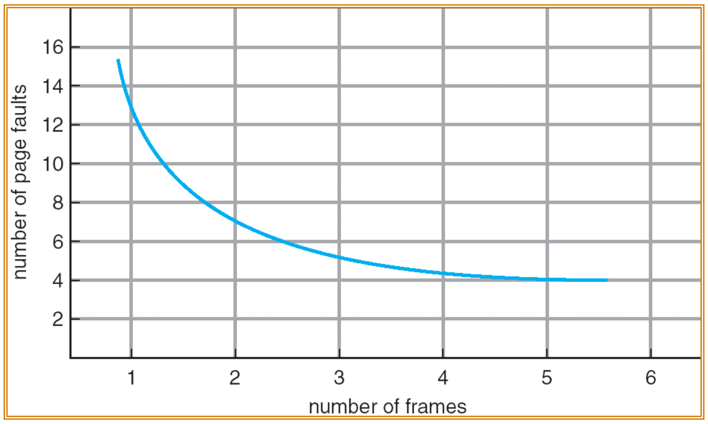
26

Page Replacement Algorithms

- Which pages to consider for replacement?
  - Can we replace kernel pages?
- Not all pages in main memory can be selected for replacement
- Some frames are locked (cannot be paged out):
  - much of the kernel is held on locked frames as well as key control structures and I/O buffers
- OK, which unlocked pages to consider for replacement?
  - Only those processes that have suffered the page fault
  - Consider all unlocked pages
  - related to the resident set management strategy:
    - how many page frames are to be allocated to each process? We will discuss this later

Graph of Page Faults Versus The Number of Frames

More frames we allocate to the process, less page faults will occur



Presentation and Evaluation of Algorithms

- Shall present and evaluate algorithms using the chains of page references such as the following:

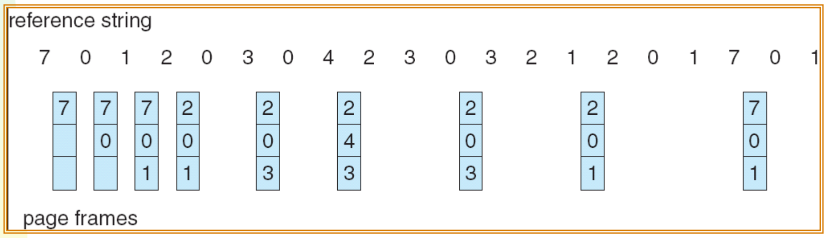
  2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

- Careful: such sequences are not random
  - Remember locality of reference.
- Evaluation made using such examples are not really sufficient to justify our general conclusions

OPT Algorithm

- Good news:
  - There is an optimal algorithm that produces the fewest number of page faults
  - Any idea how it might work?
    - selects for replacement the page for which the time to the next reference is the longest



reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2	2	2	2	2	7
	0	0	0	0	4	0	0	0
		1	1	3	3	3	1	1

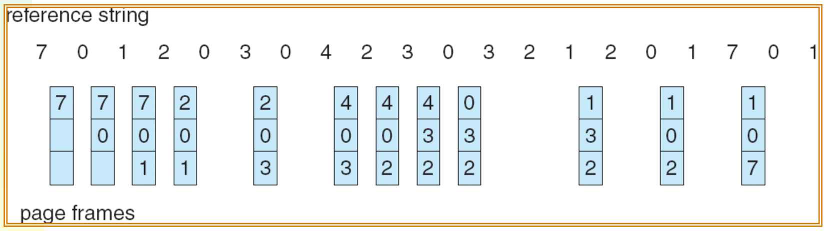
page frames

OPT Algorithm

- **Bad news:**
  - **impossible to implement (need to know the future)**
  - **Still useful: serves as a standard to compare with the other algorithms we shall study:**
    - **Least recently used (LRU)**
    - **First-in, first-out (FIFO)**
    - **Clock**
    - **Counting algorithms**

The LRU Algorithm

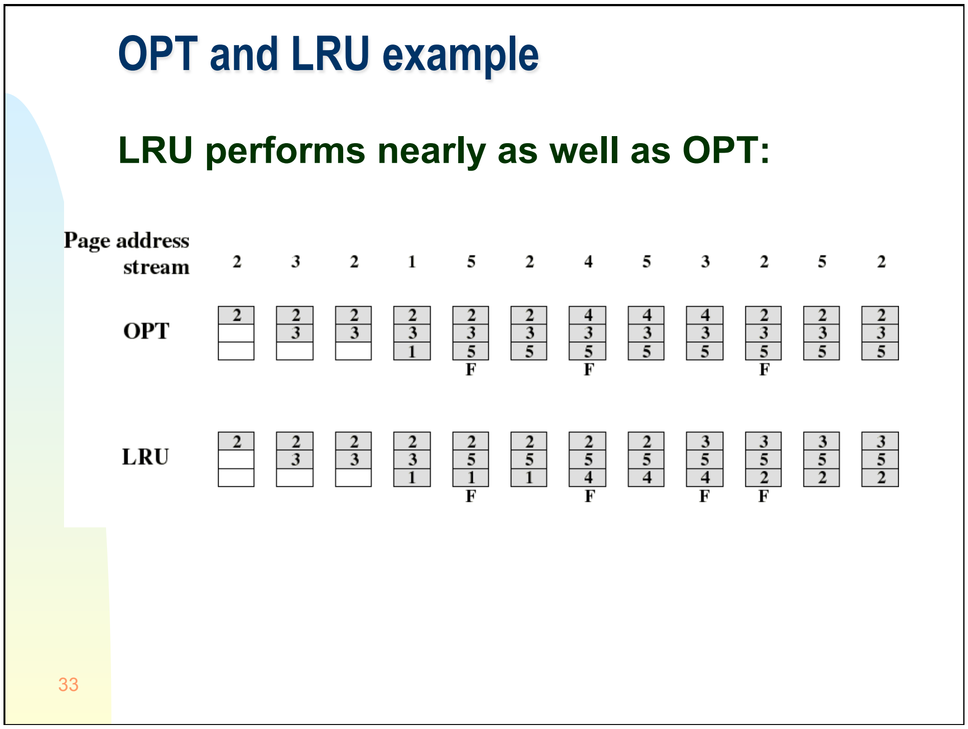
- **Least Recently Used**
- **Replaces the page that has not been referenced for the longest time**
  - **By the principle of locality, this should be the page least likely to be referenced in the near future**
  - **performs nearly as well as the optimal policy**



reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

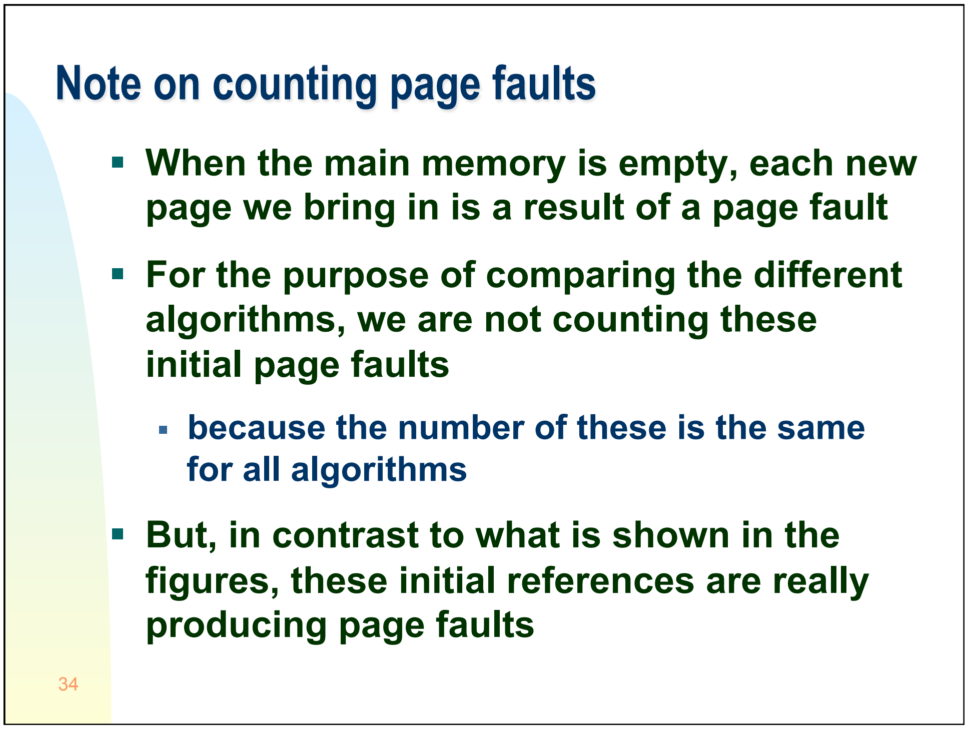
page frames



OPT and LRU example

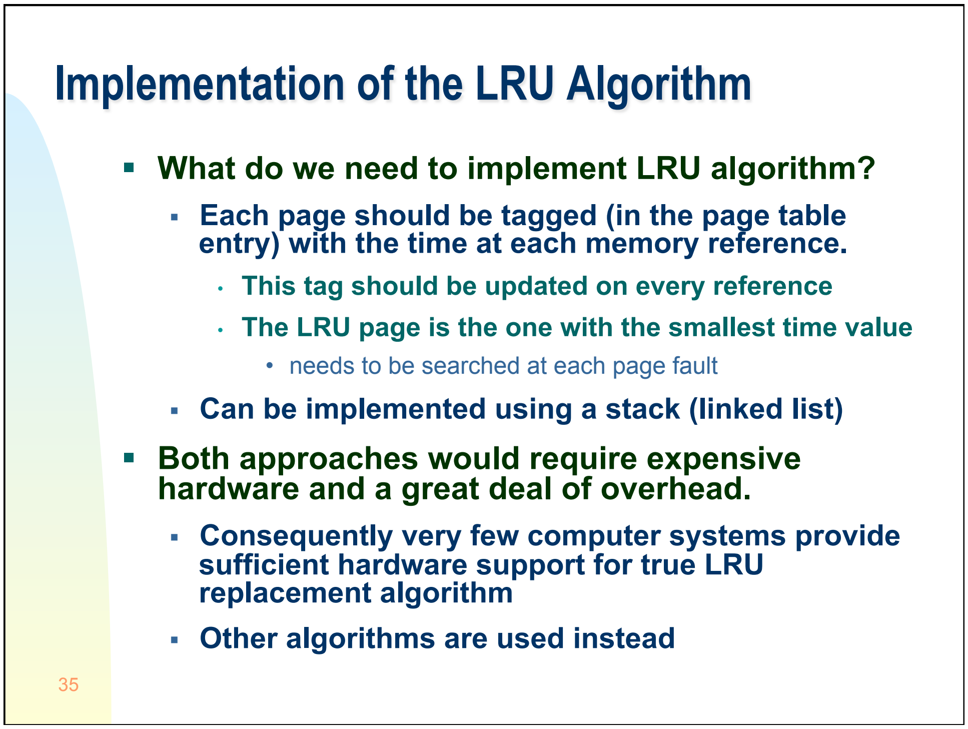
LRU performs nearly as well as OPT:

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2 3	2 3	2 3 1	2 3 5 F	2 3 5	4 3 5 F	4 3 5	4 3 5	2 3 5 F	2 3 5	2 3 5
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2



Note on counting page faults

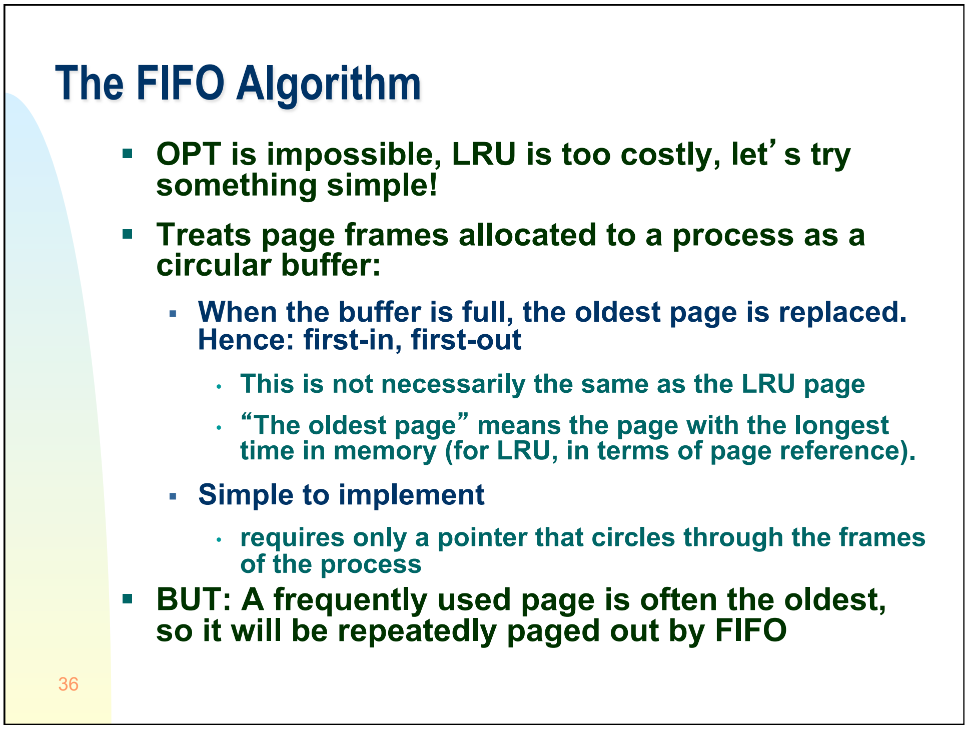
- When the main memory is empty, each new page we bring in is a result of a page fault
- For the purpose of comparing the different algorithms, we are not counting these initial page faults
  - because the number of these is the same for all algorithms
- But, in contrast to what is shown in the figures, these initial references are really producing page faults



Implementation of the LRU Algorithm

- What do we need to implement LRU algorithm?
  - Each page should be tagged (in the page table entry) with the time at each memory reference.
    - This tag should be updated on every reference
    - The LRU page is the one with the smallest time value
      - needs to be searched at each page fault
  - Can be implemented using a stack (linked list)
- Both approaches would require expensive hardware and a great deal of overhead.
  - Consequently very few computer systems provide sufficient hardware support for true LRU replacement algorithm
  - Other algorithms are used instead

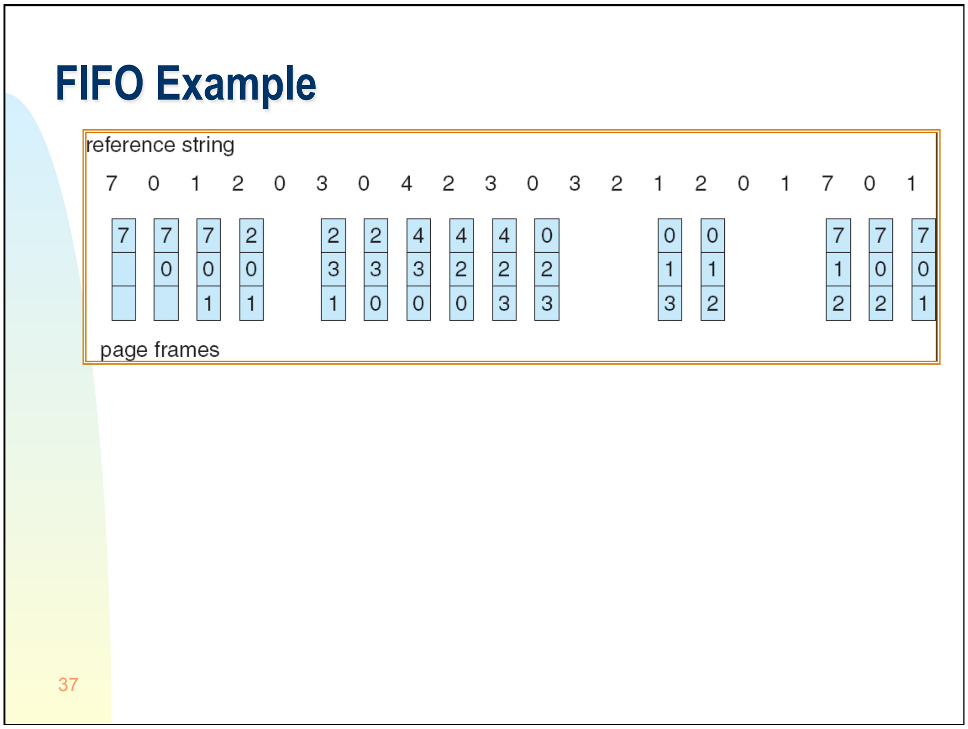
35



The FIFO Algorithm

- OPT is impossible, LRU is too costly, let’ s try something simple!
- Treats page frames allocated to a process as a circular buffer:
  - When the buffer is full, the oldest page is replaced. Hence: first-in, first-out
    - This is not necessarily the same as the LRU page
    - “The oldest page” means the page with the longest time in memory (for LRU, in terms of page reference).
  - Simple to implement
    - requires only a pointer that circles through the frames of the process
- BUT: A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO

36

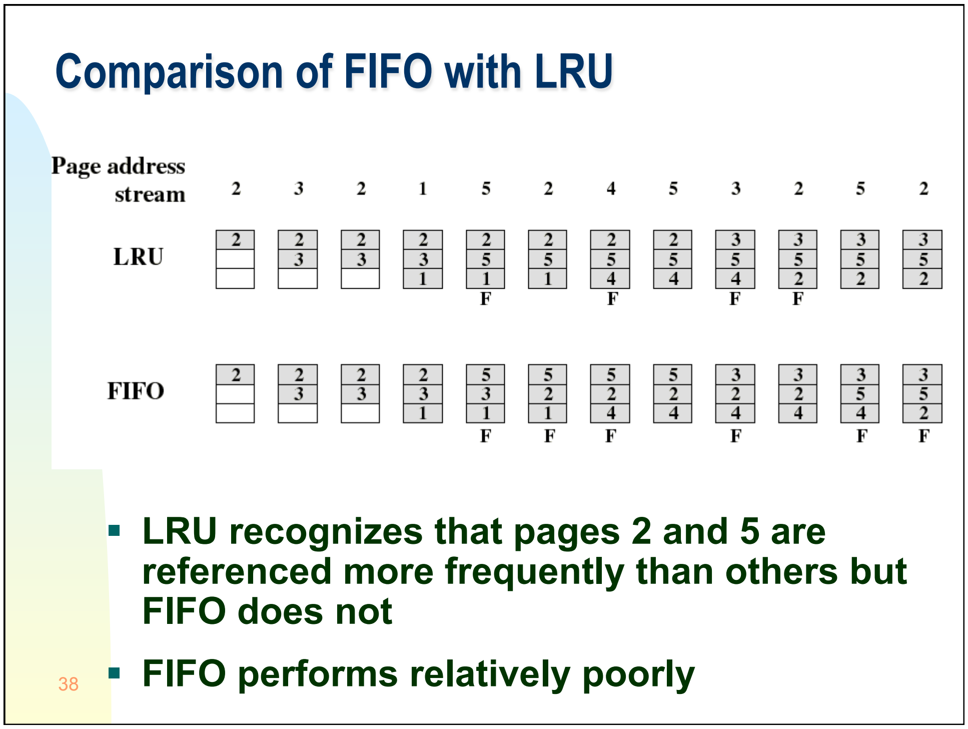


FIFO Example

reference string

page frames

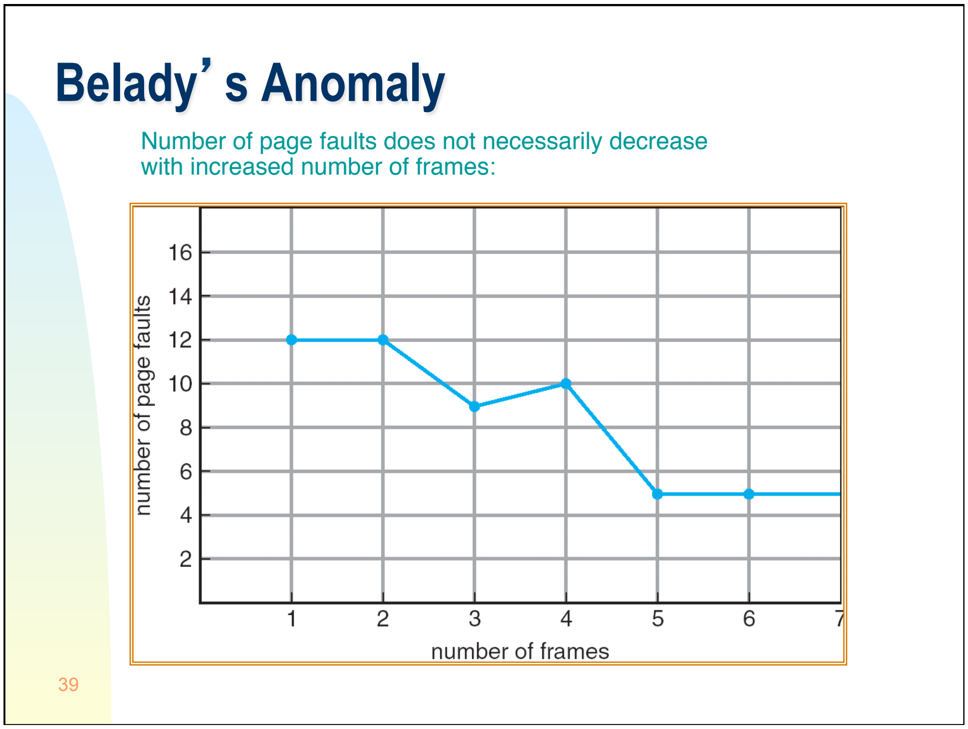
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1



Comparison of FIFO with LRU

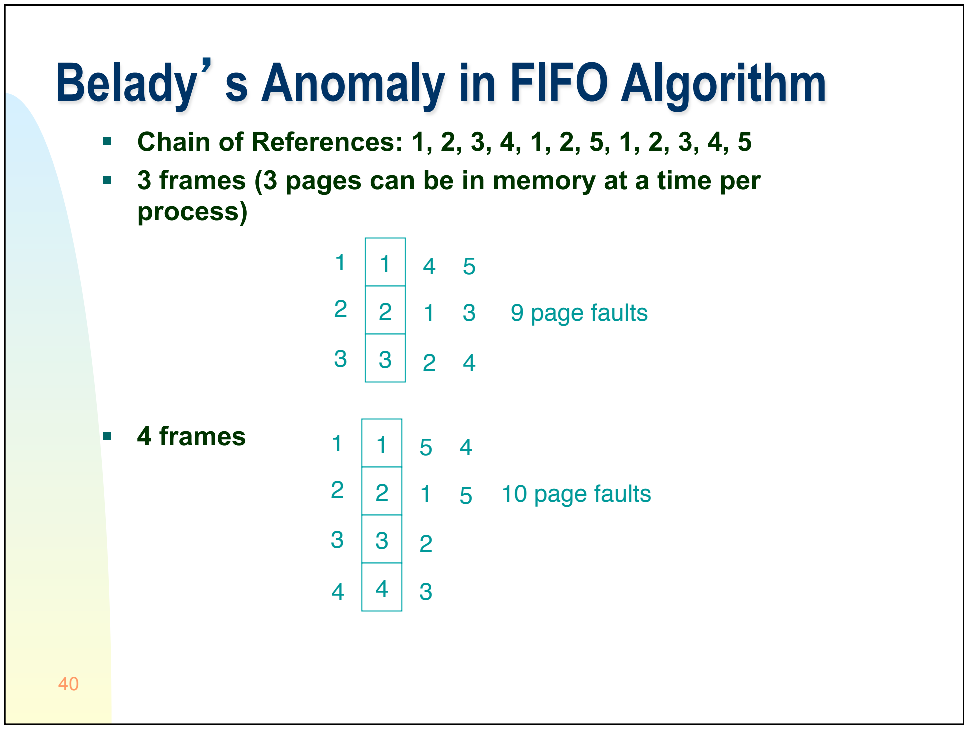
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F

- LRU recognizes that pages 2 and 5 are referenced more frequently than others but FIFO does not
- FIFO performs relatively poorly



Belady’s Anomaly

Number of page faults does not necessarily decrease with increased number of frames:



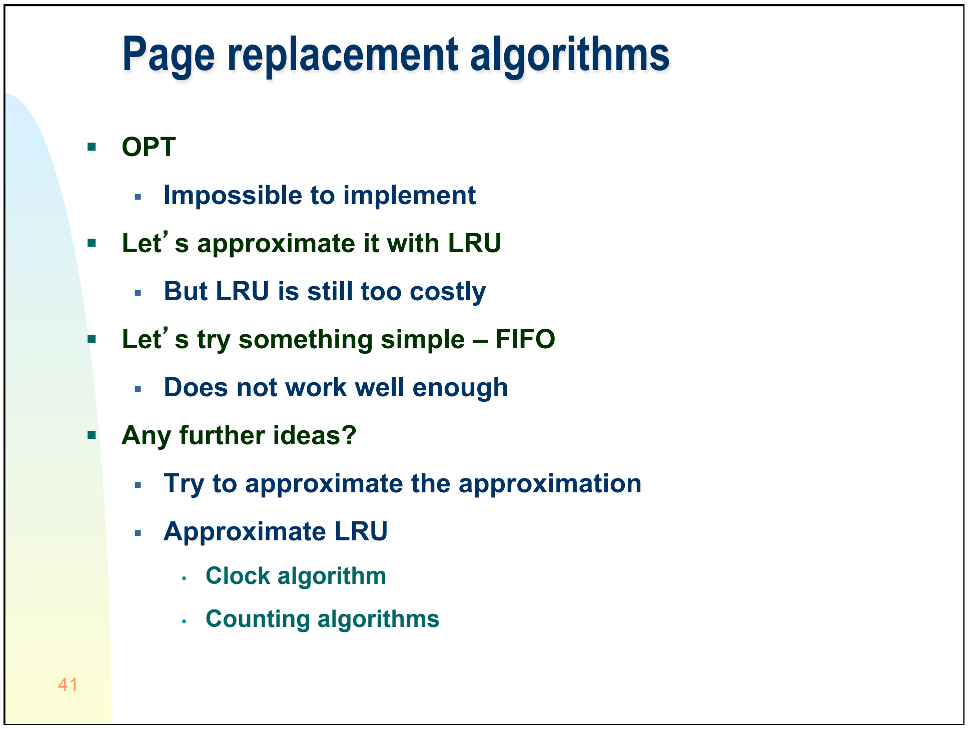
Belady’s Anomaly in FIFO Algorithm

- **Chain of References: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **3 frames (3 pages can be in memory at a time per process)**

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

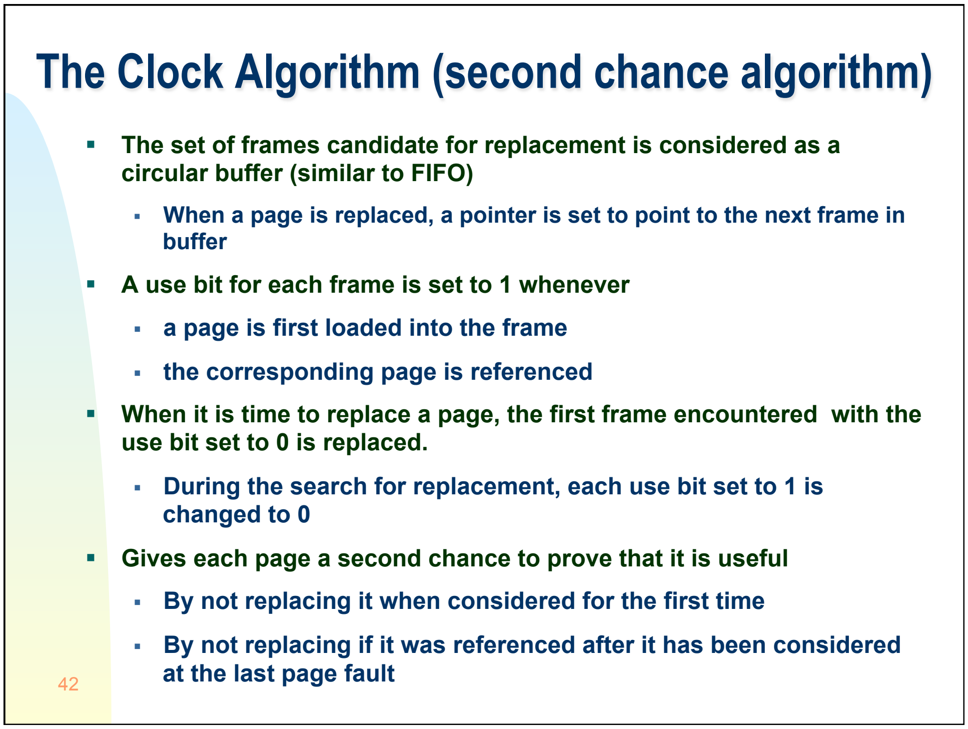
- **4 frames**

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		



Page replacement algorithms

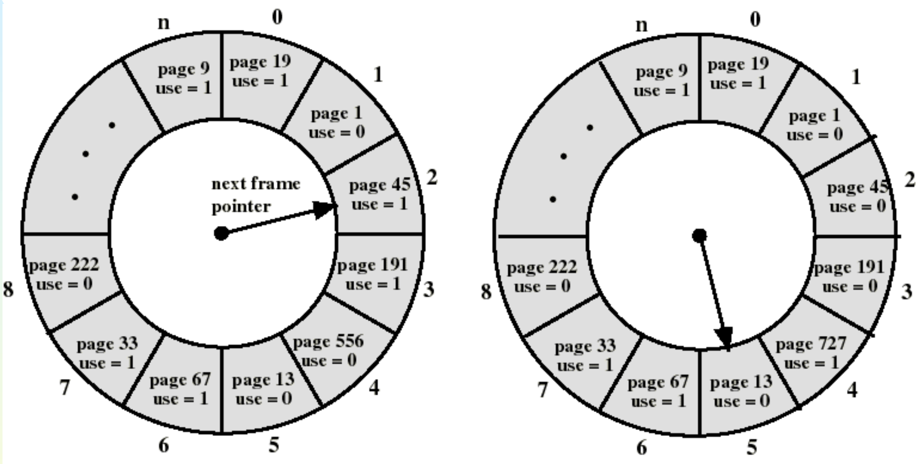
- OPT
  - Impossible to implement
- Let' s approximate it with LRU
  - But LRU is still too costly
- Let' s try something simple – FIFO
  - Does not work well enough
- Any further ideas?
  - Try to approximate the approximation
  - Approximate LRU
    - Clock algorithm
    - Counting algorithms



The Clock Algorithm (second chance algorithm)

- The set of frames candidate for replacement is considered as a circular buffer (similar to FIFO)
  - When a page is replaced, a pointer is set to point to the next frame in buffer
- A use bit for each frame is set to 1 whenever
  - a page is first loaded into the frame
  - the corresponding page is referenced
- When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.
  - During the search for replacement, each use bit set to 1 is changed to 0
- Gives each page a second chance to prove that it is useful
  - By not replacing it when considered for the first time
  - By not replacing if it was referenced after it has been considered at the last page fault

The Clock Policy: an example



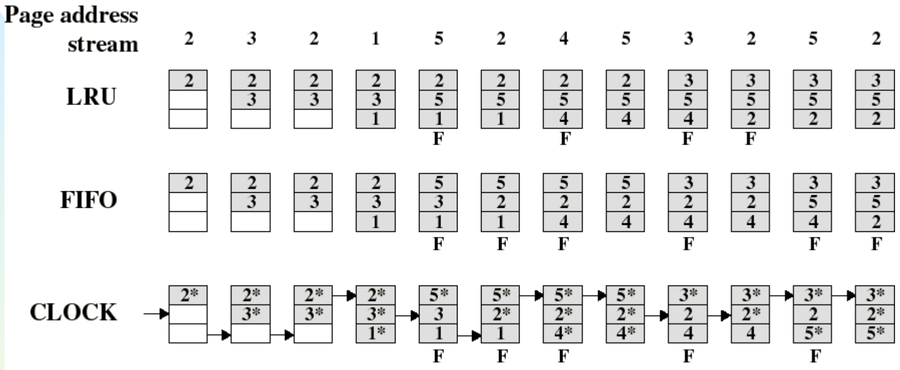
(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

Page 727 is loaded into frame 4.

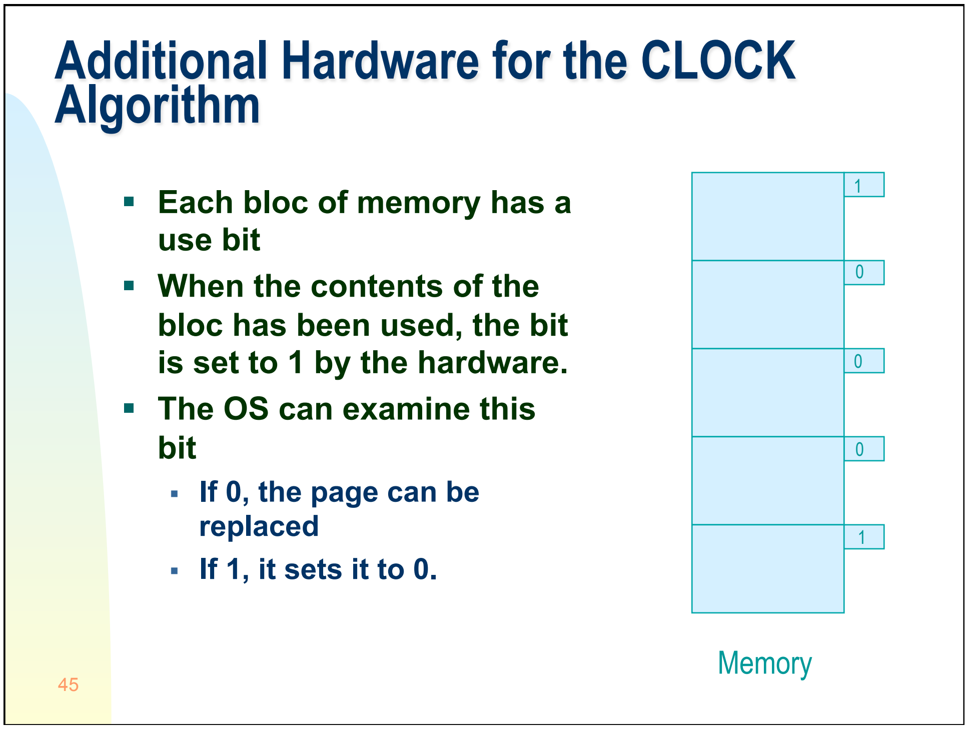
The next victim is 5, and then 8 (unless referenced).

Comparison of Clock with FIFO and LRU



Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2
FIFO	2	2 3	2 3	2 3 1	5 3 1 F	5 2 1 F	5 2 4 F	5 2 4	3 2 4 F	3 2 4	3 5 4 F	3 5 2 F
CLOCK	2*	2* 3*	2* 3*	2* 3* 1*	5* 3 1 F	5* 2* 1 F	5* 2* 4* F	5* 2* 4*	3* 2 4 F	3* 2* 4	3* 2 5* F	3* 2* 5*

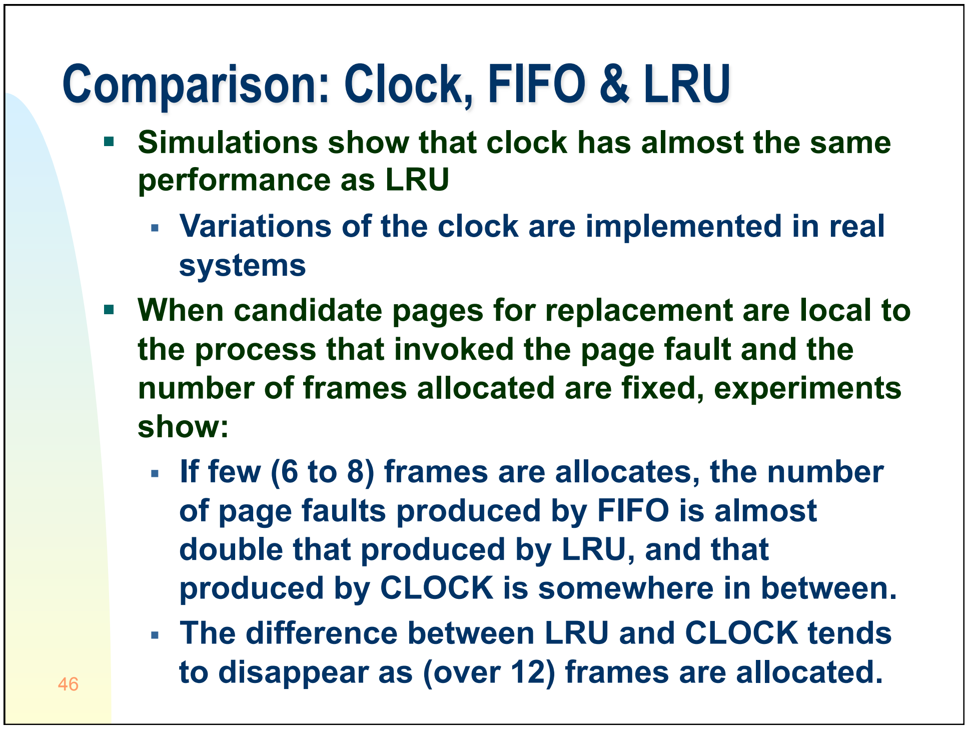
- Asterisk indicates that the corresponding use bit is set to 1
- Clock protects frequently referenced pages by setting the use bit to 1 at each reference
- LRU = 3+4, FIFO = 3+6, Clock = 3+5



Additional Hardware for the CLOCK Algorithm

- Each bloc of memory has a use bit
- When the contents of the bloc has been used, the bit is set to 1 by the hardware.
- The OS can examine this bit
  - If 0, the page can be replaced
  - If 1, it sets it to 0.

1
0
0
0
1
Memory



Comparison: Clock, FIFO & LRU

- Simulations show that clock has almost the same performance as LRU
  - Variations of the clock are implemented in real systems
- When candidate pages for replacement are local to the process that invoked the page fault and the number of frames allocated are fixed, experiments show:
  - If few (6 to 8) frames are allocates, the number of page faults produced by FIFO is almost double that produced by LRU, and that produced by CLOCK is somewhere in between.
  - The difference between LRU and CLOCK tends to disappear as (over 12) frames are allocated.

Comparing Algorithms (Stallings)

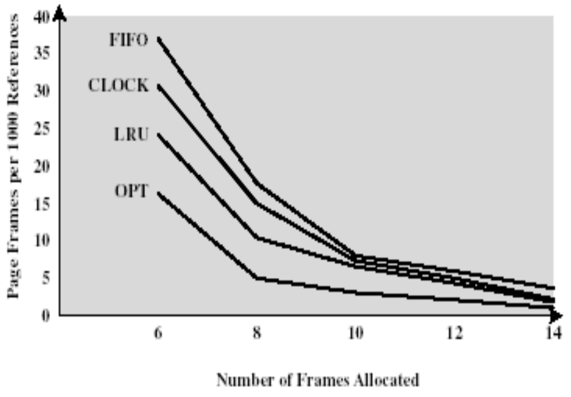


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

Counting Algorithms

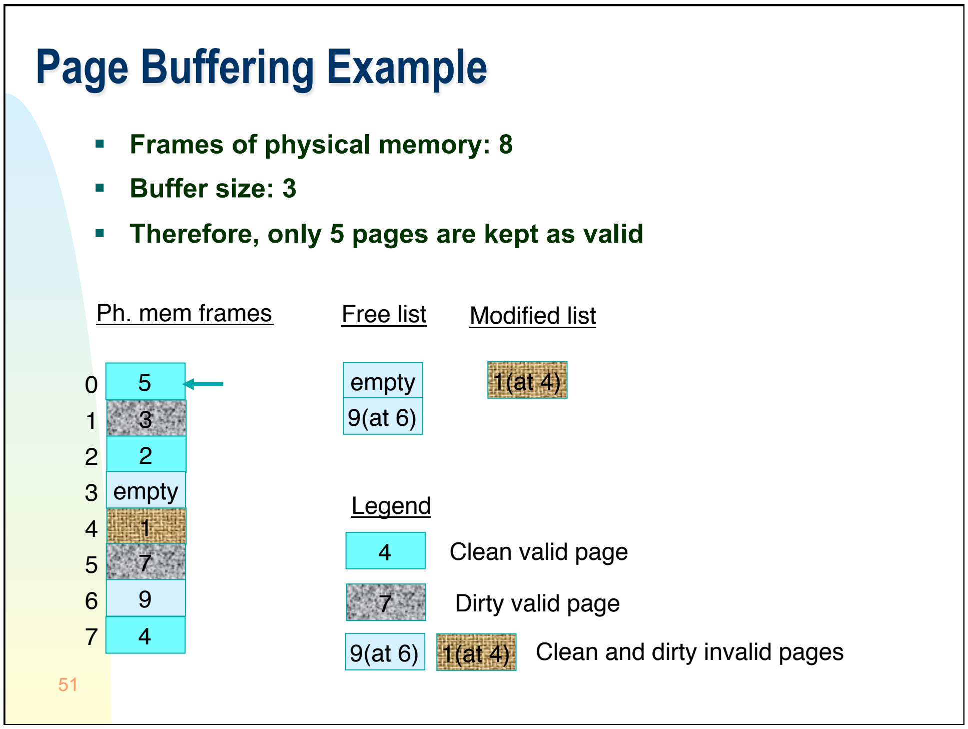
- **Keep a counter of the number of references that have been made to each page**
- LFU Algorithm**: replaces page with smallest count**
- MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used**
- **Implementation of these algorithms are expensive and rarely used.**

Page Buffering

- Pages to be replaced are kept in main memory for a while to guard against poorly performing replacement algorithms such as FIFO
- Two lists of pointers are maintained: each entry points to a frame selected for replacement
  - a free page list for frames that have not been modified since brought in (no need to swap out)
  - a modified page list for frames that have been modified (need to write them out)
- A frame to be replaced has a pointer added to the tail of one of the lists and the present bit is cleared in corresponding page table entry
  - but the page remains in the same memory frame

Page Buffering

- At each page fault the two lists are first examined to see if the needed page is still in main memory
  - If it is, we just need to set the present bit in the corresponding page table entry (and remove the matching entry in the relevant page list)
  - If it is not, then the needed page is brought in, it is placed in the frame pointed by the head of the free frame list (overwriting the page that was there)
    - the head of the free frame list is moved to the next entry
  - (the frame number in the page table entry could be used to scan the two lists, or each list entry could contain the process id and page number of the occupied frame)
- The modified list also serves to write out modified pages in cluster (rather than individually)



Page Buffering Example

- Frames of physical memory: 8
- Buffer size: 3
- Therefore, only 5 pages are kept as valid

Ph. mem frames

0	5 ←
1	3
2	2
3	empty
4	1
5	7
6	9
7	4

Free list

empty
9(at 6)

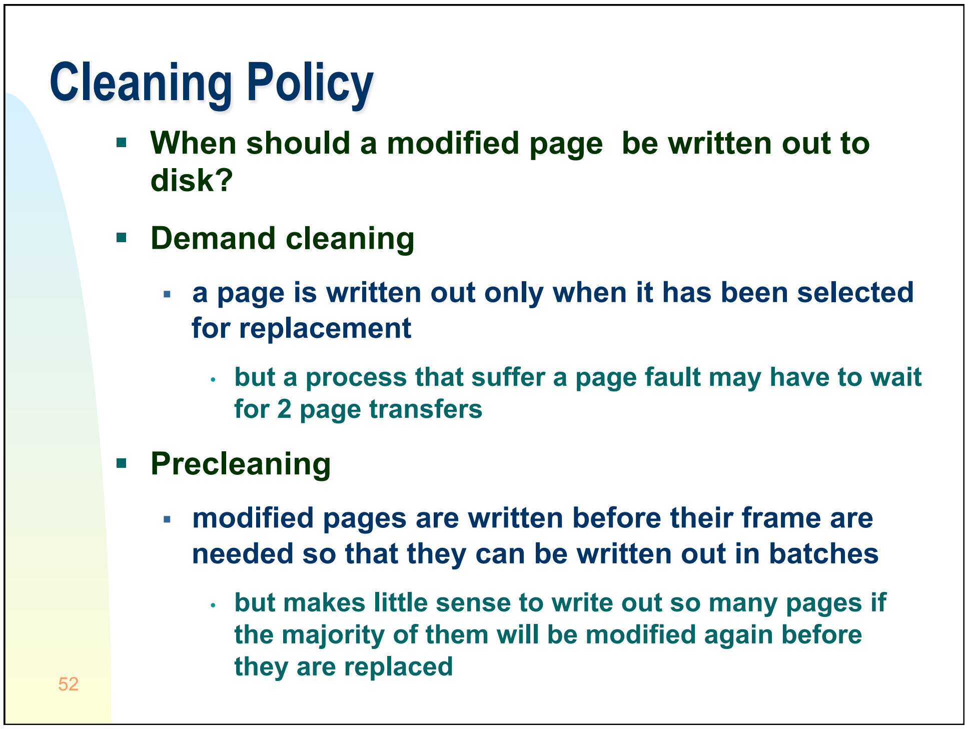
Modified list

1(at 4)

Legend

4	Clean valid page
7	Dirty valid page
9(at 6) 1(at 4)	Clean and dirty invalid pages

51



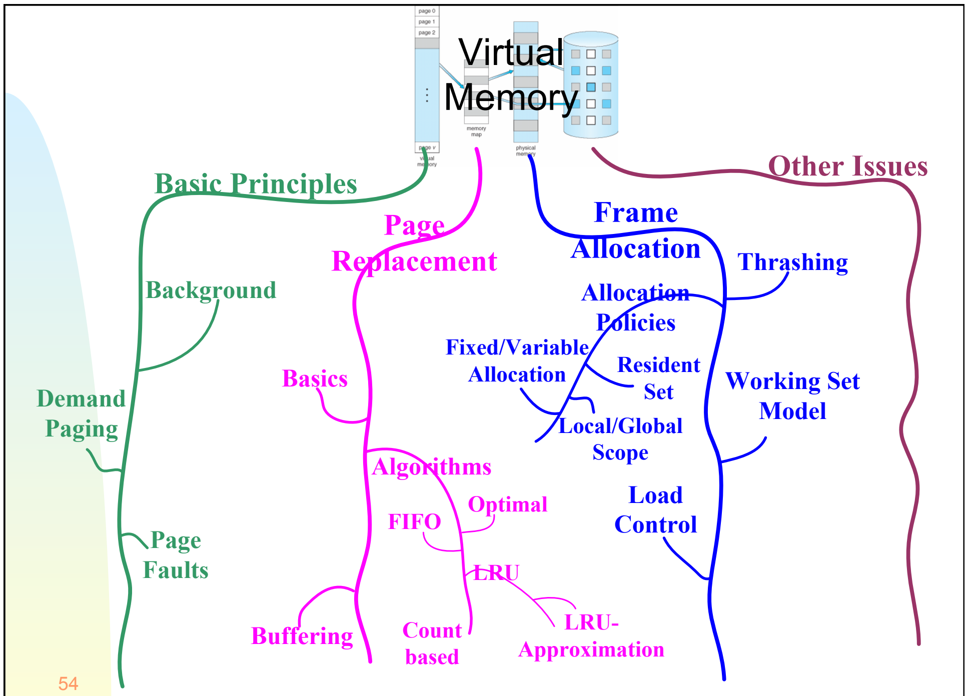
Cleaning Policy

- When should a modified page be written out to disk?
- Demand cleaning
  - a page is written out only when it has been selected for replacement
    - but a process that suffer a page fault may have to wait for 2 page transfers
- Precleaning
  - modified pages are written before their frame are needed so that they can be written out in batches
    - but makes little sense to write out so many pages if the majority of them will be modified again before they are replaced

52

Cleaning Policy

- A good compromise can be achieved with page buffering
  - recall that pages chosen for replacement are maintained either on a free (unmodified) list or on a modified list
  - pages on the modified list can be periodically written out in batches and moved to the free list
  - a good compromise since:
    - not all dirty pages are written out but only those chosen for replacement
    - writing is done in batch



Virtual Memory

Basic Principles

Background

Demand Paging

Page Faults

Page Replacement

Basics

Algorithms

FIFO

Optimal

LRU

LRU-Approximation

Count based

Buffering

Frame Allocation

Allocation Policies

Fixed/Variable Allocation

Resident Set

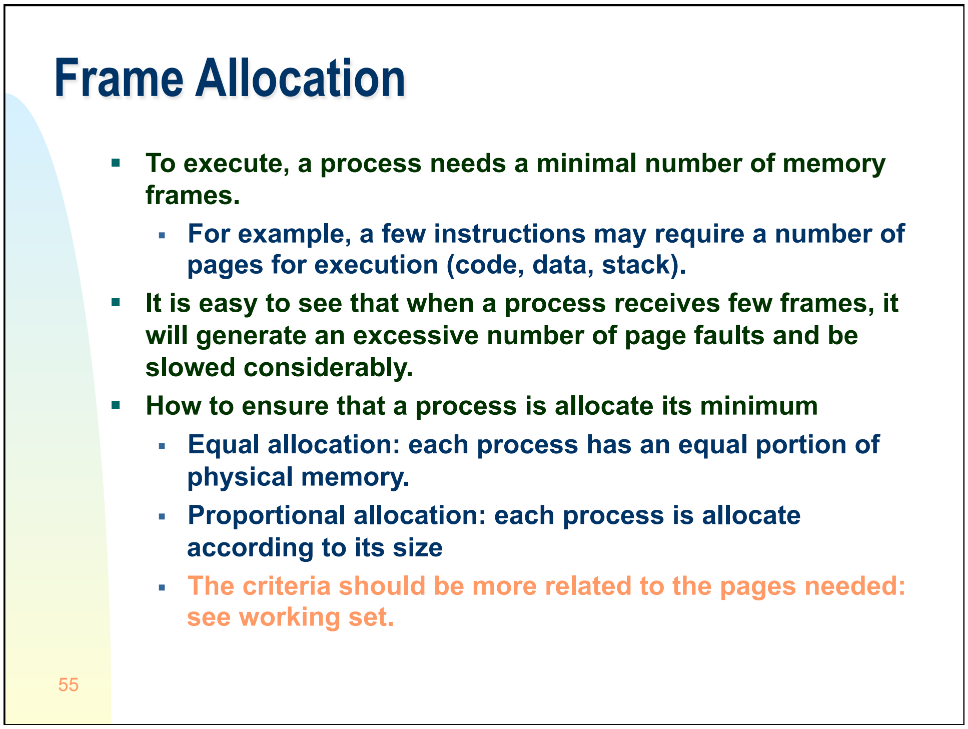
Local/Global Scope

Thrashing

Working Set Model

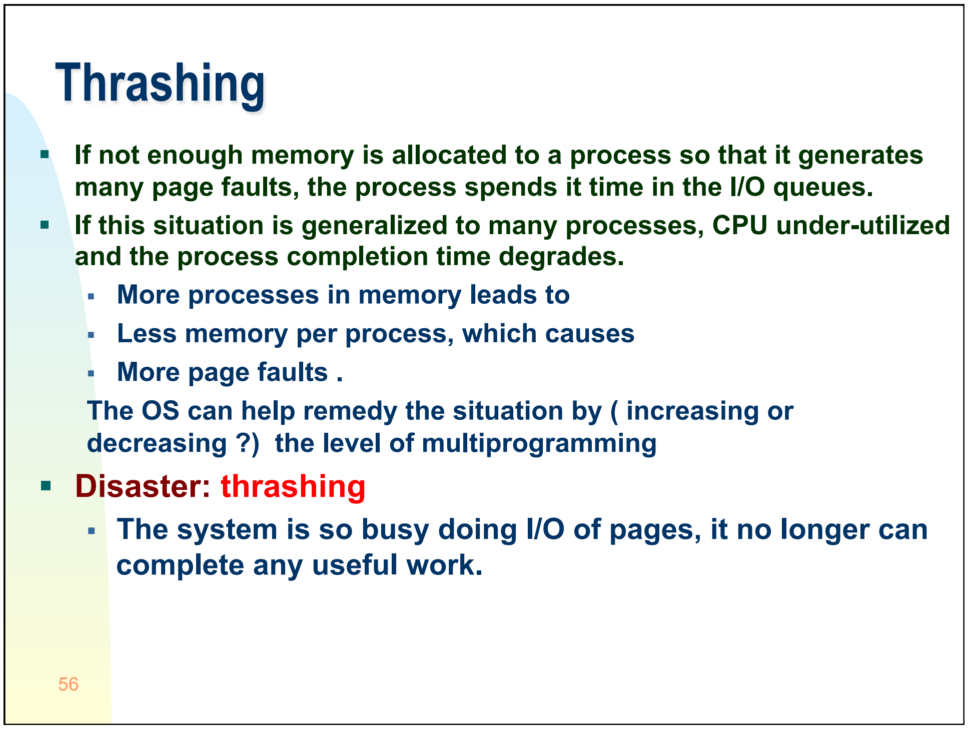
Load Control

Other Issues



Frame Allocation

- To execute, a process needs a minimal number of memory frames.
  - For example, a few instructions may require a number of pages for execution (code, data, stack).
- It is easy to see that when a process receives few frames, it will generate an excessive number of page faults and be slowed considerably.
- How to ensure that a process is allocate its minimum
  - Equal allocation: each process has an equal portion of physical memory.
  - Proportional allocation: each process is allocate according to its size
  - The criteria should be more related to the pages needed: see working set.

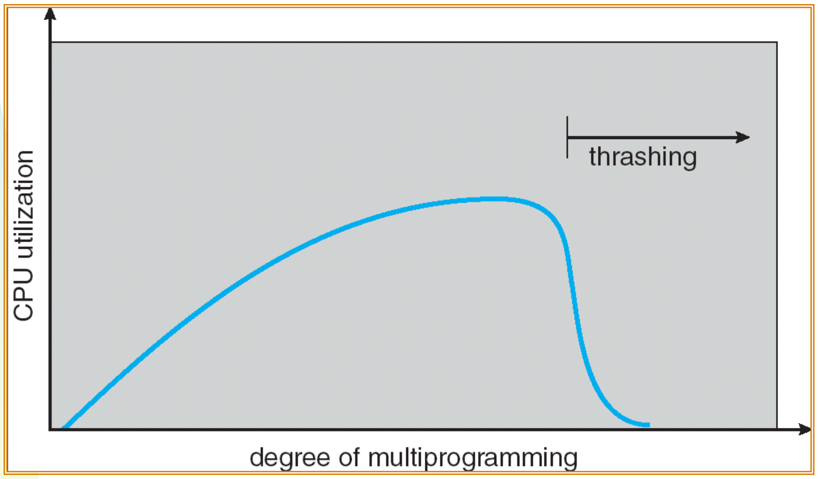


Thrashing

- If not enough memory is allocated to a process so that it generates many page faults, the process spends it time in the I/O queues.
- If this situation is generalized to many processes, CPU under-utilized and the process completion time degrades.
  - More processes in memory leads to
  - Less memory per process, which causes
  - More page faults .

  The OS can help remedy the situation by (increasing or decreasing ?) the level of multiprogramming
- Disaster: thrashing
  - The system is so busy doing I/O of pages, it no longer can complete any useful work.

Thrashing

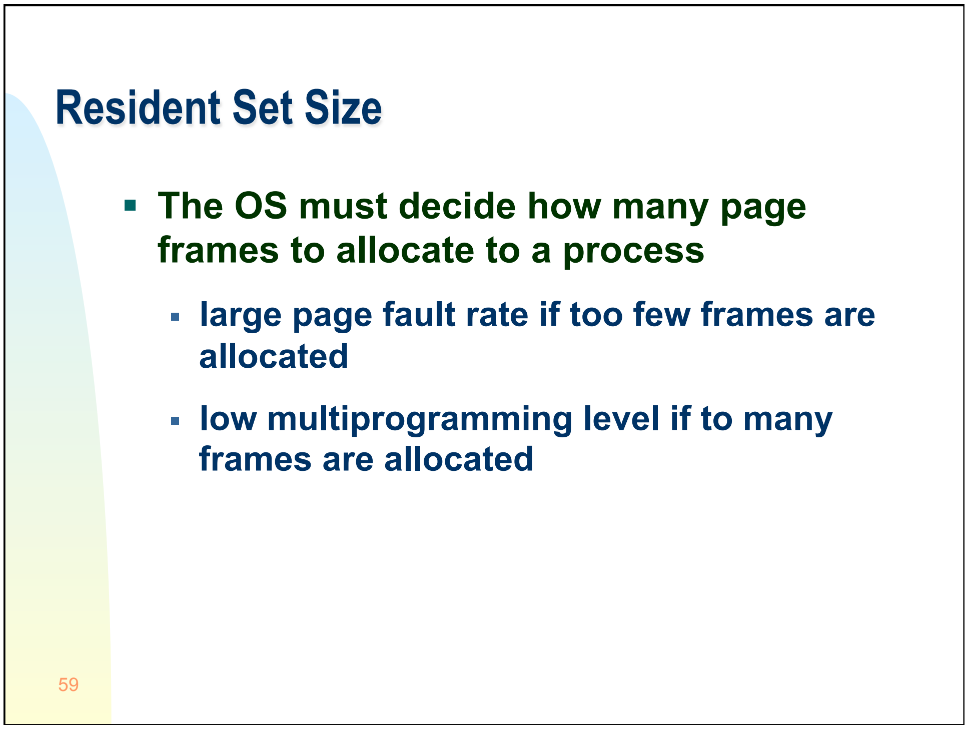


Thrashing

We really need to avoid thrashing!

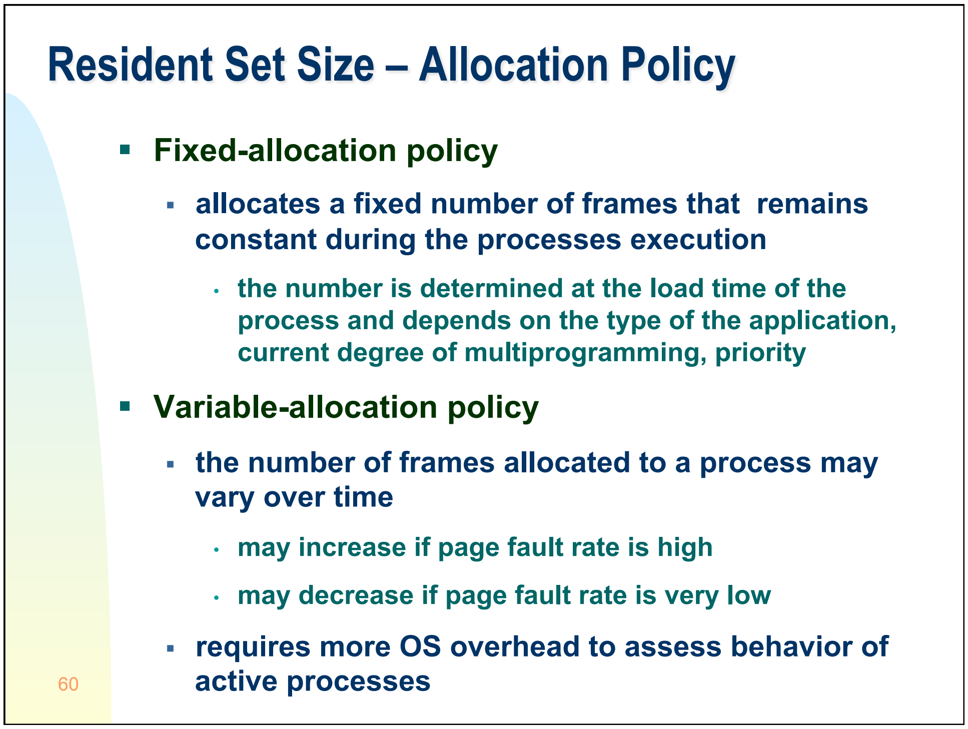
How to do that?

- **By managing the resident set sizes of processes**
- **By controlling the degree of multiprogramming**
  - **Load control**



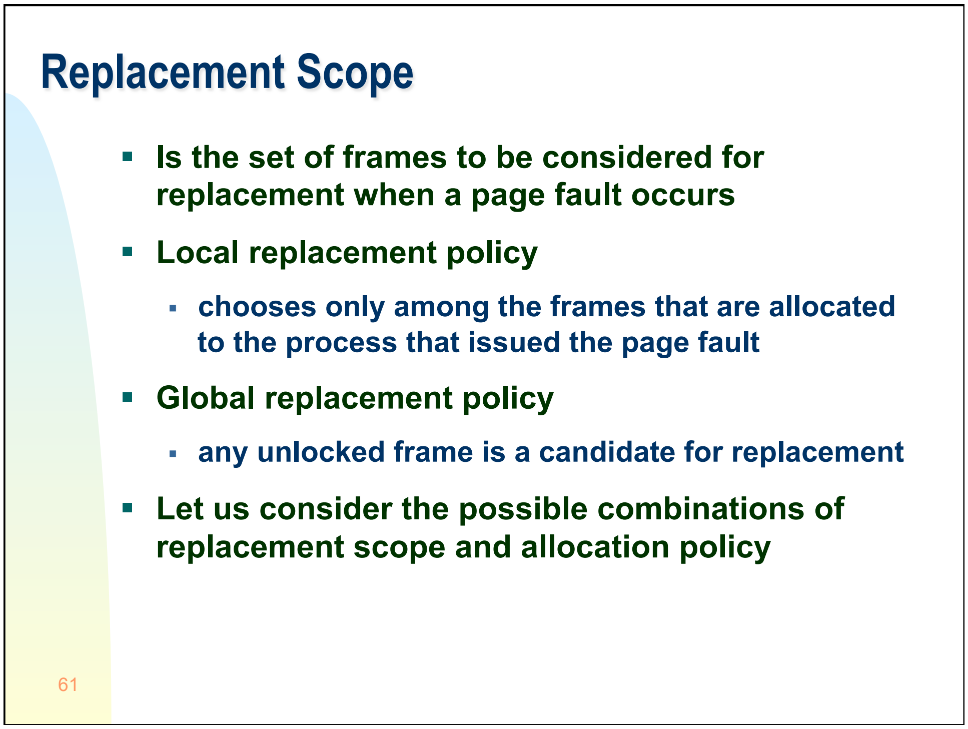
Resident Set Size

- The OS must decide how many page frames to allocate to a process
  - large page fault rate if too few frames are allocated
  - low multiprogramming level if to many frames are allocated



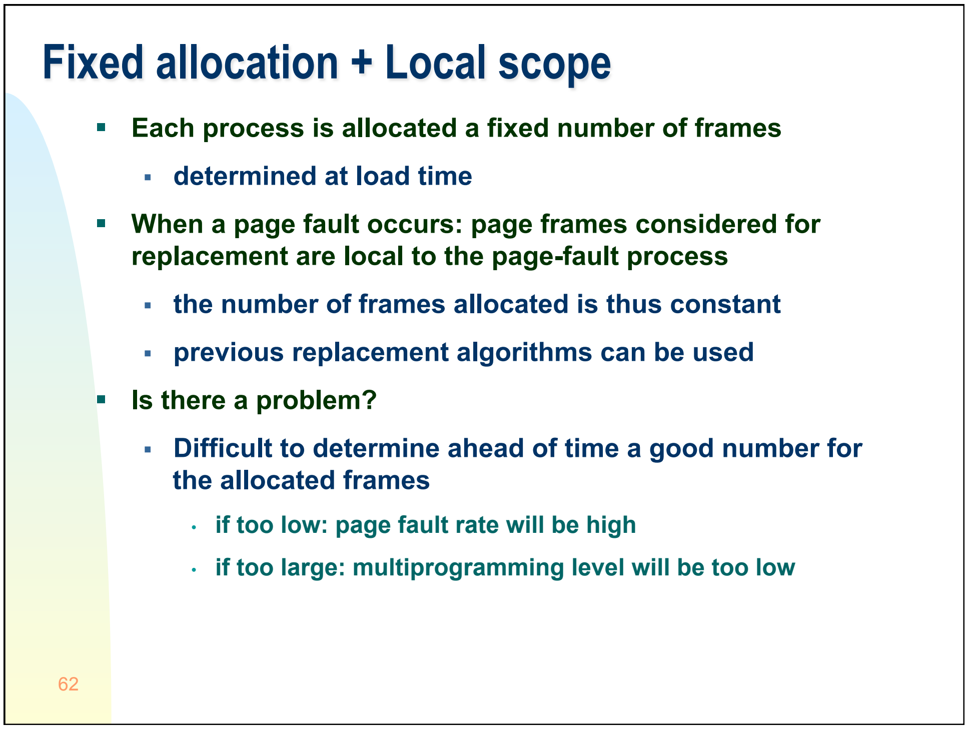
Resident Set Size – Allocation Policy

- Fixed-allocation policy
  - allocates a fixed number of frames that remains constant during the processes execution
    - the number is determined at the load time of the process and depends on the type of the application, current degree of multiprogramming, priority
- Variable-allocation policy
  - the number of frames allocated to a process may vary over time
    - may increase if page fault rate is high
    - may decrease if page fault rate is very low
  - requires more OS overhead to assess behavior of active processes



Replacement Scope

- Is the set of frames to be considered for replacement when a page fault occurs
- Local replacement policy
  - chooses only among the frames that are allocated to the process that issued the page fault
- Global replacement policy
  - any unlocked frame is a candidate for replacement
- Let us consider the possible combinations of replacement scope and allocation policy



Fixed allocation + Local scope

- Each process is allocated a fixed number of frames
  - determined at load time
- When a page fault occurs: page frames considered for replacement are local to the page-fault process
  - the number of frames allocated is thus constant
  - previous replacement algorithms can be used
- Is there a problem?
  - Difficult to determine ahead of time a good number for the allocated frames
    - if too low: page fault rate will be high
    - if too large: multiprogramming level will be too low

Fixed allocation + Global scope

- How to do that?
- Impossible to achieve
  - if all unlocked frames are candidates for replacement, the number of frames allocated to a process will necessarily vary over time

Variable allocation + Global scope

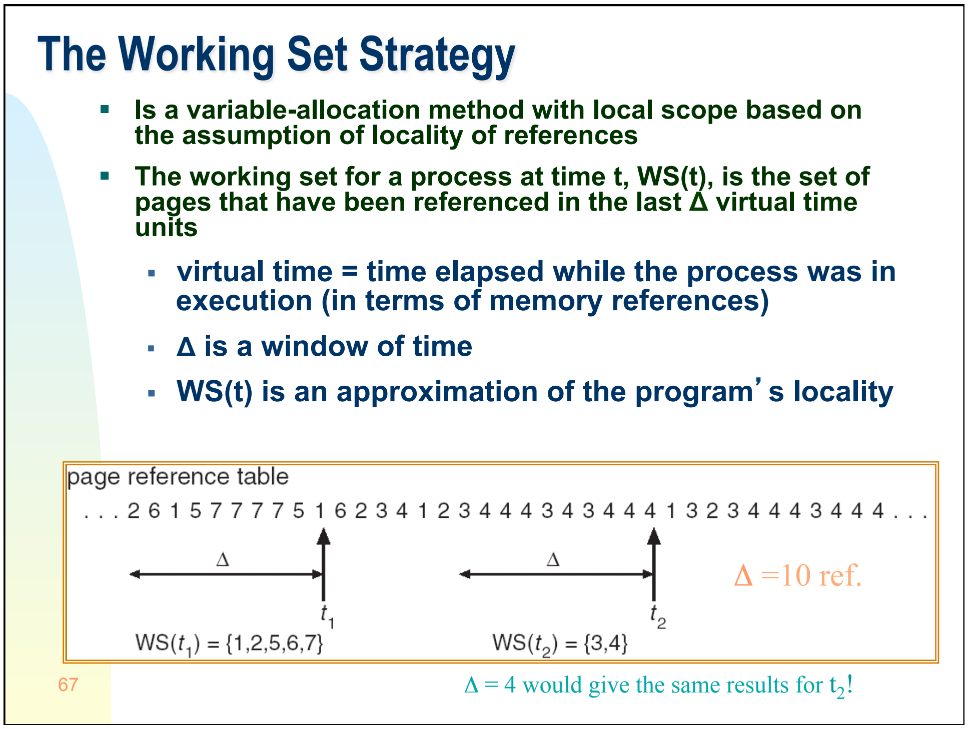
- Simple to implement--adopted by many OS (like Unix SVR4)
- A list of free frames is maintained
  - when a process issues a page fault, a free frame (from this list) is allocated to it
  - Hence the number of frames allocated to a page fault process increases
- Page replacement is necessary when the free list is empty
  - The page replacement algorithm is applied to all available frames
    - The choice for the process that will loose a frame is arbitrary: far from optimal
- Page buffering can alleviate this problem since a page may be reclaimed if it is referenced again soon

Variable allocation + Local scope

- May be the best combination (used by Windows NT)
- Allocate at load time a certain number of frames to a new process based on application type
  - use either prepaging or demand paging to fill up the allocation
- When a page fault occurs, select the page to replace from the resident set of the process that suffers the fault
- Evaluate periodically the allocation provided and increase or decrease it to improve overall performance
- How to do this evaluation?
  - Working set strategy

Working Set

- The working set of a process at a point in execution consists of the set of pages the process requires for execution without too many page faults.



The Working Set Strategy

- Is a variable-allocation method with local scope based on the assumption of locality of references
- The working set for a process at time t, WS(t), is the set of pages that have been referenced in the last Δ virtual time units
  - virtual time = time elapsed while the process was in execution (in terms of memory references)
  - Δ is a window of time
  - WS(t) is an approximation of the program's locality

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

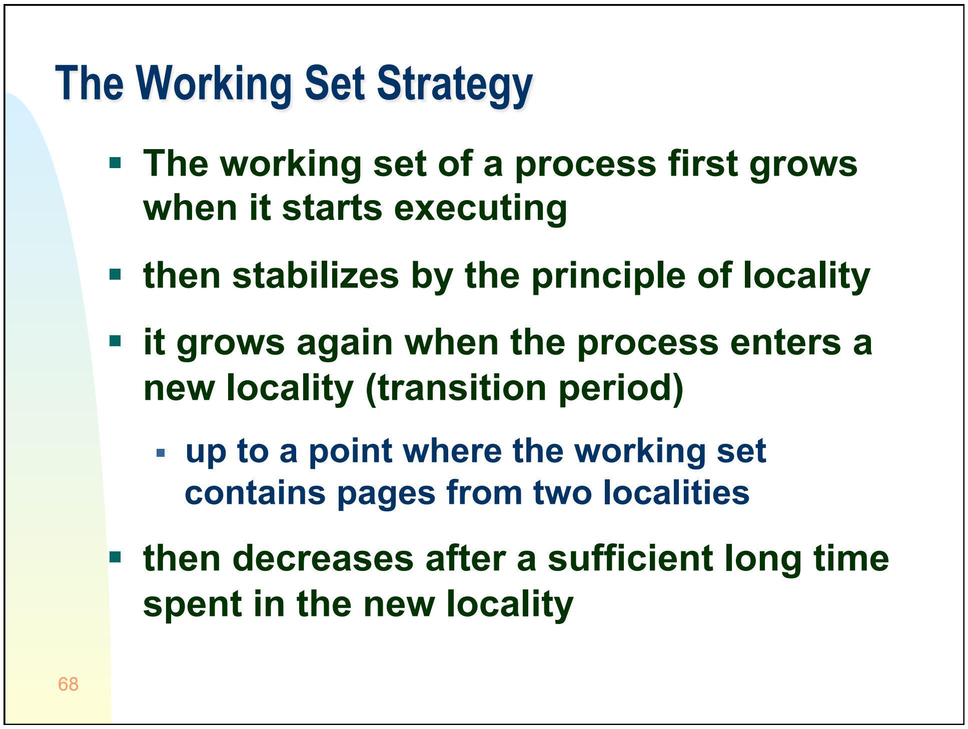
$WS(t_1) = \{1,2,5,6,7\}$

$WS(t_2) = \{3,4\}$

Δ =10 ref.

67

Δ = 4 would give the same results for t_2!



The Working Set Strategy

- The working set of a process first grows when it starts executing
- then stabilizes by the principle of locality
- it grows again when the process enters a new locality (transition period)
  - up to a point where the working set contains pages from two localities
- then decreases after a sufficient long time spent in the new locality

68

Working Set and Page Faults

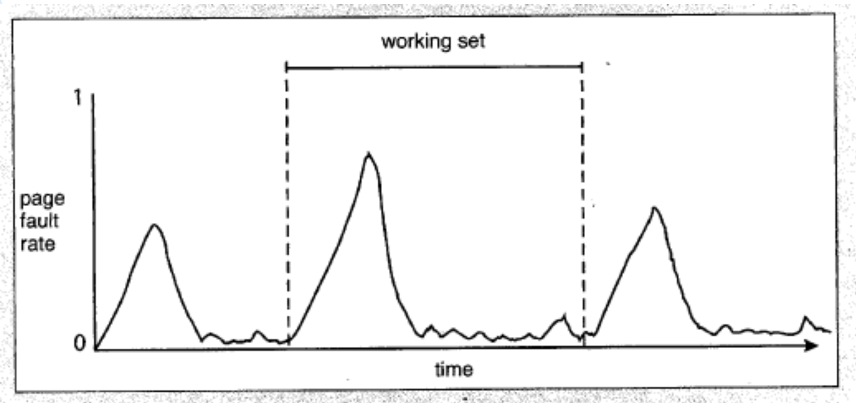


Figure 9.22 Page fault rate over time.

The Working Set Strategy

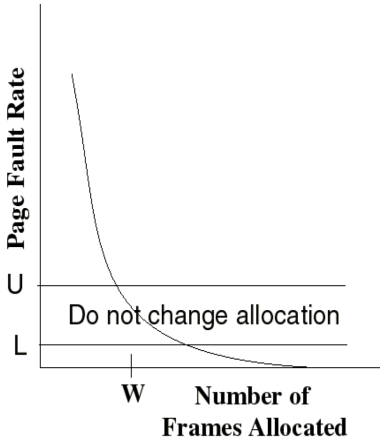
- the working set concept suggest the following strategy to determine the resident set size
  - Monitor the working set for each process
  - Periodically remove from the resident set of a process those pages that are not in the working set (apply LRU).
  - When the resident set of a process is smaller than its working set, allocate more frames to it
    - If not enough free frames are available, suspend the process (until more frames are available); ie: a process may execute only if its working set can be loaded in main memory

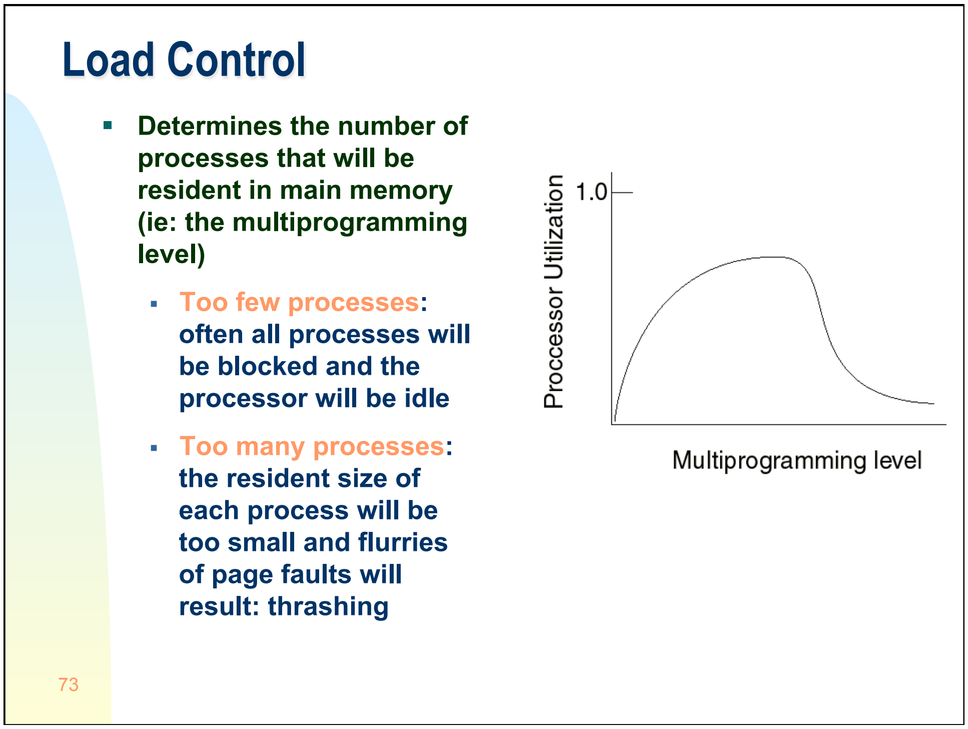
The Working Set Strategy

- **Practical problems with this working set strategy:**
  - **measurement of the working set for each process is impractical**
    - **necessary to stamp the referenced page with virtual time at every memory reference**
    - **necessary to maintain a time-ordered queue of referenced pages for each process**
  - **the optimal value for Δ is unknown and time varying**
- Any solution?**:**
  - **rather than monitoring the working set, monitor the page fault rate!**

The Page-Fault Frequency Strategy

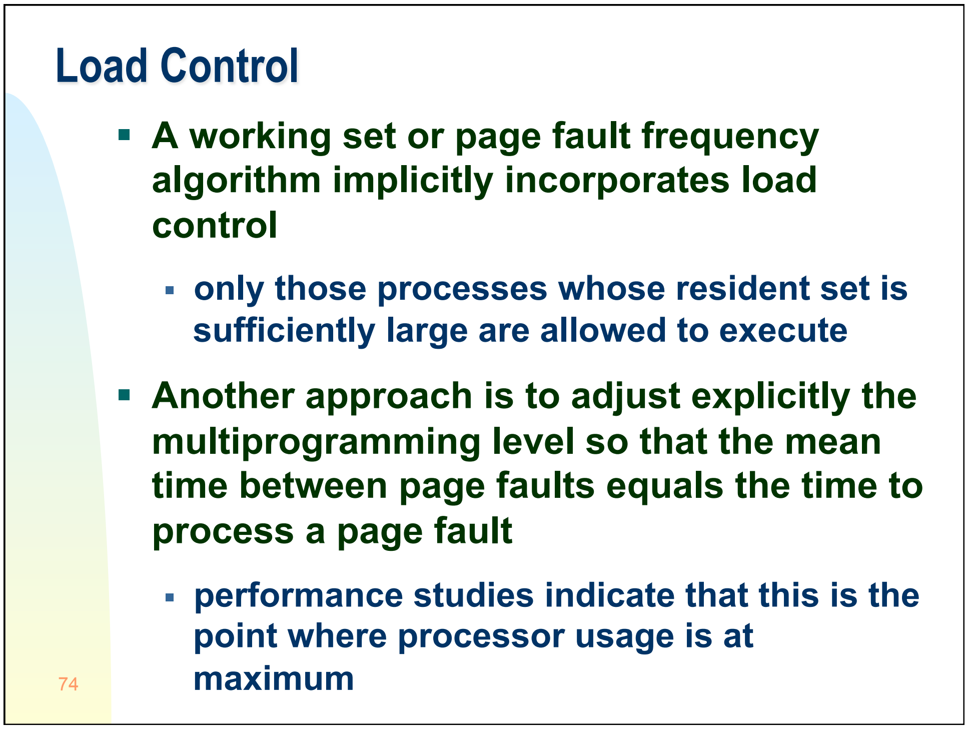
- **Define an upper bound U and lower bound L for page fault rates**
- **Allocate more frames to a process if fault rate is higher than U**
- **Allocate less frames if fault rate is < L**
- **The resident set size should be close to the working set size W**
- **We suspend the process if the PFR > U and no more free frames are available**





Load Control

- Determines the number of processes that will be resident in main memory (ie: the multiprogramming level)
  - Too few processes: often all processes will be blocked and the processor will be idle
  - Too many processes: the resident size of each process will be too small and flurries of page faults will result: thrashing



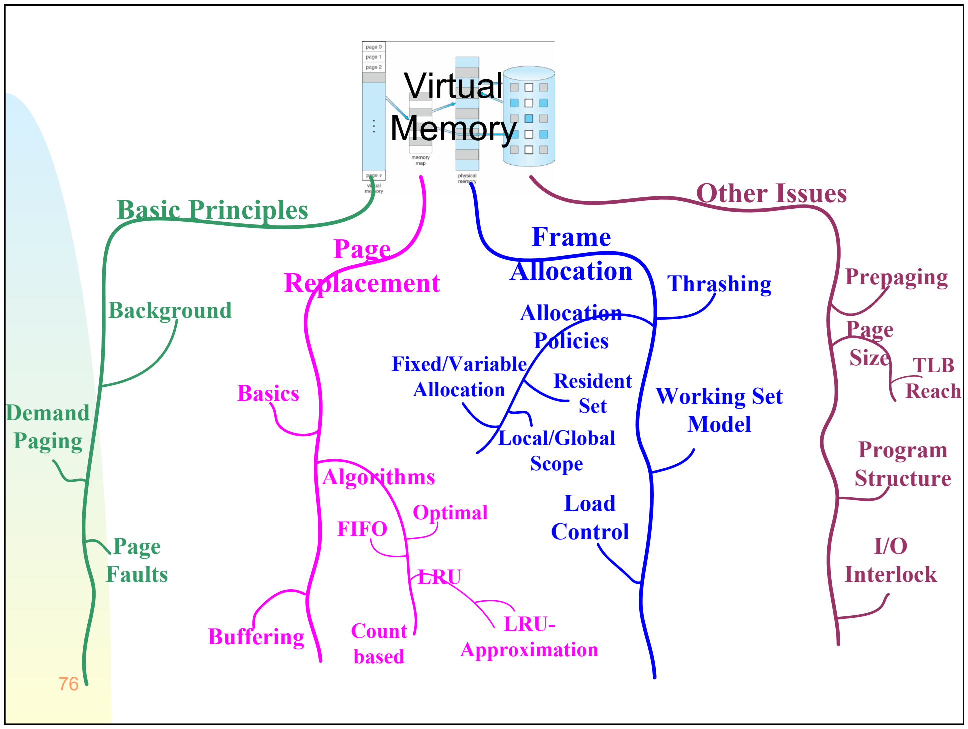
Load Control

- A working set or page fault frequency algorithm implicitly incorporates load control
  - only those processes whose resident set is sufficiently large are allowed to execute
- Another approach is to adjust explicitly the multiprogramming level so that the mean time between page faults equals the time to process a page fault
  - performance studies indicate that this is the point where processor usage is at maximum

Process Suspension

- Explicit load control requires that we sometimes swap out (suspend) processes
- Possible victim selection criteria:
  - Faulting process
    - this process may not have its working set in main memory so it will be blocked anyway
  - Last process activated
    - this process is least likely to have its working set resident
  - Process with smallest resident set
    - this process requires the least future effort to reload
  - Largest process
    - will yield the most free frames

75



Virtual Memory

Basic Principles

Background

Demand Paging

Page Faults

Page Replacement

Basics

Algorithms

FIFO

Optimal

LRU

LRU-Approximation

Count based

Buffering

Frame Allocation

Allocation Policies

Fixed/Variable Allocation

Resident Set

Local/Global Scope

Load Control

Thrashing

Working Set Model

Other Issues

Prepaging

Page Size

TLB Reach

Program Structure

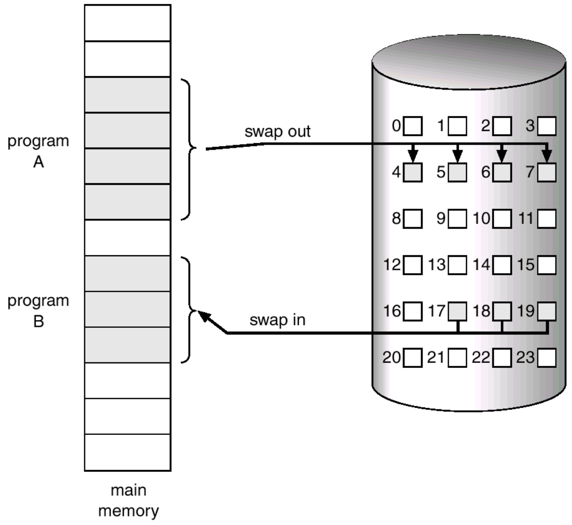
I/O Interlock

76

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Possible to remember related pages
  - E.g. when suspending a process, assume that pages resident in memory make up the working set
  - Can reload all pages when process becomes swapped back into memory.

Transferring Pages Efficiently

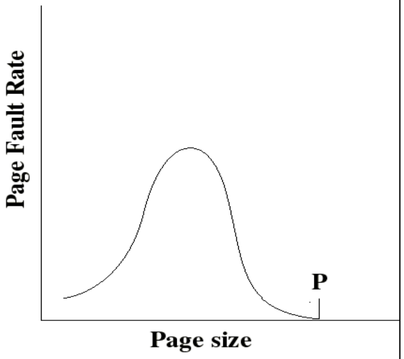


The Page Size Issue

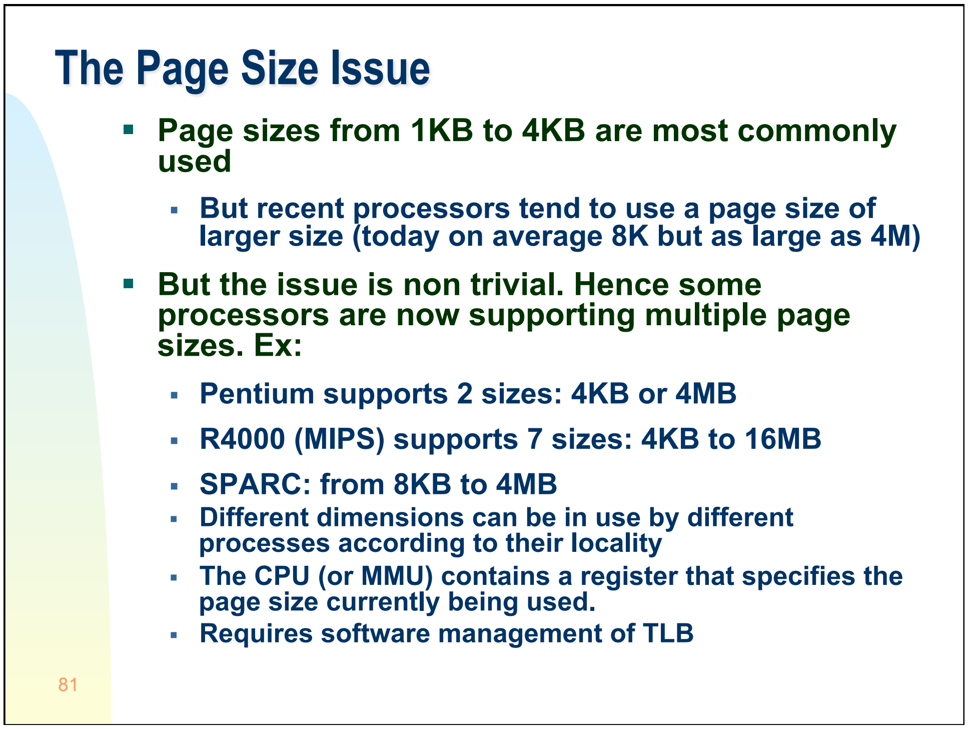
- Page size is defined by hardware; always a power of 2 for more efficient logical to physical address translation.
  - But exactly which size to use is a difficult question
- Advantages of small pages:
  - Less internal fragmentation
  - Better resolution: reads only code/data required for execution.
- Advantages of large pages:
  - More efficient I/O
  - Less pages in memory
    - Increases the TLB hit ratio
    - Page table is smaller

Page Size – Large or Small

- Assume fixed total memory for process
- With a small size,
  - Large page table means portion of it in virtual memory, double page fault
  - Principle of locality means
  - Low number of page faults. Why?
- By increasing the size
  - Less pages can be kept in RAM.
  - Each page contains more unused code.
  - More page faults. Why?
- Eventually the size of the page is sufficiently large to contain the complete program (no page faults). Is this correct?

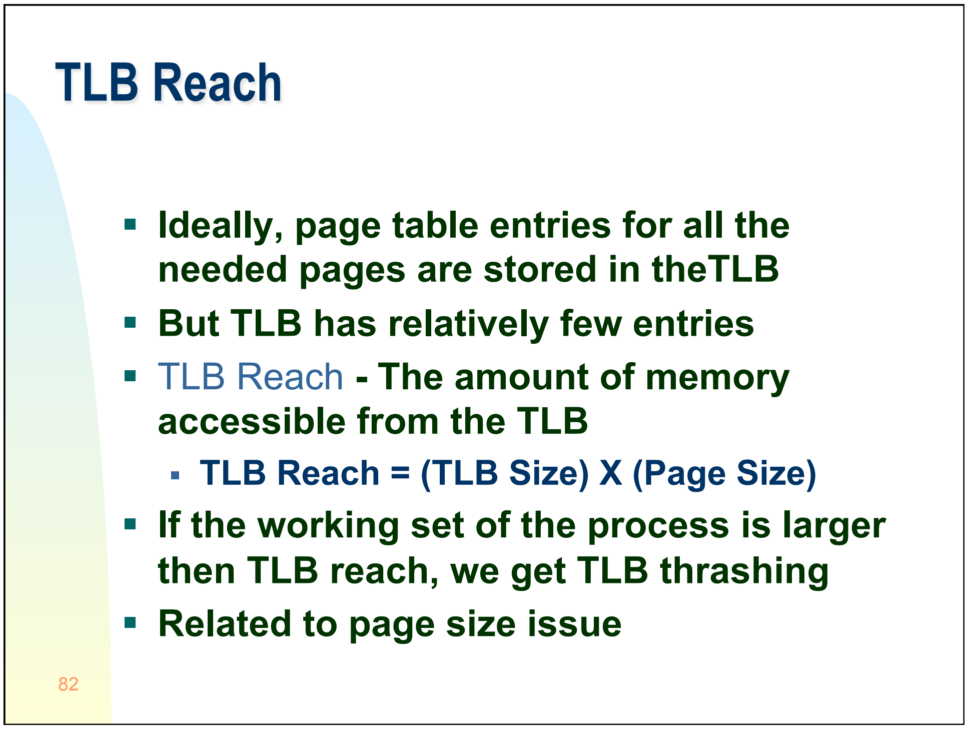


Stallings



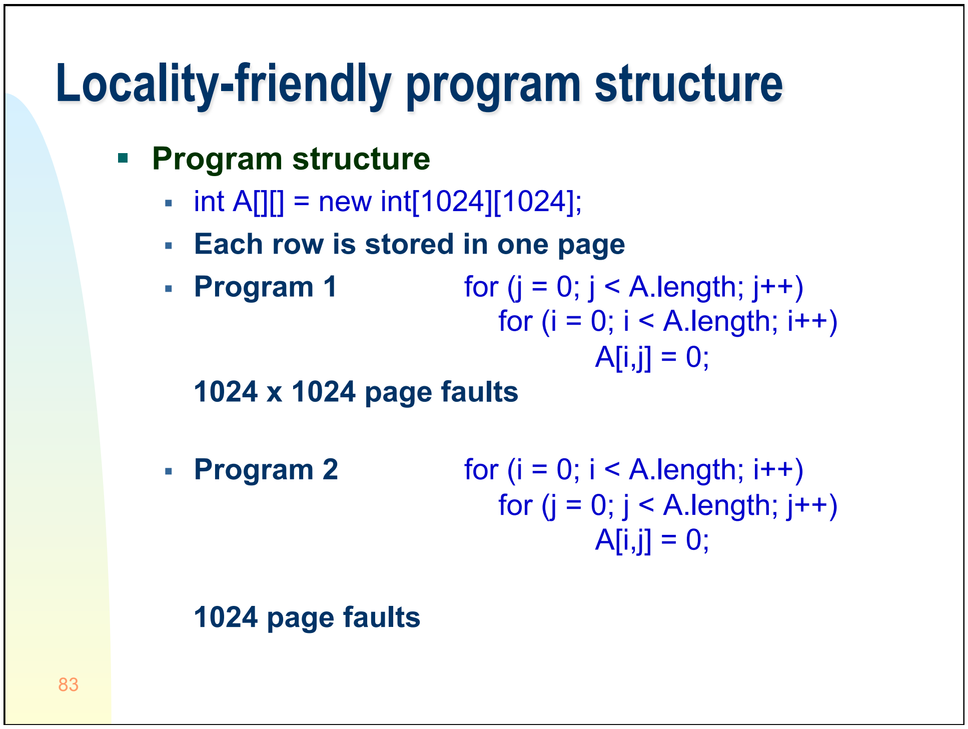
The Page Size Issue

- Page sizes from 1KB to 4KB are most commonly used
  - But recent processors tend to use a page size of larger size (today on average 8K but as large as 4M)
- But the issue is non trivial. Hence some processors are now supporting multiple page sizes. Ex:
  - Pentium supports 2 sizes: 4KB or 4MB
  - R4000 (MIPS) supports 7 sizes: 4KB to 16MB
  - SPARC: from 8KB to 4MB
  - Different dimensions can be in use by different processes according to their locality
  - The CPU (or MMU) contains a register that specifies the page size currently being used.
  - Requires software management of TLB



TLB Reach

- Ideally, page table entries for all the needed pages are stored in theTLB
- But TLB has relatively few entries
- TLB Reach - The amount of memory accessible from the TLB
  - TLB Reach = (TLB Size) X (Page Size)
- If the working set of the process is larger then TLB reach, we get TLB thrashing
- Related to page size issue



Locality-friendly program structure

- Program structure
  - int A[][] = new int[1024][1024];
  - Each row is stored in one page
  - Program 1

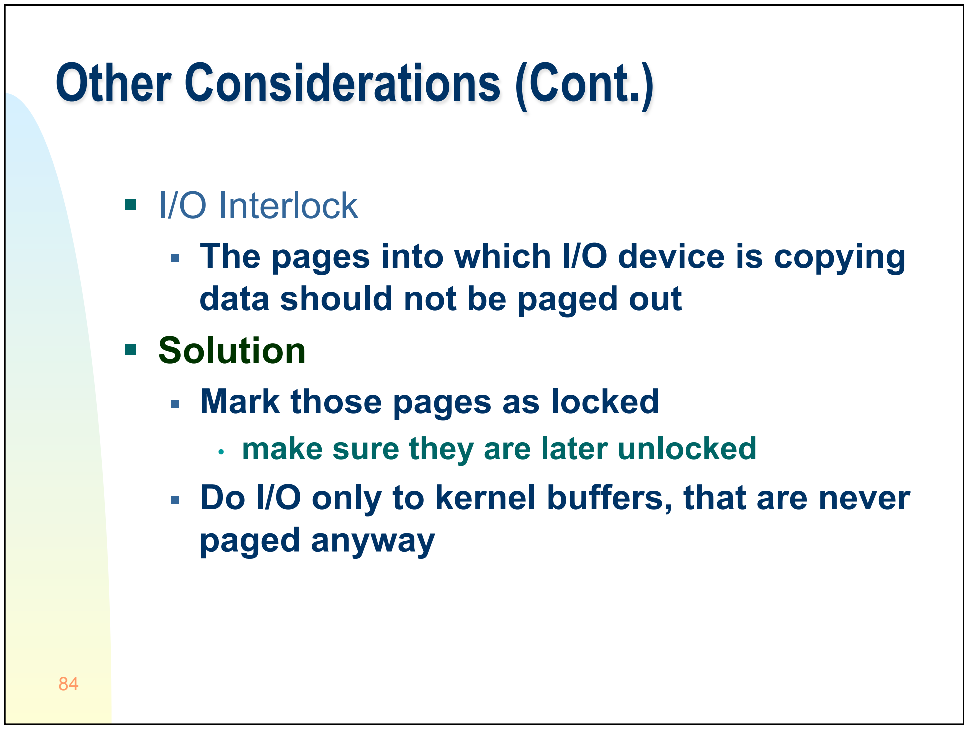
```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
             A[i,j] = 0;
```

1024 x 1024 page faults

- Program 2

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
             A[i,j] = 0;
```

1024 page faults



Other Considerations (Cont.)

- I/O Interlock
  - The pages into which I/O device is copying data should not be paged out
- Solution
  - Mark those pages as locked
    - make sure they are later unlocked
  - Do I/O only to kernel buffers, that are never paged anyway

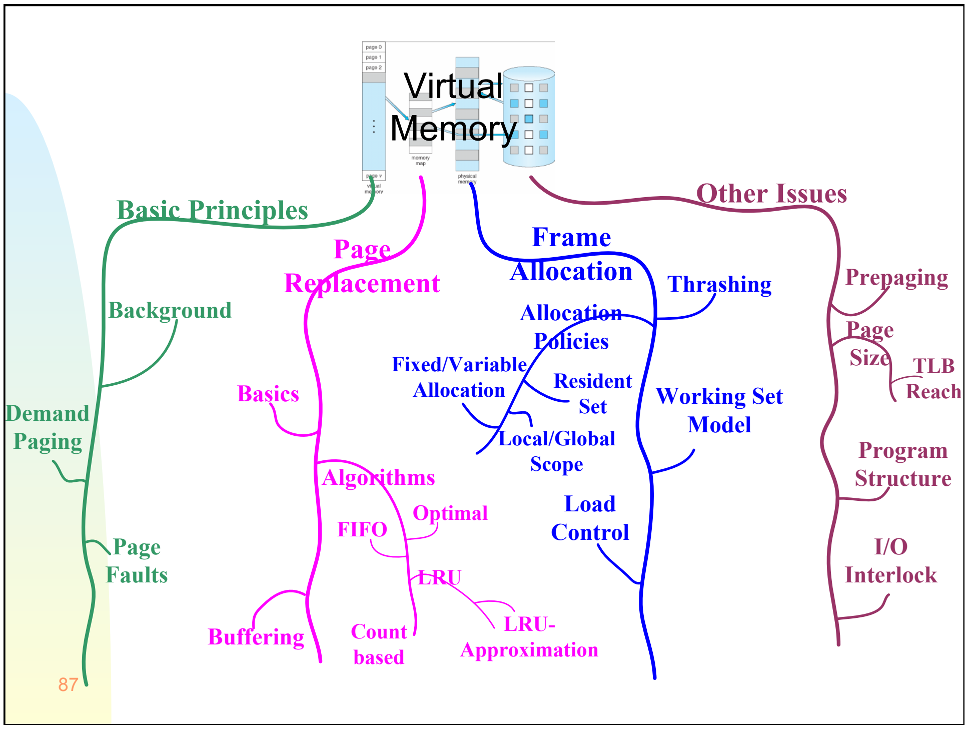
Windows XP

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page.
- Processes are assigned working set minimum and working set maximum
- Working set minimum is the minimum number of frames the process is guaranteed to have in memory
- A process may be assigned as many frames up to its working set maximum
  - Uses local scope for page replacement
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory
- Working set trimming removes frames from processes that have frames in excess of their working set minimum

Solaris

- Maintains a list of free frames to assign faulting processes
  - If there are less then *lotsfree* (1/64 physical memory) free frames, *pageout* process is started to try to free some more pages
- *Pageout* process
  - uses modified clock algorithm + page buffering
  - The aggressiveness of paging out depends on how much free memory remains
    - How many pages/s to evaluate, how often to run it, how much time to give a page to prove itself as useful
- If *pageout* is unable to keep enough free pages, the system starts to swap out whole processes
- + other optimizations (shared libraries, priority paging)

Virtual Memory



- Basic Principles
  - Background
  - Demand Paging
  - Page Faults
- Page Replacement
  - Basics
  - Algorithms
    - FIFO
    - Optimal
    - LRU
    - LRU-Approximation
    - Count based
  - Buffering
- Frame Allocation
  - Allocation Policies
    - Fixed/Variable Allocation
    - Resident Set
    - Local/Global Scope
  - Thrashing
  - Working Set Model
  - Load Control
- Other Issues
  - Prepaging
  - Page Size
    - TLB Reach
  - Program Structure
  - I/O Interlock

87