Slide 1

Part 8: Virtual Memory

- Introduction
  - Swapping, Virtual Memory Support
- Demand Paging
  - Page Fault, Performance of Demand Paging, Page Replacement
- Page-Replacement Algorithms
  - FIFO, Optimal, LRU, LRU Implementation, and LRU Approximation
- Allocation of Frames
  - Fixed vs. Variable Allocation, Local vs. Global Replacement
- Thrashing
  - Cause of Thrashing, Working-Set Model, Page-fault Frequency Scheme

An analysis of this topic is complicated by the fact that memory management is a complex interrelationship between processor hardware and operating system software.

Slide 2

Introduction - Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Swapping can be used to
  - Make space available for processes that require more memory
  - Increase the *degree of multiprogramming*

"Swapping-out" means to copy the entire process from memory to the backing store (a hard-disk). It will free up the memory occupied by the process and hence allow more processes to be executed. This will increase the degree of multiprogramming (i.e., the number of processes that can be concurrently executed).

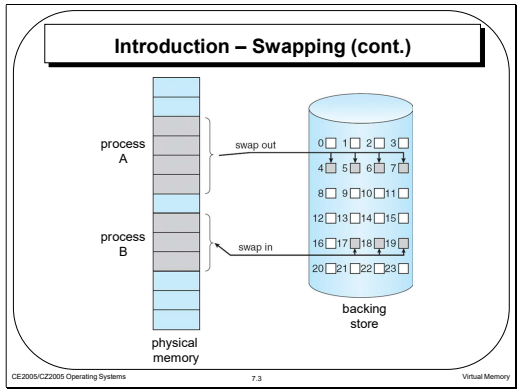
Processes in waiting or ready state can be swapped out

A ready queue can consist of processes whose image is either in memory or in backing store. Hence, context-switch time in such swapping system is fairly high.

Swapping operations incur high overheads (involving disk I/O).

Swapping out/in decision is made by the mid-term scheduler – the swapper.

Slide 3



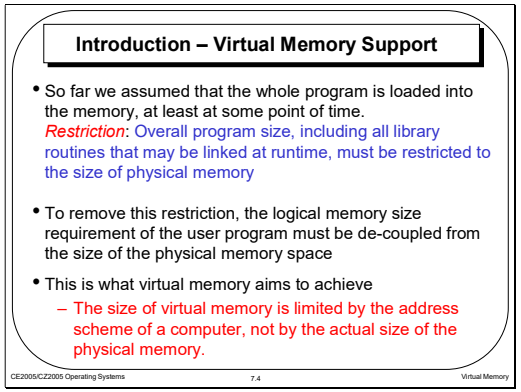
scheduler's
long term -> job
mid term -> swapper
short term -> CPU

A process may be swapped in and out of physical memory such that it occupies different regions of physical memory at different times during the course of execution. This requires address binding must be done during execution time.

After swapping out, the memory frames for process A becomes available. After swapping in, the memory frames are occupied.

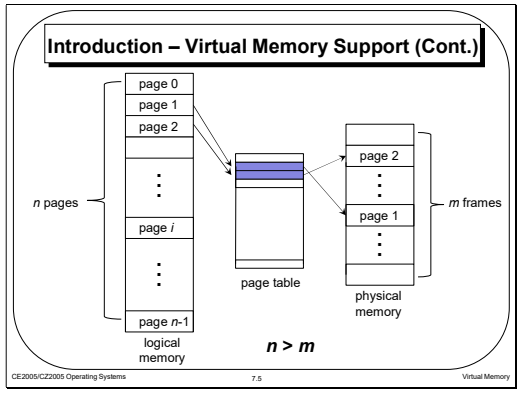
Policies for swapping out processes can be based on process priority, size of process, state of process, etc.

Slide 4



Although swapping (and some other techniques) can be used to make space available for processes that require more memory and to increase the degree of multiprogramming, fundamentally the whole process image is still required to be loaded into the memory at least at some point of time.

Slide 5



To understand what the key issue is, and why virtual memory was a matter of much debate, let us examine again the task of the operating system with respect to virtual memory. Consider a large process, consisting of a long program plus a number of arrays of data. Over any short period of time, execution may be confined to a small section of the program (e.g., a subroutine) and access to perhaps only one or two arrays of data. If this is so, then it would clearly be wasteful to load in dozens of pages for that process when only a few pages will be used during that stage of process execution. We can make better use of memory by loading in just a few pages. This is possible because:

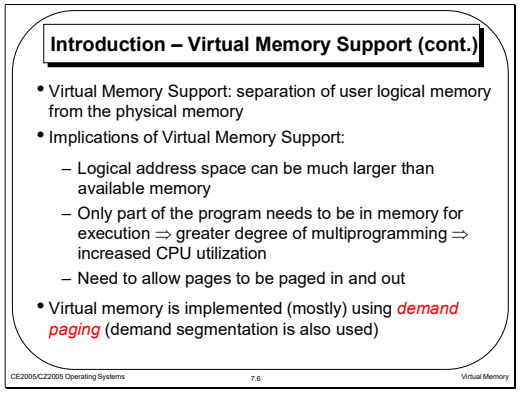
1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a page of a process can be in and out of physical memory such that it occupies different regions of physical memory at different times during the course of process's execution.

2. A process may be broken up into a number of pieces (pages or segments) and these pieces need not be contiguously located in physical memory during execution.

Then, if the program branches to an instruction or references a data item on a page not in physical memory, a fault is triggered. This tells the operating system to bring in the desired page.

Thus, the size of virtual memory is limited by the address scheme of a computer, not by the actual size of the physical memory.

Slide 6

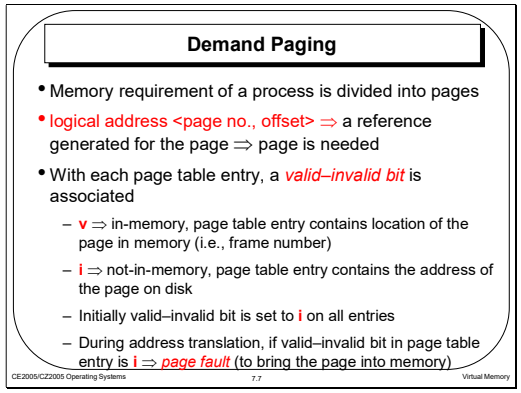


Backing store (hard-disk) can be considered as an extension of the physical memory, which contains all process images. So, using virtual memory support, only part of a process' logical address space is in memory and the rest can always be found in backing store (disk). Logical address space can be much larger than physical address space. Since at any one time, only a few pages of any given process are in memory, more processes can be maintained in memory. Furthermore, time is saved because unused pages are not loaded in and out of memory.

[For virtual memory to be practical and effective, two ingredients are needed. First, there must be hardware support for the paging scheme to be employed. Second, the operating system must include software for managing the movement of pages between memory and backing store.]

We illustrate how virtual memory can be implemented using demand paging.

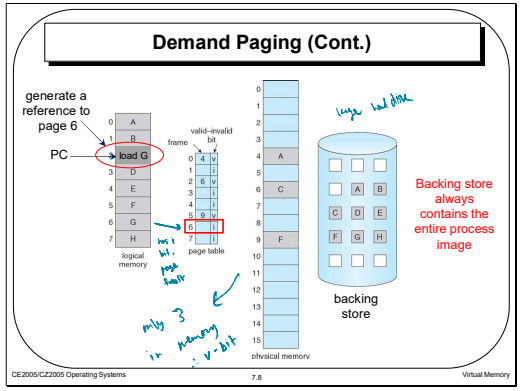
Slide 7



For demand paging, a page is loaded into the memory only when it is needed.

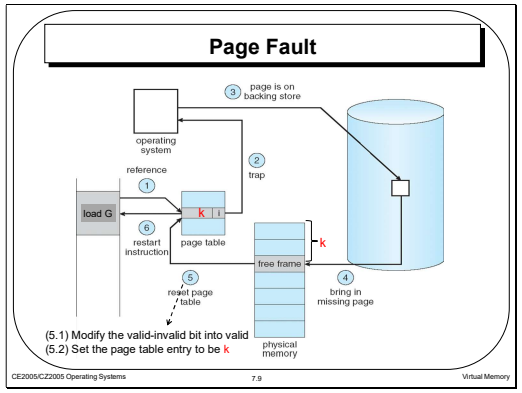
If valid-invalid bit in page table entry is i, the entry contains the address of the page on disk.

Slide 8



The pages that are not present in the memory can be always located in the backing store.

Slide 9



1. If there is a reference to a page, valid/invalid bit of the page table is checked to determine if the page is in the memory.

2. If the page is not in the memory, trap to OS to bring in the page. The state of interrupted process is saved.

3. OS uses the page table entry to locate the page in backing store.

4. OS finds a free frame (i.e., using free frame list & page replacement algorithm), and schedules a disk operation to read the desired page into the newly 7allocated frame.

5. When the disk read is complete, page table entry is modified. Valid/invalid bit is changed to valid and entry now contains the frame number where the paper is in the memory.

6. The state of interrupted process is restored, and the instruction is now re-started.

Slide 10

Page Fault (Cont.)

1. If there is a reference to a page, page table is checked to determine if the page is in the memory.
2. If the page is not in the memory, trap to OS to bring in the page. The state of interrupted process is saved.
3. OS uses the page table entry to locate the page in backing store.
4. OS finds a free frame (i.e., using free frame list & page replacement algorithm), and schedules a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, page table entry is modified.
6. The state of interrupted process is restored, and the instruction is now re-started.

Slide 11

Performance of Demand Paging

- Let p = probability of Page Fault, $0 \le p \le 1.0$
  - if p = 0 no page faults
  - if p = 1, every reference causes a fault
- Effective Access Time (EAT)

$$EAT = (1 - p) \times \text{memory access time} + p \times \text{page fault time}$$

- In practice, memory access time << page fault time.

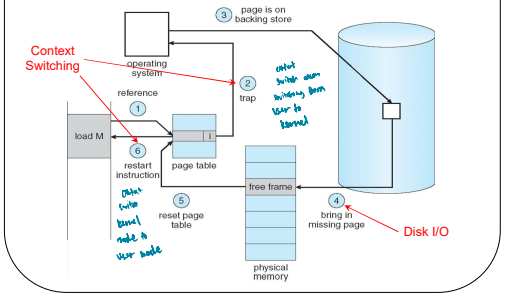
Effective access time = expected access time

If the system generates 10 page faults for every 100 page accesses, then p=10%.

The difference can be over 100 times.

Slide 12

Performance of Demand Paging (Cont.)

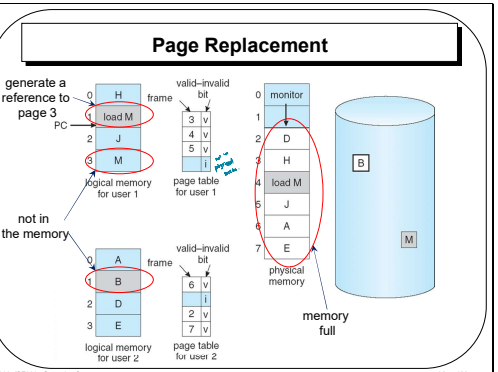


Three major components of page fault time:

- Servicing the page fault interrupt
- Read in a new page (longest component)
- Restart the process

Slide 13

Page Replacement



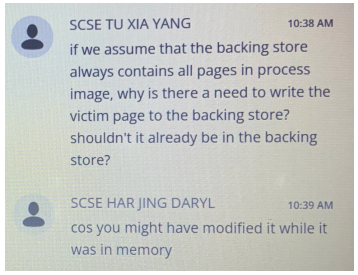
Need for page replacement

Slide 14

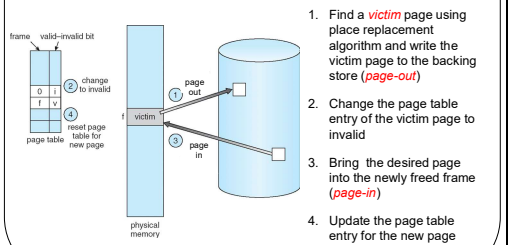
Page Replacement (Cont.)

- When page fault occurs, what happens if there is no free frame?
  victim page
  Solution: Find some page in memory, not really in use, and page it out (i.e., replace page)
- If no free frame is available, then two page transfers may be necessary, one out and one in. This increases further the page fault time!
- Use a *page replacement algorithm* to locate the page to be replaced. Desirably, the algorithm should result in minimum number of page faults

Slide 15



Page Replacement (Cont.)



1. Find a *victim* page using place replacement algorithm and write the victim page to the backing store (*page-out*)
2. Change the page table entry of the victim page to invalid
3. Bring the desired page into the newly freed frame (*page-in*)
4. Update the page table entry for the new page

When replacing pages, **only** *dirty pages* are actually written to backing store

page-in/page-out is different from swap-in/swap-out.

Slide 16

Page Replacement (Cont.)

- This overhead can be minimized by not writing pages that have not been modified since they were brought into memory. Such pages are identified by using a *modify* (or *dirty*) bit
- When replacing pages, **only** *dirty pages* are actually written to backing store
- Page replacement completes separation between logical memory and physical memory - large virtual memory can be provided on a smaller physical memory

We can reduce this overhead by using a modify bit (or dirty bit). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from secondary storage. In this case, we must write the page to storage. If the modify bit is not set, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to storage: it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.

swap in/out ≠ page in/out
↓
specific page

Slide 17

Objective: minimize no. of page faults

Page-Replacement Algorithms

- Objective is to have an algorithm with lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string), counting the number of page faults
- In all our examples, the ref. string used is

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

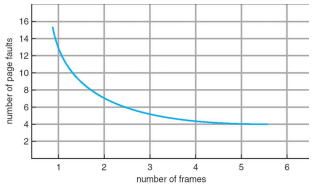
We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a reference string. We can generate reference strings artificially (by using a random-number generator, for example), or we can trace a given system and record the address of each memory reference.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page p, then any references to page p that immediately follow will never cause a page fault.

Slide 18

Page-Replacement Algorithms (Cont.)

- Obviously, as the # of available frames increases, the # of page faults will decrease:



Belady's Anomaly: more frames $\not\Rightarrow$ fewer page faults

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available.

We would expect that as the number of frames available increases, the number of page faults decreases. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

Slide 19

Outline

- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- Second-Chance Algorithm (or Clock Algorithm)

view animations on NTU Learn

Slide 20

First-In-First-Out (FIFO) Algorithm

- The easiest and simplest algorithm.

  Replace the page that has been *loaded* in the memory for the longest period of time

- Example using 3 free frames:

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	4	4	4	5	5	5	5	5	5
F2		2	2	2	1	1	1	1	1	3	3	3
F3			3	3	3	2	2	2	2	2	4	4
	P	P	P	P	P	P	P			P	P	

9 page faults

Slide 21

FIFO Algorithm (Cont.)

– Example using 4 free frames:

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	1	1	1	5	5	5	5	4	4
F2		2	2	2	2	2	2	1	1	1	1	5
F3			3	3	3	3	3	3	2	2	2	2
F4				4	4	4	4	4	4	3	3	3
	P	P	P	P			P	P	P	P	P	P

10 page faults !

- Belady's Anomaly
  – more frames ⇏ fewer page faults

Slide 22

Optimal Algorithm

- Replace the page that will not be used for the longest period of time.

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	1	1	1	1	1	1	1	4	4
F2		2	2	2	2	2	2	2	2	2	2	2
F3			3	3	3	3	3	3	3	3	3	3
F4				4	4	4	5	5	5	5	5	5
	P	P	P	P			P				P	

6 page faults

- *Problem: How will the OS know the future references?*

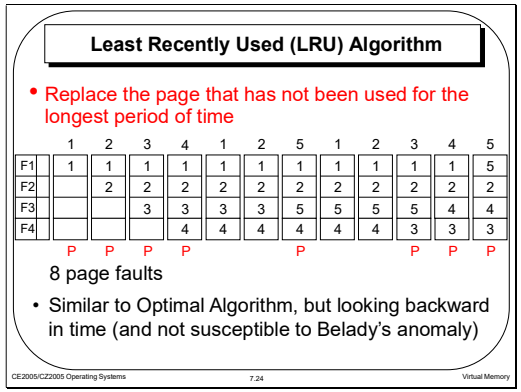
Slide 23

Optimal Algorithm (Cont.)

- Theoretically best but practically infeasible
- Used as a benchmark to measure how well other algorithms perform
- Does not suffer from Belady's anomaly:
  Add a frame ⇒ more look-ahead ⇒
  Inclusion property : Pages loaded in *n* frames is always a subset of pages in *n+1* frames

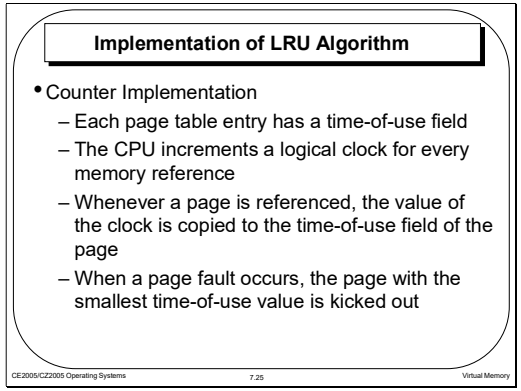
Inclusion property: the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames. In this case, it guarantees that if there is no page fault using n frames, there will be no page faults using n+1 frames.

Slide 24



For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory. So, LRU does not suffer from Belady's anomaly.

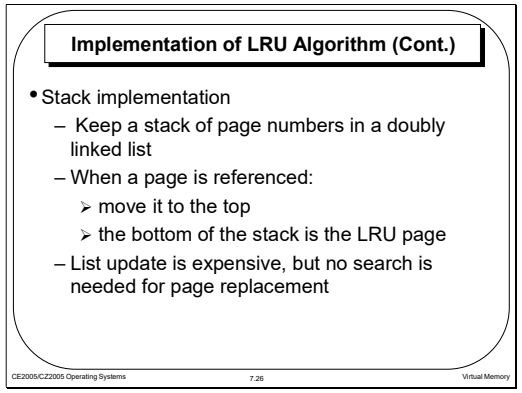
Slide 25



In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

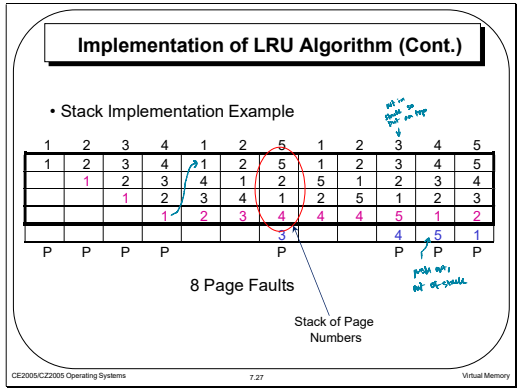
Associated overheads

Slide 26

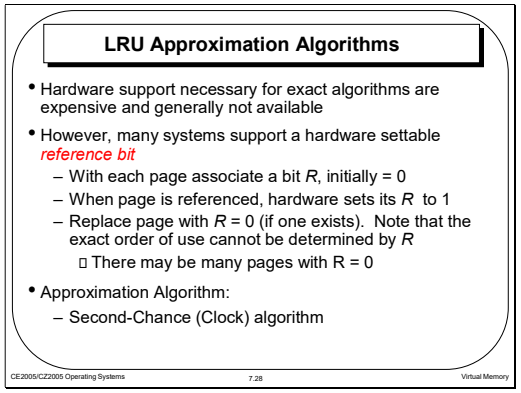


Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom. Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page.

Slide 27



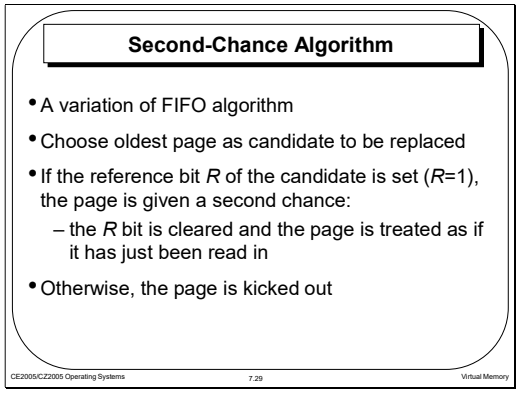
Slide 28



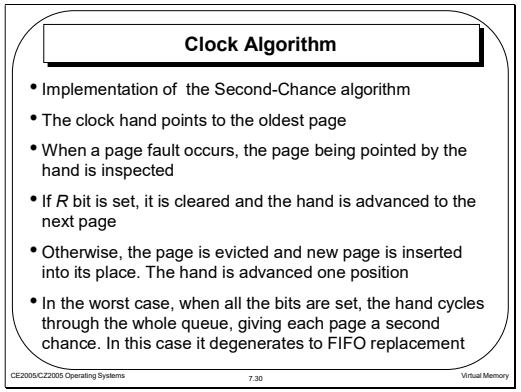
Not many computer systems provide sufficient hardware support for true LRU page replacement. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a process executes, the bit associated with every page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Slide 29

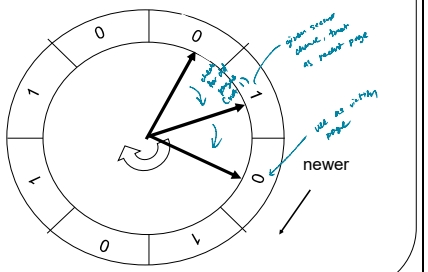


Slide 30



Slide 31

Clock Algorithm (Cont.)



Slide 32

Example: Clock Algorithm

Page Number	R Bit	1	2	3	4	1	2	5	1	2	3	4	5
F1		1,0	1,0	1,0	1,0	1,1	1,1	1,0	1,1	1,1	1,1	1,0	1,0
F2			2,0	2,0	2,0	2,0	2,1	2,0	2,0	2,1	2,1	2,0	2,0
F3				3,0	3,0	3,0	3,0	5,0	5,0	5,0	5,0	4,0	4,0
F4					4,0	4,0	4,0	4,0	4,0	4,0	3,0	3,0	5,0
		P	P	P	P			P			P	P	P

Circular Queue — 8 Page Faults

The second digit of each entry is the *R* bit. The shaded entry is where the clock hand points to. It moves upwards. When reaching the top, it will go back to the bottom

Slide 33

Allocation of Frames

- The OS must decide how many pages to bring into physical memory
  - the smaller the amount of memory allocated to each process, the more processes can reside in memory
  - small number of pages loaded increases page faults
  - beyond a certain size, further allocations of pages will not effect the page fault rate

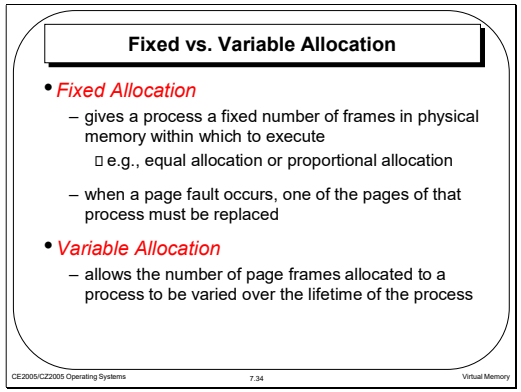
With paged virtual memory, it is not necessary and indeed may not be possible to bring all of the pages of a process into physical memory to prepare it for execution. Thus, the operating system must decide how many pages to bring in, that is, how much physical memory to allocate to a particular process. Several factors come into play:

• The smaller the amount of memory allocated to a process, the more processes that can reside in physical memory at any one time. This increases the probability that the operating system will find at least one ready process at any given time and hence reduces the time lost due to swapping.

• If a relatively small number of pages of a process are in physical memory, then, despite the principle of locality, the rate of page faults will be rather high.

• Beyond a certain size, additional allocation of physical memory to a particular process will have no noticeable effect on the page fault rate for that process because of the principle of locality.

Slide 34



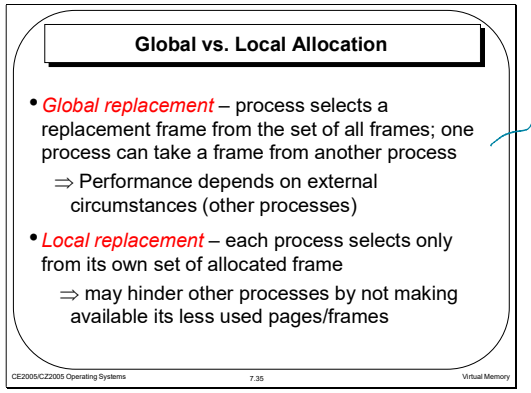
With these factors in mind, two sorts of policies are to be found in contemporary operating systems.

A **fixed-allocation policy** gives a process a fixed number of frames in physical memory within which to execute. That number is decided at initial load time (process creation time) and may be determined based on the type of process (interactive, batch, type of application) or may be based on guidance from the programmer or system manager. With a fixed-allocation policy, whenever a page fault occurs in the execution of a process, one of the pages of that process must be replaced by the needed page.

A **variable-allocation policy** allows the number of page frames allocated to a process to be varied over the lifetime of the process. Ideally, a process that is suffering persistently high levels of page faults, indicating that the principle of locality only holds in a weak form for that process, will be given additional page frames to reduce the page fault rate; whereas a process with an exceptionally low page fault rate, indicating that the process is quite well behaved from a locality point of view, will be given a reduced allocation, with the hope that this will not noticeably increase the page fault rate. The use of a variable-allocation policy relates to the concept of replacement scope, as explained in the next subsection.

The variable-allocation policy would appear to be the more powerful one. However, the difficulty with this approach is that it requires the operating system to assess the behavior of active processes. This inevitably requires software overhead in the operating system and is dependent on hardware mechanisms provided by the processor platform.

Slide 35



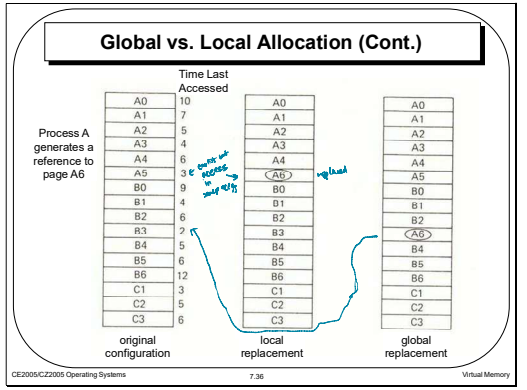
The scope of a replacement strategy can be categorized as global or local. Both types of policies are activated by a page fault when there are no free page frames.

A **local replacement policy chooses only among the resident pages** of the process that generated the page fault in selecting a page to replace.

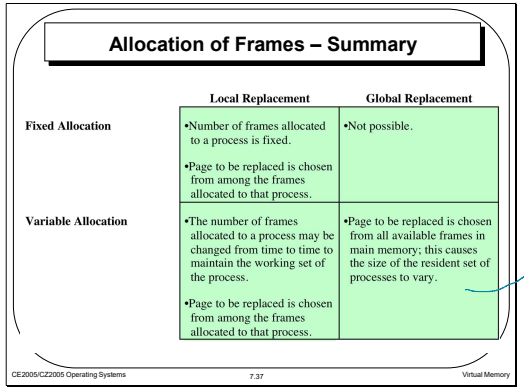
A **global replacement policy considers all unlocked pages in physical memory as candidates for** replacement, regardless of which process owns a particular page.

While it happens that local policies are easier to analyze, there is no convincing evidence that they perform better than global policies, which are attractive because of their simplicity of implementation and minimal overhead [CARR84, MAEK87].

Slide 36



Slide 37



There is a correlation between replacement scope and resident set size.

A fixed resident set implies a local replacement policy: To hold the size of a resident set fixed, a page that is removed from physical memory must be replaced by another page from the same process.

A variable-allocation policy can clearly employ a global replacement policy: The replacement of a page from one process in physical memory with that of another causes the allocation of one process to grow by one page and that of the other to shrink by one page.

With a fixed-allocation policy, it is necessary to decide ahead of time the amount of allocation to give to a process. This could be decided on the basis of the type of application and the amount requested by the program. The drawback to this approach is twofold: If allocations tend to be too small, then there will be a high page fault rate, causing the entire multiprogramming system to run slowly. If allocations tend to be unnecessarily large, then there will be too few programs in physical memory and there will be either considerable processor idle time or considerable time spent in swapping.

The variable-allocation, local-scope strategy attempts to overcome the problems with a global-scope strategy. It can be summarized as follows:
1. When a new process is loaded into physical memory, allocate to it a certain number of page frames as its resident set, based on application type, program request, or other criteria. Use either prepaging or demand paging to fill up the allocation.
2. When a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault.
3. From time to time, reevaluate the allocation provided to the process, and increase or decrease it to improve overall performance.

With this strategy, the decision to increase or decrease a resident set size is a deliberate one and is based on an assessment of the likely future demands of active processes. Because of this evaluation, such a strategy is more complex than a simple global replacement policy. However, it may yield better performance. The key elements of the variable-allocation, local-scope strategy are the criteria used to determine resident set size and the timing of changes. One specific strategy that has received much attention in the literature is known as the working set strategy .

The variable-allocation, global-scope perhaps the easiest to implement and has been adopted in a number of operating systems. At any given time, there are a number of processes in physical memory, each with a certain number of frames allocated to it. Typically, the operating system also maintains a list of free frames. When a page fault occurs, a free frame is added to the resident set of a process and the page is brought in. Thus, a process experiencing page faults will gradually grow in size, which should help reduce overall page faults in the system.

The difficulty with this approach is in the replacement choice. When there are no free frames available, the operating system must choose a page currently in memory to replace. The selection is made from among all of the frames in memory, except for locked frames such as those of the kernel. Using any of the policies discussed in the preceding subsection, the page selected for replacement can belong to any of the resident processes; there is no discipline to determine which process should lose a page from its resident set. Therefore, the process that suffers the reduction in resident set size may not be optimum.

Slide 38

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high.
  - This leads to low CPU utilization
- A common wisdom to improve CPU utilization is to increase the degree of multiprogramming
  - Another process is added to the system
  - CPU utilization drops further

CE2005/CZ2005 Operating Systems 7.38 Virtual Memory

How can this happen?

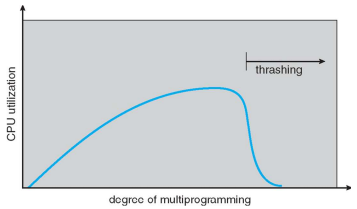
Consider what occurs if a process does not have "enough" frames—that is, it does not have the minimum number of frames it needs to support all pages required during the current stage of execution. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called thrashing: The system spends most of its time paging-in and out rather than executing instructions. The avoidance of thrashing was a major research area in the 1970s and led to a variety of complex but effective algorithms. In essence, the operating system tries to guess, based on recent history, which pieces are least likely to be used in the near future.

local replacement result in less thrashing than global replacement

Slide 39

Thrashing (Cont.)



- *Thrashing* ≡ a process is busy bringing pages in and out (no work is being done)

CE2005/CZ2005 Operating Systems 7.39 Virtual Memory

As the multiprogramming level increases from a small value, one would expect to see processor utilization rise, because there is less chance that all resident processes are blocked. However, a point is reached at which the average resident set is inadequate. At this point, the number of page faults rises dramatically, and processor utilization collapses.

CPU

no of frames

a lot of disk IO

Slide 40

Strategies to combat Thrashing

- Two approaches are used:
  - Working-Set Model
  - Page-Fault Frequency

CE2005/CZ2005 Operating Systems 7.40 Virtual Memory

Slide 41

Working-Set Model

- *Locality of Reference*: Processes tend to refer to pages in a localised manner
  - *Temporal Locality*: locations referred to just before are likely to be referred to again (i.e., location references are clustered over time)
  - *Spatial Locality*: code & data are usually clustered physically
- This observation led to the Working-Set Model

To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it "needs"? One strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.

Except for branch and procedure/function calls, program execution is sequential. Hence, in most cases the next instruction to be fetched immediately follows the last instruction fetched.

Most of iterative constructs consist of a relatively small number of instructions repeated many times. For the duration of iteration, computation is therefore confined to a small contiguous portion of a program.

It is rare to have a long chain of nested procedure/function invocations. Rather, a program confined to a rather narrow window of procedure-invocation depth. Thus, over a short period of time references to instructions tend to be localized to a few procedures.

In many programs, much of the computation involves processing data structures, such as arrays or sequences of records. In many cases, successive references to these data structures require access to closely located data items.

The principle of locality states that program and data references within a process tend to cluster. Hence, the assumption that only a few pieces of a process will be needed over a short period of time is valid. Also, it should be possible to make intelligent guesses about which pieces of a process will be needed in the near future, which avoids thrashing.

Slide 42

Working-Set Model

Locality Example

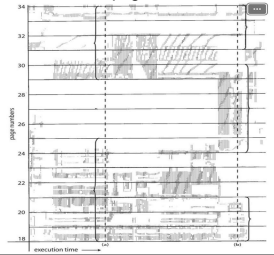
```
sum = 0;
for (I =0; I < n; i++)
  sum += a[i];
return sum;
```

- Data
  - Temporal: `sum` referenced in each iteration
  - Spatial: elements of array `a[]` accessed in consecutive manner in the code
- Instructions
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

Slide 43

Working-Set Model (Cont.)

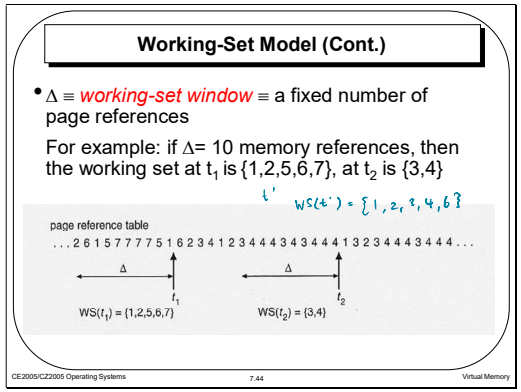
- During the lifetime of the process, references are confined to a subset of pages



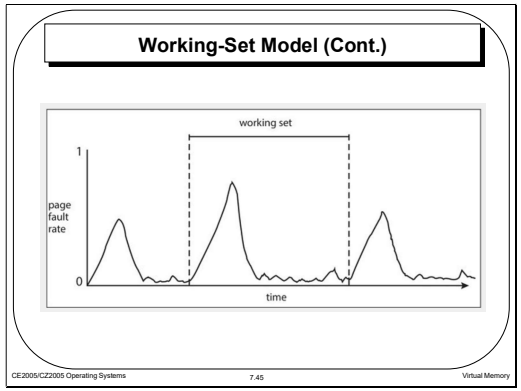
One way to confirm the principle of locality is to look at the logical memory access pattern of a process. The figure illustrates the concept of locality and how a process's locality changes over time. At time (a), the locality is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}. At time (b), the locality changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}. Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.

The figure shown on this slide is a rather famous diagram that dramatically illustrates the principle of locality [HATF72].

Slide 44



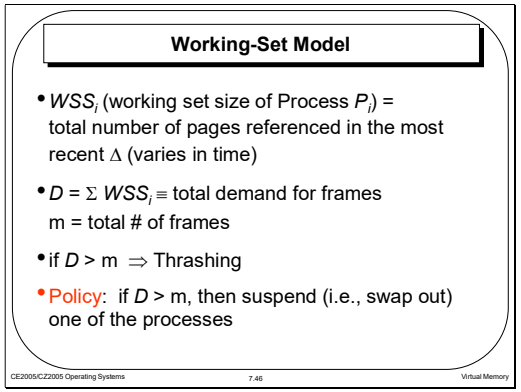
Slide 45



There is a direct relationship between the working set of a process and its page-fault rate. Typically, the working set of a process changes over time as references to data and code sections move from one locality to another. Assuming there is sufficient memory to store the working set of a process (that is, the process is not thrashing), the page-fault rate of the process will transition between peaks and valleys over time.

A peak in the page-fault rate occurs when we begin demand-paging a new locality. However, once the working set of this new locality is in memory, the page-fault rate falls. When the process moves to a new working set, the page-fault rate rises toward a peak once again, returning to a lower rate once the new working set is loaded into memory. The span of time between the start of one peak and the start of the next peak represents the transition from one working set to another.

Slide 46

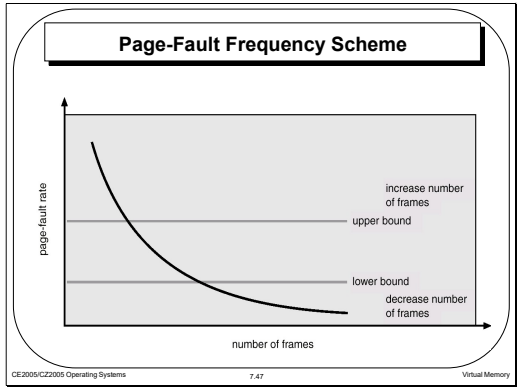


There are a number of problems associated with the working set model.

The optimal value of Δ is unknown and in any case would vary during a process execution.

If Δ too small, it will not encompass entire locality; and if Δ too large, it will encompass several localities.

Slide 47



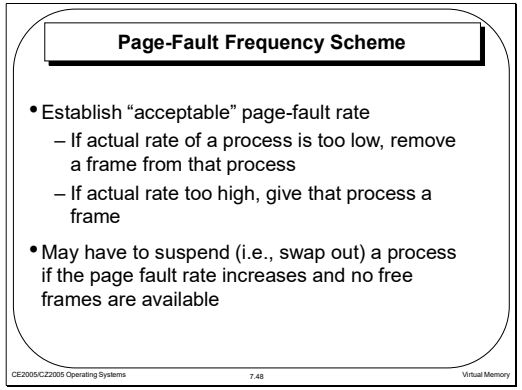
Establish “acceptable” page-fault rate

If actual rate of a process is too low, remove a frame from that process

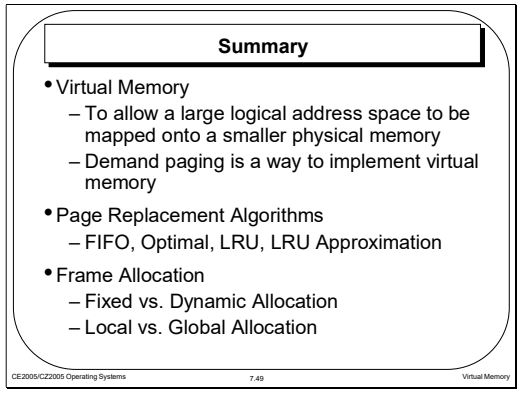
If actual rate too high, give that process a frame

May have to suspend (i.e., swap out) a process if the page fault rate increases and no free frames are available

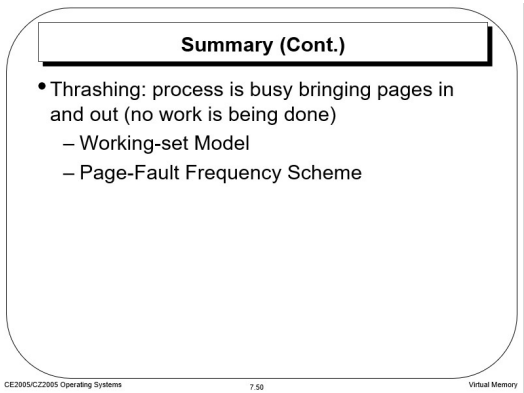
Slide 48



Slide 49



Slide 50



To use the processor and the I/O facilities efficiently, it is desirable to maintain as many processes in physical memory as possible. In addition, it is desirable to free programmers from size restrictions in program development.

The way to address both of these concerns is virtual memory. With virtual memory, all address references are logical references that are translated at run time to real addresses. This allows a process to be located anywhere in physical memory and for that location to change over time. Virtual memory also allows a process to be broken up into pieces. These pieces need not be contiguously located in physical memory during execution and, indeed, it is not even necessary for all of the pieces of the process to be in physical memory during execution.

Design Issues: page replacement policies; frame allocation policies

Performance Issue: thrashing

True or False

Logical address needs to be translated to physical address during execution if address binding is done in load time. False → logical addr & physical addr the same in this case, can be used directly → abs addr format

Memory compaction is used to reduce internal fragmentation. False → used for external fragmentation

Consider a paging system with page table stored in memory. Using translation look-aside buffer (TLB) will always reduce effective memory access time. False → if hit ratio is low, increase effective memory access time

Using inverted page table increases address translation time. True → cause have to search ENTIRE address translation time

If segmentation is used for memory allocation, the logical address space of a process will remain contiguous in physical memory. False
↳ should be non-contiguous

True or False

Context switch contributes the most to the overhead of handling page faults. False → Disk I/O contribute the most to overheads

As the number of available page frames increases, the number of page faults will always decrease no matter which page replacement algorithm is used. False → recall belady's anomaly → increase no of frame doesn't neccessarily mean no. of page faults decrease

Using demand paging for virtual memory support, CPU utilization can be always improved by increasing the degree of multi-programming False → thrashing concept used here

Variable frame allocation implies a global policy must be used for page replacement. False

Thrashing can be detected by evaluating the level of CPU utilization as compared to the level of multiprogramming. True → thrashing ↓ low CPU utilization & a lot of disk I/O ↓ loading too many processes