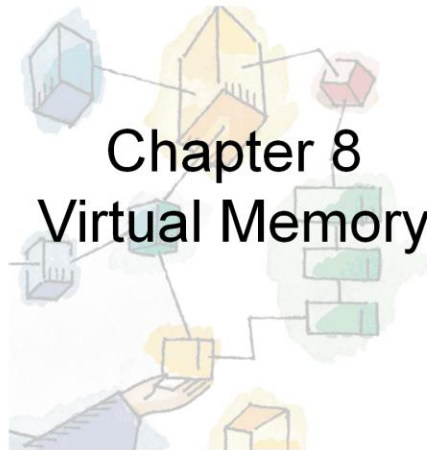
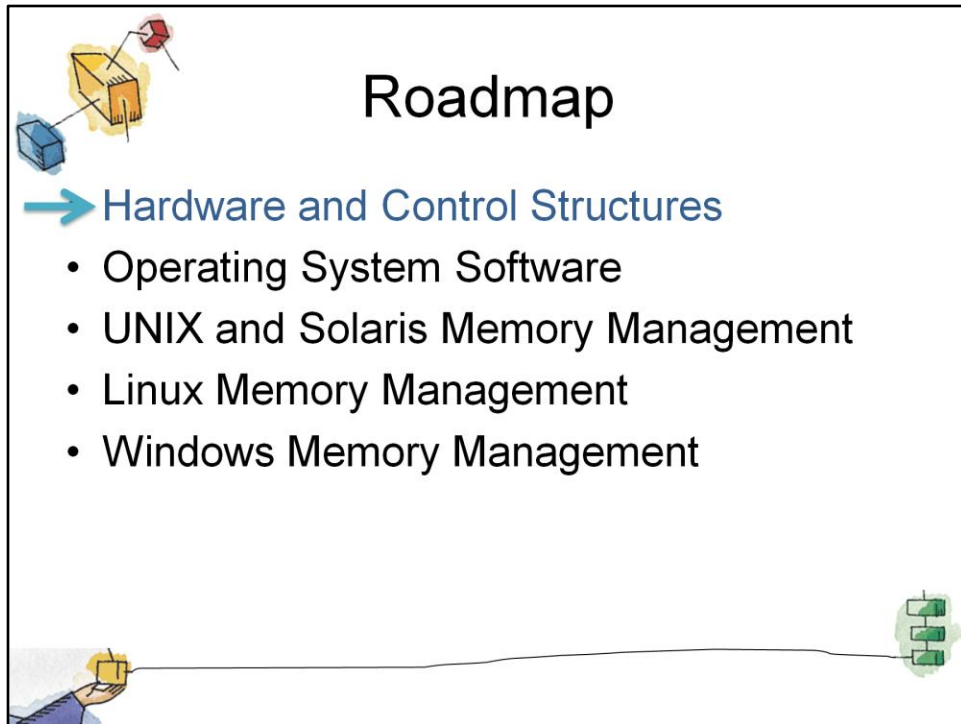


*Operating Systems:
Internals and Design Principles, 6/E*
William Stallings



Dave Bremer
Otago Polytechnic, N.Z.
©2008, Prentice Hall

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.



Memory management is a complex interrelationship between processor hardware and operating system software.

We focus first on the hardware aspect of virtual memory, looking at the use of paging, segmentation, and combined paging and segmentation.

Then we look at the issues involved in the design of a virtual memory facility in operating systems.



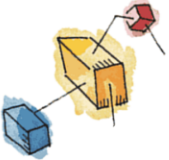
Terminology

Table 8.1 Virtual Memory Terminology

Virtual memory	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.
Virtual address	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
Virtual address space	The virtual storage assigned to a process.
Address space	The range of memory addresses available to a process.
Real address	The address of a storage location in main memory.





Some key terms related to virtual memory.



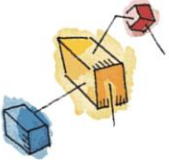
Key points in Memory Management

- 1) Memory references are logical addresses dynamically translated into physical addresses at run time
 - A process may be swapped in and out of main memory occupying different regions at different times during execution
- 2) A process may be broken up into pieces that do not need to be located contiguously in main memory





Two characteristics of paging and segmentation are the keys to this breakthrough in memory management:

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time.
 - A process may be swapped in and out of main memory occupying different regions of main memory at different times during the course of 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 main memory during execution.
 - The combination of dynamic run-time address translation and the use of a page or segment table permits this.



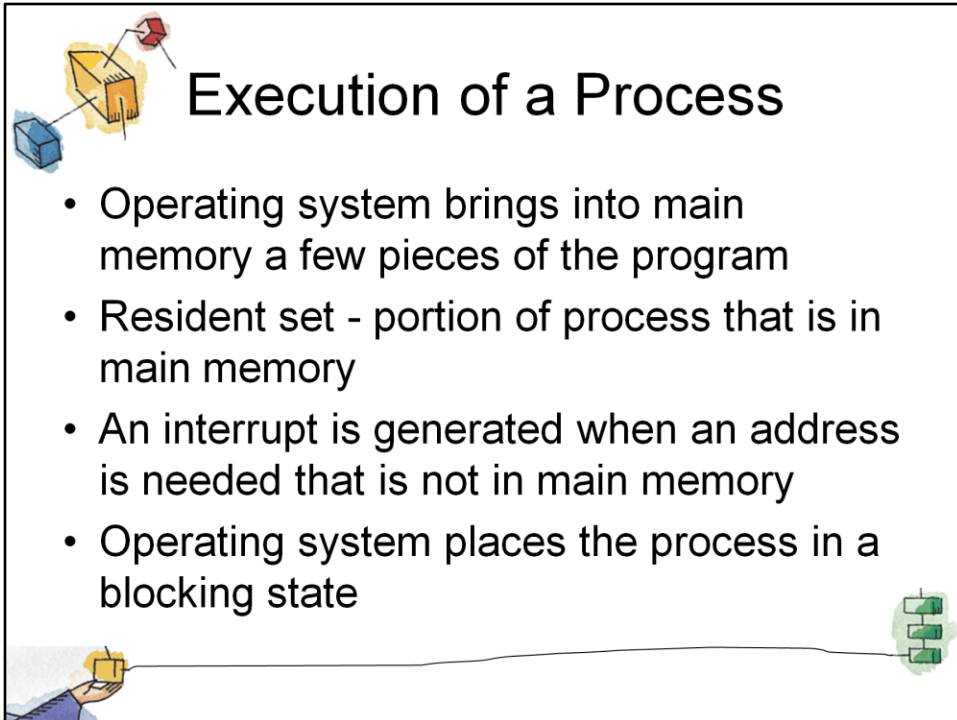
Breakthrough in Memory Management

- **If both** of those two characteristics are present,
 - then it is not necessary that all of the pages or all of the segments of a process be in main memory during execution.
- If the next instruction, and the next data location are in memory then execution can proceed
 - at least for a time



If the piece (segment or page) that holds the next instruction to be fetched and the piece that holds the next data location to be accessed are in main memory,

- then execution may proceed (at least for a time).



Execution of a Process

- Operating system brings into main memory a few pieces of the program
- Resident set - portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory
- Operating system places the process in a blocking state

Suppose that it is time to bring a new process into memory.

The operating system begins by bringing in only one or a few pieces, to include the initial program piece and the initial data piece to which those instructions refer.

The portion of a process that is actually in main memory at any time is defined to be the **resident set of the process**.

- As the process executes, things proceed smoothly as long as all memory references are to locations that are in the resident set.
- Using the segment or page table, the processor always is able to determine whether this is so.

If the processor encounters a logical address that **is not** in main memory,

- it generates an interrupt indicating a memory access fault.

The operating system puts the interrupted process in a blocking state and takes control.

Execution of a Process

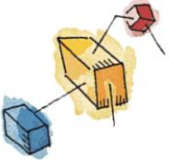
- Piece of process that contains the logical address is brought into main memory
 - Operating system issues a disk I/O Read request
 - Another process is dispatched to run while the disk I/O takes place
 - An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state

For the execution of this process to proceed later, the operating system will need to bring into main memory the piece of the process that contains the logical address that caused the access fault.

For this purpose, the operating system issues a disk I/O read request.



After the I/O request has been issued, the operating system can dispatch an other process to run while the disk I/O is performed.

Once the desired piece has been brought into main memory, an I/O interrupt is issued, giving control back to the operating system, which places the affected process back into a Ready state.



Implications of this new strategy

- More processes may be maintained in main memory
 - Only load in some of the pieces of each process
 - With so many processes in main memory, it is very likely a process will be in the Ready state at any particular time
- A process may be larger than all of main memory




1. More processes may be maintained in main memory.

- Because we are only going to load some of the pieces of any particular process, there is room for more processes.
- This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in a Ready state at any particular time.


2. A process may be larger than all of main memory.

- Without the scheme a programmer must be acutely aware of how much memory is available.
- If the program being written is too large, the programmer must devise ways to structure the program into pieces that can be loaded separately in some sort of overlay strategy.
- With virtual memory based on paging or segmentation, that job is left to the operating system and the hardware. As far as the programmer is concerned, he or she is dealing with a huge memory, the size associated with disk storage.
- The operating system automatically loads pieces of a process into main memory as required.

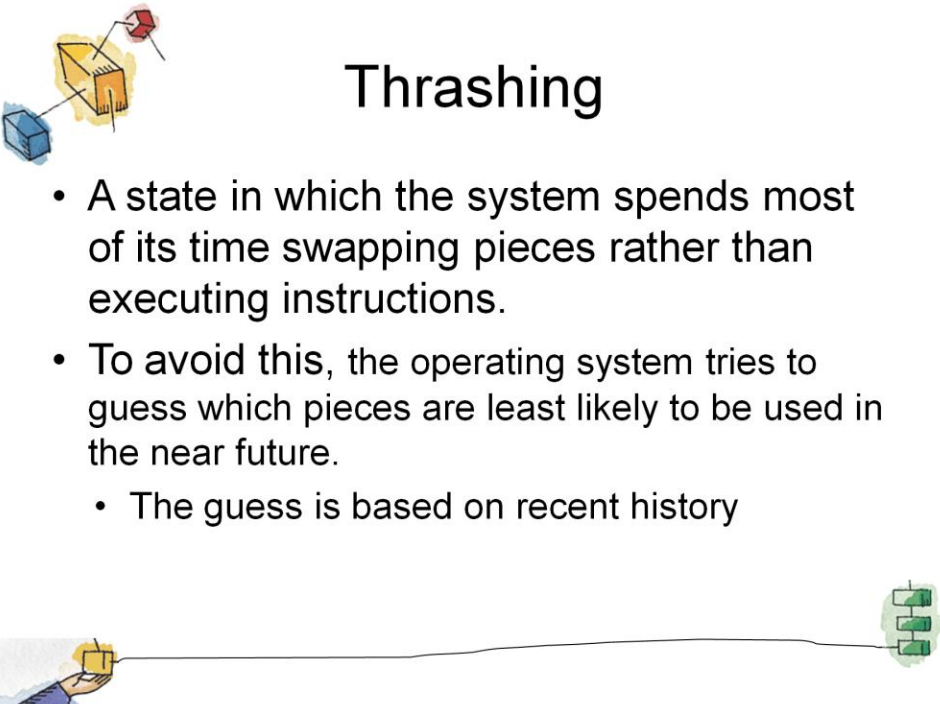


Real and Virtual Memory

- Real memory
 - Main memory, the actual RAM
- Virtual memory
 - Memory on disk
 - Allows for effective multiprogramming and relieves the user of tight constraints of main memory



Refer to table 8.2 in the book for a comparison of the characteristics of paging and segmentation, with and without the use of virtual memory.



Thrashing

- A state in which the system spends most of its time swapping pieces rather than executing instructions.
- To avoid this, the operating system tries to guess which pieces are least likely to be used in the near future.
 - The guess is based on recent history

In the steady state, practically all of main memory will be occupied with process pieces, so that the processor and operating system have direct access to as many processes as possible.

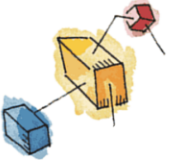
Thus, when the operating system brings one piece in, it must throw another out.

If it throws out a piece just before it is used, then it will just have to go get that piece again almost immediately.

- Too much of this leads to a condition known as thrashing
- The system spends most of its time swapping pieces 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.



Principle of Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Therefore it is possible to make intelligent guesses about which pieces will be needed in the future
- This suggests that virtual memory may work efficiently

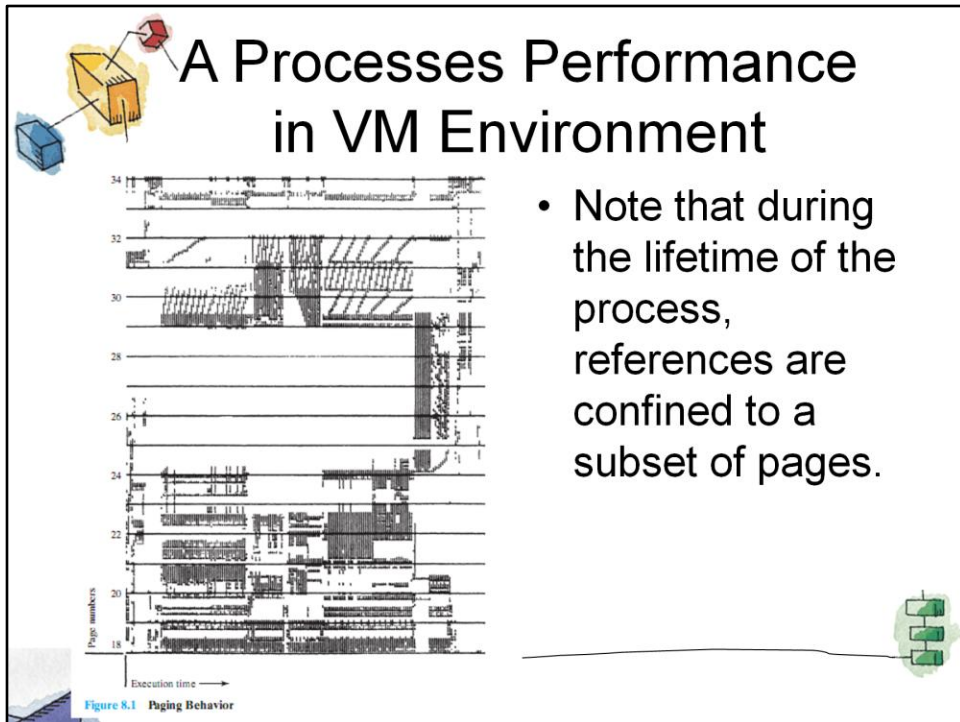


The principle of locality states that program and data references within a process tend to cluster.

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.

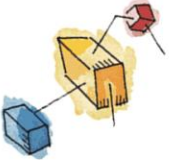
One way to confirm the principle of locality is to look at the performance of processes in a virtual memory environment.



Give a brief description of this figure to the students – the detail probably not be clear in the presentation, but it is the overall clustering that is important and interesting


Figure 8.1 is a rather famous diagram that dramatically illustrates the principle of locality [HATF72].

Note that during the lifetime of the process, references are confined to a subset of pages.




Support Needed for Virtual Memory

- Hardware must support paging and segmentation
- Operating system must be able to manage the movement of pages and/or segments between secondary memory and main memory



For virtual memory to be practical and effective, two ingredients are needed.



- 1) there must be hardware support for the paging and/or segmentation scheme to be employed.
- 2) the operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory.



Paging

- Each process has its own page table
- Each page table entry contains the frame number of the corresponding page in main memory
- Two extra bits are needed to indicate:
 - whether the page is in main memory or not
 - Whether the contents of the page has been altered since it was last loaded

(see next slide)



In the discussion of simple paging, we indicated that each process has its own page table.

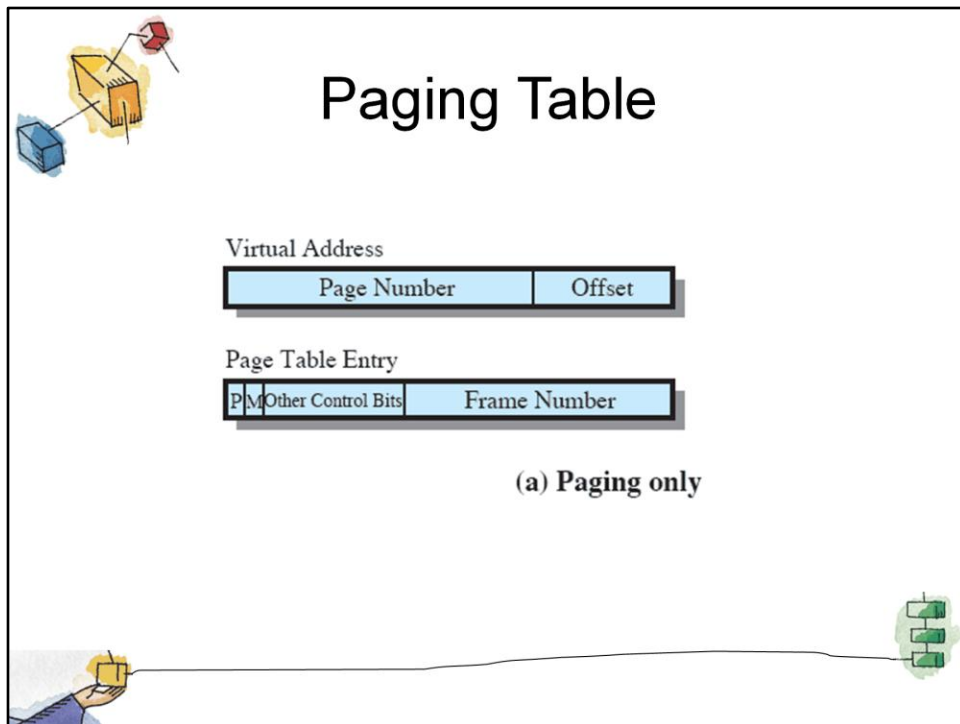
- When all of its pages are loaded into main memory, the page table for a process is created and loaded into main memory.

Each page table entry contains the frame number of the corresponding page in main memory.

A page table is also needed for a virtual memory scheme based on paging.

Again, it is typical to associate a unique page table with each process. **(Figure 8.2a – next slide).**

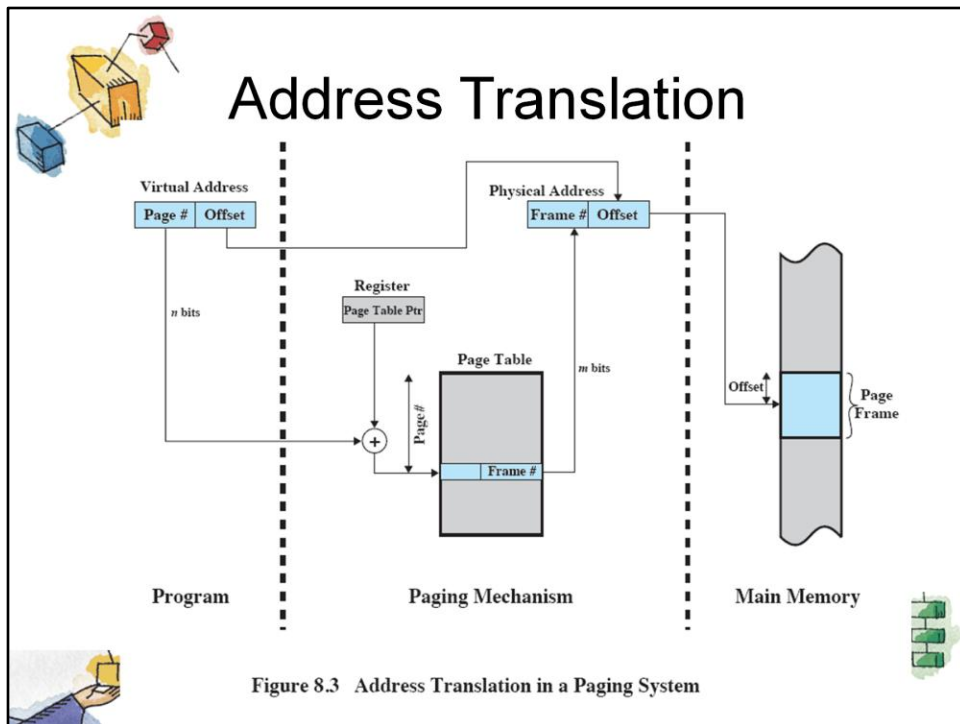
- The page table entries become more complex because only some of the pages of a process may be in main memory.
- A bit is needed in each page table entry to indicate whether the corresponding page is present (P) in main memory or not.
- If the bit indicates that the page is in memory, then the entry also includes the frame number of that page.
- The page table entry includes a modify (M) bit, indicating whether the contents of the corresponding page have been altered since the page was last loaded



This is a repeat of the notes on the previous slide

It is typical to associate a unique page table with each process. **(Figure 8.2a – this slide).**

- A bit is needed in each page table entry to indicate whether the corresponding page is present (P) in main memory or not.
- If the bit indicates that the page is in memory, then the entry also includes the frame number of that page.
- The page table entry includes a modify (M) bit, indicating whether the contents of the corresponding page have been altered since the page was last loaded



The basic mechanism for reading a word from memory involves using a page table to translate a virtual/logical address, consisting of page number and offset,

Into a physical address, consisting of frame number and offset,

Because the page table is of variable length, depending on the size of the process, we cannot expect to hold it in registers.

Instead, it must be in main memory to be accessed.

Figure 8.3 suggests a hardware implementation.

When a particular process is running, a register holds the starting address of the page table for that process.

- The page number of a virtual address is used to index that table and look up the corresponding frame number.
- This is combined with the offset portion of the virtual address to produce the desired real address.

Typically, the page number field is longer than the frame number field ($n > m$).



Page Tables

- Page tables are also stored in virtual memory
- When a process is running, part of its page table is in main memory



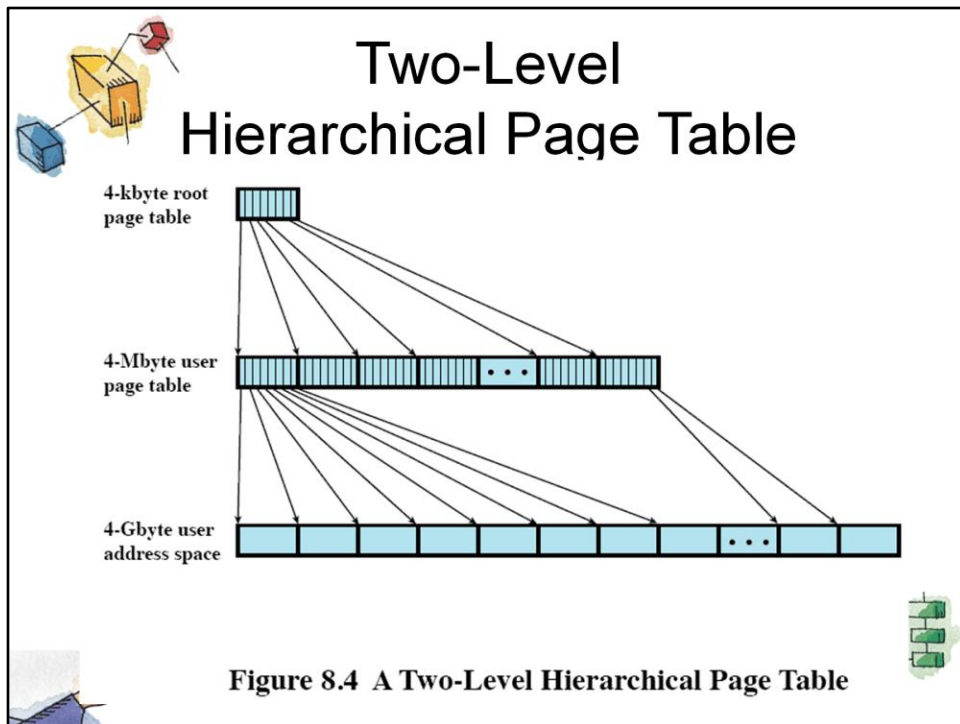


Figure 8.4 shows an example of a two-level scheme typical for use with a 32-bit address.

If we assume byte-level addressing and 4-kbyte (2^{12}) pages,

- then the 4-Gbyte (2^{32}) virtual address space is composed of 220pages.

If each of these pages is mapped by a 4-byte page table entry (PTE), we can create a user page table composed of 2^{20} PTEs requiring 4 Mbyte (2^{22}) bytes.

This huge user page table, occupying 2^{10} pages, can be kept in virtual memory and mapped by a root page table with 2^{10} PTEs occupying

4 Kbyte (2^{12}) of main memory.

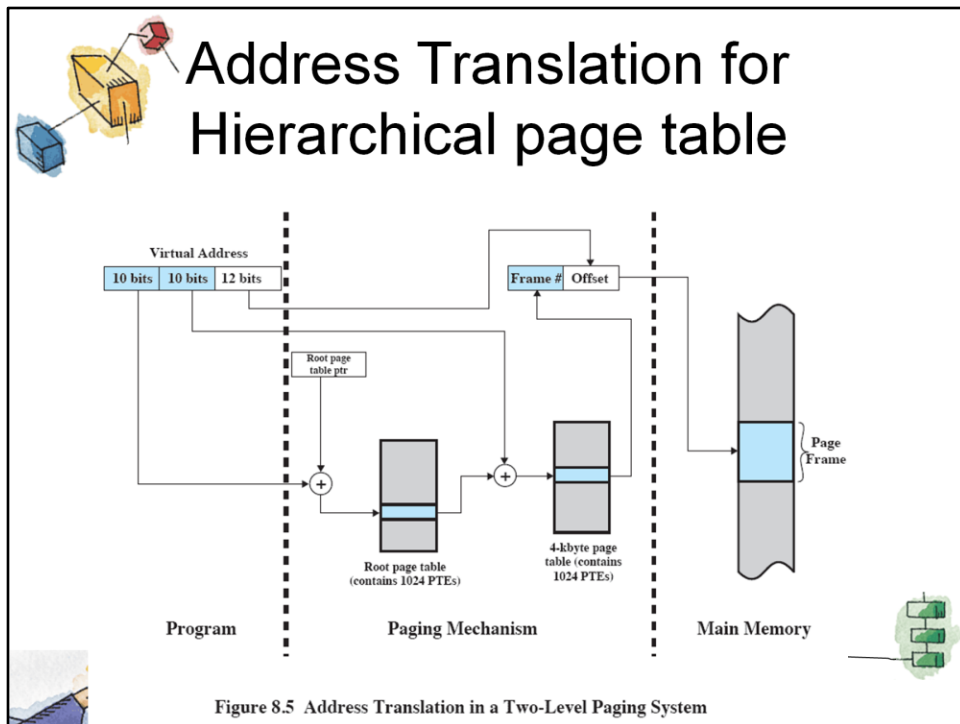


Figure 8.5 shows the steps involved in address translation for this scheme.

The root page always remains in main memory.

The first 10 bits of a virtual address are used to index into the root page to find a PTE for a page of the user page table.

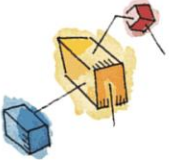
- If that page is **not** in main memory, a page fault occurs.
- If that page **is** in main memory, then the next 10 bits of the virtual address index into the user PTE page to find the PTE for the page that is referenced by the virtual address.



Page tables grow proportionally



- A drawback of the type of page tables just discussed is that their size is proportional to that of the virtual address space.
- An alternative is Inverted Page Tables





Inverted Page Table

- Used on PowerPC, UltraSPARC, and IA-64 architecture
- Page number portion of a virtual address is mapped into a hash value
- Hash value points to inverted page table
- Fixed proportion of real memory is required for the tables regardless of the number of processes



The page number portion of a virtual address is mapped into a hash value using a simple hashing function.

The hash value is a pointer to the inverted page table, which contains the page table entries.


There is one entry in the inverted page table for each real memory page frame rather than one per virtual page.

- Thus a fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported.

Because more than one virtual address may map into the same hash table entry, a chaining technique is used for managing the overflow.

The hashing technique results in chains that are typically short—between one and two entries.



The page table's structure is called inverted because it indexes page table entries by frame number rather than by virtual page number



Inverted Page Table

Each entry in the page table includes:

- Page number
- Process identifier
 - The process that owns this page.
- Control bits
 - includes flags, such as valid, referenced, etc
- Chain pointer
 - the index value of the next entry in the chain.



Page number:

- This is the page number portion of the virtual address.

Process identifier:

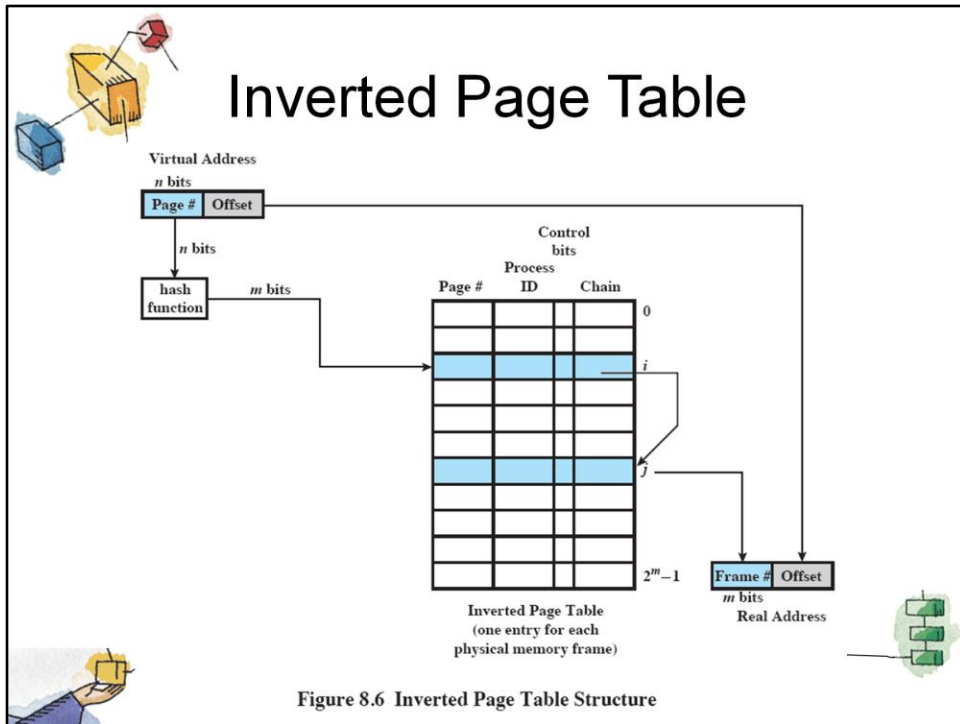
- The process that owns this page.
- The combination of page number and process identifier identify a page within the virtual address space of a particular process.

Control bits:

- This field includes flags, such as valid, referenced, and modified; and protection and locking information.

Chain pointer:

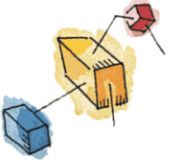
- Contains the index value (number between 0 and $2^m - 1$) of the next entry in the chain.
- Otherwise, Null if there are no chained entries for this entry.



Refer to previous description of the fields



In this example, the virtual address includes an n -bit page number, with $n > m$.

The hash function maps the n -bit page number into an m -bit quantity, which is used to index into the inverted page table.



Translation Lookaside Buffer

- Each virtual memory reference can cause two physical memory accesses
 - One to fetch the page table
 - One to fetch the data
- To overcome this problem a high-speed cache is set up for page table entries
 - Called a Translation Lookaside Buffer (TLB)
 - Contains page table entries that have been most recently used



This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used.



TLB Operation

- Given a virtual address,
 - processor examines the TLB
- If page table entry is present (TLB hit),
 - the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss),
 - the page number is used to index the process page table





Looking into the Process Page Table

- First checks if page is already in main memory
 - If not in main memory a page fault is issued
- The TLB is updated to include the new page entry



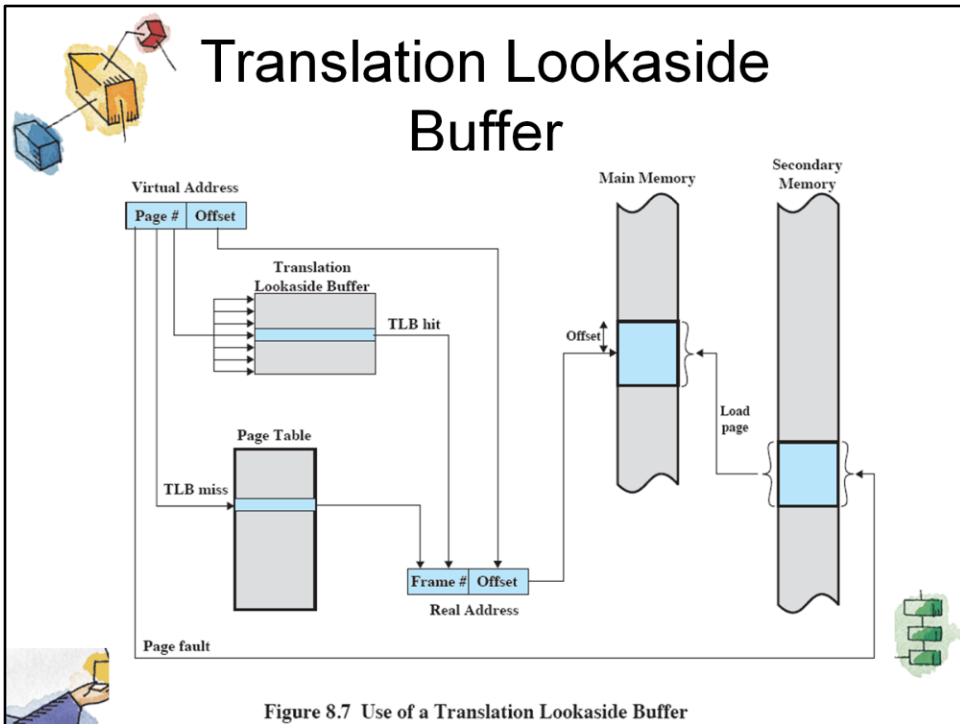
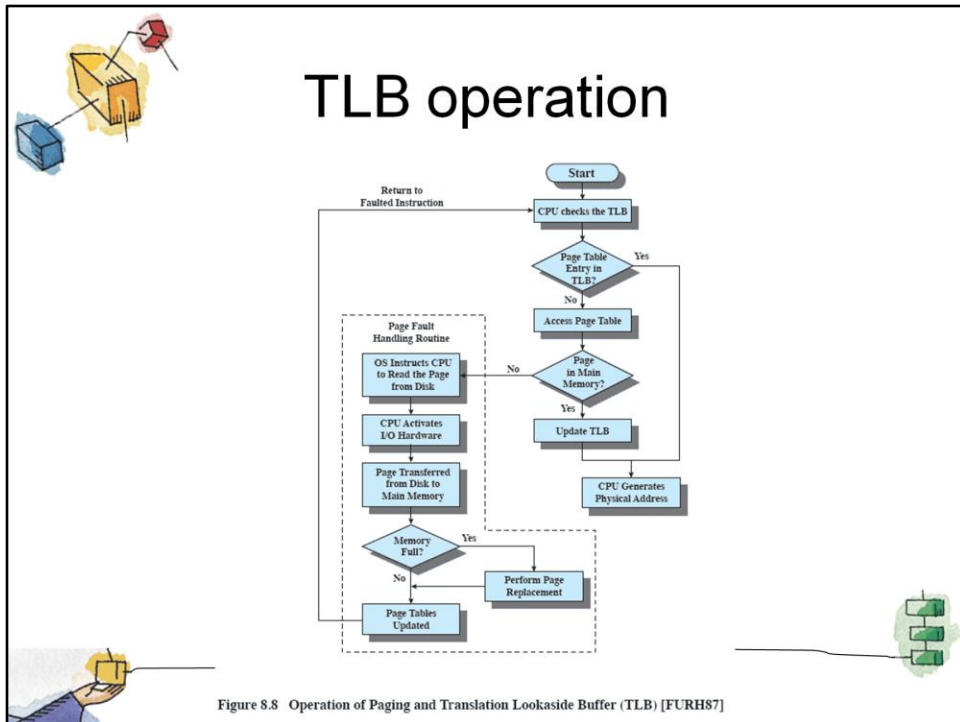



Figure 8.7 Use of a Translation Lookaside Buffer



The flowchart shows that if the desired page is not in main memory, a page fault interrupt causes the page fault handling routine to be invoked.


To keep the flowchart simple, the fact that the operating system may dispatch another process while disk I/O is underway is not shown.

By the principle of locality, most virtual memory references will be to locations in recently used pages. Therefore, most references will involve page table entries in the cache.



Associative Mapping

- As the TLB only contains some of the page table entries we cannot simply index into the TLB based on the page number
 - Each TLB entry must include the page number as well as the complete page table entry
- The process is able to simultaneously query numerous TLB entries to determine if there is a page number match

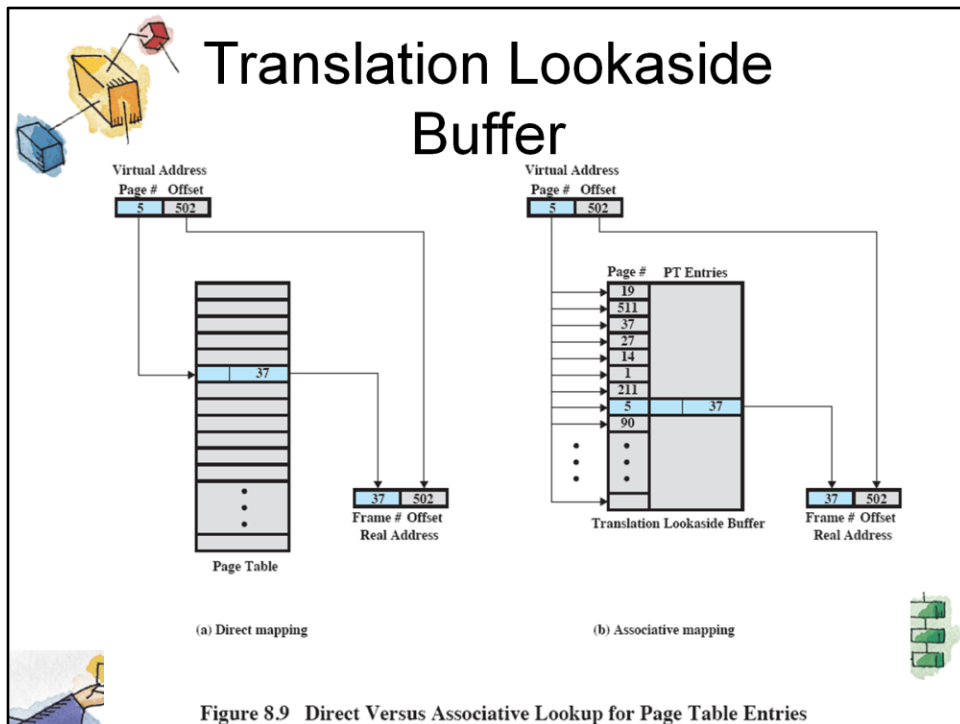


Because the TLB only contains some of the entries in a full page table, we cannot simply index into the TLB based on page number.

- Instead, each entry in the TLB must include the page number as well as the complete page table entry.

The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number.

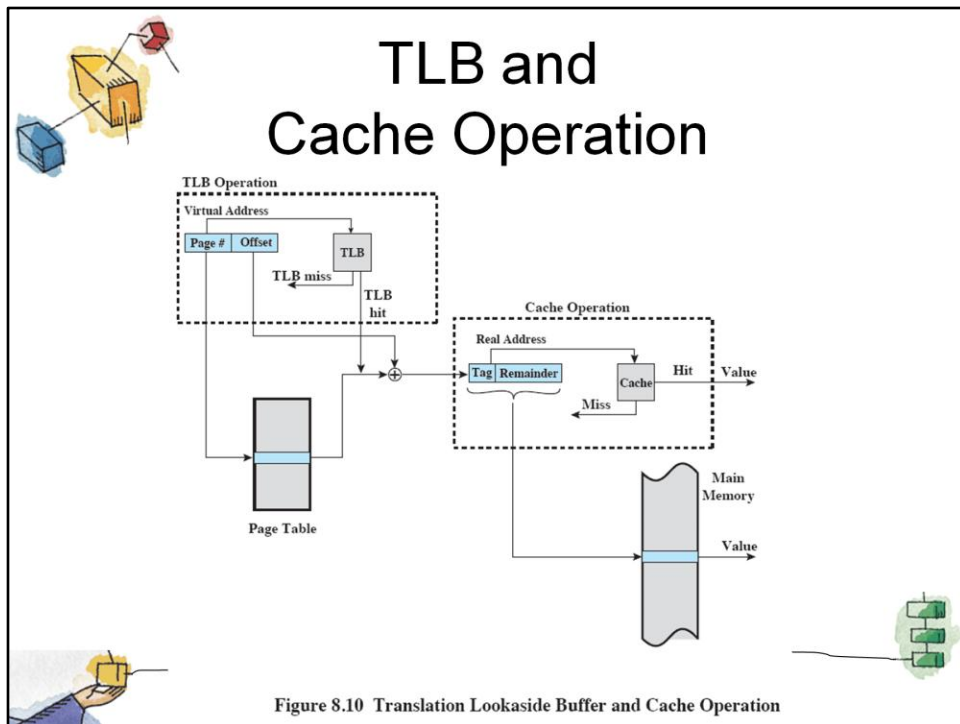
This technique is referred to as associative mapping.



This slide contrasts associative mapping with direct mapping or indexing, used for lookup in the page table.

The design of the TLB also must consider the way in which entries are organized in the TLB and which entry to replace when a new entry is brought in.

These issues must be considered in any hardware cache design.




The virtual memory mechanism must interact with the cache system (not the TLB cache, but the main memory cache)

A virtual address will generally be in the form of a page number, offset.



First, the memory system consults the TLB to see if the matching page table entry is present.

- If **it is**, the real (physical) address is generated by combining the frame number with the offset.
- If **not**, the entry is accessed from a page table.
 - Once the real address is generated, which is in the form of a tag and a remainder, the cache is consulted to see if the block containing that word is present.
 - If so, it is returned to the CPU.
 - If not, the word is retrieved from main memory.



Page Size

- Smaller page size, less amount of internal fragmentation
- But Smaller page size, more pages required per process
 - More pages per process means larger page tables
- Larger page tables means large portion of page tables in virtual memory



An important hardware design decision is the size of page to be used.

There are several factors to consider.

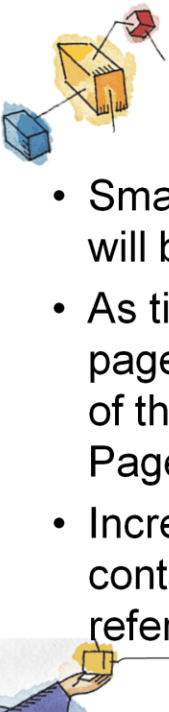
- **internal fragmentation.** Clearly, the smaller the page size, the less the amount of internal fragmentation. To optimize the use of main memory, we would like to reduce internal fragmentation.
- **BUT the smaller the page, the greater the number of pages required per process.** More pages per process means larger page tables. For large programs in a heavily multiprogrammed environment, this may mean that some portion of the page tables of active processes must be in virtual memory, not in main memory.
- The physical characteristics of most secondary-memory devices, which are rotational, favour a larger page size for more efficient block transfer of data



Page Size

- Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better





Further complications to Page Size

- Small page size, large number of pages will be found in main memory
- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.
- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.

Complicating these matters is the effect of page size on the rate at which page faults occur.

This behavior, is based on the principle of locality.

If the page size is very small, then ordinarily a relatively large number of pages will be available in main memory for a process.

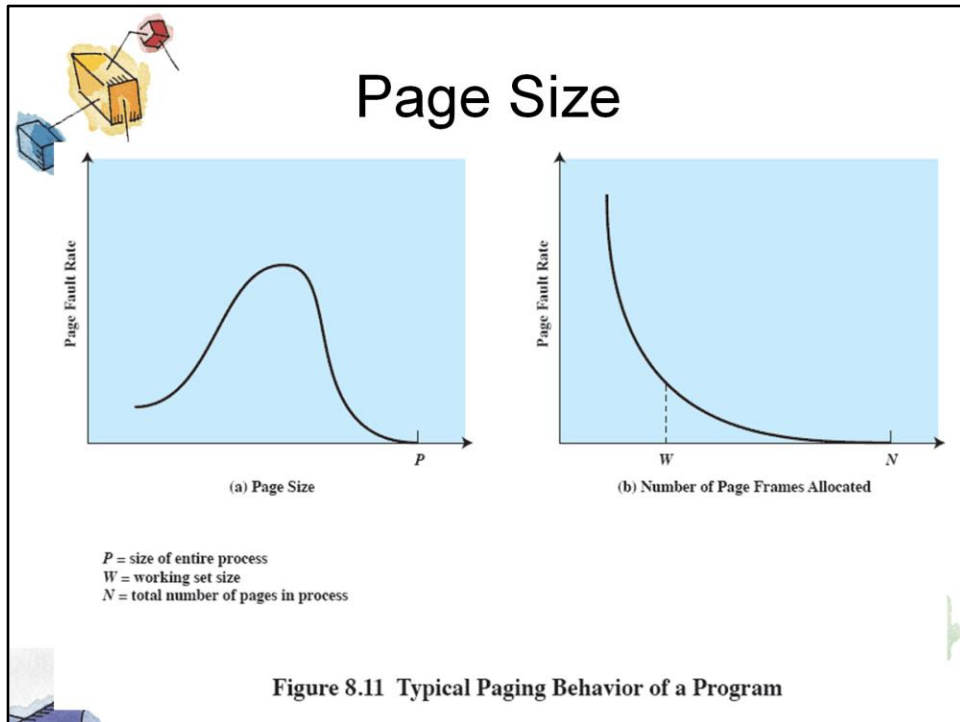
- After a time, the pages in memory will all contain portions of the process near recent references.
- Thus, the page fault rate should be low.

As the size of the page is increased, each individual page will contain locations further and further from any particular recent reference.

- Thus the effect of the principle of locality is weakened and the page fault rate begins to rise.

Eventually, however, the page fault rate will begin to fall as the size of a page approaches the size of the entire process (point *P* in the diagram).

- When a single page encompasses the entire process, there will be no page faults.




Much of this is explained on the previous slide

A further complication is that the page fault rate is also determined by the number of frames allocated to a process.

Figure 8.11b shows that, for a fixed page size, the fault rate drops as the number of pages maintained in main memory grows.



- Thus, a software policy (the amount of memory to allocate to each process) interacts with a hardware design decision (page size).



Example Page Size

Table 8.3 Example Page Sizes

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit word
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBMAS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBMPowerPC	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

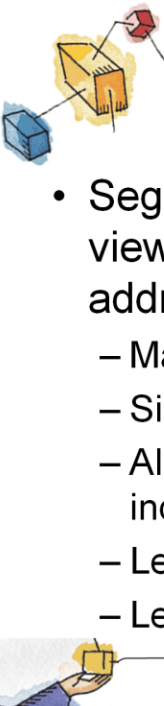



Finally, the design issue of page size is related to the size of physical main memory and program size.

At the same time that main memory is getting larger, the address space used by applications is also growing.

The trend is most obvious on personal computers and workstations, where applications are becoming increasingly complex.

- Furthermore, contemporary programming techniques used in large programs tend to decrease the locality of references within a process



Segmentation

- Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments.
 - May be unequal, dynamic size
 - Simplifies handling of growing data structures
 - Allows programs to be altered and recompiled independently
 - Lends itself to sharing data among processes
 - Lends itself to protection

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments.

Segments may be of unequal, indeed dynamic, size.

Memory references consist of a form of address (segment number, offset).

This organization has a number of advantages to the programmer over a nonsegmented address space:

1. It simplifies the handling of growing data structures.

- If the programmer doesn't know ahead of time how large a particular data structure will become, it is necessary to guess unless dynamic segment sizes are allowed. With segmented virtual memory, the data structure can be assigned its own segment, and the operating system will expand or shrink the segment as needed.
- If a segment that needs to be expanded is in main memory and there is insufficient room, the operating system may move the segment to a larger area of main memory, if available, or swap it out. The enlarged segment would be swapped back in at the next opportunity.

2. It allows programs to be altered and recompiled independently,

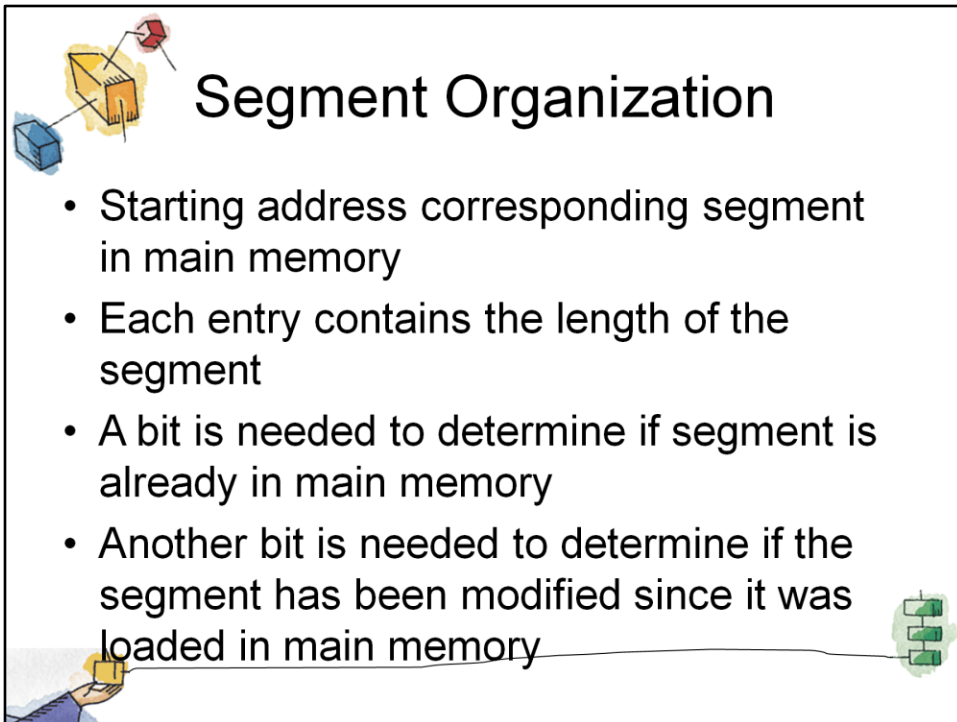
- without requiring the entire set of programs to be relinked and reloaded.
- Again, this is accomplished using multiple segments.

3. It lends itself to sharing among processes.

- A programmer can place a utility program or a useful table of data in a segment that can be referenced by other processes.

4. It lends itself to protection.

- Because a segment can be constructed to contain a well-defined set of programs or data, the programmer or system administrator can assign access privileges in a convenient fashion.



Segment Organization

- Starting address corresponding segment in main memory
- Each entry contains the length of the segment
- A bit is needed to determine if segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory

Each segment table entry contains the starting address of the corresponding segment in main memory, as well as the length of the segment.

The same device, a segment table, is needed when we consider a virtual memory scheme based on segmentation.

- Again, it is typical to associate a unique segment table with each process.

But because only some of the segments of a process may be in main memory, a bit is needed in each segment table entry to indicate whether the corresponding segment is present in main memory or not.

- If the bit indicates that the segment is in memory, then the entry also includes the starting address and length of that segment.

Another control bit in the segmentation table entry is a **modify bit**, indicating whether the contents of the corresponding segment have been altered since the segment was last loaded into main memory.

- If there has been no change, then it is not necessary to write the segment out when it comes time to replace the segment in the frame that it currently occupies.

Other control bits may also be present.



Segment Table Entries

Virtual Address

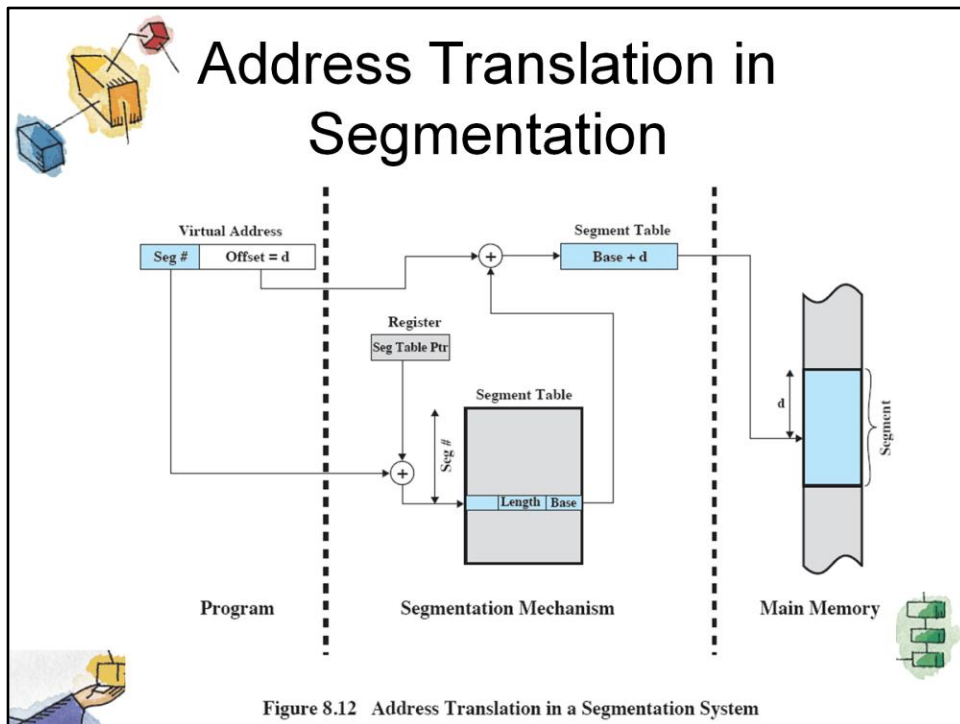
Segment Number	Offset
----------------	--------

Segment Table Entry

P/M	Other Control Bits	Length	Segment Base
-----	--------------------	--------	--------------

(b) Segmentation only





Reading a word from memory involves the translation of a virtual, or logical, address, consisting of segment number and offset, into a physical address, using a segment table.

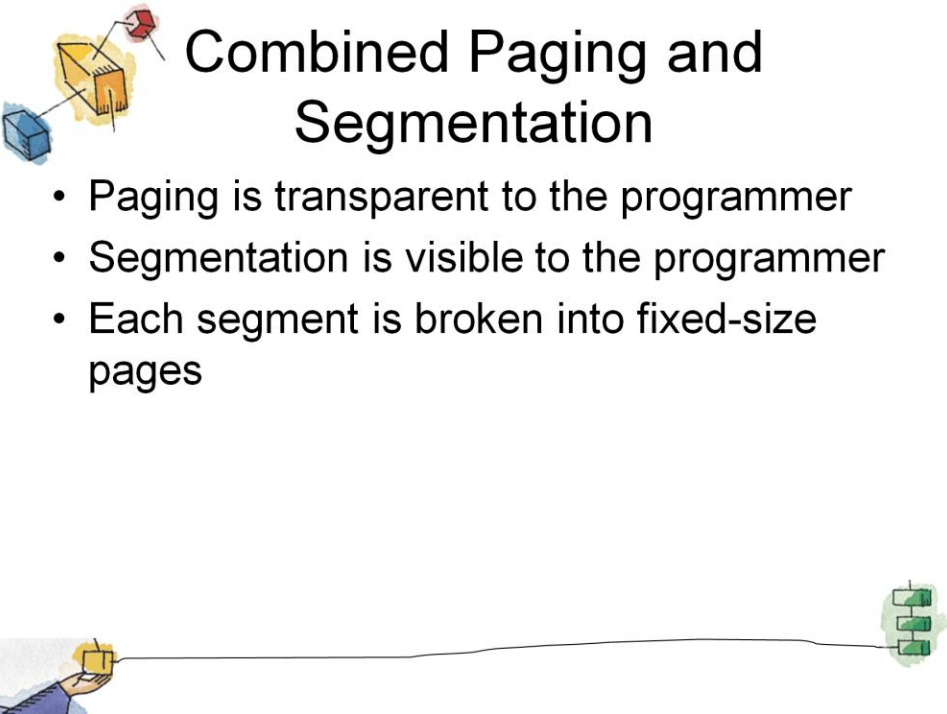
We cannot expect to hold the segment table in registers because it is of variable length, depending on the size of the process

- Instead, it must be in main memory to be accessed.

This figure suggests a hardware implementation of this scheme (note similarity to Figure 8.3).

When a particular process is running, a register holds the starting address of the segment table for that process.

- The segment number of a virtual address is used to index that table and look up the corresponding main memory address for the start of the segment.
- This is added to the offset portion of the virtual address to produce the desired real address.



Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size pages

In a combined paging/segmentation system, a user's address space is broken up into a number of segments,

- at the discretion of the programmer.

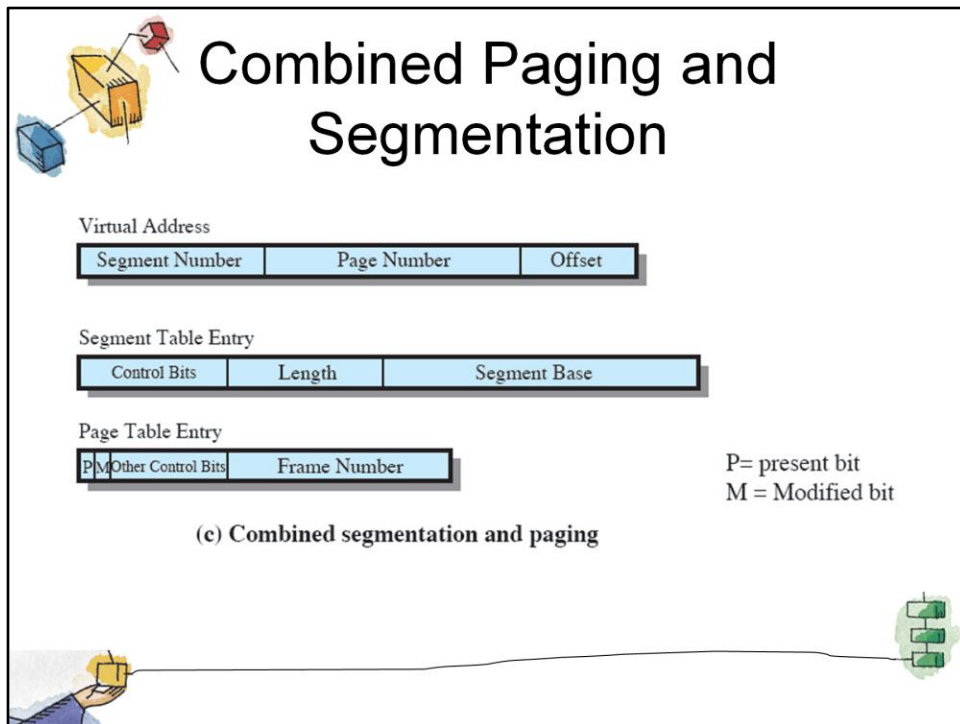
Each segment is, in turn, broken up into a number of fixed-size pages,

- which are equal in length to a main memory frame.

If a segment has length less than that of a page, the segment occupies just one page.

From the programmer's point of view, a logical address still consists of a segment number and a segment offset.

From the system's point of view, the segment offset is viewed as a page number and page offset for a page within the specified segment.



This figure (8.2c) suggests the segment table entry and page table entry formats.

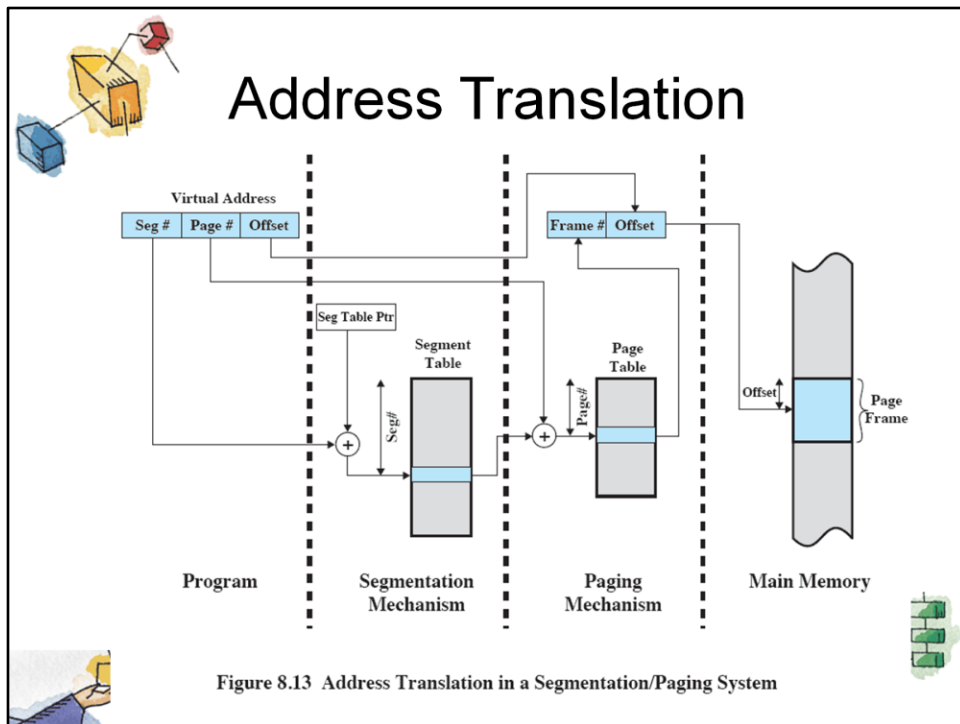
As before, the segment table entry contains the length of the segment.

It also contains a base field, which now refers to a page table.

- The present and modified bits are not needed because these matters are handled at the page level.
- Other control bits may be used, for purposes of sharing and protection.

The page table entry is essentially the same as is used in a pure paging system.

- Each page number is mapped into a corresponding frame number if the page is present in main memory.
- The modified bit indicates whether this page needs to be written back out when the frame is allocated to another page.
- There may be other control bits dealing with protection or other aspects of memory management.



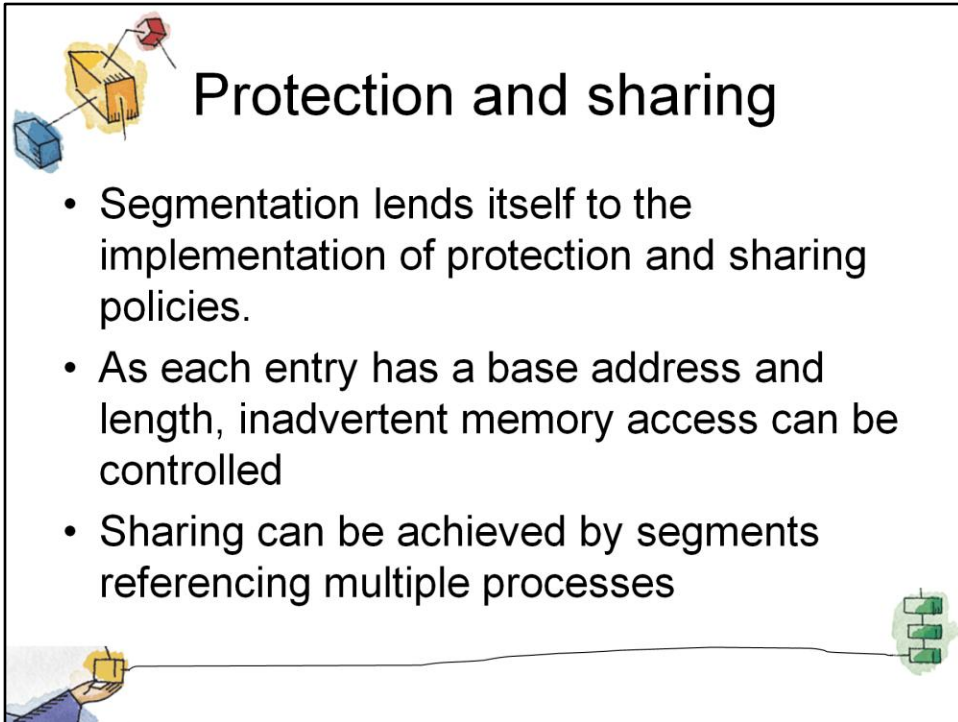
This figure suggests a structure to support combined paging/segmentation (note similarity to Figure 8.5).

Associated with each process is a segment table and a number of page tables, one per process segment.

When a particular process is running, a register holds the starting address of the segment table for that process.

Presented with a virtual address, the processor uses the segment number portion to index into the process segment table to find the page table for that segment.

- Then the page number portion of the virtual address is used to index the page table and look up the corresponding frame number.
- This is combined with the offset portion of the virtual address to produce the desired real address.



Protection and sharing

- Segmentation lends itself to the implementation of protection and sharing policies.
- As each entry has a base address and length, inadvertent memory access can be controlled
- Sharing can be achieved by segments referencing multiple processes

Segmentation lends itself to the implementation of protection and sharing policies.

As each segment table entry includes a length as well as a base address, a program cannot inadvertently access a main memory location beyond the limits of a segment.

To achieve sharing, it is possible for a segment to be referenced in the segment tables of more than one process.

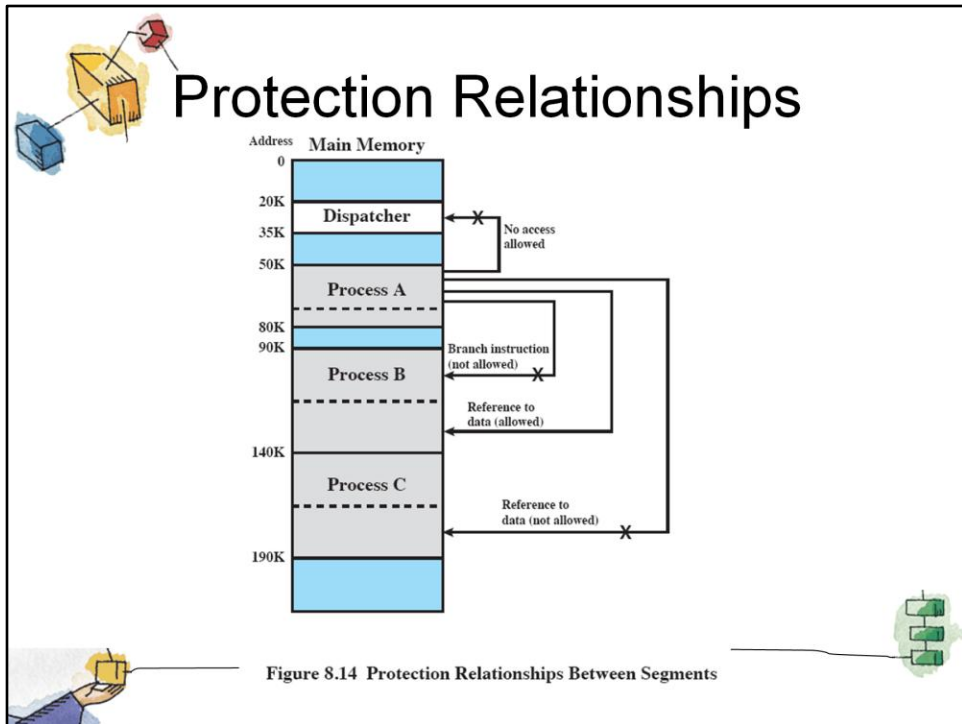


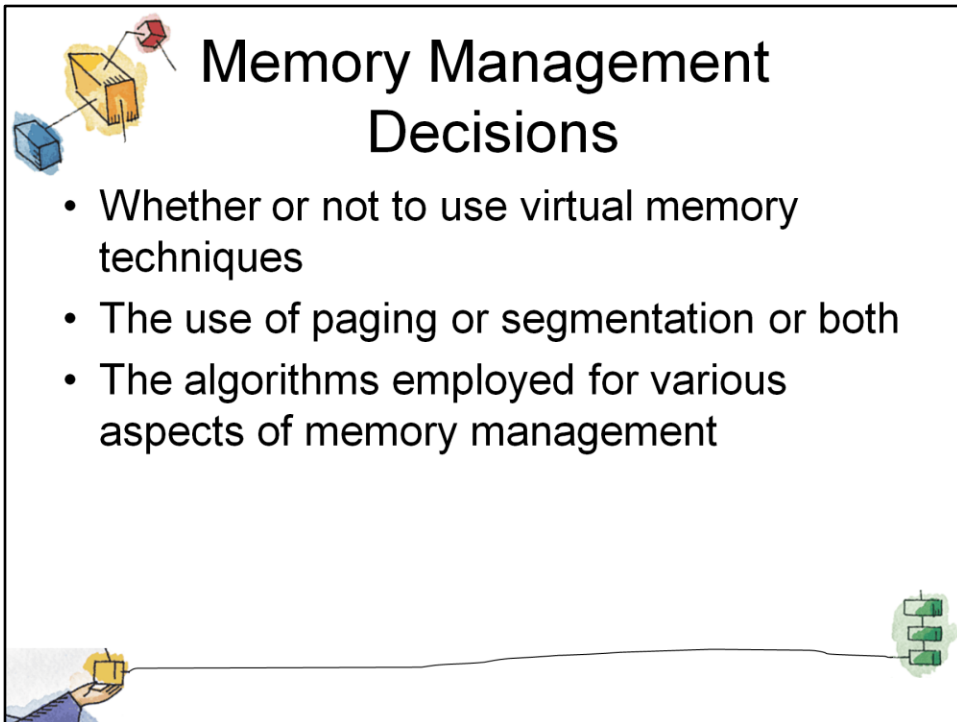
Figure 8.14 illustrates the types of protection relationships that can be enforced in such a system.



Roadmap

- Hardware and Control Structures
- Operating System Software
 - UNIX and Solaris Memory Management
 - Linux Memory Management
 - Windows Memory Management



The slide features a title 'Memory Management Decisions' in a large, bold, black font. To the left of the title is a cluster of colorful icons: a blue cube, a yellow cube, a red cube, and a yellow cube with a red cube on top. Below the title is a bulleted list with three items. At the bottom left, a hand is shown holding a yellow cube. At the bottom right, there is a green icon consisting of three stacked rectangular blocks. A thin black line runs horizontally across the bottom of the slide, connecting the hand icon to the green icon.


Memory Management Decisions

- Whether or not to use virtual memory techniques
- The use of paging or segmentation or both
- The algorithms employed for various aspects of memory management

The design of the memory management portion of an operating system depends on three fundamental areas of choice:

- Whether or not to use virtual memory techniques
- The use of paging or segmentation or both
- The algorithms employed for various aspects of memory management

The first two are determined largely by hardware support. But, with few exceptions (DOS for old PC's) all major OS's support Virtual Memory.





Key Design Elements

Table 8.4 Operating System Policies for Virtual Memory

<p>Fetch Policy</p> <ul style="list-style-type: none"> Demand Prepaging <p>Placement Policy</p> <p>Replacement Policy</p> <p>Basic Algorithms</p> <ul style="list-style-type: none"> Optimal Least recently used (LRU) First-in-first-out (FIFO) Clock Page buffering 	<p>Resident Set Management</p> <p>Resident set size</p> <ul style="list-style-type: none"> Fixed Variable <p>Replacement Scope</p> <ul style="list-style-type: none"> Global Local <p>Cleaning Policy</p> <ul style="list-style-type: none"> Demand Precleaning <p>Load Control</p> <ul style="list-style-type: none"> Degree of multiprogramming
---	---

- **Key aim: Minimise page faults**
 - No definitive best policy

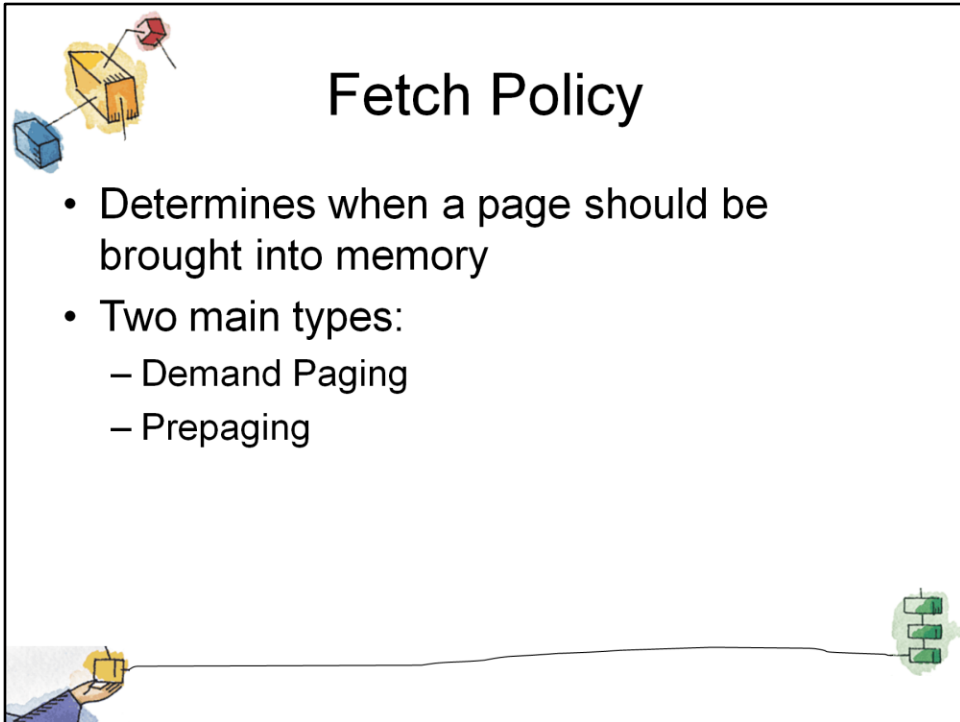
We would like to minimize the rate at which page faults occur, because page faults cause considerable software overhead.

- At a minimum, the overhead includes deciding which resident page or pages to replace,
- and the I/O of exchanging pages.
- Also, the operating system must schedule another process to run during the page I/O, causing a process switch.

In all of the areas referred to in Table 8.4, there is no definitive policy that works best.

The performance of any particular set of policies depends on

- main memory size,
- the relative speed of main and secondary memory,
- the size and number of processes competing for resources,
- and the execution behaviour of individual programs.

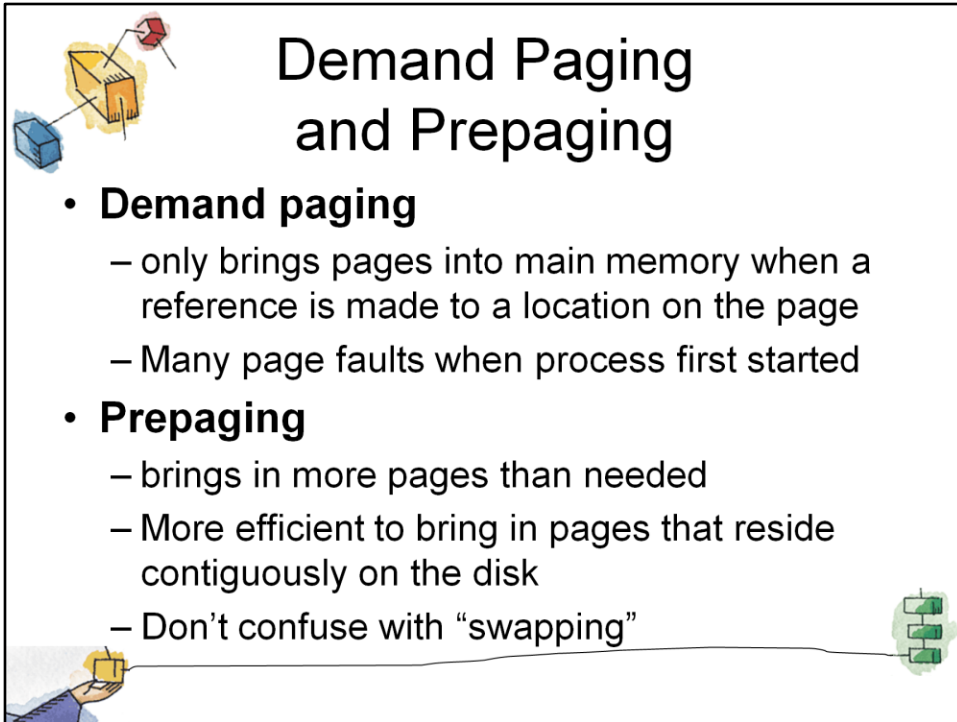
A diagram titled "Fetch Policy" enclosed in a black rectangular border. In the top-left corner, there is a small illustration of a blue box, a yellow box, and a red box with arrows pointing towards a central yellow box. In the bottom-left corner, a hand is shown holding a yellow box. In the bottom-right corner, there is a small illustration of a green box with a white outline. The title "Fetch Policy" is centered at the top in a large, black, sans-serif font. Below the title, there is a bulleted list of three items.

Fetch Policy

- Determines when a page should be brought into memory
- Two main types:
 - Demand Paging
 - Prepaging

The fetch policy determines when a page should be brought into main memory.

The two common alternatives are demand paging and prepaging



Demand Paging and Prepaging

- **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
- **Prepaging**
 - brings in more pages than needed
 - More efficient to bring in pages that reside contiguously on the disk
 - Don't confuse with “swapping”

With demand paging,

- A page is brought into main memory only when a reference is made to a location on that page.
- When a process is first started, there will be a flurry of page faults.
- As more and more pages are brought in, the principle of locality suggests that most future references will be to pages that have recently been brought in.
- Thus, after a time, matters should settle down and the number of page faults should drop to a very low level.

With prepaging,

- Pages other than the one demanded by a page fault are brought in.
- If the pages of a process are stored contiguously in secondary memory, then it is more efficient to bring in a number of contiguous pages at one time rather than bringing them in one at a time over an extended period.
- Of course, this policy is ineffective if most of the extra pages that are brought in are not referenced.

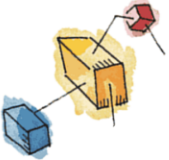
The prepaging policy could be employed either when a process first starts up, in which case the programmer would somehow have to designate desired pages, or every time a page fault occurs.

The latter would seem preferable because it is invisible to the programmer.

However, the utility of prepaging has not been established.



Prepaging should not be confused with swapping.

- When a process is swapped out of memory and put in a suspended state, all of its resident pages are moved out.
- When the process is resumed, all of the pages that were previously in main memory are returned to main memory.



Placement Policy

- Determines where in real memory a process piece is to reside
- Important in a segmentation system
- Paging or combined paging with segmentation hardware performs address translation

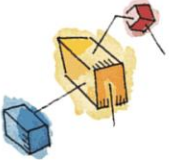


The **placement policy** determines where in real memory a process piece is to reside.

In a pure segmentation system, the placement policy is an important design issue; policies such as best-fit, first-fit, and etc are possible alternatives.



However, for a system that uses either pure paging or paging combined with segmentation,

placement is usually irrelevant because the address translation hardware and the main memory access hardware can perform their functions for any page-frame combination with equal efficiency.



Replacement Policy

- When all of the frames in main memory are occupied and it is necessary to bring in a new page, the replacement policy determines which page currently in memory is to be replaced.

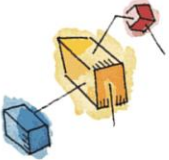


This topic is sometimes difficult to explain because several interrelated concepts are involved:

- How many page frames are to be allocated to each active process
- Whether the set of pages to be considered for replacement should be limited to those of the process that caused the page fault or encompass all the page frames in main memory
- Among the set of pages considered, which particular page should be selected for replacement


When all of the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault,

the replacement policy determines which page currently in memory is to be replaced.

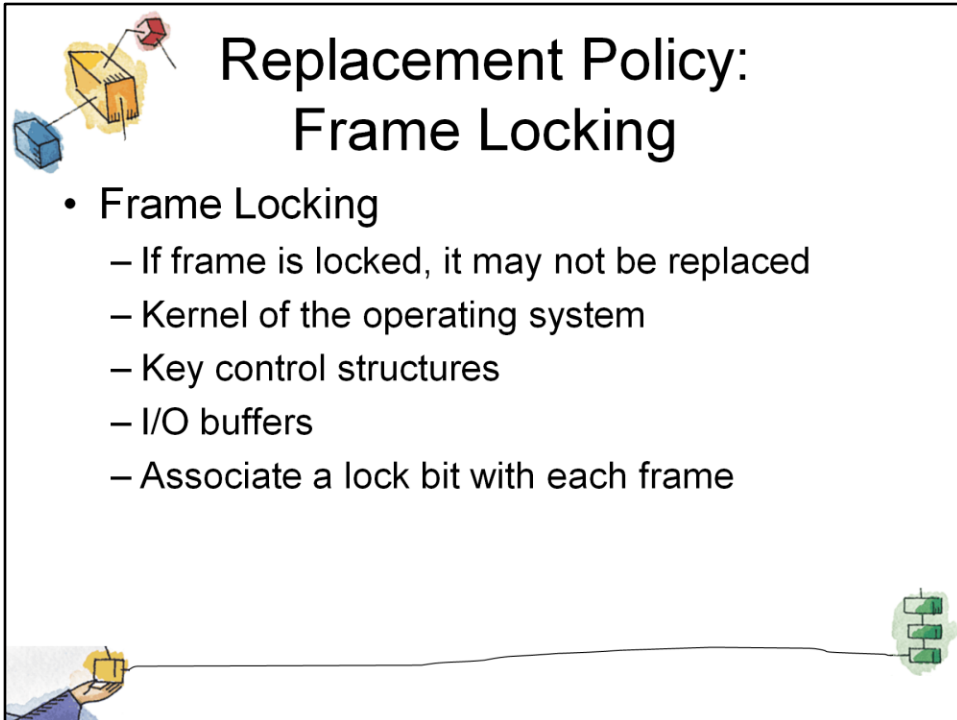


But...

- Which page is replaced?
- Page removed should be the page least likely to be referenced in the near future
 - How is that determined?
 - Principal of locality again
- Most policies predict the future behavior on the basis of past behavior



One tradeoff that must be considered is that the more elaborate and sophisticated the replacement policy, the greater the hardware and software overhead to implement it



Replacement Policy: Frame Locking

- Frame Locking
 - If frame is locked, it may not be replaced
 - Kernel of the operating system
 - Key control structures
 - I/O buffers
 - Associate a lock bit with each frame

Camera jumps to animation of Page Replacement Algorithm
<http://gaia.ecs.csus.edu/~zhangd/oscal/PagingApplet.html>

One restriction on replacement policy needs to be mentioned before looking at various algorithms:

- Some of the frames in main memory may be locked.
- When a frame is locked, the page currently stored in that frame may not be replaced.
- Much of the kernel of the operating system is held on locked frames, as well as key control structures.
- In addition, I/O buffers and other time-critical areas may be locked into main memory frames.

Locking is achieved by associating a lock bit with each frame.

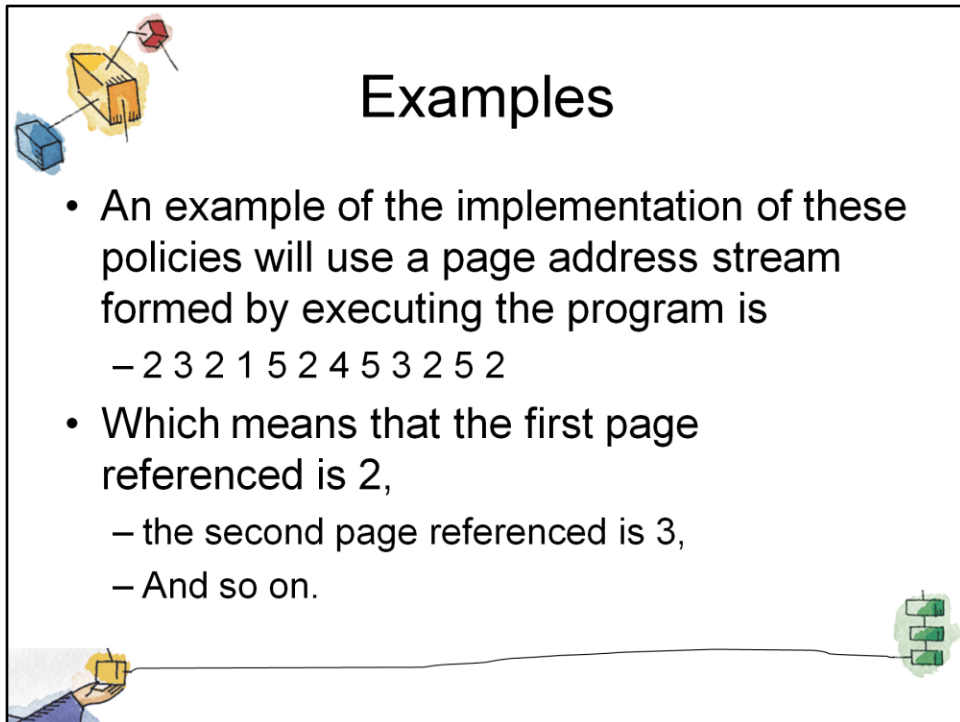
This bit may be kept in a frame table as well as being included in the current page table.



Basic Replacement Algorithms

- There are certain basic algorithms that are used for the selection of a page to replace, they include
 - Optimal
 - Least recently used (LRU)
 - First-in-first-out (FIFO)
 - Clock
- Examples

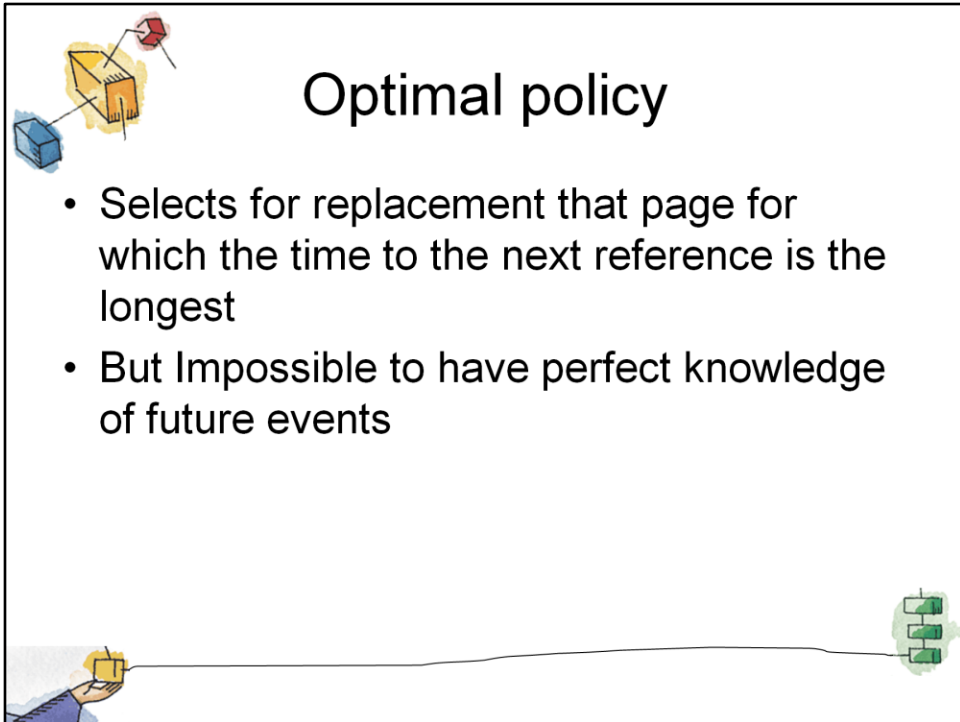




Examples

- An example of the implementation of these policies will use a page address stream formed by executing the program is
 - 2 3 2 1 5 2 4 5 3 2 5 2
- Which means that the first page referenced is 2,
 - the second page referenced is 3,
 - And so on.

Figure 8.15 gives an example of the optimal policy. The example assumes a fixed frame allocation (fixed resident set size) for this process of three frames. The execution of the process requires reference to five distinct pages. The page address stream formed by executing the program is 232152453252 which means that the first page referenced is 2, the second page referenced is 3, and so on. The optimal policy produces three page faults after the frame allocation has been filled.



The diagram is enclosed in a black rectangular border. In the top-left corner, there is a cluster of icons: a blue cube, a yellow cube, and a red cube, with lines connecting them. In the bottom-left corner, a hand is shown holding a yellow cube. In the bottom-right corner, there is a green icon representing a stack of three pages. A thin black line extends from the hand towards the stack of pages.

Optimal policy

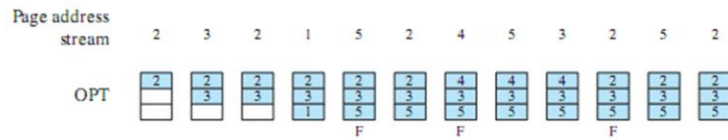
- Selects for replacement that page for which the time to the next reference is the longest
- But Impossible to have perfect knowledge of future events

The optimal policy selects for replacement that page for which the time to the next reference is the longest.

- This policy results in the fewest number of page faults.
- **BUT** Clearly, this policy is impossible to implement, because it would require the operating system to have perfect knowledge of future events.
- However, it does serve as a standard against which to judge real world algorithms.



Optimal Policy Example

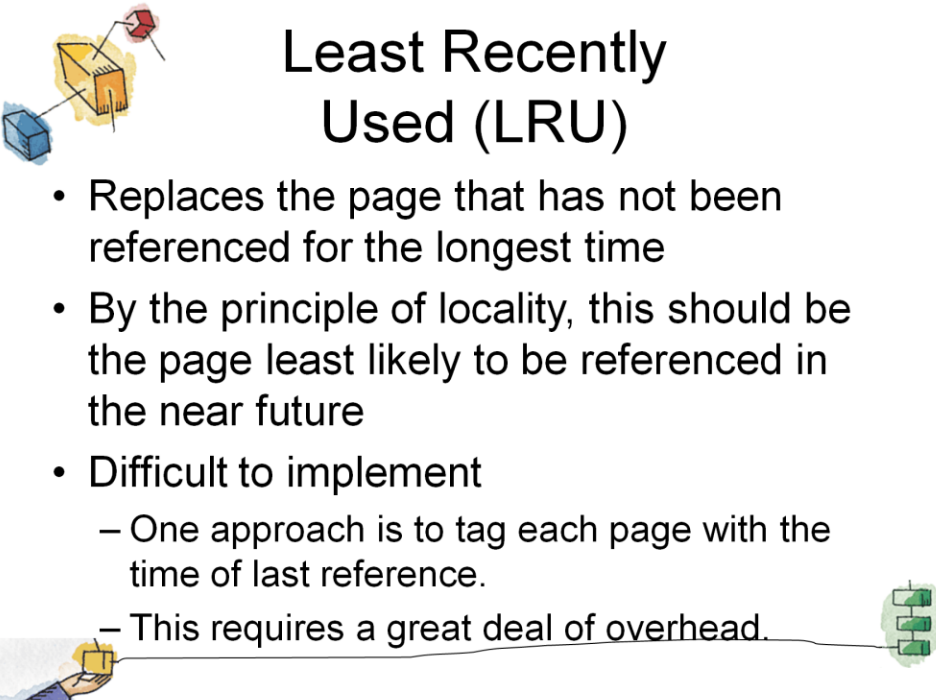


F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms

- The optimal policy produces three page faults after the frame allocation has been filled.





Least Recently Used (LRU)

- 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
- Difficult to implement
 - One approach is to tag each page with the time of last reference.
 - This requires a great deal of overhead.

The least recently used (LRU) policy replaces the page in memory 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.

And, in fact, the LRU policy does nearly as well as the optimal policy.

The problem with this approach is the difficulty in implementation.

One approach would be to tag each page with the time of its last reference;

- this would have to be done at each memory reference, both instruction and data.
- Even if the hardware would support such a scheme, the overhead would be tremendous.
- Alternatively, one could maintain a stack of page references, again an expensive prospect.



LRU Example

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

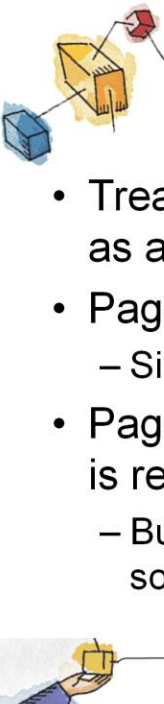
F

F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms


- The LRU policy does nearly as well as the optimal policy.
 - In this example, there are four page faults





First-in, first-out (FIFO)

- Treats page frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
 - Simplest replacement policy to implement
- Page that has been in memory the longest is replaced
 - But, these pages may be needed again very soon if it hasn't truly fallen out of use



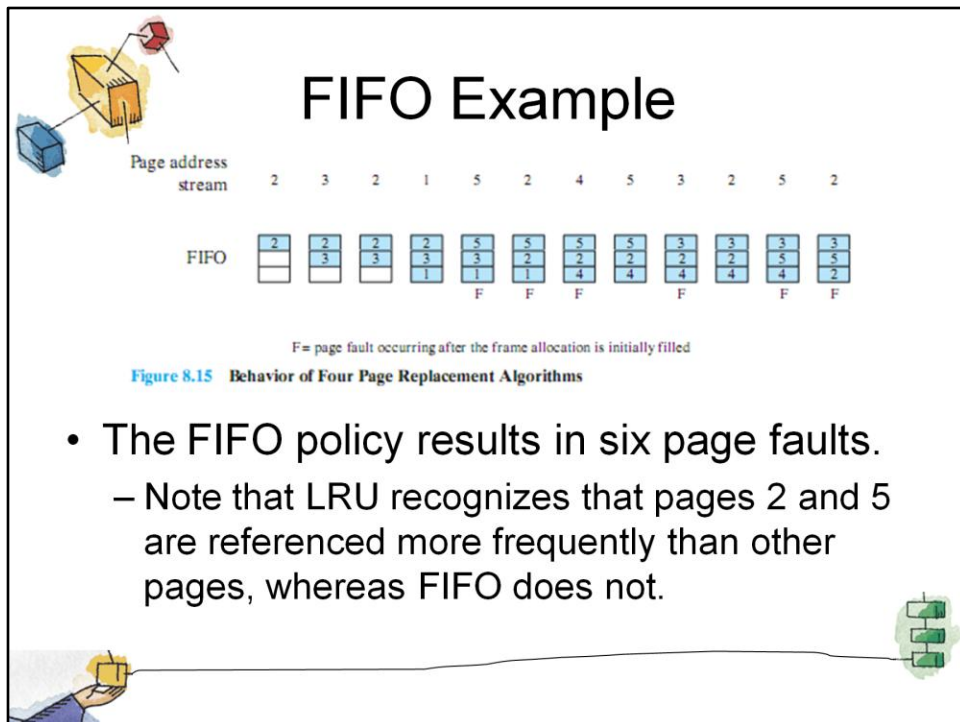
The first-in-first-out (FIFO) policy treats the page frames allocated to a process as a circular buffer, and pages are removed in round-robin style.

All that is required is a pointer that circles through the page frames of the process.

This is one of the simplest page replacement policies to implement.

The logic behind this choice is that one is replacing the page that has been in memory the longest:

- A page fetched into memory a long time ago may have now fallen out of use.
- This reasoning will often be wrong, because there will often be regions of program or data that are heavily used throughout the life of a program.
- Those pages will be repeatedly paged in and out by the FIFO algorithm.



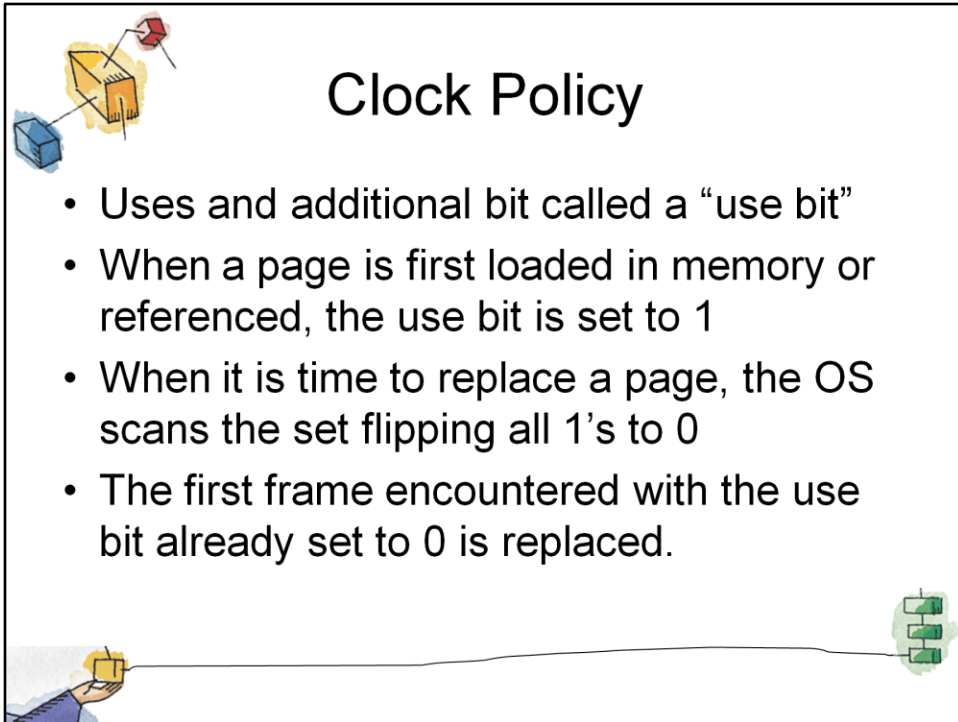
Continuing our example in Figure 8.15, the FIFO policy results in six page faults.

Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

Although the LRU policy does nearly as well as an optimal policy, it is difficult to implement and imposes significant overhead.

- On the other hand, the FIFO policy is very simple to implement but performs relatively poorly.

Over the years, operating system designers have tried a number of other algorithms to approximate the performance of LRU while imposing little overhead.



Clock Policy

- Uses an additional bit called a “use bit”
- When a page is first loaded in memory or referenced, the use bit is set to 1
- When it is time to replace a page, the OS scans the set flipping all 1’s to 0
- The first frame encountered with the use bit already set to 0 is replaced.

Camera jumps to animation of clock policy at <http://gaia.ecs.csus.edu/~zhangd/oscal/ClockFiles/Clock.htm>

The simplest form of clock policy requires the association of an additional bit with each frame, referred to as the use bit.

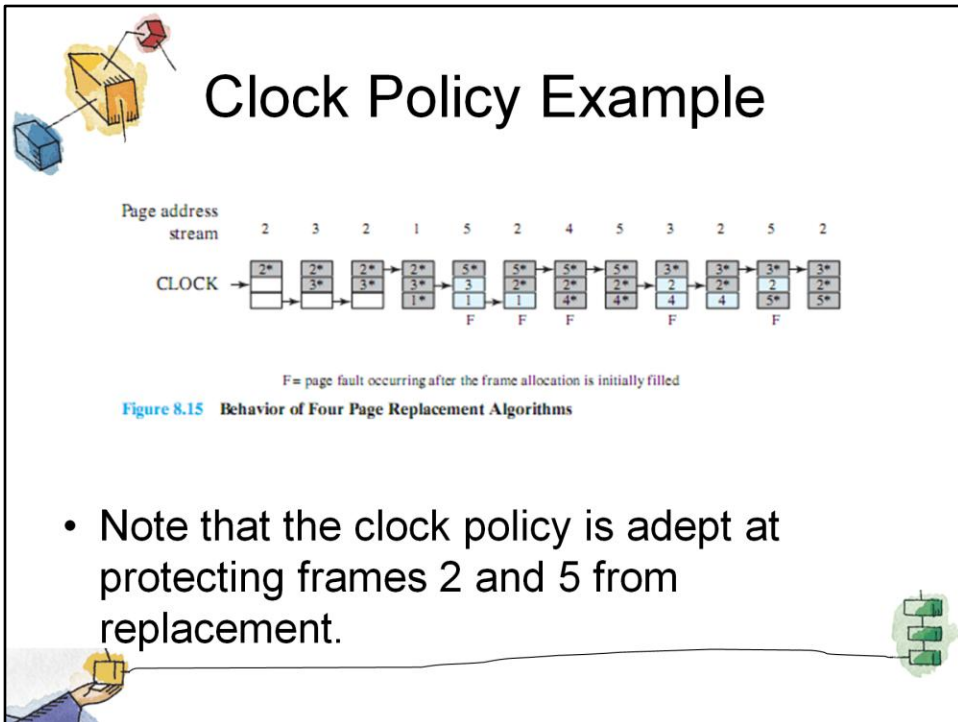
When a page is first loaded into a frame in memory, the use bit for that frame is set to 1.

- Whenever the page is subsequently referenced (after the reference that generated the page fault), its use bit is set to 1.

The set of frames that are candidates for replacement is considered to be a circular buffer, with which a pointer is associated.

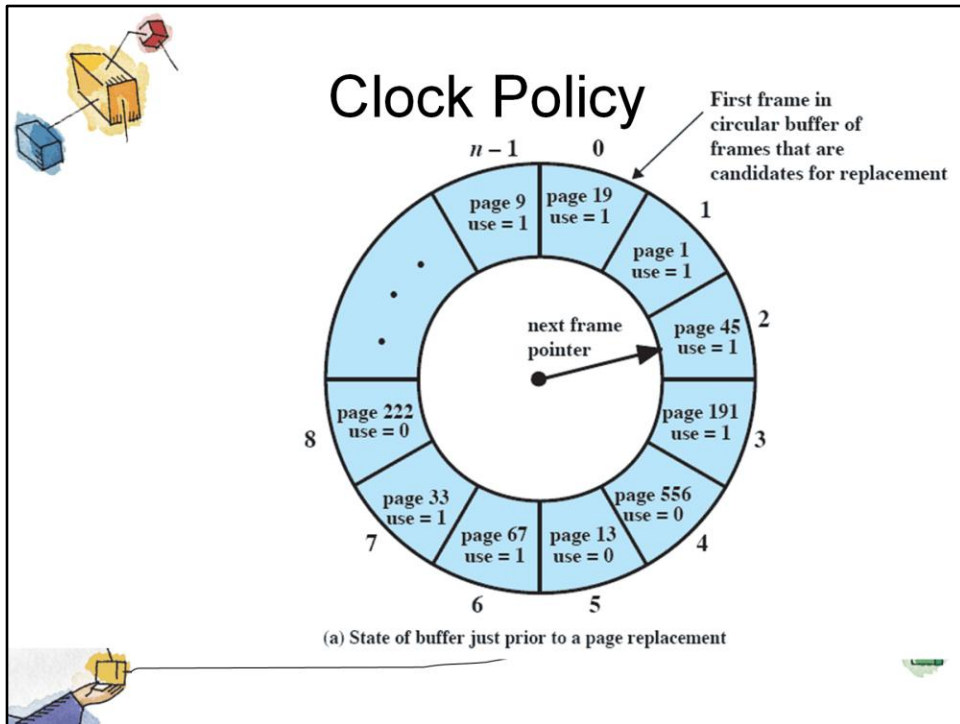
- When a page is replaced, the pointer is set to indicate the next frame in the buffer after the one just updated.
- When it comes time to replace a page, the operating system scans the buffer to find a frame with a use bit set to zero.
- Each time it encounters a frame with a use bit of 1, it resets that bit to zero and continues on.
- If any of the frames in the buffer have a use bit of zero at the beginning of this process, the first such frame encountered is chosen for replacement.
- If all of the frames have a use bit of 1, then the pointer will make one complete cycle through the buffer, setting all the use bits to zero, and stop at its original position, replacing the page in that frame.

This policy is similar to FIFO, except that, in the clock policy, any frame with a use bit of 1 is passed over by the algorithm.



The presence of an asterisk indicates that the corresponding use bit is equal to 1, and the arrow indicates the current position of the pointer.

Note that the clock policy is adept at protecting frames 2 and 5 from replacement.



Continued on next slide

This figure (8.16) provides an example of the simple clock policy mechanism.

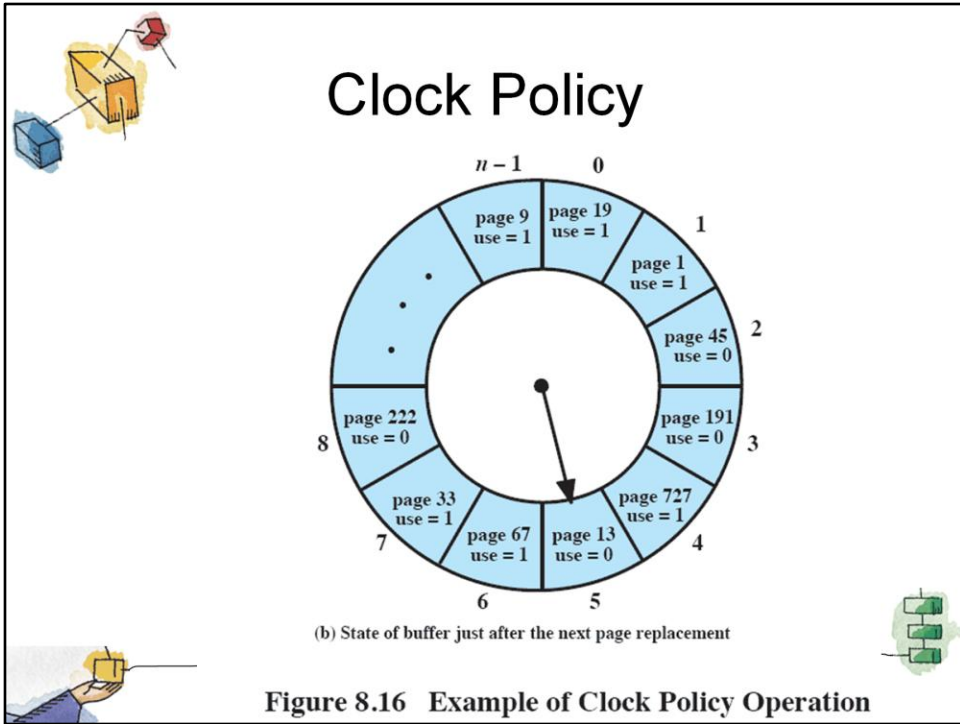
A circular buffer of n main memory frames is available for page replacement.

Just prior to the replacement of a page from the buffer with incoming page 727,

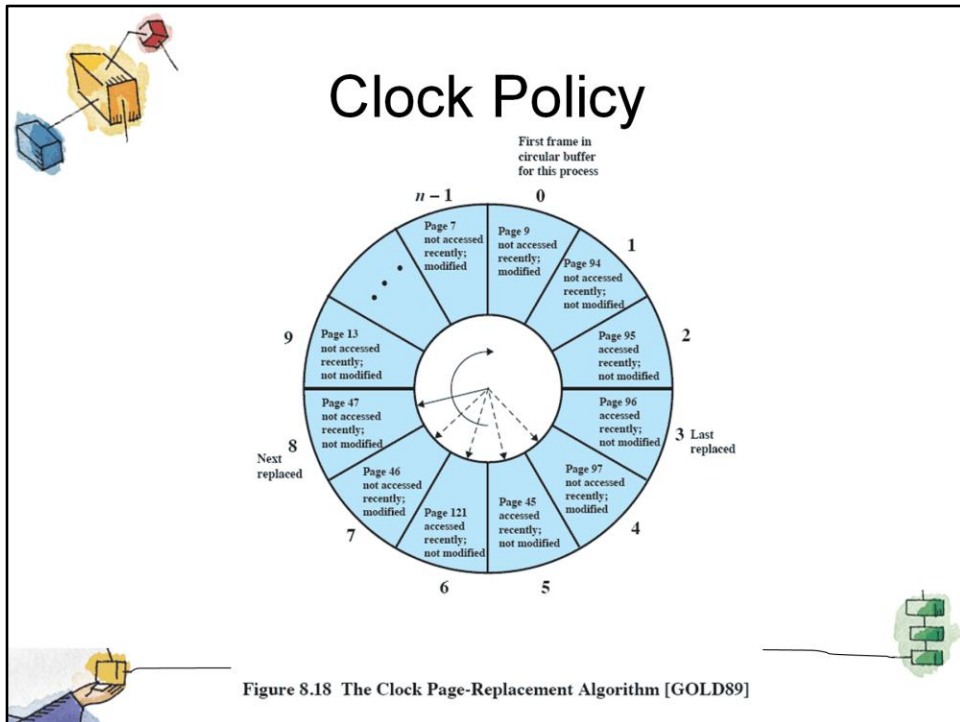
- the next frame pointer points at frame 2, which contains page 45.

The clock policy is now executed.

- Because the use bit for page 45 in frame 2 is equal to 1, this page is not replaced.
- Instead, the use bit is set to zero and the pointer advances.
- Similarly, page 191 in frame 3 is not replaced; its use bit is set to zero and the pointer advances.
- In the next frame, frame 4, the use bit is set to 0. Therefore, page 556 is replaced with page 727.
- The use bit is set to 1 for this frame and the pointer advances to frame 5, completing the page replacement procedure.



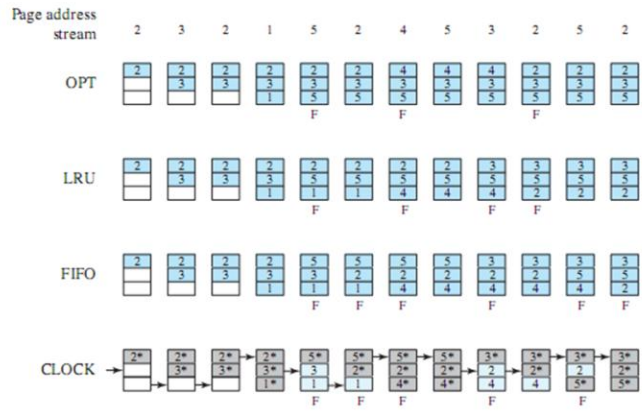
Explanation on previous slide



Explanation two slides previous



Combined Examples



F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms



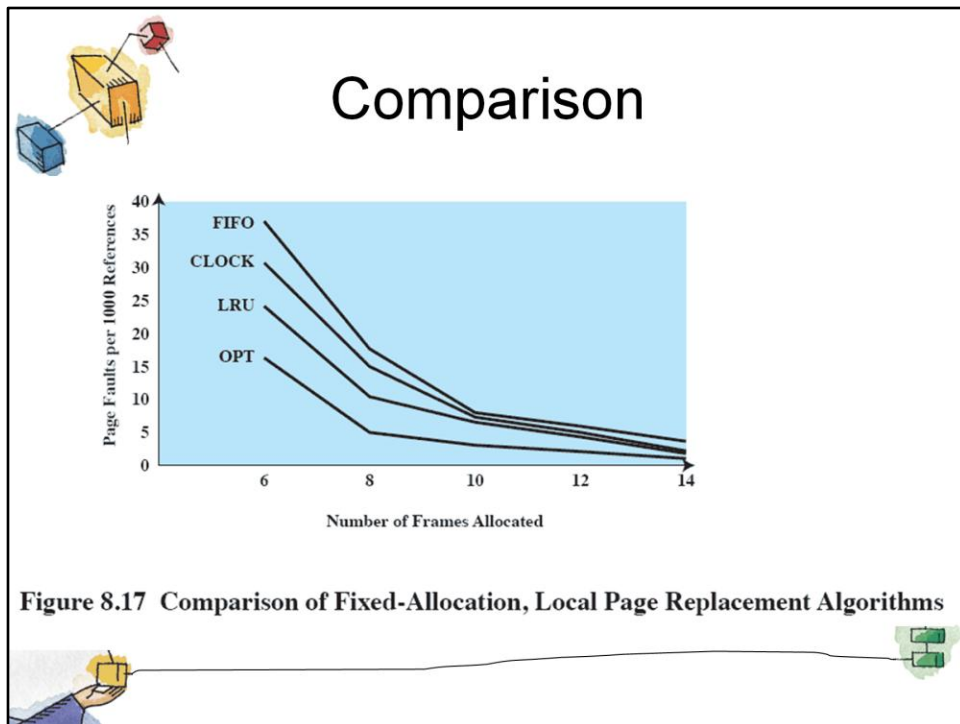


Figure 8.17 shows the results of an experiment reported in, which compares the four algorithms that we have been discussing;

It is assumed that the number of page frames assigned to a process is fixed.

The results are based on the execution of 0.25×10^6 references in a FORTRAN program, using a page size of 256 words.

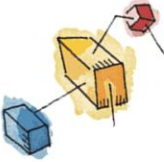
Baer ran the experiment with frame allocations of 6, 8, 10, 12, and 14 frames.

The differences among the four policies are most striking at small allocations, with FIFO being over a factor of 2 worse than optimal.

All four curves have the same shape as the idealized behavior shown in Figure 8.11b.



In order to run efficiently, we would like to be to the right of the knee of the curve (with a small page fault rate) while at the same time keeping a small frame allocation (to the left of the knee of the curve).

These two constraints indicate that a desirable mode of operation would be at the knee of the curve.



Page Buffering

- LRU and Clock policies both involve complexity and overhead
 - Also, replacing a modified page is more costly than unmodified as needs written to secondary memory
- Solution: Replaced page is added to one of two lists
 - Free page list if page has not been modified
 - Modified page list

Although LRU and the clock policies are superior to FIFO, they both involve complexity and overhead not suffered with FIFO.

There is also the related issue that the cost of replacing a page that has been modified is greater than for one that has not, because the former must be written back out to secondary memory.

An interesting strategy that can improve paging performance and allow the use of a simpler page replacement policy is page buffering.

E.G. VAX VMS approach.

The page replacement algorithm is simple FIFO.

To improve performance, a replaced page is not lost but rather is assigned to one of two lists:

- the **free page** list if the page has not been modified, or
- the **modified page** list if it has.

Note that the page is not physically moved about in main memory;

instead, the entry in the page table for this page is removed and placed in either the free or modified page list.

The free page list is a list of page frames available for reading in pages.

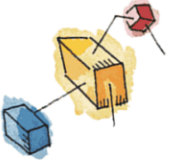
- When a page is to be read in, the page frame at the head of the list is used, destroying the page that was there.
- When an unmodified page is to be replaced, it remains in memory and its page frame is added to the tail of the free page list.
- Similarly, when a modified page is to be written out and replaced, its page frame is added to the tail of the modified page list.

The important aspect of these manoeuvres is that the page to be replaced remains in memory.

- If the process references that page, it is returned to the resident set of that process at little cost.
- In effect, the free and modified page lists act as a cache of pages.



The modified page list serves another useful function:

Modified pages are written out in clusters rather than one at a time - significantly reducing the number of I/O operations and therefore the amount of disk access time.



Replacement Policy and Cache Size

- Main memory size is getting larger and the locality of applications is decreasing.
 - So, cache sizes have been increasing
- With large caches, replacement of pages can have a performance impact
 - improve performance by supplementing the page replacement policy with a policy for page placement in the page buffer



Main memory size is getting larger and the locality of applications is decreasing.
In compensation, cache sizes have been increasing.

Large cache sizes, even multimegabyte ones, are now feasible design alternatives.

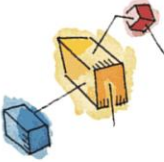
With a large cache, the replacement of virtual memory pages can have a performance impact.

- If the page frame selected for replacement is in the cache, then that cache block is lost as well as the page that it holds.

In systems that use some form of page buffering, it is possible to improve cache performance by supplementing the page replacement policy with a policy for page placement in the page buffer.



Most operating systems place pages by selecting an arbitrary page frame from the page buffer; typically a first-in-first-out discipline is used.

- Careful page placement strategy can result in 10 to 20% fewer cache misses than naive placement.



Resident Set Management

- The OS must decide how many pages to bring into main memory
 - The smaller the amount of memory allocated to each process, the more processes that can reside in memory.
 - Small number of pages loaded increases page faults.
 - Beyond a certain size, further allocations of pages will not affect 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 main memory to prepare it for execution.

Thus, the operating system must decide how many pages to bring in

IE , how much main 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 main 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 main memory, then, despite the principle of locality, the rate of page faults will be rather high
- Beyond a certain size, additional allocation of main memory to a particular process will have no noticeable effect on the page fault rate for that process because of the principle of locality.



Resident Set Size

- Fixed-allocation
 - Gives a process a fixed number of pages within which to execute
 - When a page fault occurs, one of the pages of that process must be replaced
- Variable-allocation
 - Number of pages allocated to a process varies over the lifetime of the process

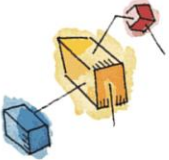


A **fixed-allocation policy** gives a process a fixed number of frames in main memory within which to execute.

- 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.
- 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 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



Replacement Scope

- The scope of a replacement strategy can be categorized as *global* or *local*.
 - Both types 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
 - A global replacement policy considers all unlocked pages in main memory

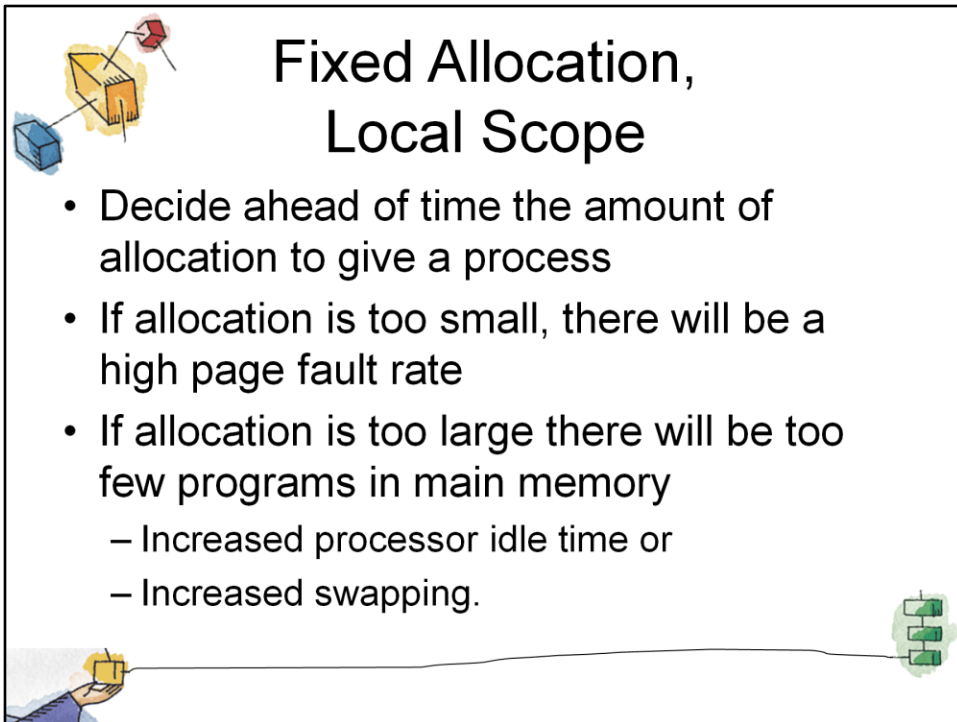


The scope of a replacement strategy can be categorized as global or local.

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 main memory as candidates for replacement, regardless of which process owns a particular page.



Fixed Allocation, Local Scope

- Decide ahead of time the amount of allocation to give a process
- If allocation is too small, there will be a high page fault rate
- If allocation is too large there will be too few programs in main memory
 - Increased processor idle time or
 - Increased swapping.

A process that is running in main memory with a fixed number of frames.


When a page fault occurs, the operating system must choose which page from among the currently resident pages for this process is to be replaced.

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 based on the type of application and the amount requested by the program.



Two drawbacks:

- 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 main memory and there will either be considerable processor idle time or considerable time spent in swapping.



Variable Allocation, Global Scope

- Easiest to implement
 - Adopted by many operating systems
- Operating system keeps list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no free frame, replaces one from another process
 - Therein lies the difficulty ... which to replace.



This combination is 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 main memory, each with a certain number of frames allocated to it.

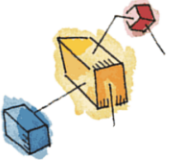
Typically, the operating system also maintains a list of free frames.

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.
- The process that suffers the reduction in resident set size may not be optimum.



Variable Allocation, Local Scope

- When new process added, allocate number of page frames based on application type, program request, or other criteria
- When page fault occurs, select page from among the resident set of the process that suffers the fault
- Reevaluate allocation from time to time



1. When a new process is loaded into main 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, re-evaluate 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.

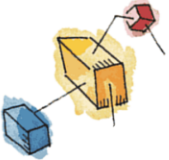


Resident Set Management Summary

Table 8.5 Resident Set Management



	Local Replacement	Global Replacement
Fixed Allocation	<ul style="list-style-type: none">• Number of frames allocated to process is fixed.• Page to be replaced is chosen from among the frames allocated to that process.	<ul style="list-style-type: none">• Not possible.
Variable Allocation	<ul style="list-style-type: none">• The number of frames allocated to a process may be changed from time to time, to maintain the working set of the process.• Page to be replaced is chosen from among the frames allocated to that process.	<ul style="list-style-type: none">• Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.





Cleaning Policy

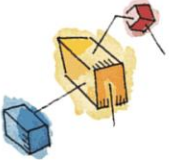
- A cleaning policy is concerned with determining when a modified page should be written out to secondary memory.
- Demand cleaning
 - A page is written out only when it has been selected for replacement
- Precleaning
 - Pages are written out in batches



A cleaning policy is the opposite of a fetch policy; it is concerned with determining when a modified page should be written out to secondary memory.



With **demand cleaning**, a page is written out to secondary memory only when it has been selected for replacement.

A **precleaning** policy writes modified pages before their page frames are needed so that pages can be written out in batches.



Cleaning Policy

- Best approach uses page buffering
- Replaced pages are placed in two lists
 - Modified and unmodified
- Pages in the modified list are periodically written out in batches
- Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page



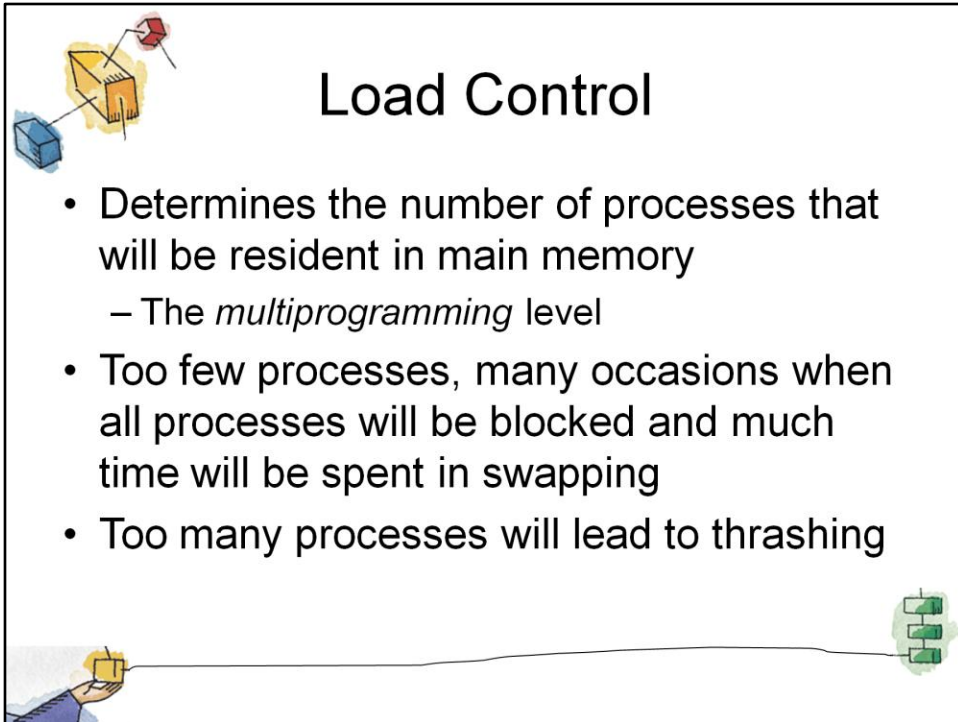
Clean only pages that are replaceable, but decouple the cleaning and replacement operations.

With page buffering, replaced pages can be placed on two lists:

- modified and unmodified.

The pages on the modified list can periodically be written out in batches and moved to the unmodified list.

A page on the unmodified list is either reclaimed if it is referenced, or lost when its frame is assigned to another page.



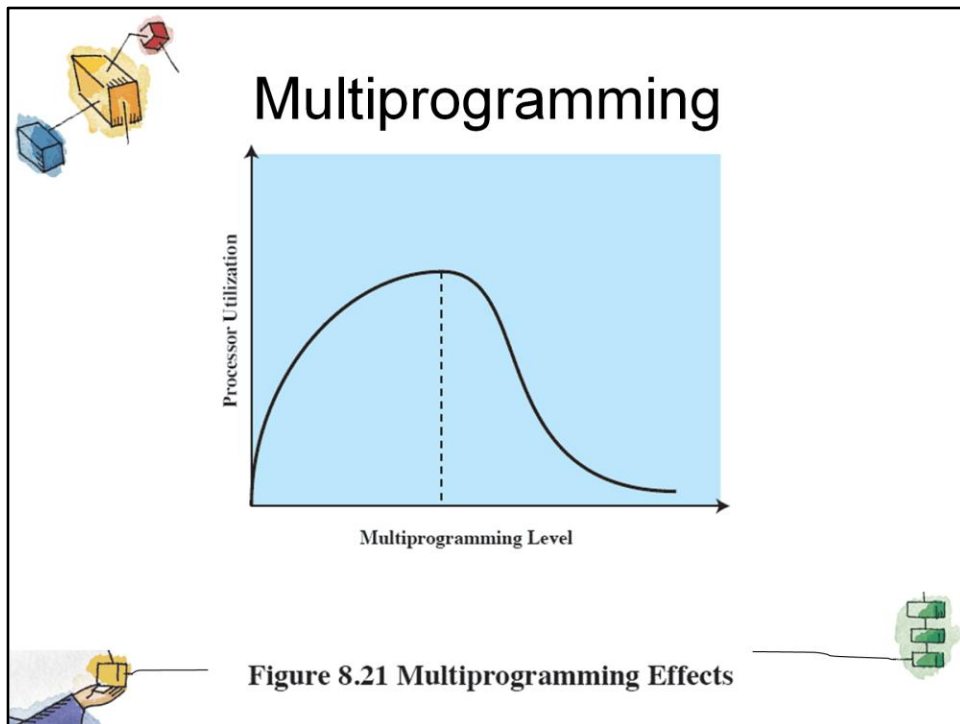
Load Control

- Determines the number of processes that will be resident in main memory
 - The *multiprogramming* level
- Too few processes, many occasions when all processes will be blocked and much time will be spent in swapping
- Too many processes will lead to thrashing

Load control is concerned with determining the number of processes that will be resident in main memory - the multiprogramming level.

The load control policy is critical in effective memory management.

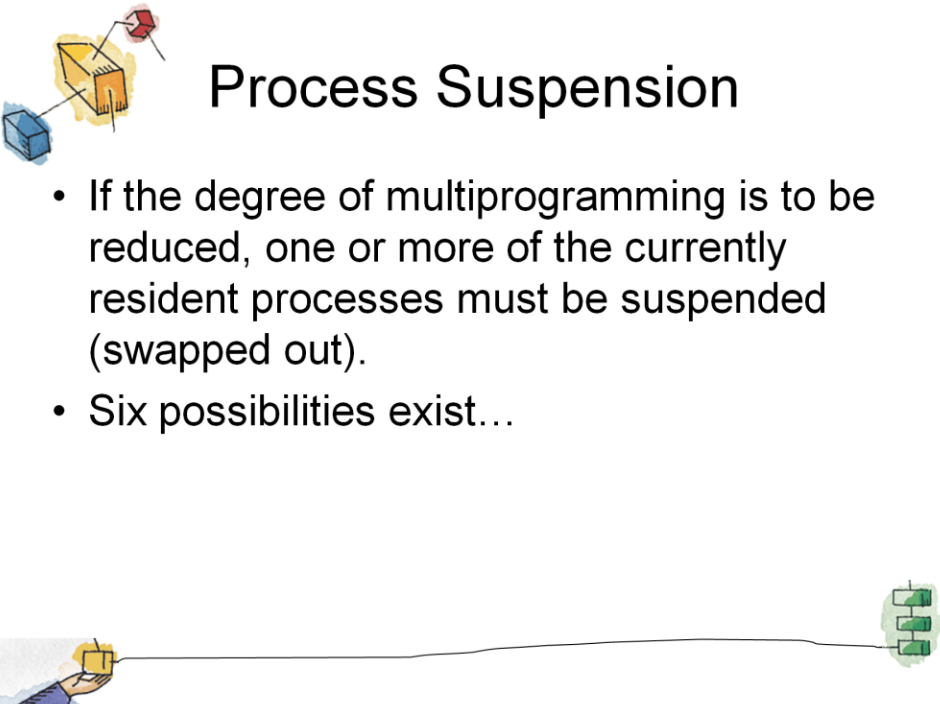
- If too few processes are resident at any one time, then there will be many occasions when all processes are blocked, and much time will be spent in swapping.
- On the other hand, if too many processes are resident, then, on average, the size of the resident set of each process will be inadequate and frequent faulting will occur resulting in thrashing.



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.



Process Suspension

- If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be suspended (swapped out).
- Six possibilities exist...

If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be suspended (swapped out). [CARR84] lists six possibilities:

Suspension policies

- Lowest priority process
- Faulting process
 - This process does 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

Lowest-priority process:

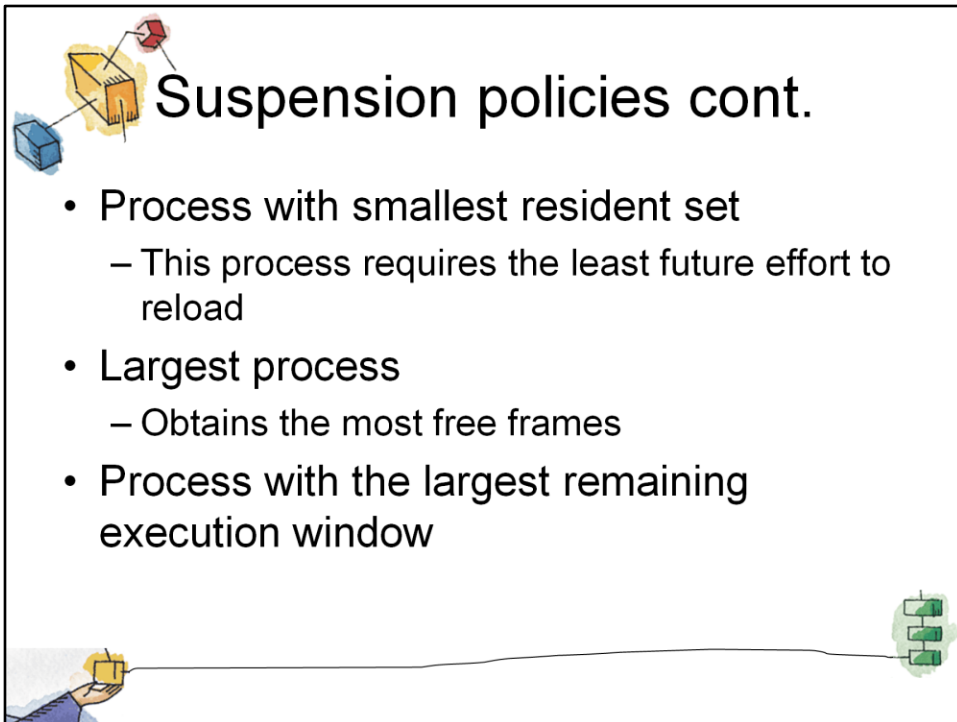
- This implements a scheduling policy decision and is unrelated to performance issues.

Faulting process:

- There is a greater probability that the faulting task does not have its working set resident, and performance would suffer least by suspending it.
- Immediate payoff because it blocks a process that is about to be blocked anyway and it eliminates the overhead of a page replacement and I/O operation.

Last process activated:

- This is the process least likely to have its working set resident.



Suspension policies cont.

- Process with smallest resident set
 - This process requires the least future effort to reload
- Largest process
 - Obtains the most free frames
- Process with the largest remaining execution window

Process with the smallest resident set:

- This will require the least future effort to reload.
- However, it penalizes programs with strong locality.

Largest process:

- This obtains the most free frames in an overcommitted memory, making additional deactivations unlikely soon.

Process with the largest remaining execution window:

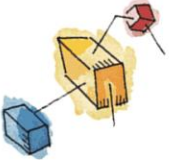
- In most process scheduling schemes, a process may only run for a certain quantum of time before being interrupted and placed at the end of the Ready queue.
- This approximates a shortest-processing-time-first scheduling discipline.



Roadmap



- Hardware and Control Structures
- Operating System Software
- UNIX and Solaris Memory Management
- Linux Memory Management
- Windows Memory Management





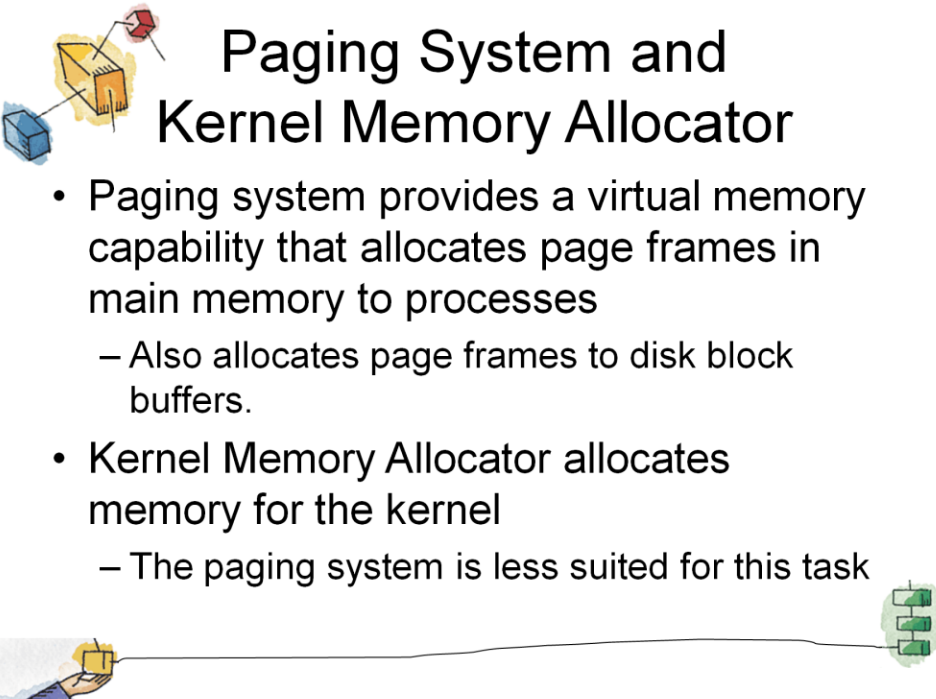
Unix

- Intended to be machine independent so implementations vary
 - Early Unix: variable partitioning with no virtual memory to paged
 - Recent Unix (SVR4 & Solaris) using paged virtual memory
- SVR4 uses two separate schemes:
 - *Paging system and a kernel memory allocator.*



Because UNIX is intended to be machine independent, its memory management scheme will vary from one system to the next.

- Earlier versions of UNIX simply used variable partitioning with no virtual memory scheme.
- Current implementations of UNIX and Solaris make use of paged virtual memory.

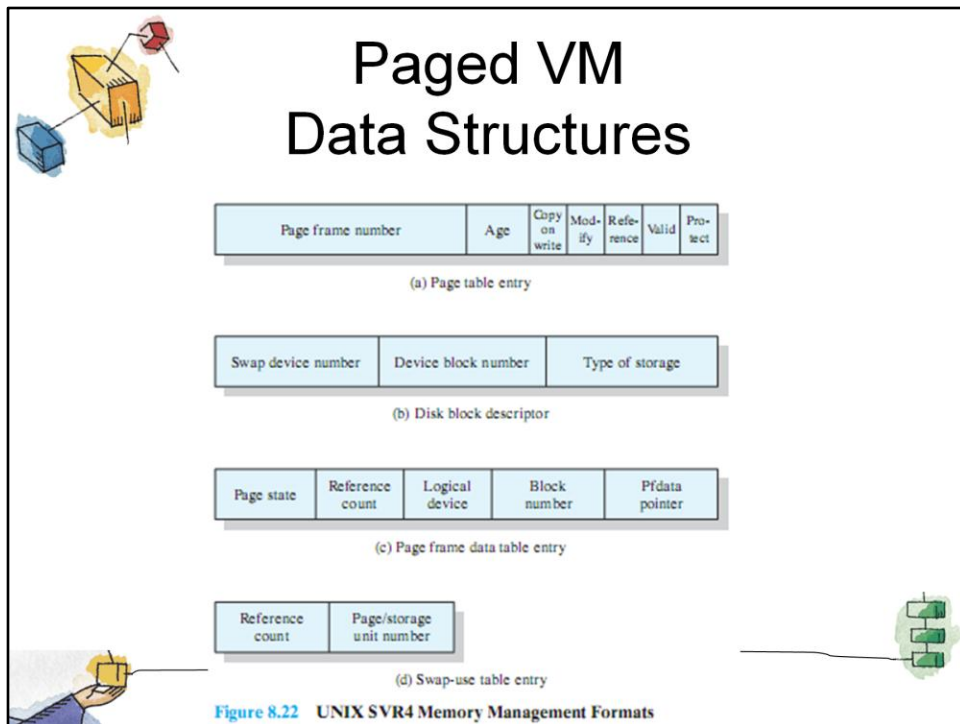


Paging System and Kernel Memory Allocator

- Paging system provides a virtual memory capability that allocates page frames in main memory to processes
 - Also allocates page frames to disk block buffers.
- Kernel Memory Allocator allocates memory for the kernel
 - The paging system is less suited for this task

provides a virtual memory capability that allocates page frames in main memory to processes and also allocates page frames to disk block buffers.

Although this is an effective memory-management scheme for user processes and disk I/O, a paged virtual memory scheme is less suited to managing the memory allocation for the kernel



Page Table:

- Typically, there will be one page table per process, with one entry for each page in virtual memory for that process.

Disk block descriptor:

- Associated with each page of a process is an entry in this table that describes the disk copy of the virtual page.

Page frame data table:

- Describes each frame of real memory and is indexed by frame number. This table is used by the replacement algorithm.

Swap-use table:

- There is one swap-use table for each swap device, with one entry for each page on the device.



Page Table Entry Fields

Page Table Entry

Page frame number

Refers to frame in real memory.

Age

Indicates how long the page has been in memory without being referenced. The length and contents of this field are processor dependent.

Copy on write

Set when more than one process shares a page. If one of the processes writes into the page, a separate copy of the page must first be made for all other processes that share the page. This feature allows the copy operation to be deferred until necessary and avoided in cases where it turns out not to be necessary.

Modify

Indicates page has been modified.

Reference

Indicates page has been referenced. This bit is set to zero when the page is first loaded and may be periodically reset by the page replacement algorithm.

Valid

Indicates page is in main memory.

Protect

Indicates whether write operation is allowed.





Disk Block Descriptor Fields

Disk Block Descriptor

Swap device number

Logical device number of the secondary device that holds the corresponding page. This allows more than one device to be used for swapping.

Device block number

Block location of page on swap device.

Type of storage

Storage may be swap unit or executable file. In the latter case, there is an indication as to whether the virtual memory to be allocated should be cleared first.





Page Frame and Swap Use fields

Page Frame Data Table Entry

Page State

Indicates whether this frame is available or has an associated page. In the latter case, the status of the page is specified: on swap device, in executable file, or DMA in progress.

Reference count

Number of processes that reference the page.

Logical device

Logical device that contains a copy of the page.

Block number

Block location of the page copy on the logical device.

Pfdata pointer

Pointer to other pfdata table entries on a list of free pages and on a hash queue of pages.

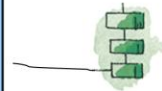
Swap-Use Table Entry


Reference count

Number of page table entries that point to a page on the swap device.

Page/storage unit number



Page identifier on storage unit.





Page Replacement

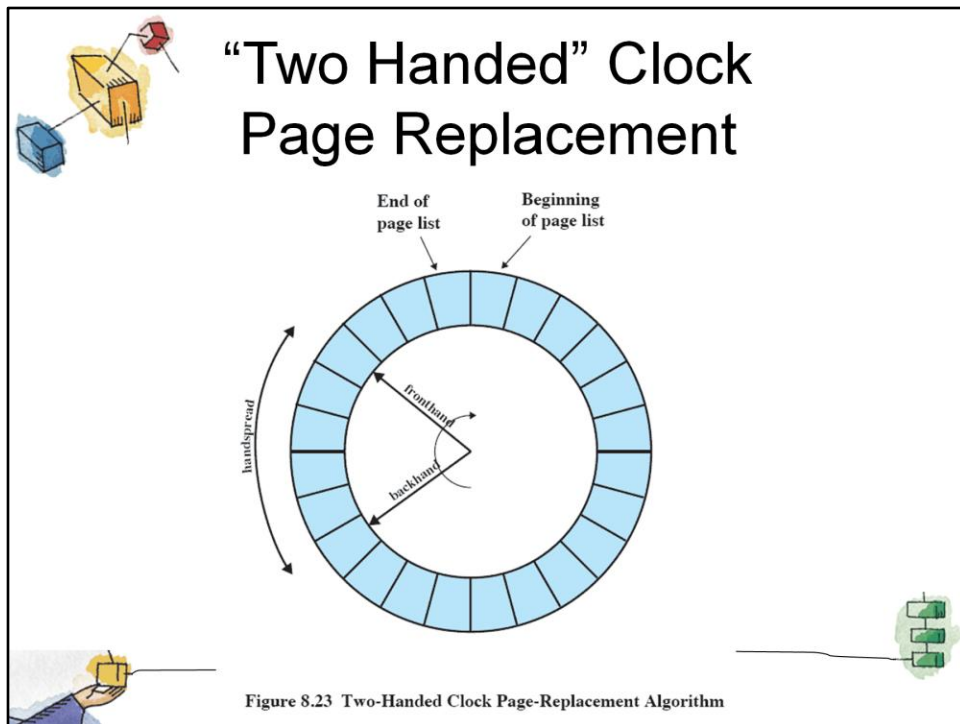
- The page frame data table is used for page replacement
- Pointers used to create several lists within the table
 - Free frame list
 - When the number of free frames drops below a threshold, the kernel will steal a number of frames to compensate.



The page frame data table is used for page replacement.

Several pointers are used to create lists within this table.

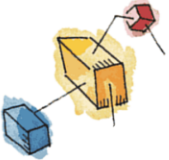
- All of the available frames are linked together in a list of free frames available for bringing in pages.
- When the number of available frames drops below a certain threshold, the kernel will steal a number of frames to compensate.



The page replacement algorithm used in SVR4 is a refinement of the clock policy algorithm known as the two-handed clock algorithm


The algorithm uses the reference bit in the page table entry for each page in memory that is eligible (not locked) to be swapped out.

- This bit is set to 0 when the page is first brought in and set to 1 when the page is referenced for a read or write.
- One hand in the clock algorithm, the fronthead, sweeps through the pages on the list of eligible pages and sets the reference bit to 0 on each page.
- Sometime later, the backhand sweeps through the same list and checks the reference bit.
- If the bit is set to 1, then that page has been referenced since the fronthead swept by; these frames are ignored.
- If the bit is still set to 0, then the page has not been referenced in the time interval between the visit by fronthead and backhand; these pages are placed on a list to be paged out.



Parameters for Two Handed Clock

- **Scanrate:**
 - The rate at which the two hands scan through the page list, in pages per second
- **Handspread:**
 - The gap between fronthand and backhand
- Both have defaults set at boot time based on physical memory



Two parameters determine the operation of the algorithm:

Scanrate: The rate at which the two hands scan through the page list, in pages per second

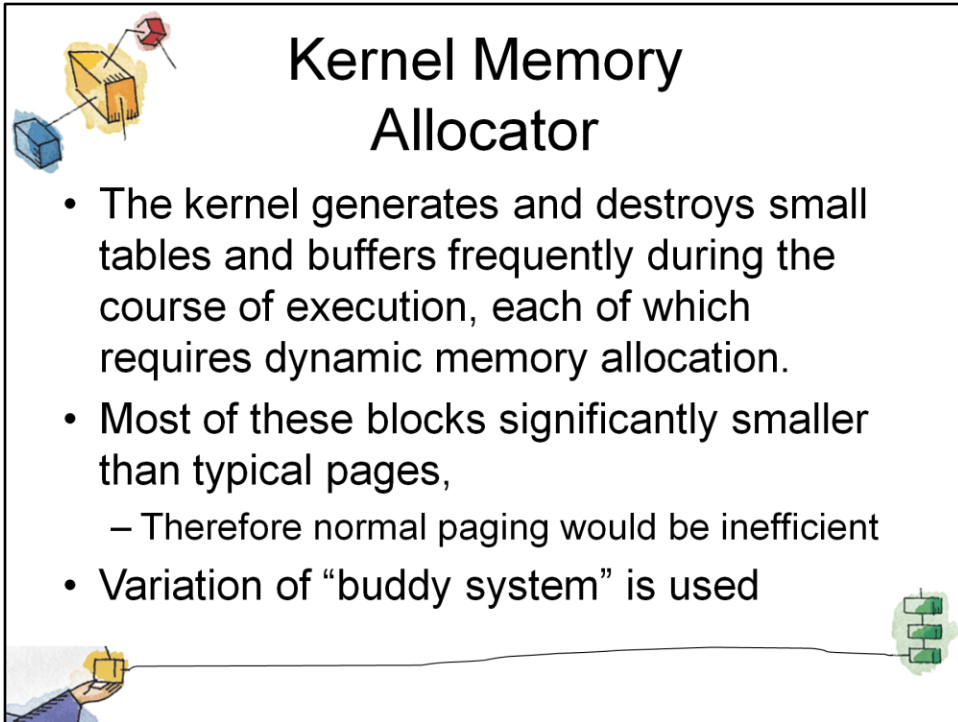
Handspread: The gap between fronthand and backhand

These two parameters have default values set at boot time based on the amount of physical memory.

The scanrate parameter can be altered to meet changing conditions.

- As the amount of free memory shrinks, the clock hands move more rapidly to free up more pages.

The handspread parameter, together with scanrate, determines the window of opportunity to use a page before it is swapped out due to lack of use.

A diagram titled "Kernel Memory Allocator" enclosed in a black border. The title is centered at the top in a large, bold, black font. To the left of the title is a small illustration of a yellow box with a red arrow pointing to it, and a blue box below it. To the right of the title is a small illustration of a green box with a red arrow pointing to it. Below the title is a bulleted list of three items. The first item is "The kernel generates and destroys small tables and buffers frequently during the course of execution, each of which requires dynamic memory allocation." The second item is "Most of these blocks significantly smaller than typical pages," followed by a sub-bullet "– Therefore normal paging would be inefficient". The third item is "Variation of 'buddy system' is used". At the bottom left of the diagram is a small illustration of a hand holding a yellow box. At the bottom right is a small illustration of a green box with a red arrow pointing to it.

Kernel Memory Allocator

- The kernel generates and destroys small tables and buffers frequently during the course of execution, each of which requires dynamic memory allocation.
- Most of these blocks significantly smaller than typical pages,
 - Therefore normal paging would be inefficient
- Variation of “buddy system” is used

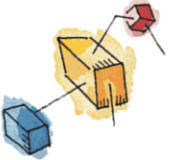
The kernel generates and destroys small tables and buffers frequently during the course of execution, each of which requires dynamic memory allocation.

Examples include:

- The pathname translation routine may allocate a buffer to copy a pathname from user space.
- The `alloca()` routine allocates STREAMS buffers of arbitrary size.
- Many UNIX implementations allocate zombie structures to retain exit status and resource usage information about deceased processes.
- In SVR4 and Solaris, the kernel allocates many objects (such as `proc` structures, `vnodes`, and file descriptor blocks) dynamically when needed.



Most of these blocks are significantly smaller than the typical machine page size,

- therefore the paging mechanism would be inefficient for dynamic kernel memory allocation.
- For SVR4, a modification of the buddy system is used.



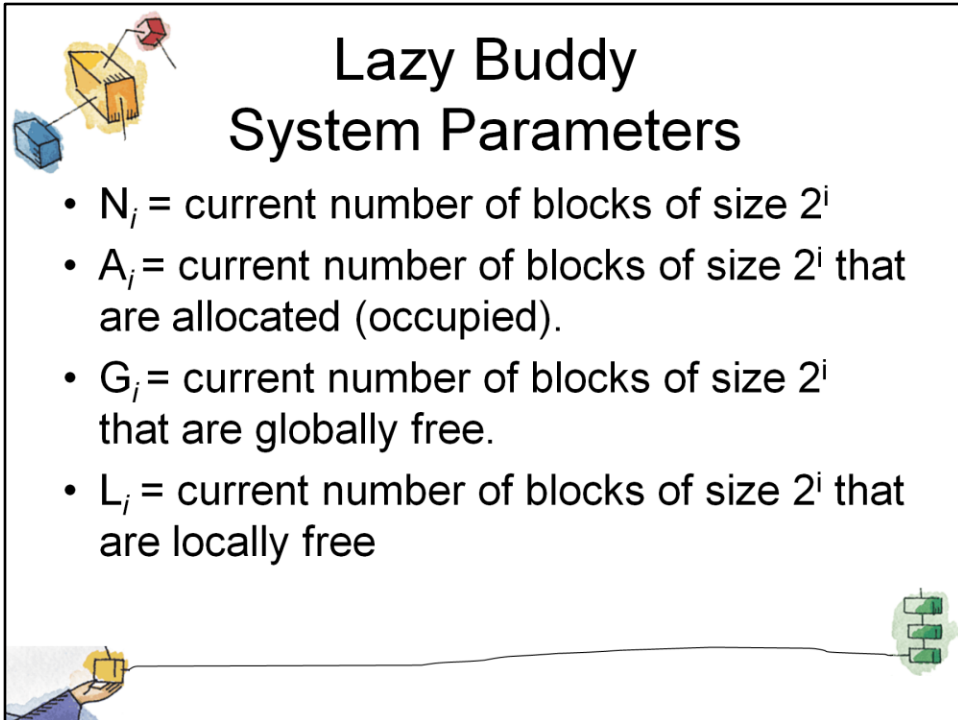
Lazy Buddy

- UNIX often exhibits steady-state behavior in kernel memory demand;
 - i.e. the amount of demand for blocks of a particular size varies slowly in time.
- To avoid unnecessary joining and splitting of blocks,
 - the lazy buddy system defers coalescing until it seems likely that it is needed, and then coalesces as many blocks as possible.



Therefore, if a block of size 2^i is released and is immediately coalesced with its buddy into a block of size 2^{i+1} , the kernel may next request a block of size 2^i , which may necessitate splitting the larger block again.

To avoid this unnecessary coalescing and splitting, the lazy buddy system defers coalescing until it seems likely that it is needed, and then coalesces as many blocks as possible.



Lazy Buddy System Parameters

- N_i = current number of blocks of size 2^i
- A_i = current number of blocks of size 2^i that are allocated (occupied).
- G_i = current number of blocks of size 2^i that are globally free.
- L_i = current number of blocks of size 2^i that are locally free

N_i = current number of blocks of size 2^i

A_i = current number of blocks of size 2^i that are allocated (occupied).

G_i = current number of blocks of size 2^i that are globally free.

- These are blocks that are eligible for coalescing;
- If the buddy of such a block becomes globally free, then the two blocks will be coalesced into a globally free block of size 2^{i+1}

- All free blocks (holes) in the standard buddy system could be considered globally free.

L_i = current number of blocks of size 2^i that are locally free.

- These are blocks that are not eligible for coalescing.
- Even if the buddy of such a block becomes free, the two blocks are not coalesced.
- Rather, the locally free blocks are retained in anticipation of future demand for a block of that size.



Lazy Buddy System Allocator

Initial value of D_i is 0

After an operation, the value of D_i is updated as follows


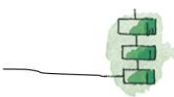
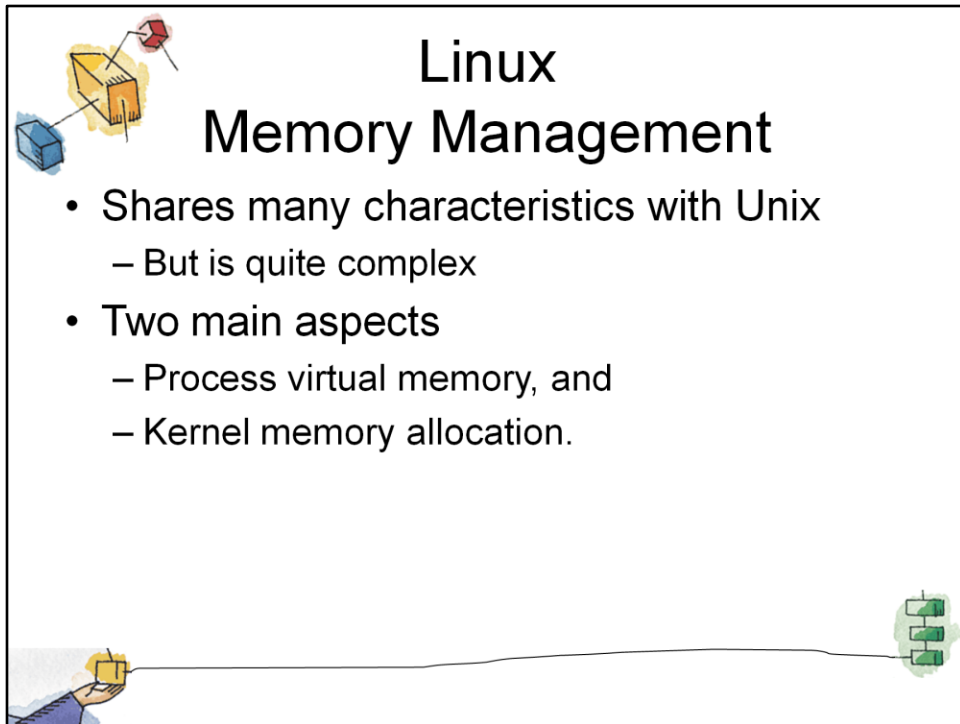
- (I)** if the next operation is a block allocate request:
- if there is any free block, select one to allocate
 - if the selected block is locally free
 - then $D_i := D_i + 2$
 - else $D_i := D_i + 1$
 - otherwise
 - first get two blocks by splitting a larger one into two (recursive operation)
 - allocate one and mark the other locally free
 - D_i remains unchanged (but D may change for other block sizes because of the recursive call)
- (II)** if the next operation is a block free request
- Case $D_i \geq 2$
 - mark it locally free and free it locally
 - $D_i := D_i - 2$
 - Case $D_i = 1$
 - mark it globally free and free it globally; coalesce if possible
 - $D_i := 0$
 - Case $D_i = 0$
 - mark it globally free and free it globally; coalesce if possible
 - select one locally free block of size 2^i and free it globally; coalesce if possible
 - $D_i := 0$
- 
- 

Figure 8.24 Lazy Buddy System Algorithm



Linux Memory Management

- Shares many characteristics with Unix
 - But is quite complex
- Two main aspects
 - Process virtual memory, and
 - Kernel memory allocation.

The slide features several illustrations: a blue cube and a yellow cube with a red dot in the top left; a hand holding a yellow cube in the bottom left; and a green circuit-like diagram in the bottom right.

Linux shares many of the characteristics of the memory management schemes of other UNIX implementations but has its own unique features.

Overall, the Linux memory-management scheme is quite complex.

Two main aspects of Linux memory management:

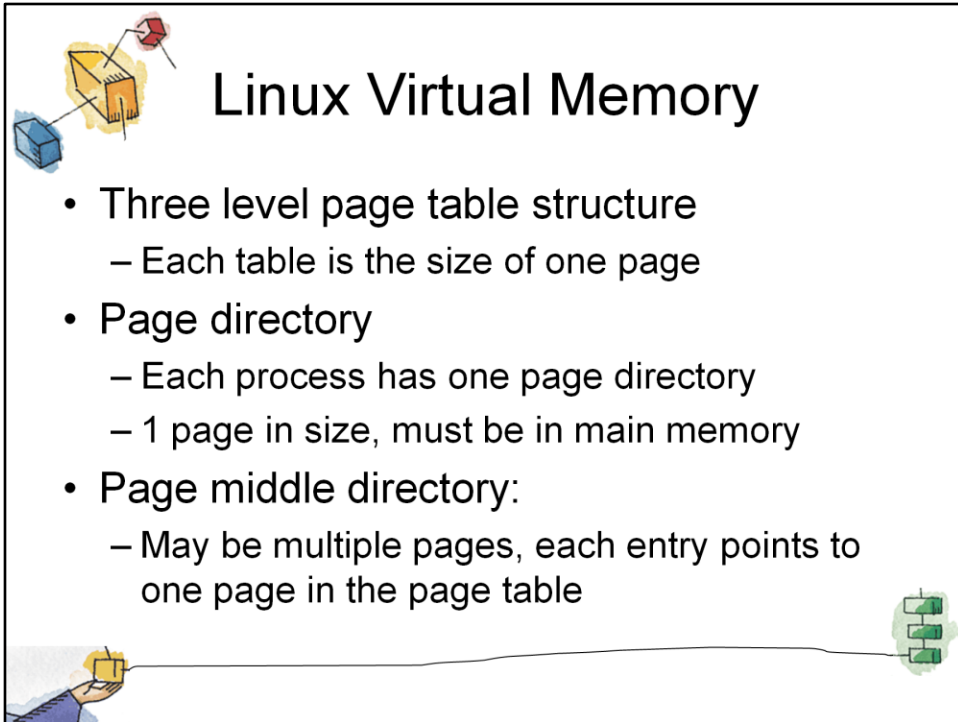
- process virtual memory, and
- kernel memory allocation.



Linux Memory Management

- Page directory
- Page middle directory
- Page table



A diagram titled "Linux Virtual Memory" enclosed in a black border. In the top-left corner, there is an illustration of a yellow cube with a red dot on top, connected by lines to a blue cube and a red cube. In the bottom-left corner, a hand is shown holding a yellow cube. In the bottom-right corner, there is a small green icon of a server rack. The main content is a bulleted list describing the three-level page table structure.

Linux Virtual Memory

- Three level page table structure
 - Each table is the size of one page
- Page directory
 - Each process has one page directory
 - 1 page in size, must be in main memory
- Page middle directory:
 - May be multiple pages, each entry points to one page in the page table

Three-level page table structure, consisting of the following types of tables

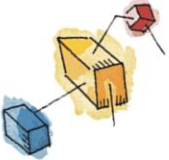
- each individual table is the size of one page:

Page directory:

- An active process has a single page directory that is the size of one page.
- Each entry in the page directory points to one page of the page middle directory.
- The page directory must be in main memory for an active process.


Page middle directory:

- May span multiple pages.
- Each entry in the page middle directory points to one page in the page table.



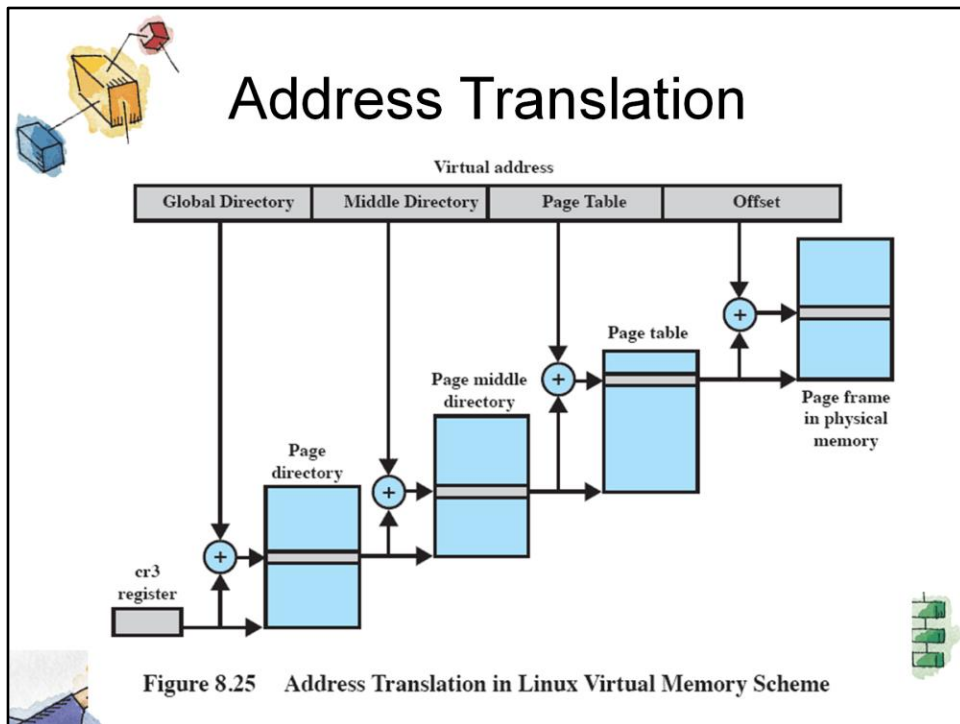
Linux Memory cont

- Page table
 - May also span multiple pages.
 - Each page table entry refers to one virtual page of the process.



Page table

- May also span multiple pages.
- Each page table entry refers to one virtual page of the process.



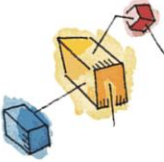
A virtual address in Linux is viewed as consisting of four fields (Figure 8.25).

The leftmost (most significant) field is used as an index into the page directory.

The next field serves as an index into the page middle directory.



The third field serves as an index into the page table.

The fourth field gives the offset within the selected page of memory.



Page Replacement

- Based on the clock algorithm
- The “use bit” is replaced with an 8-bit age variable
 - Incremented with each page access
- Periodically decrements the age bits
 - Any page with an age of 0 is “old” and is a candidate for replacement
- A form of Least Frequently Used policy



The Linux page replacement algorithm is based on the clock algorithm described in Section 8.2 (see Figure 8.16).

In the simple clock algorithm, a use bit and a modify bit are associated with each page in main memory.

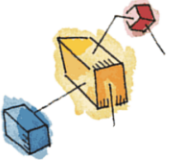
In the Linux scheme, the use bit is replaced with an 8-bit age variable.

Each time that a page is accessed, the age variable is incremented.

In the background, Linux periodically sweeps through the global page pool and decrements the age variable for each page as it rotates through all the pages in main memory.



- A page with an age of 0 is an “old” page that has not been referenced in some time and is the best candidate for replacement.
- The larger the value of age, the more frequently a page has been used in recent times and the less eligible it is for replacement.

• Thus, the Linux algorithm is a form of least frequently used policy.



Windows Memory Management

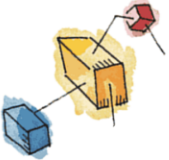
- The Windows virtual memory manager controls how memory is allocated and how paging is performed.
- Designed to operate over a variety of platforms
 - uses page sizes ranging from 4 Kbytes to 64 Kbytes.



The Windows virtual memory manager controls how memory is allocated and how paging is performed.



The memory manager is designed to operate over a variety of platforms and use page sizes ranging from 4 Kbytes to 64 Kbytes.

Intel and AMD64 platforms have 4096 bytes per page and Intel Itanium platforms have 8192 bytes per page.



Windows Virtual Address Map

- On 32 bit platforms each user process sees a separate 32 bit address space
 - Allowing 4G per process
- Some reserved for the OS,
 - Typically each user process has 32G of available virtual address space
 - With all processes sharing the same 2G system space

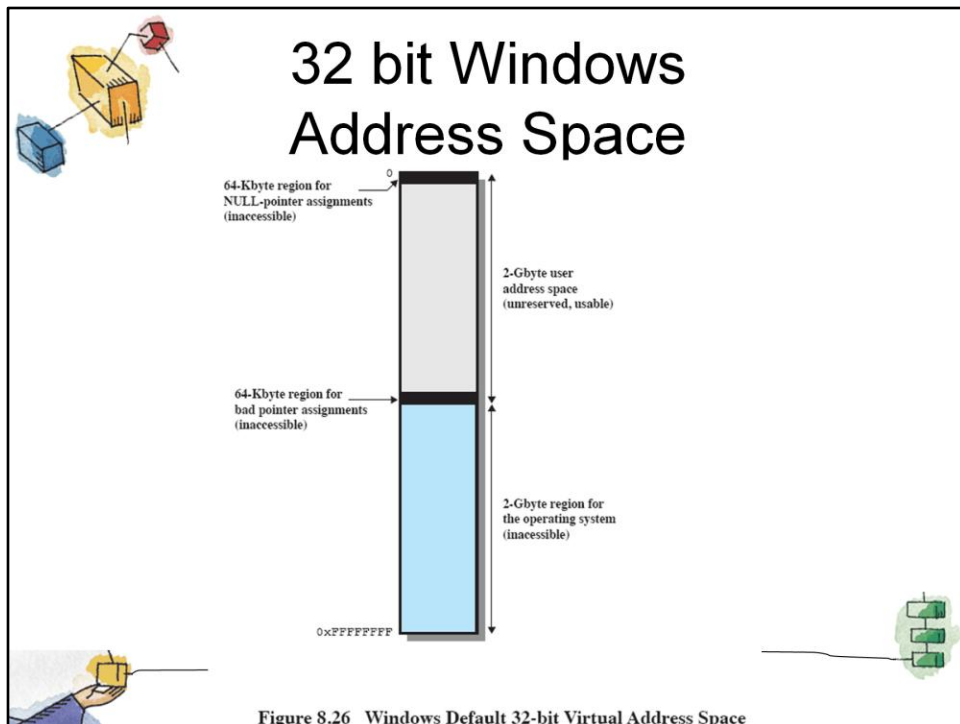


On 32-bit platforms, each Windows user process sees a separate 32-bit address space, allowing 4 Gbytes of virtual memory per process.

By default, a portion of this memory is reserved for the operating system, so each user actually has 2 Gbytes of available virtual address space

- and all processes share the same 2 Gbytes of system space.

There is an option that allows user space to be increased to 3 Gbytes, leaving 1 Gbyte for system space to support large memory-intensive applications on servers with multiple gigabytes of RAM

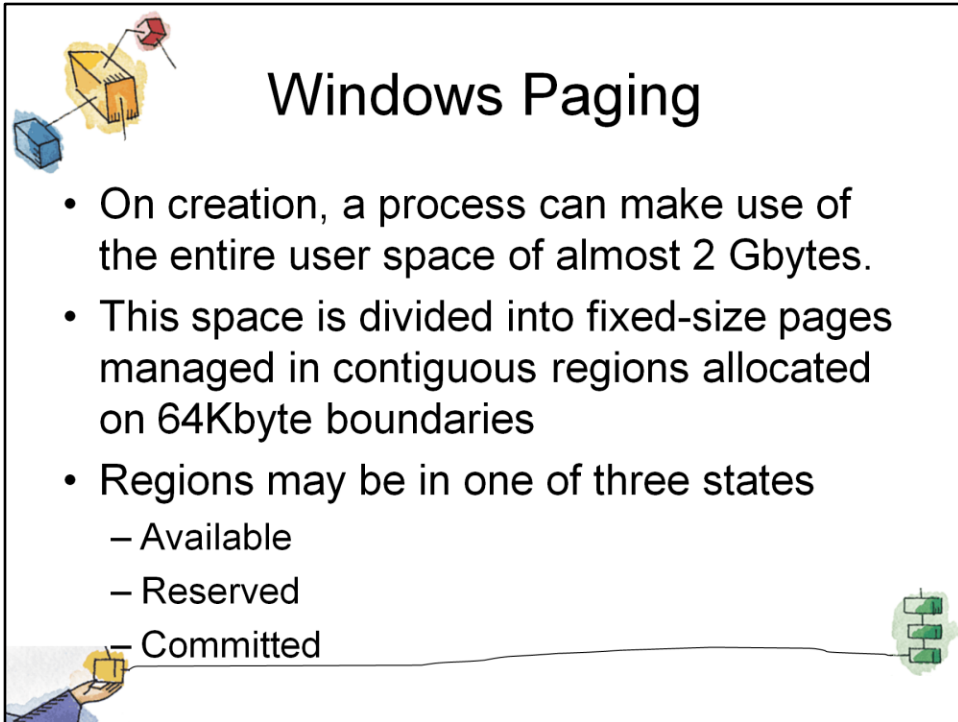


This figure shows the default virtual address space seen by a normal 32-bit user process.

It consists of four regions:

- 0x00000000 to 0x0000FFFF:
 - Set aside to help programmers catch NULL-pointer assignments.
- 0x00010000 to 0x7FFEFFFF:
 - Available user address space. This space is divided into pages that may be loaded into main memory.
- 0x7FFF0000 to 0x7FFFFFFF:
 - A guard page inaccessible to the user.
 - This page makes it easier for the operating system to check on out-of-bounds pointer references.
- 0x80000000 to 0xFFFFFFFF:
 - System address space.
 - This 2-Gbyte process is used for the Windows Executive, Kernel, and device drivers.

On 64-bit platforms, 8TB of user address space is available in Windows Vista.



Windows Paging

- On creation, a process can make use of the entire user space of almost 2 Gbytes.
- This space is divided into fixed-size pages managed in contiguous regions allocated on 64Kbyte boundaries
- Regions may be in one of three states
 - Available
 - Reserved
 - Committed

When a process is created, it can in principle make use of the entire user space of almost 2 Gbytes.

This space is divided into fixed-size pages, any of which can be brought into main memory, but the operating system manages them in contiguous regions allocated on 64-Kbyte boundaries.

A region can be in one of three states:

Available:

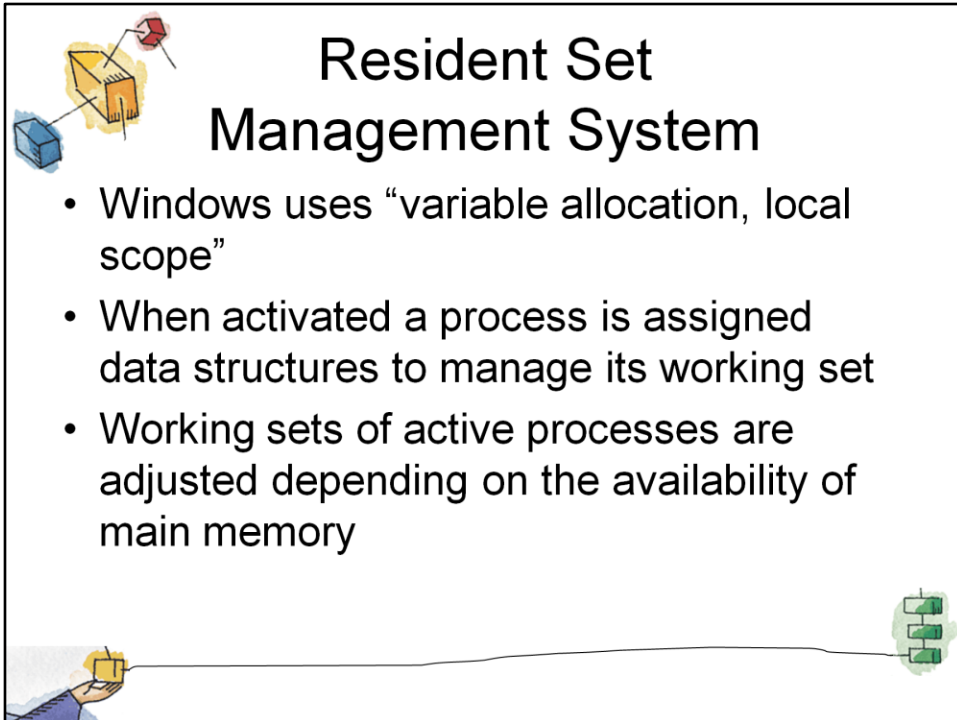
- Addresses not currently used by this process.

Reserved:

- Addresses that the virtual memory manager has set aside for a process so they cannot be allocated to another use (e.g., preserving space for a stack to grow).

Committed:

- Addresses for which the virtual memory manager has initialized for use by the process to access virtual memory pages.
- These pages can reside either on disk or in physical memory.



Resident Set Management System

- Windows uses “variable allocation, local scope”
- When activated a process is assigned data structures to manage its working set
- Working sets of active processes are adjusted depending on the availability of main memory

The resident set management scheme used by Windows is variable allocation, local scope

When a process is first activated, it is assigned data structures to manage its working set.

- As the pages needed by the process are brought into physical memory the memory manager uses the data structures to keep track of the pages assigned to the process.

Working sets of active processes are adjusted using the following general conventions:

- When main memory is plentiful, the virtual memory manager allows the resident sets of active processes to grow.
 - To do this, when a page fault occurs, a new physical page is added to the process but no older page is swapped out, resulting in an increase of the resident set of that process by one page
- When memory becomes scarce, the virtual memory manager recovers memory for the system by removing less recently used pages out of the working sets of active processes, reducing the size of those resident sets.