THE UNIVERSITY OF NEWCASTLE AUSTRALIA

FACULTY OF ENGINEERING AND BUILT ENVIRONMENT

www.newcastle.edu.au

OPERATING SYSTEMS

Week 7

Much of the material on these slides comes from the recommended textbook by William Stallings

Detailed content

Weekly program

- ✓ Week 1 – Operating System Overview
- ✓ Week 2 – Processes and Threads
- ✓ Week 3 – Scheduling
- ✓ Week 4 – Real-time System Scheduling and Multiprocessor Scheduling
- ✓ Week 5 – Concurrency: Mutual Exclusion and Synchronization
- ✓ Week 6 – Concurrency: Deadlock and Starvation
- ❑ **Week 7 – Memory Management I**
- ❑ Week 8 – Memory Management II
- ❑ Week 9 – Disk and I/O Scheduling
- ❑ Week 10 – File Management
- ❑ Week 11 – Security and Protection
- ❑ Week 12 – Revision of the course
- ❑ Week 13 – Extra revision (if needed)

Key Concepts From Last Lecture

- Three general approaches to deadlock: prevention, avoidance, detection
- Deadlock Prevention guarantees that deadlock will not occur
  - By assuring that one of the necessary conditions for deadlock will not occur
- Deadlock Avoidance involves analysis of each new resource request to determine if it could lead to deadlock and granting it only if deadlock is not possible
  - Banker's algorithm for resource allocation denial
- Deadlock detection OS always grant resource requests but periodically OS check for deadlock and take action to break the deadlock
  - Deadlock detection algorithm
  - Recovery techniques

Week 07 Lecture

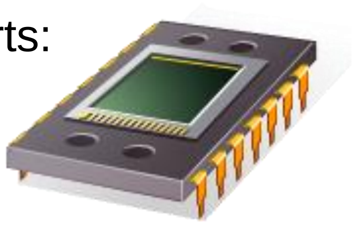
Memory Management

- ❑ Memory management requirements
- ❑ Memory Partitioning
- ❑ Fixed Partitioning
- ❑ Dynamic Partitioning
- ❑ Paging and Segmentation
- ❑ Virtual Memory
- ❑ VM mechanism, principle, advantage
- ❑ Paging with VM
- ❑ Translation Lookaside Buffer
- ❑ Segmentation with VM
- ❑ Combined Paging and Segmentation

Videos to watch before lecture

Memory management

- In a uni-programming system, main memory is divided into two parts:
  - one part for the operating system (resident monitor, kernel)
  - one part for the program currently being executed.

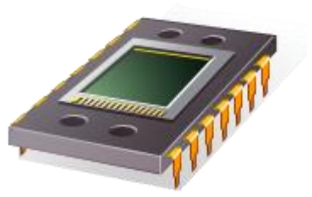


- In a multiprogramming system, the "user" part of memory must be further subdivided to accommodate multiple processes.
  - The task of subdivision is carried out dynamically by the operating system and is known as **memory management.**

- Effective memory management is vital in a multiprogramming system.
  - If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle.
  - Thus memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

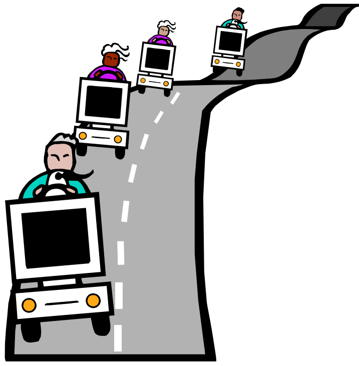
Memory management requirements

- Memory management is intended to satisfy the following requirements:
  1. Relocation
  2. Protection
  3. Sharing
  4. Logical organization
  5. Physical organization

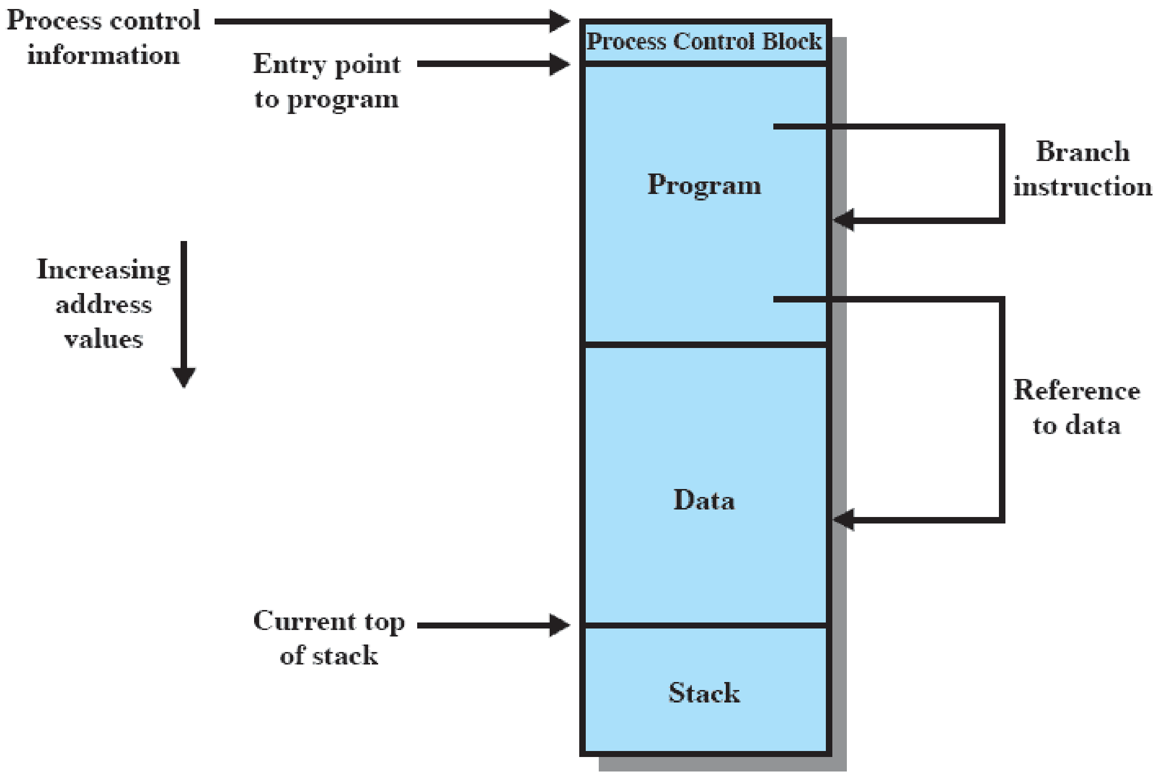


1. Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program
- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization
- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
  - May need to *relocate* the process to a different area of memory



Addressing requirements

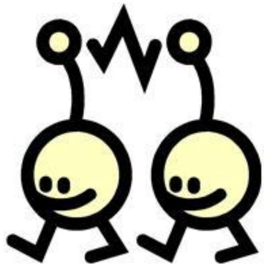


2. Protection

- Each process should be **protected** against **accidental** or **intentional interference** by other processes.
- Location of a program in main memory is unpredictable
- Memory references generated by a process must be checked at run time
- Mechanisms that support relocation also support protection
- Protection must be **provided by the hardware**, because:
  - Addresses are not known until runtime;
  - The OS cannot predict the address references of a user program;

3. Sharing

- Advantageous to allow each process access to the same copy of the program rather than have their own separate copy
- Memory management must allow controlled access to shared areas of memory without compromising protection
  - – Processes which run the same code should run the same address space
- Mechanisms used to support relocation also support sharing capabilities



4. Logical organisation

- Main memory in a computer system is organized as a linear, or one-dimensional, address space,
  - consisting of a sequence of bytes or words.
- Secondary memory, at its physical level, is similarly organized.
- Most programs are organized into modules
  - some of which are unmodifiable (read only, execute only)
  - some of which contain data that may be modified.
- If the operating system and computer hardware can effectively deal with user programs and data in the form of modules of some sort, then a number of advantages can be realized
- The memory management system should provide a **logical view** of memory:
  - Modules can be written and compiled separately with cross referencing resolved at runtime.
  - Different degrees of protection should be available to different modules.
  - Modules should be sharable as designated by the application programmer.

5. Physical organisation

- The responsibility for main/secondary memory management could be assigned to the individual programmer, but this is impractical and undesirable
- Memory available for a program plus its data may be insufficient
  - *overlaying* allows various modules to be assigned the same region of memory but is time consuming to program
- In a multi-programming environment, programmer does not know how much space will be available

Memory partitioning

- Memory management brings processes into main memory for execution by the processor
  - Involves virtual memory
  - Based on segmentation and paging

- Partitioning
  - Used in several variations in some now-obsolete operating systems
  - Does not involve virtual memory

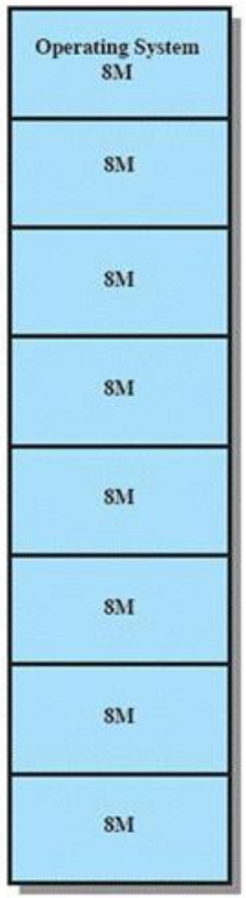
Fixed Partitioning

Two types:

- Equal-size partitions
- Unequal-size partitions

- Equal-size partitions
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition

- The operating system can swap out a process if all partitions are full and no process is in the Ready or Running state

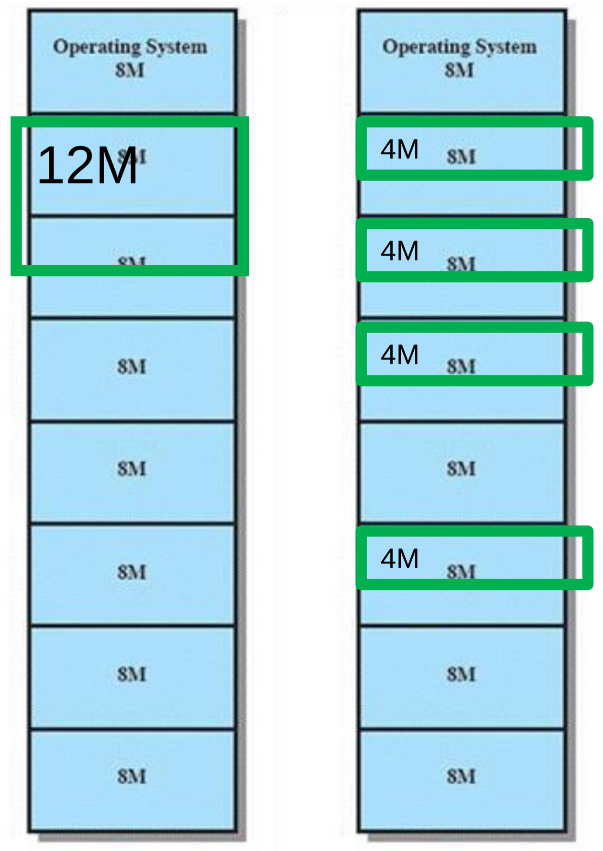


(a) Equal-size partitions

Equal-size Fixed Partitioning:

Disadvantages:

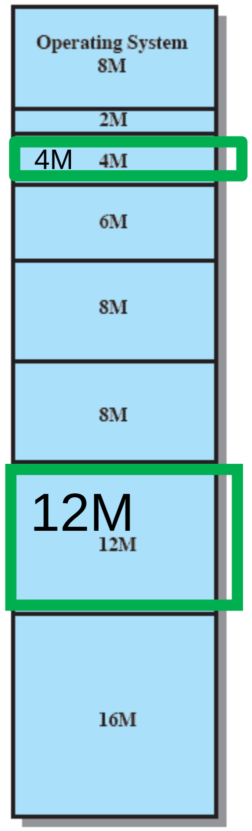
- A program may be too big to fit in a partition
  - Program needs to be designed with the use of overlays
- Main memory utilization is inefficient
  - Any program, regardless of size, occupies an entire partition
  - ***Internal fragmentation***
    - Wasted space due to the block of data loaded being smaller than the partition



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Unequal-size Fixed Partitioning

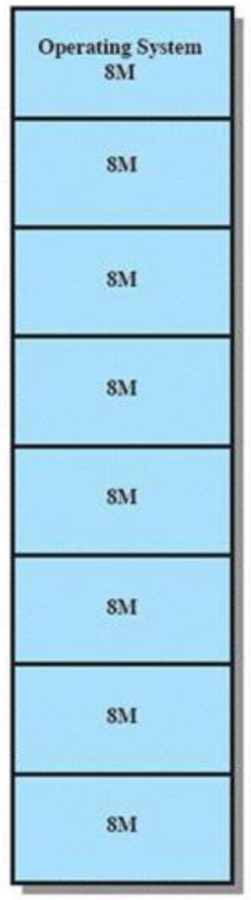
- Using unequal size partitions helps lessen the problems
  - Programs up to 16M can be accommodated without overlays
  - Partitions smaller than 8M allow smaller programs to be accommodated with less internal fragmentation



(b) Unequal-size partitions

Placement Algorithm: Fixed Equal Partitions

- With equal size partitions, placement is easy
  - Just place a program into any available unused partition.
  - If no partition is available, then one process must be swapped out to make room for the new one.



(a) Equal-size partitions

Placement Algorithm: Fixed Unequal Partitions

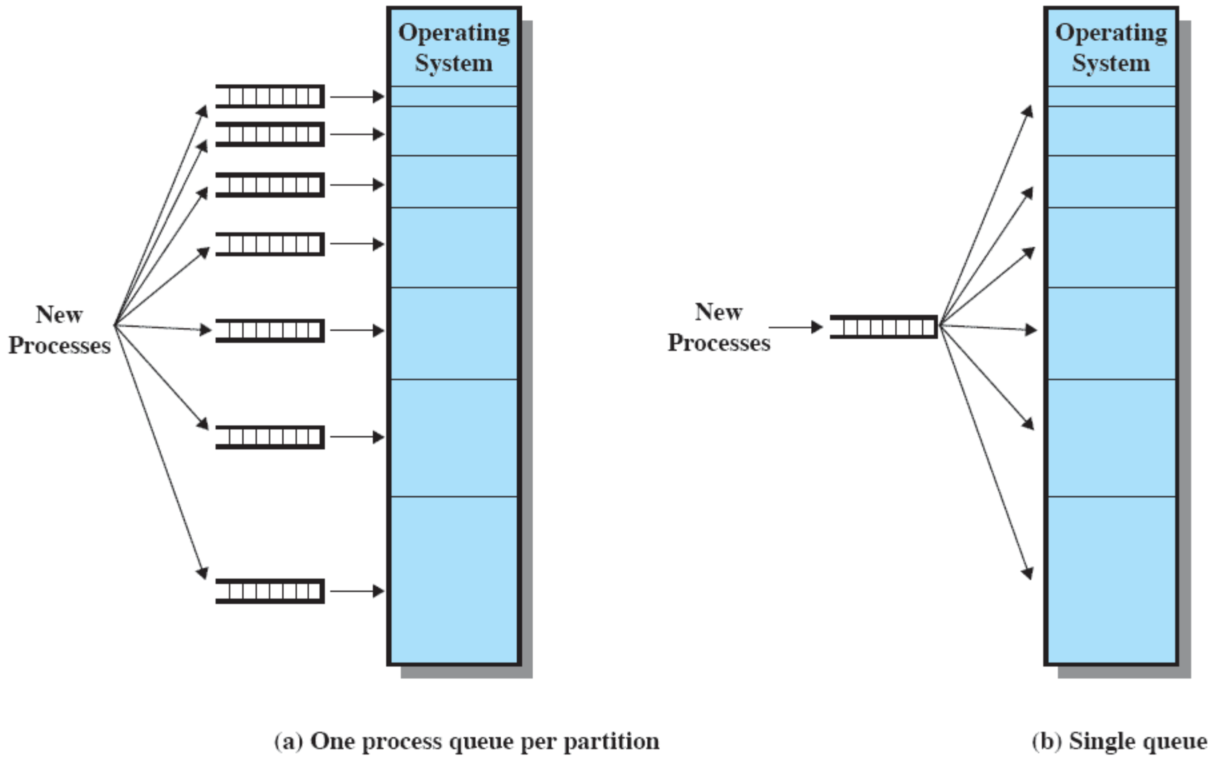


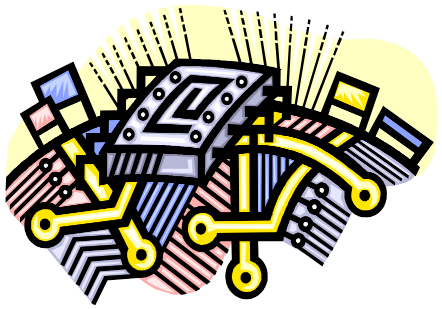
Figure 7.3 Memory Assignment for Fixed Partitioning

Equal VS unequal-size partitions

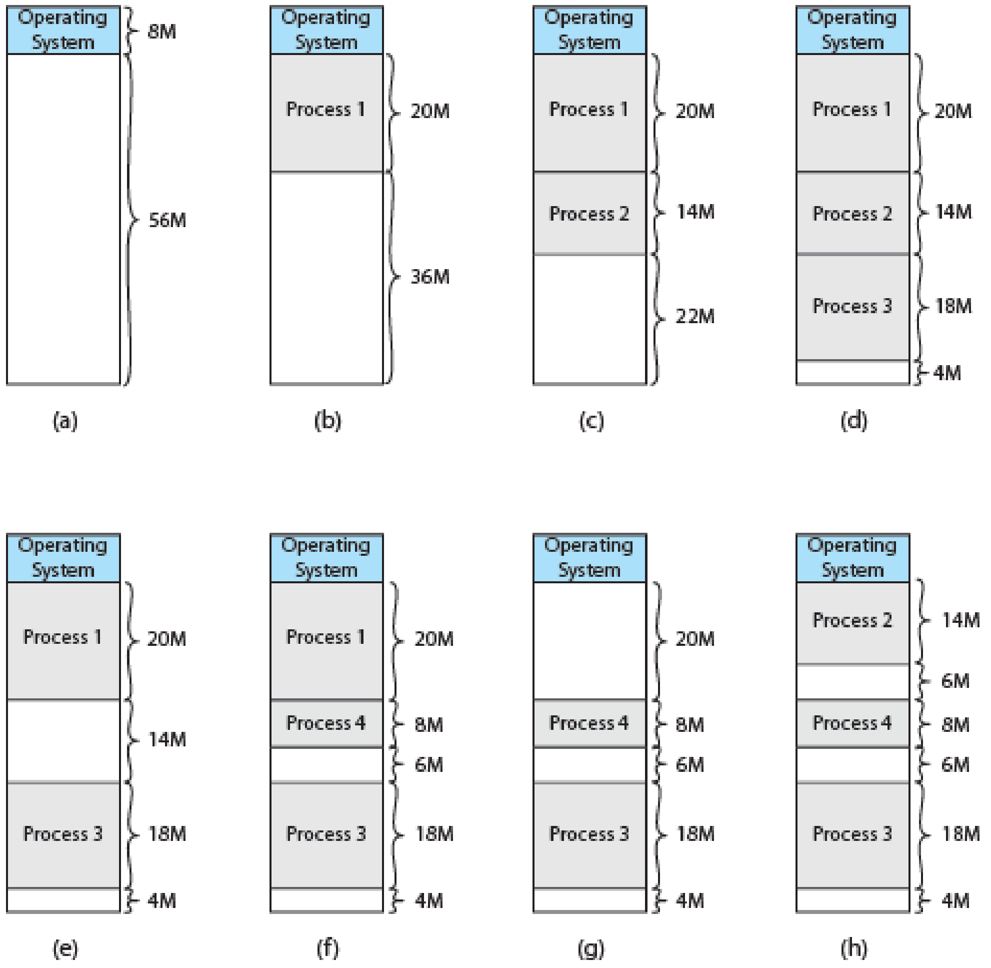
- **Equal Sized Partitions:**
  - Simple and require minimal OS software and processing overhead
- **Unequal-Sized Partitions:**
  - **Advantages:**
    - Provides a degree of flexibility
  - **Disadvantages:**
    - The number of partitions specified at system generation time limits the number of active processes in the system
    - Small jobs will not utilize partition space efficiently

Dynamic partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as it requires
- This technique was used by IBM's mainframe operating system, OS/MVT



Effect of dynamic partitioning



Effect of dynamic partitioning

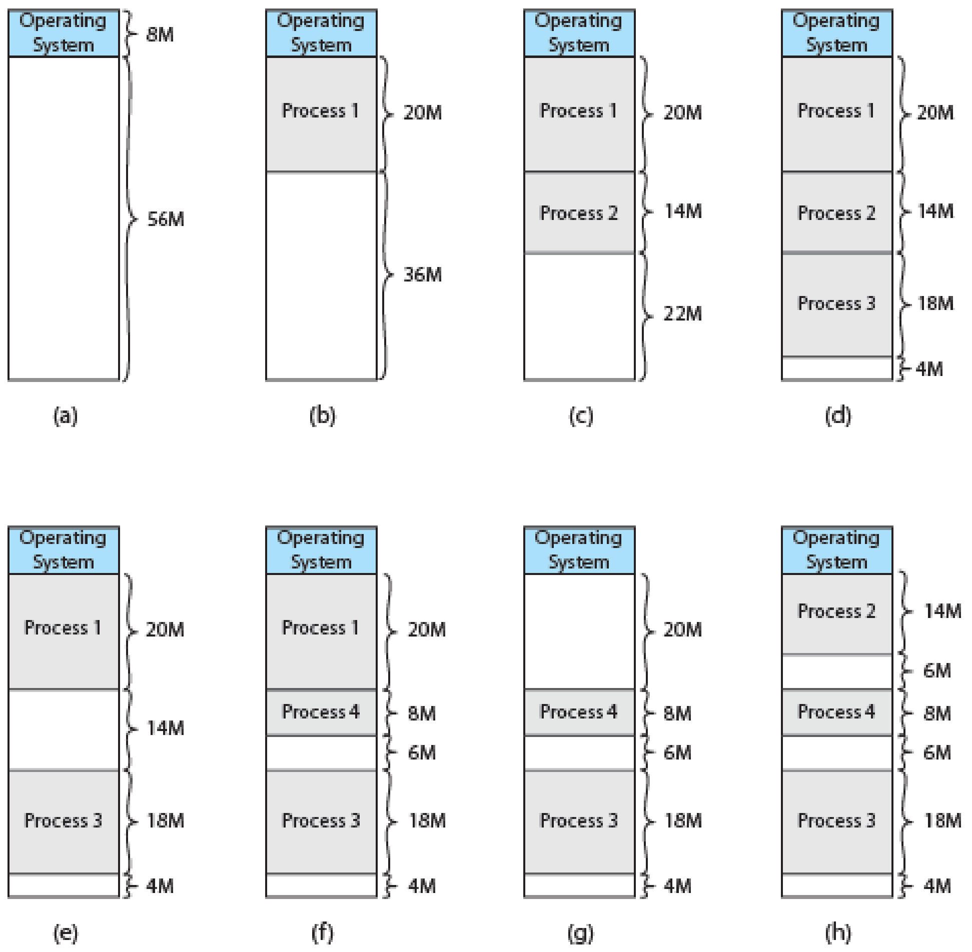


Figure 7.4 The Effect of Dynamic Partitioning

external fragmentation: memory that is external to all partitions becomes increasingly fragmented

Dynamic partitioning

External Fragmentation

- memory becomes more and more fragmented
- memory utilization declines

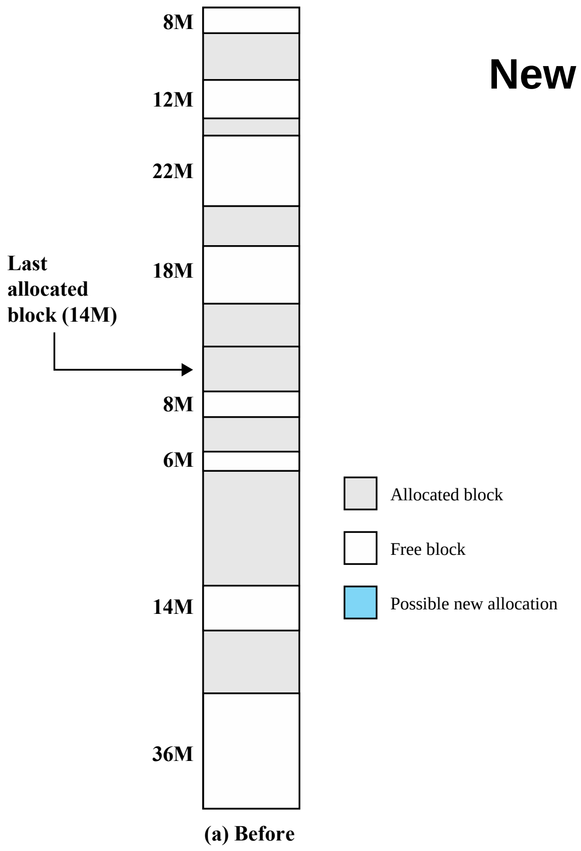
Compaction

- technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous
- free memory is together in one block
- time consuming and wastes CPU time

Placement algorithms

Best-fit	First-fit	Next-fit
• Chooses the block that is closest in size to the request	• Begins to scan memory from the **beginning** and chooses the first available block that is large enough	• Begins to scan memory from the location of the **last placement** and chooses the next available block that is large enough

Memory configuration example



New allocation request of 16MB block

First fit?

Best fit?

Next fit?

Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Memory configuration example

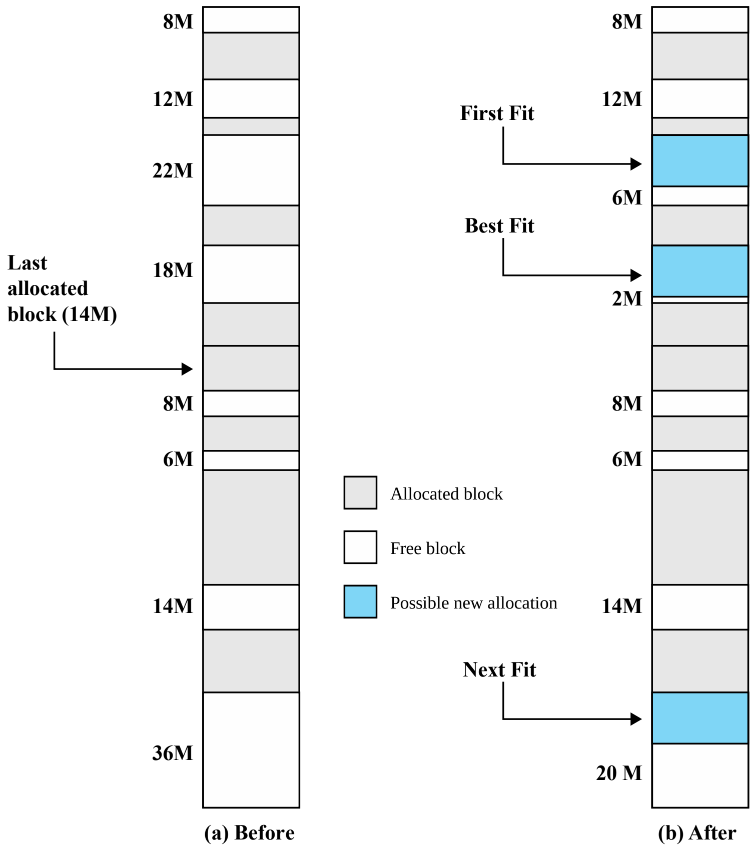


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Replacement algorithms

- There will be situations
  - All the processes in main memory are blocked
  - Insufficient memory, even after compaction, for an additional process
- OS will swap one of its processes to make room for a new process or a Ready-Suspended process

- Which process to replace?
  - But before that how to relocate a process?

Addresses

Logical

- reference to a memory location independent of the current assignment of data to memory

Relative

- address is expressed as a location relative to some known point

Physical or Absolute

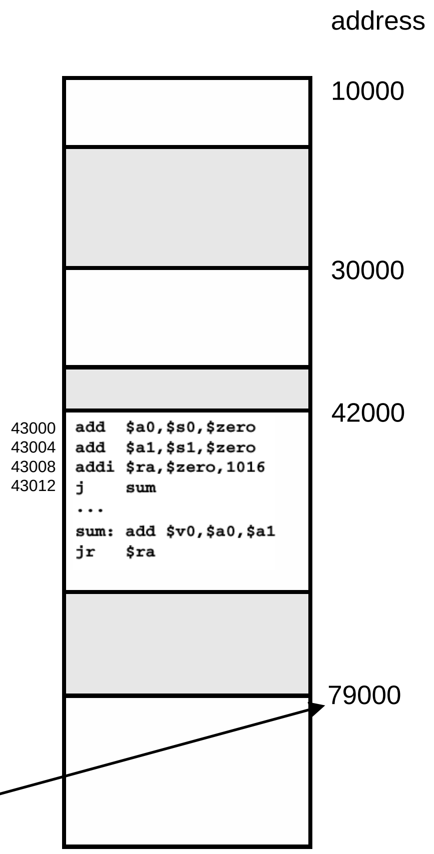
- actual location in main memory

Addresses: Relative VS Physical

```
main( ) {
    ...
    s = sum (a, b);
    ...
}
```

```
int sum(int x, int y) {
    return x + y;
}
```

```
address
  1000 add  $a0,$s0,$zero   # $a0 = x
  1004 add  $a1,$s1,$zero   # $a1 = y
  1008 addi $ra,$zero,1016  # $ra=1016
  1012 j    sum             # jump to sum
  1016 ...
  2000 sum: add $v0,$a0,$a1
  2004 jr   $ra             # jump to 1016
```



```
43000  add  $a0,$s0,$zero
43004  add  $a1,$s1,$zero
43008  addi $ra,$zero,1016
43012  j    sum
       ...
       sum: add $v0,$a0,$a1
       jr   $ra
```

Relative address + base address = Physical address

Relocation

- Programs that employ relative addresses in memory are loaded using **dynamic run-time loading**

- Typically, all of the memory references in the loaded process are relative to the origin of the program.

- Thus a hardware mechanism is needed for translating relative addresses to physical main memory addresses at the time of execution of the instruction that contains the reference.

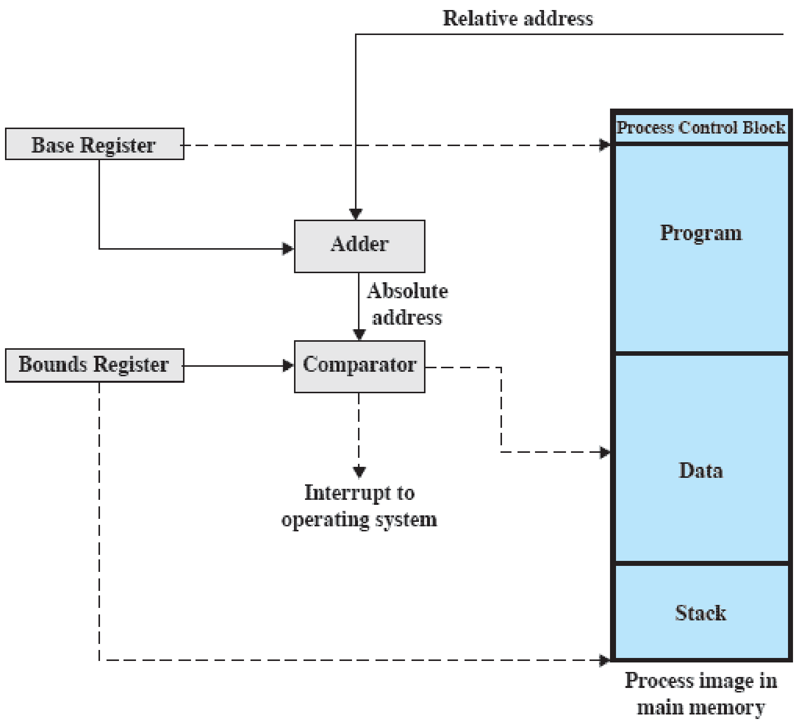


Figure 7.8 Hardware Support for Relocation

Relocation

- When a process is assigned to the Running state, a special processor register, sometimes called the *base register*, is loaded with the starting address in main memory of the program.

- There is also a “bounds” register that indicates the ending location of the program
  - These values must be set when the program is loaded into memory or when the process image is swapped in.

- During the course of execution of the process, relative addresses are encountered.
  - These include the contents of the instruction register,
  - instruction addresses that occur in branch and call instructions,
  - data addresses that occur in load and store instructions.

- Each such relative address goes through two steps of manipulation by the processor.
  - First, the value in the base register is added to the relative address to produce an absolute address.
  - Second, the resulting address is compared to the value in the bounds register. If the address is within bounds, then the instruction execution may proceed. Otherwise, an interrupt is generated to the operating system, which must respond to the error in some fashion.

- This scheme allows programs to be swapped in and out of memory during the course of execution. It also provides a measure of protection
  - Each process image is isolated by the contents of the base and bounds registers and safe from unwanted accesses by other processes.

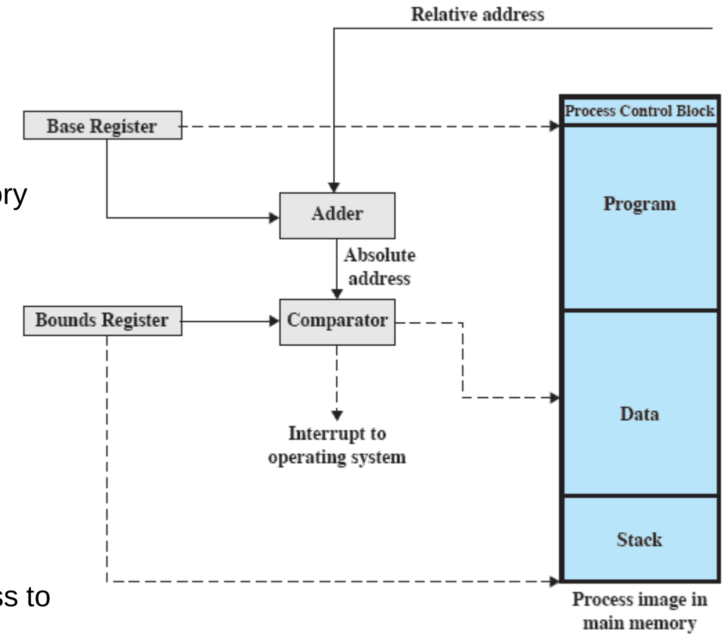
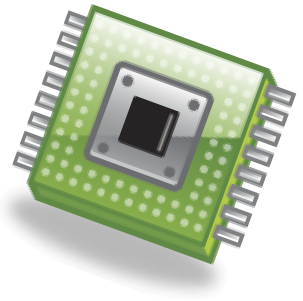


Figure 7.8 Hardware Support for Relocation

Memory management terms

Frame	A fixed-length block of main memory.
Page	A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.
Segment	A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).

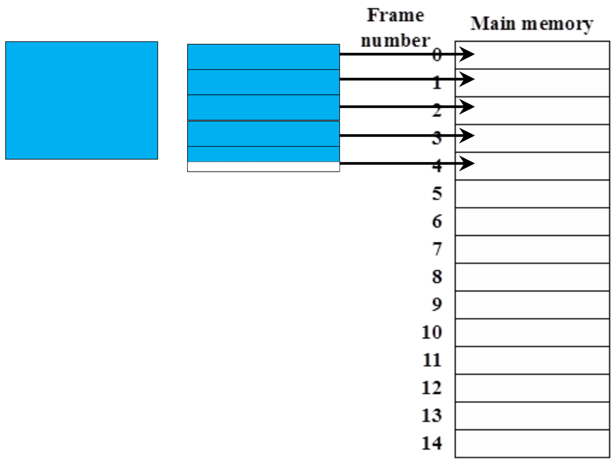


Paging

- Both unequal fixed-size and variable-size partitions are inefficient in the use of memory;
  - The former results in internal fragmentation,
  - The latter in external fragmentation.

- Suppose, however, that main memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of the same size.
  - Then the chunks of a process, known as **pages ,** could be assigned to available chunks of memory, known as **page frames , page frames.**

- Any wasted space in memory for each process is due to internal fragmentation consisting of only a fraction of the last page of a process. There is no external fragmentation.



Paging

- Partition memory into equal fixed-size chunks that are relatively small
- Process is also divided into small fixed-size chunks of the same size

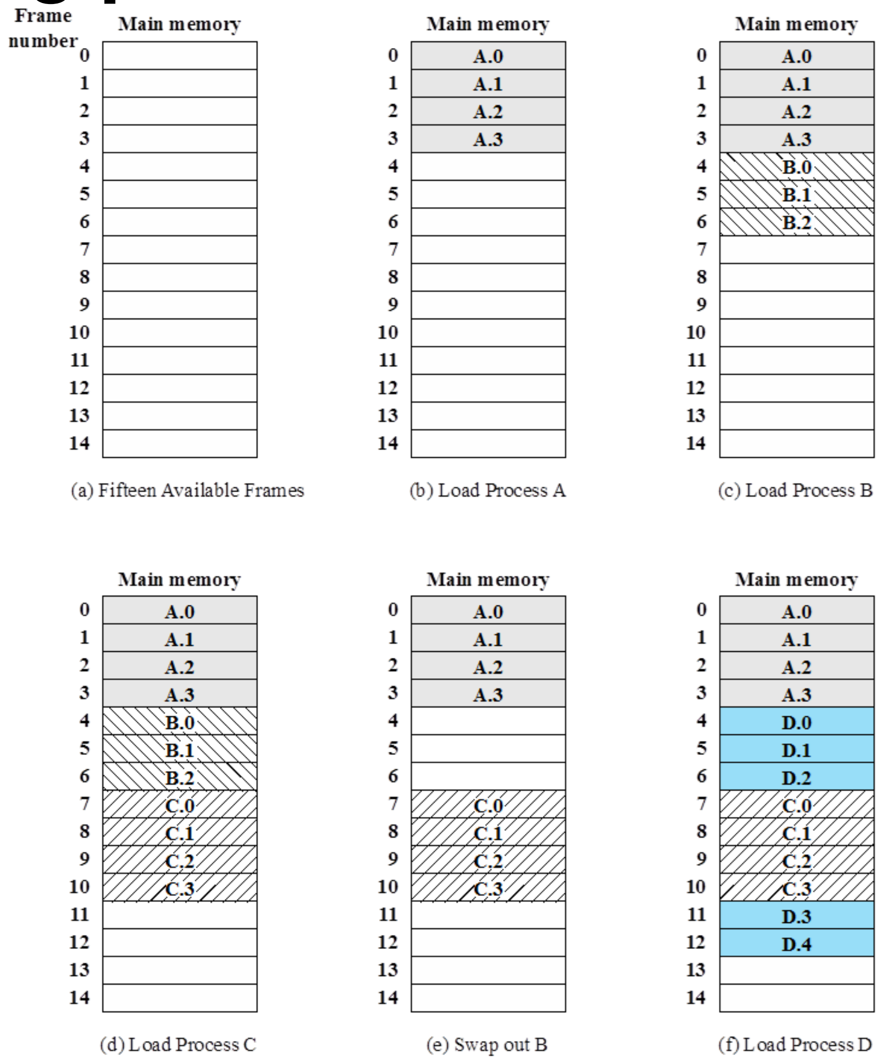
Pages

- chunks of a process

Frames

- available chunks of memory

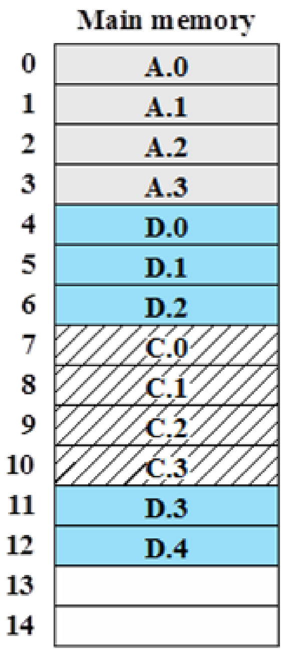
Assigning process to free frames


Frame number
Main memory
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
(a) Fifteen Available Frames
A.0 A.1 A.2 A.3
(b) Load Process A
A.0 A.1 A.2 A.3 B.0 B.1 B.2
(c) Load Process B
A.0 A.1 A.2 A.3 B.0 B.1 B.2 C.0 C.1 C.2 C.3
(d) Load Process C
A.0 A.1 A.2 A.3 C.0 C.1 C.2 C.3
(e) Swap out B
A.0 A.1 A.2 A.3 D.0 D.1 D.2 C.0 C.1 C.2 C.3 D.3 D.4
(f) Load Process D

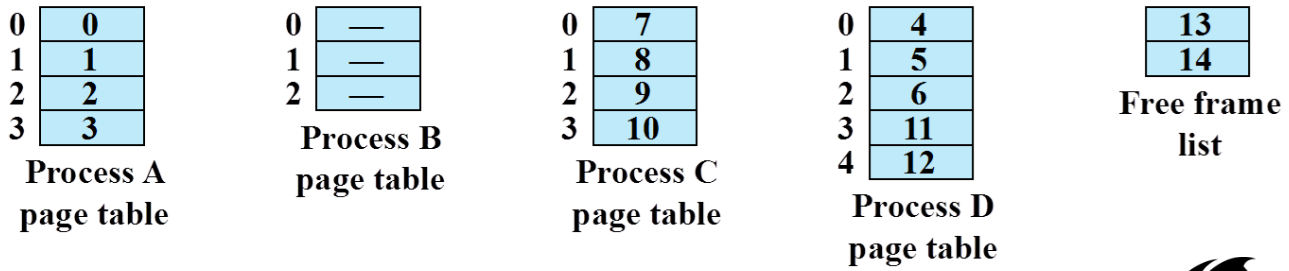
Page table

- Maintained by operating system for each process
- Contains the frame location for each page in the process
- Processor must know how to access for the current process
- Used by processor to produce a physical address

Data structures

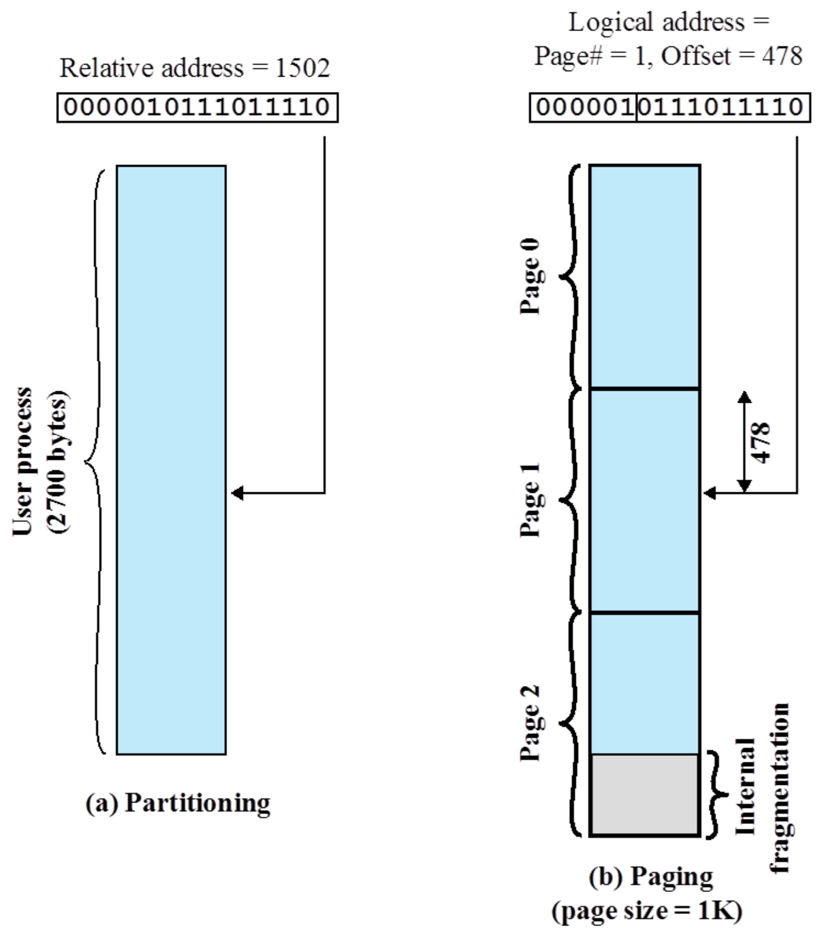


(f) Load Process D



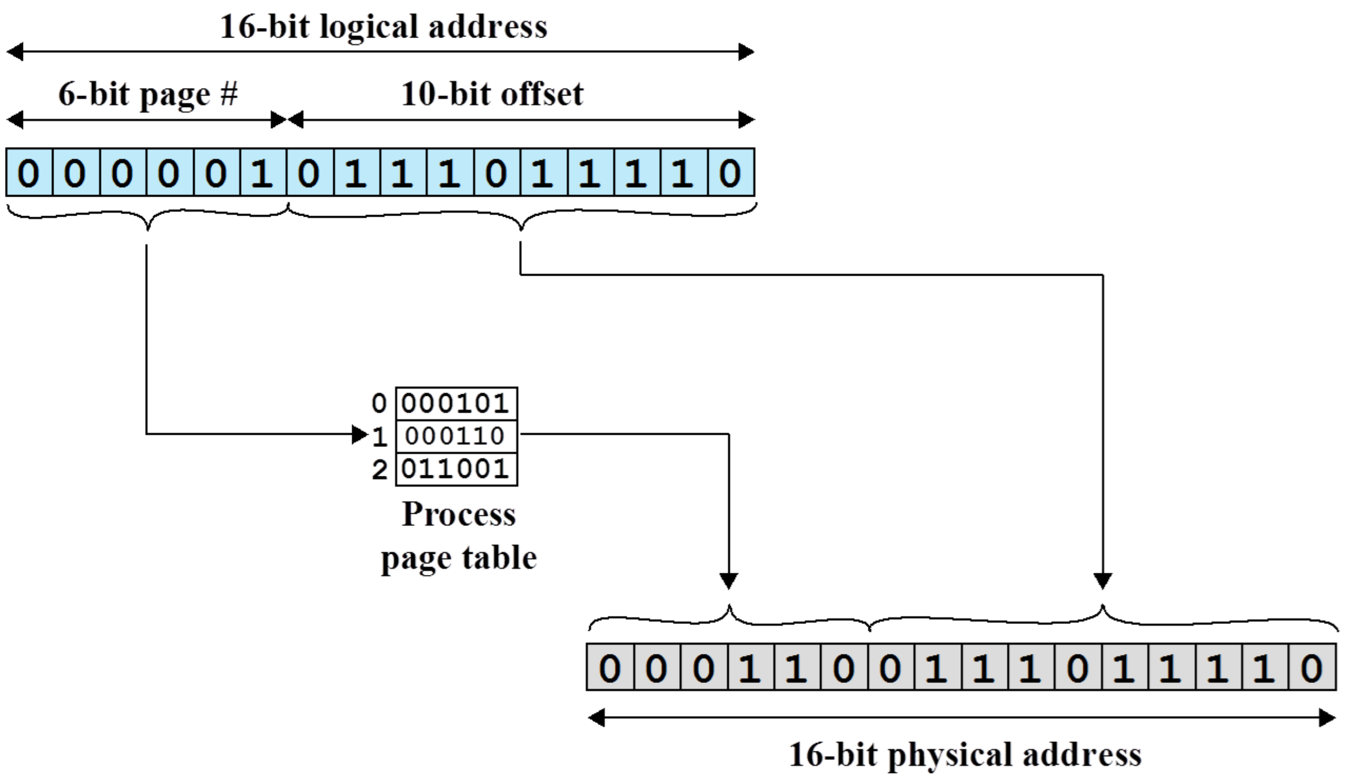
11/09/2018

Logical addresses



- Consider page/frame size a power of 2
- 16-bit addresses are used, and the page size is 1K (1,024 bytes)
- The relative address 1502, in binary form, is 0000010111011110.
- With a page size of 1K, an offset field of 10 bits is needed, leaving 6 bits for the page number.
  - Thus a program can consist of a maximum of 2^6 = 64 pages of 1K bytes each.
- Relative address 1502 in page-offset format
  0000010111011110
- Relative address 1502 corresponds to an offset of 478 (0111011110) on page 1 (000001), which yields the same 16-bit number, 0000010111011110.

Logical to physical address translation: Paging



(a) Paging

Segmentation

- A program can be subdivided into segments
  - **May vary in length**
  - There is a maximum length
- Addressing consists of two parts:
  - Segment number
  - An offset
- Similar to dynamic partitioning
  - Requires all segments to be in main memory for execution
  - No internal fragmentation, but suffers from external fragmentation
    - External fragmentation should be less. Why?
  - **Contrast:** a program may occupy more than one partition which need not to be contiguous…

Segmentation

- Usually visible to programmer - unlike paging
- Provided as a convenience for organizing programs and data
- Typically the programmer or the compiler will assign programs and data to different segments
- For purposes of modular programming the program or data may be further broken down into multiple segments
  - the principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation

Logical addresses: Revisited

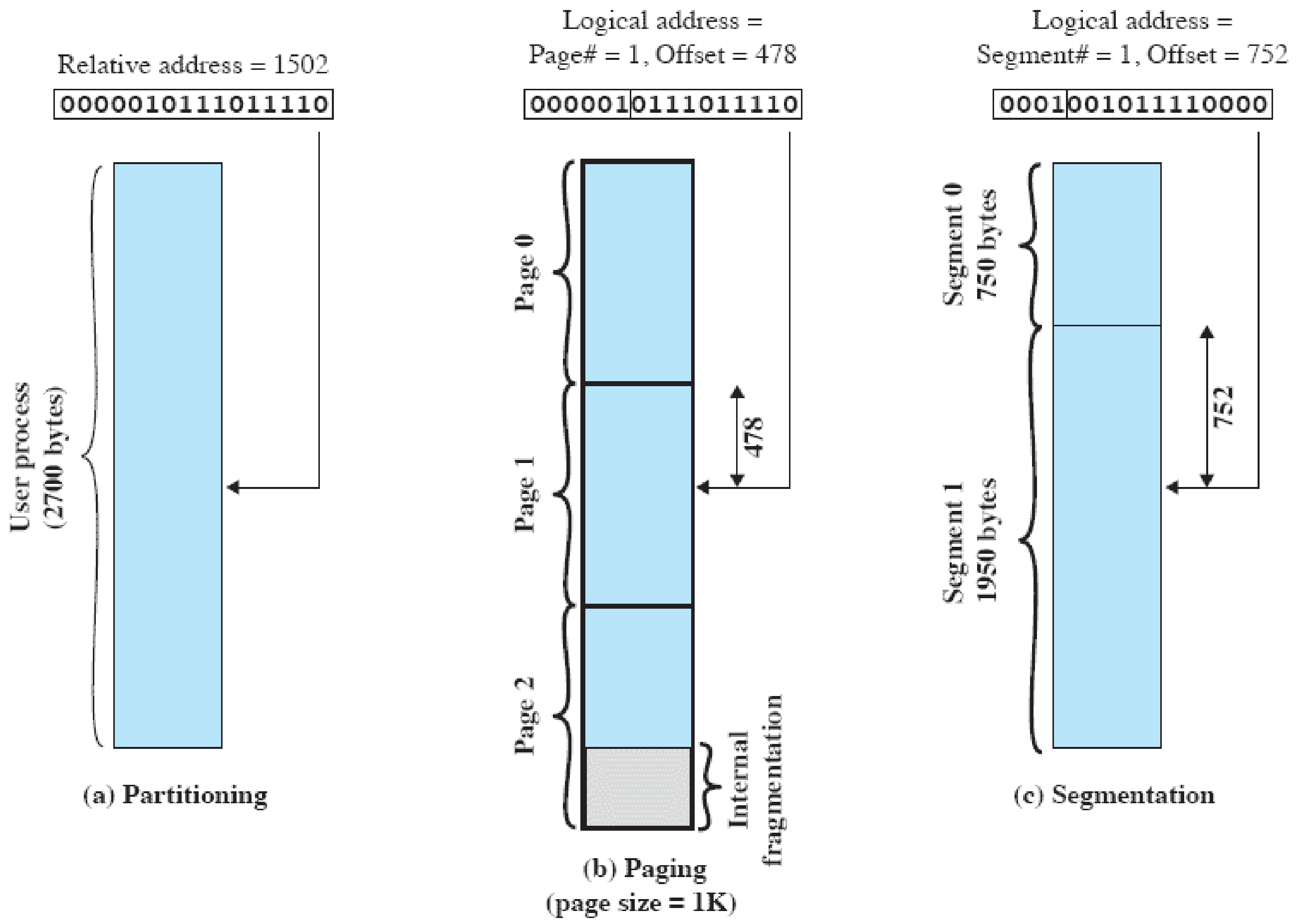
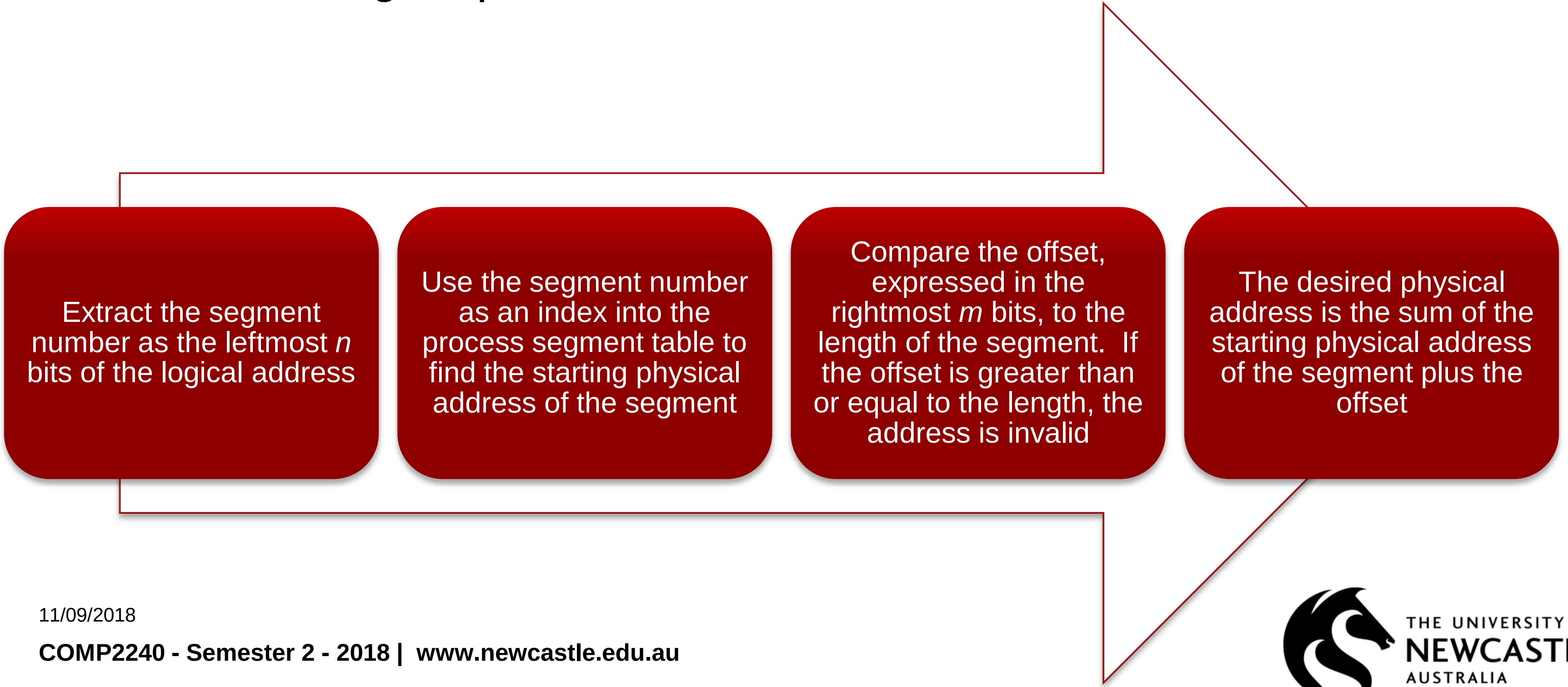


Figure 7.11 Logical Addresses

Address Translation

- Another consequence of unequal size segments is that there is no simple relationship between logical addresses and physical addresses
- The following steps are needed for address translation:



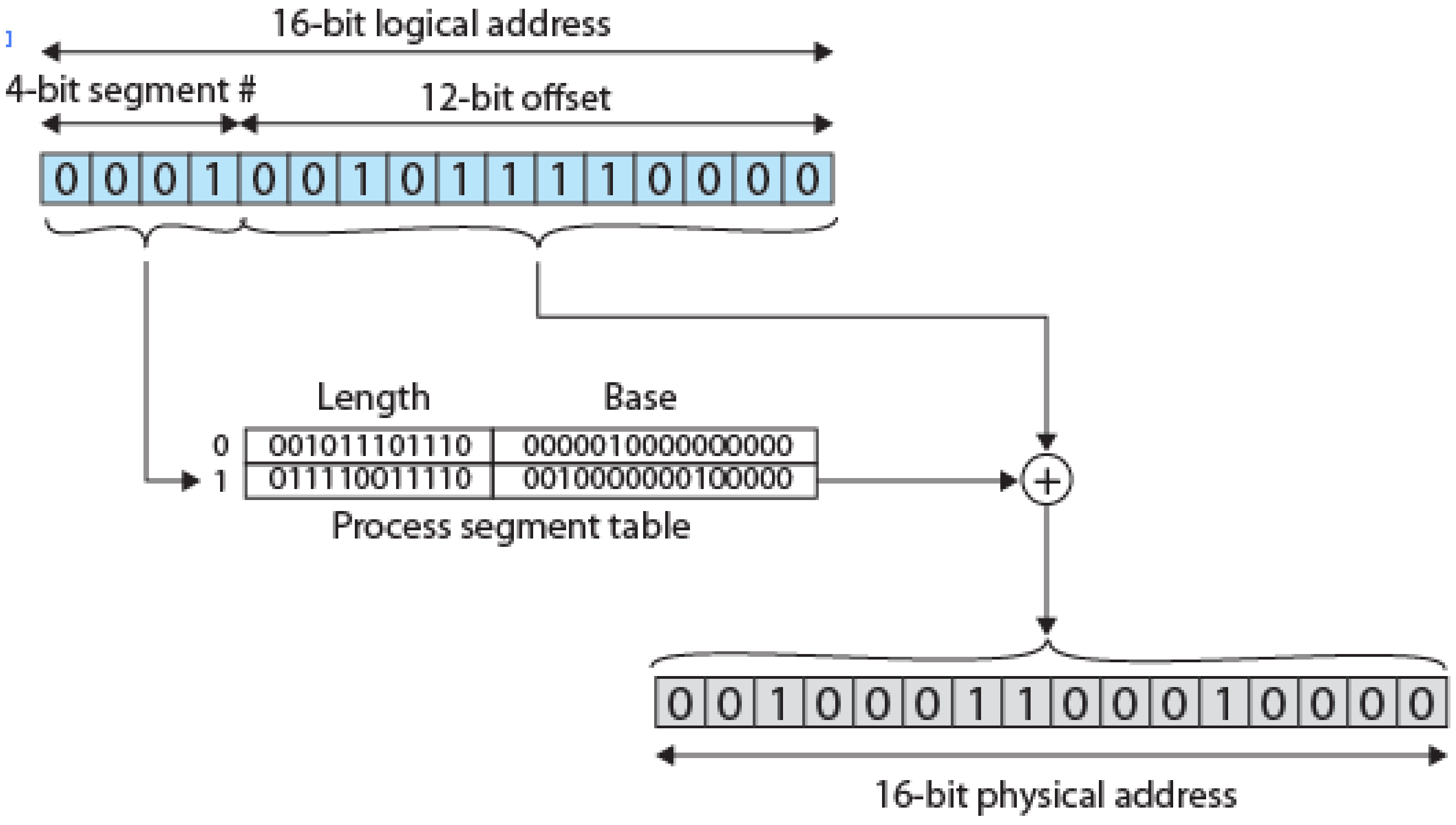
Extract the segment number as the leftmost *n* bits of the logical address

Use the segment number as an index into the process segment table to find the starting physical address of the segment

Compare the offset, expressed in the rightmost *m* bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid

The desired physical address is the sum of the starting physical address of the segment plus the offset

Logical to physical address translation: Segmentation

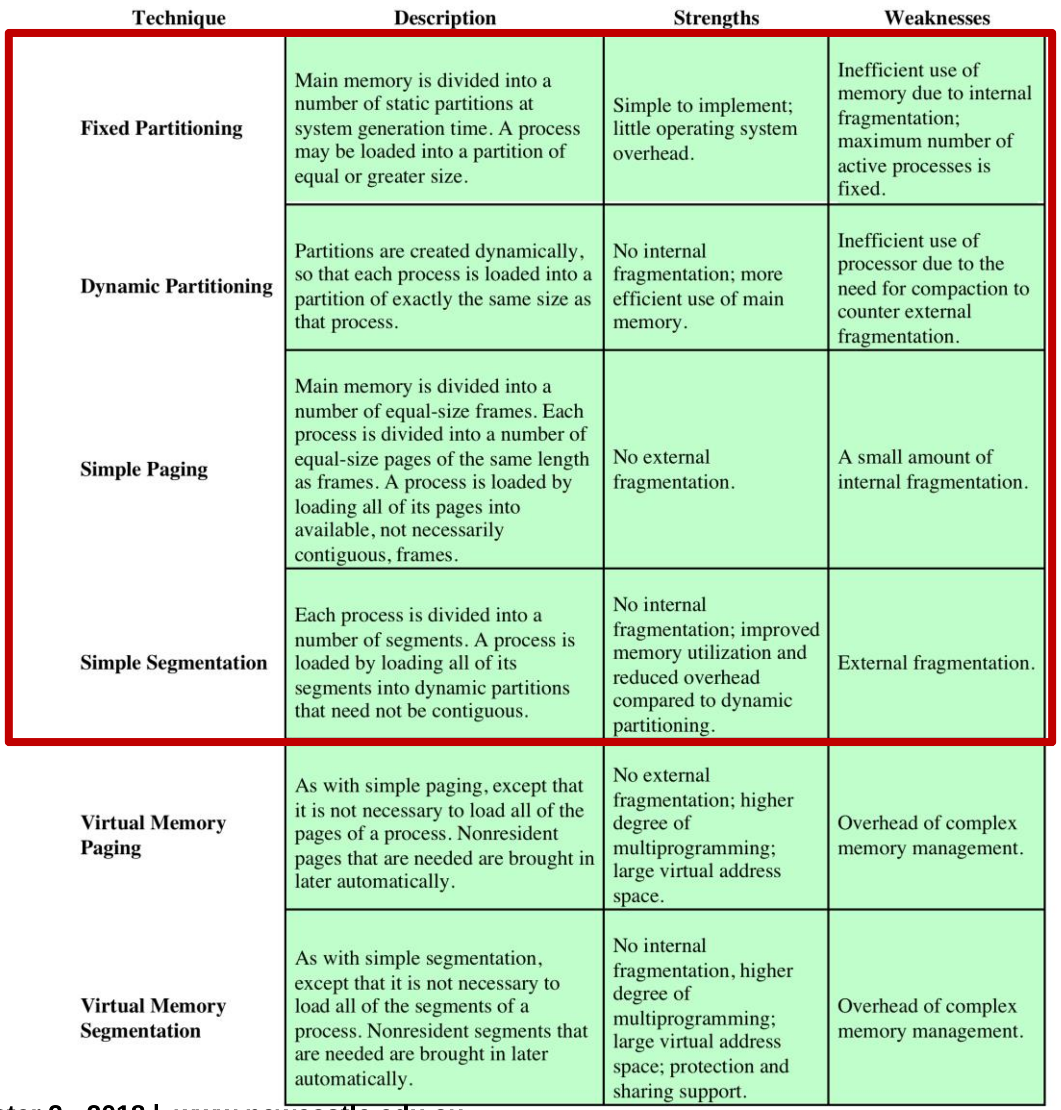


(b) Segmentation

Memory management summary

- One of the most important and complex tasks of an operating system
- Needs to be treated as a resource to be allocated to and shared among a number of active processes
- Desirable to maintain as many processes in main memory as possible
- Desirable to free programmers from size restriction in program development
- Basic tools are *paging* and *segmentation* (possible to combine)
  - Paging – small fixed-sized pages
  - Segmentation – pieces of varying size

Memory management techniques: review



Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

Virtual memory

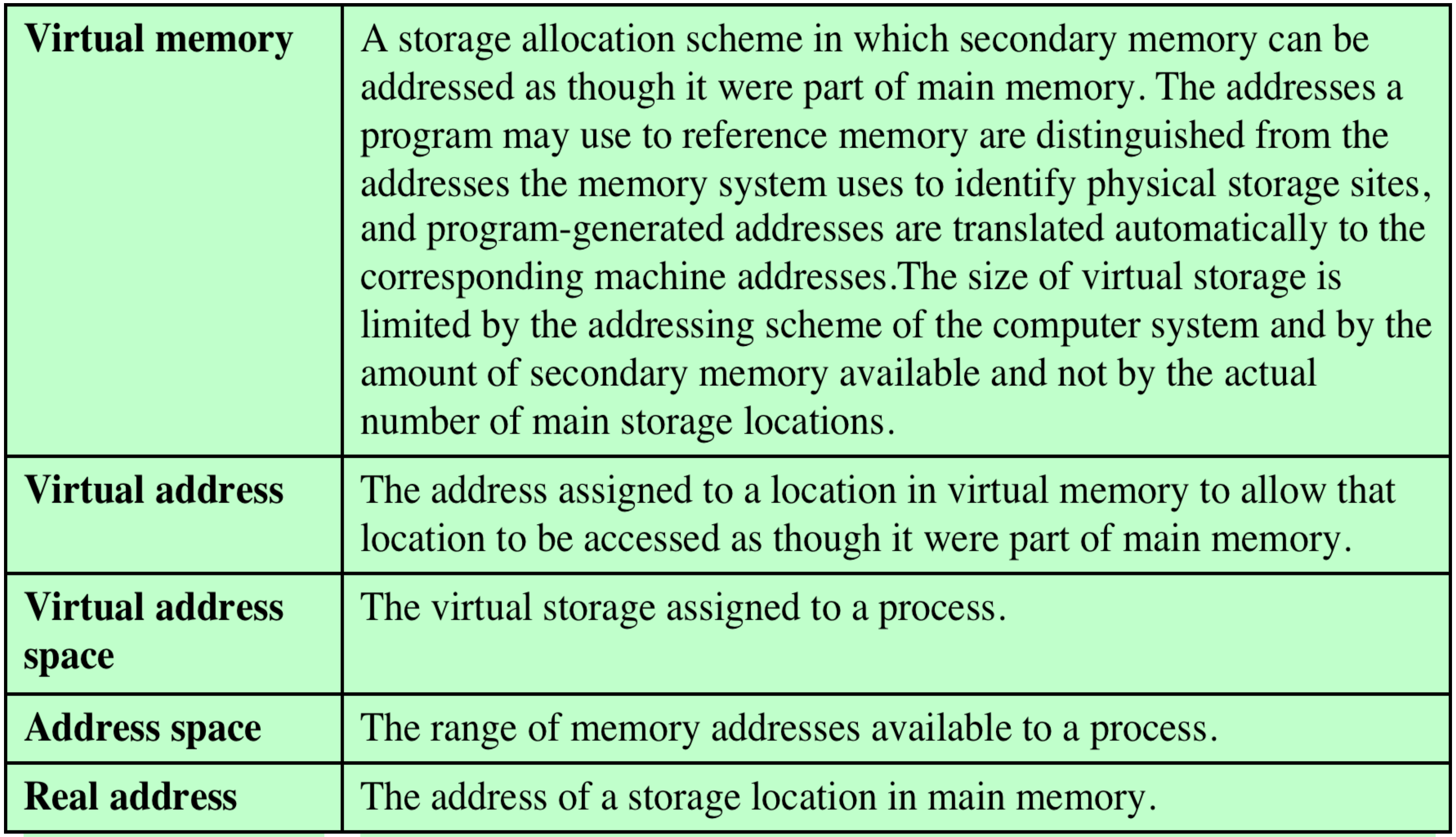
- Two characteristics of simple paging/segmentation:
  1. All memory references are **logical addresses** that can be translated to **physical addresses** at **runtime**.
     - Thus a process may be swapped in and out of memory and need not occupy the same region at different times during the execution.
  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.
     - This is achieved using a **page** or **segment** table.
- **Thus it is not necessary for *all* pages or segments to be in main memory for all the duration of the execution of a program.**

Virtual memory

- A **virtual memory system** ensures that the currently executing parts of a process are in memory, when required. It is possible that the parts which are not currently executing or accessed are on disk (secondary storage).

- Virtual memory gives the machine a **very large** (virtual) address space

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.

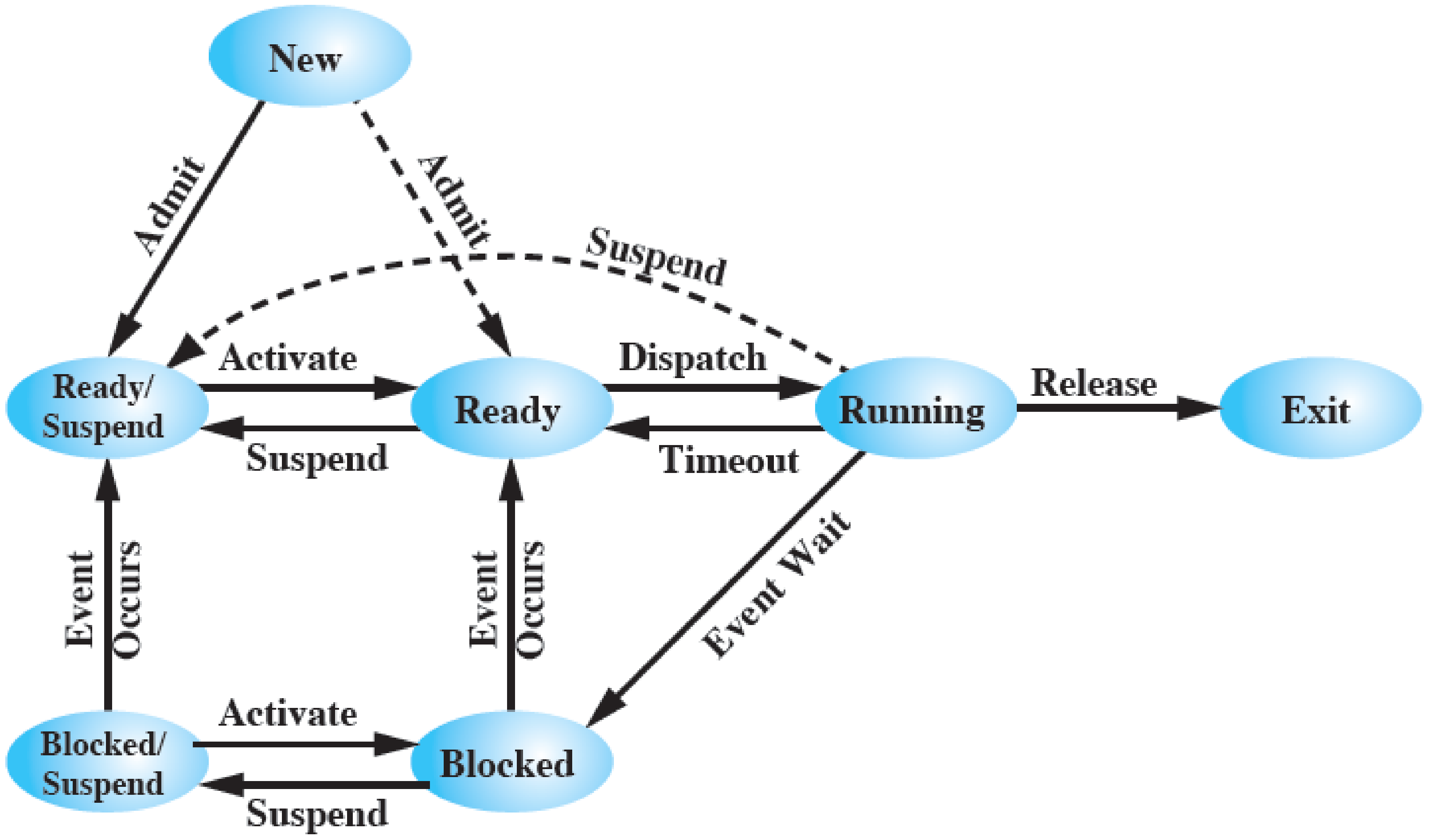
Virtual memory execution

- Those <u>pieces (pages or segments)</u> of a process which are currently in main memory form the **resident set** of the process.

- The resident set is marked in the page/segment table of a process.

- When a process requests a memory address ***A*** :
  - if *A* is in the resident set of pieces then the request is serviced normally;
  - if *A* is **not** in the resident set of pieces then a "fault" occurs:

1. an interrupt is generated;
2. the OS blocks the process;
3. the OS issues an I/O request for the required piece of the process;
4. the OS dispatches another process to run while the I/O request is performed.
5. when the I/O is complete, the OS places the process into the **ready** queue.


New
Admit
Admit
Suspend
Activate
Dispatch
Release
Ready/ Suspend
Ready
Running
Exit
Suspend
Timeout
Event Occurs
Event Occurs
Event Wait
Activate
Blocked/ Suspend
Blocked
Suspend

Virtual memory: Advantages

- **Advantages** of virtual memory include:
  1. **More processes** may be in memory at once.
     - The scheduler has a larger choice of processes which are ready to run, and the probability of idleness is decreased.
  2. **More memory**: it is possible for a process to be larger than main memory.
     - The programmer need not be concerned with the details of address translation: the *virtual memory* appears the same as *real memory*.

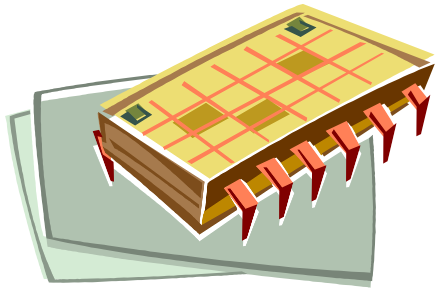
- There is some **overhead** required for maintaining virtual memory
  – Every time a page is swapped in or out, the process may be suspended and the page or segment tables need to be updated

- However, virtual memory has **proved to be a big gain in efficiency**.
  – The user is freed from the tight constraints of main memory.
  – Multiprogramming becomes more effective.

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

Real/virtual paging and segmentation

Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation
Main memory partitioned into small fixed-size chunks called frames		Main memory not partitioned	
Program broken into pages by the compiler or memory management system		Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)	
Internal fragmentation within frames		No internal fragmentation	
No external fragmentation		External fragmentation	
Operating system must maintain a page table for each process showing which frame each page occupies		Operating system must maintain a segment table for each process showing the load address and length of each segment	
Operating system must maintain a free frame list		Operating system must maintain a list of free holes in main memory	
Processor uses page number, offset to calculate absolute address		Processor uses segment number, offset to calculate absolute address	
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory for the process to run. Segments may be read in as needed
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk

Thrashing

A state in which the system spends most of its time swapping process pieces rather than executing instructions

To avoid this, the operating system tries to guess, based on recent history, which pieces are least likely to be used in the near future

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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
- Avoids thrashing

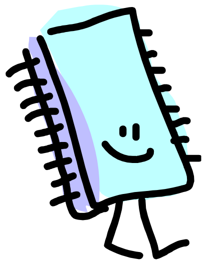
Support needed for virtual memory

For virtual memory to be practical and effective:

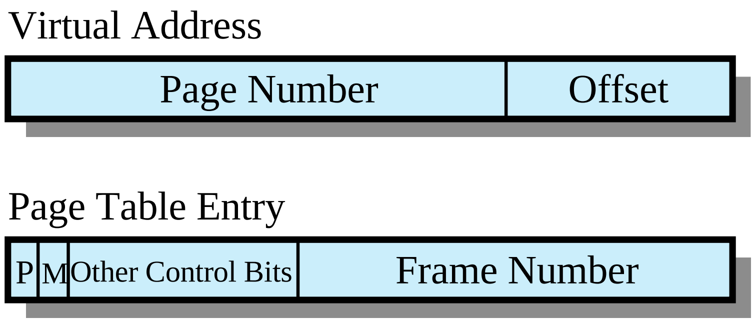
- Hardware must support paging and segmentation
- Operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory

Paging

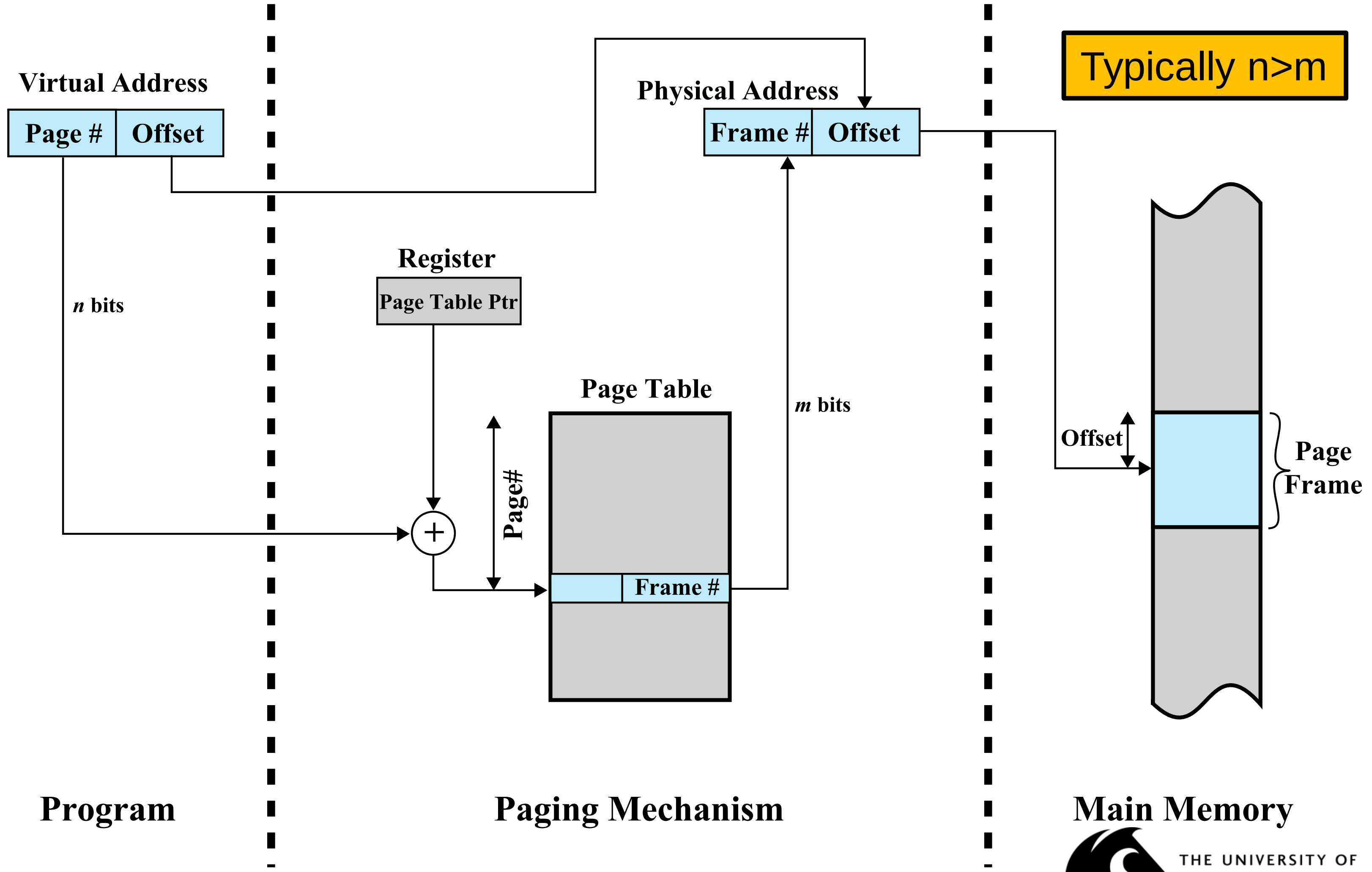
- The term *virtual memory* is usually associated with systems that employ paging
- Use of paging to achieve virtual memory was first reported for the Atlas computer
- Each process has its own page table
  - each page table entry (PTE) contains the frame number of the corresponding page in main memory



Memory management formats



Address translation



Address translation

- 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.

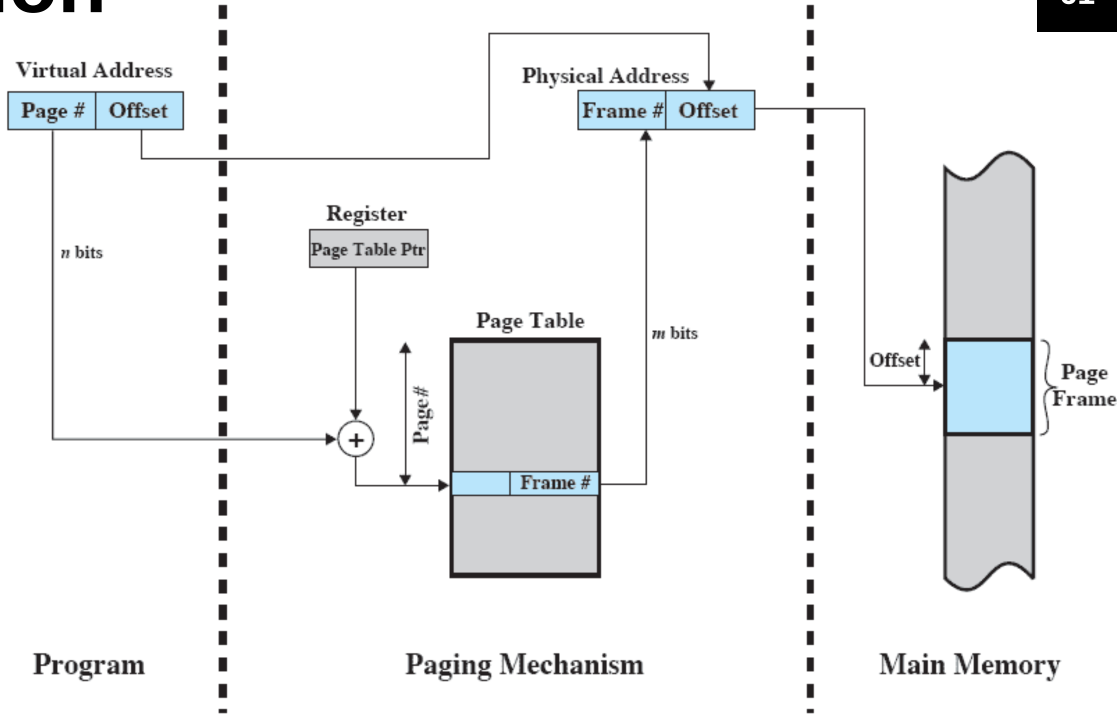


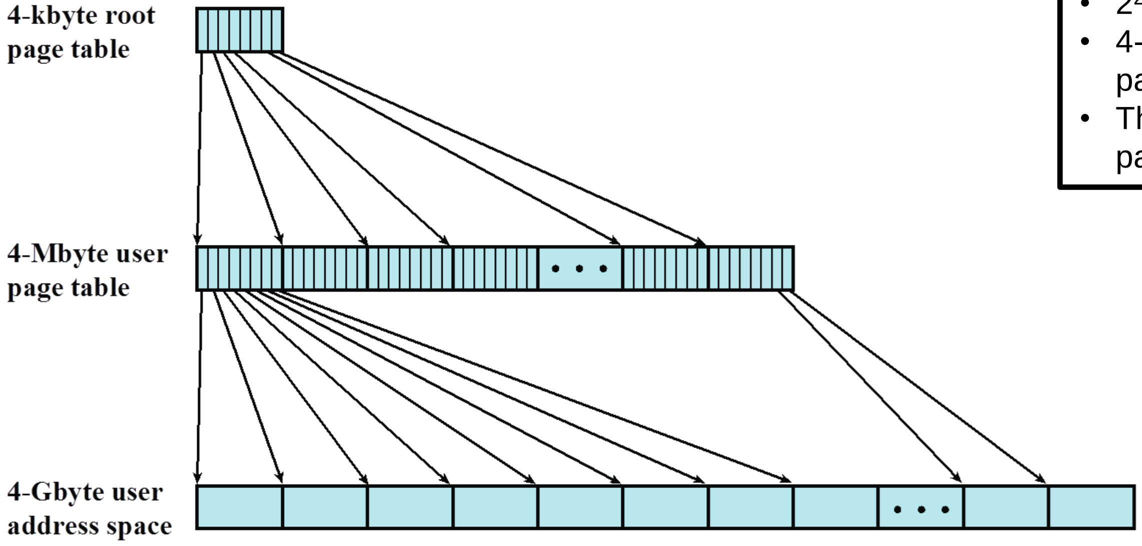
Figure 8.3 Address Translation in a Paging System

- 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).

Two level hierarchical page table

- A process can occupy huge amount of virtual memory
  - In VAX each process can be upto 2^{31} = 2 Gbytes of virtual memory
- If 2^9=512 byte size page is used then 2^{22} page table entries!
- Page tables are subject to paging..

- 32 bit address
- 2^{20} pages of size 4-Kbyte (2^{12})
- 2^{20} entries in the page table
- 4-byte/ entry means the size of the page table is 4 Mbyte (2^{22})
- The page table itself needs 2^{10} pages (4-Kbte page size)



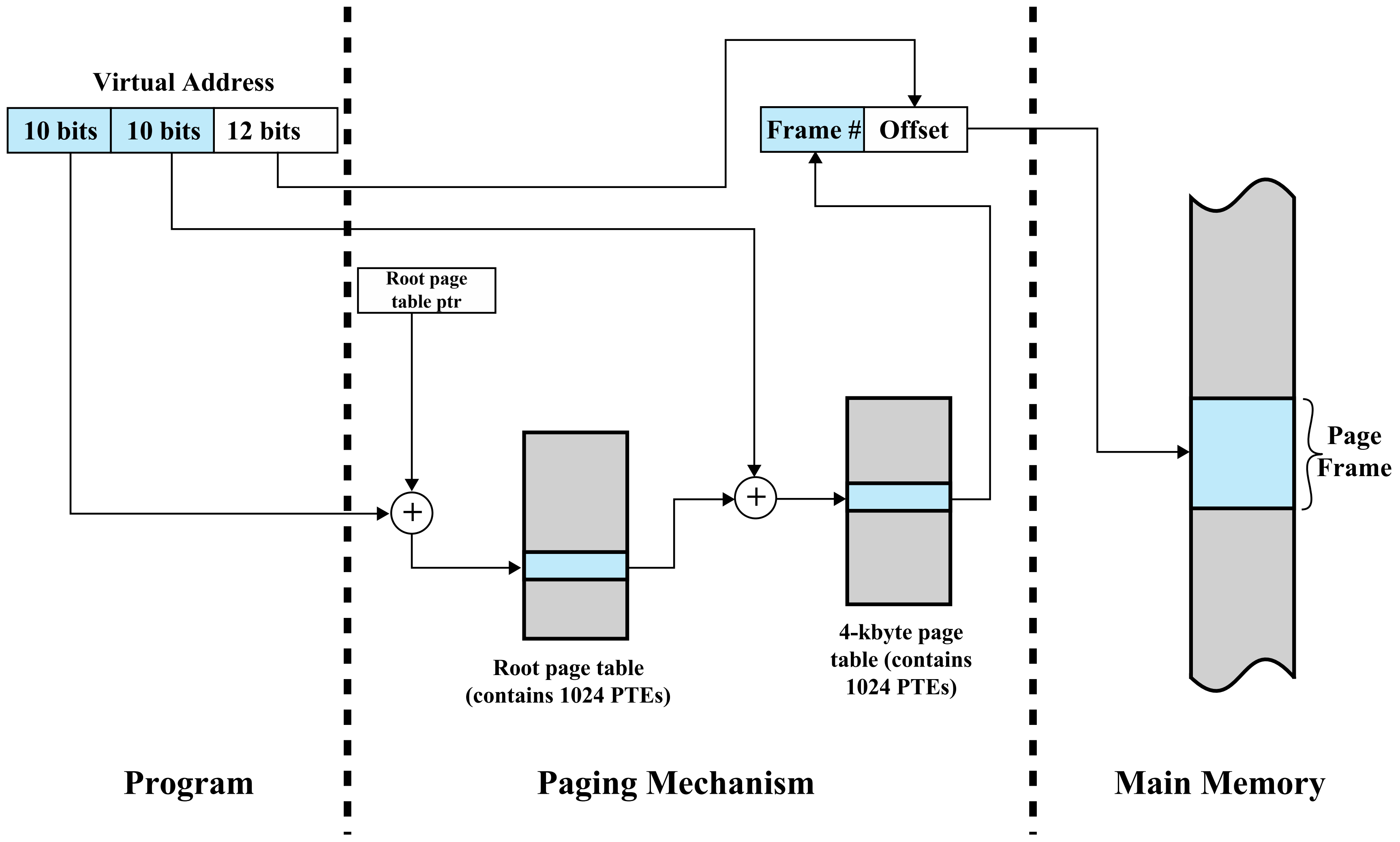


Figure 8.4 Address Translation in a Two-Level Paging System

Inverted page table

- A drawback of previous page tables is that their size is proportional to that of the virtual address space
- An alternative is for the page number portion of a virtual address to be 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 or virtual pages supported
- Structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number

Inverted page table

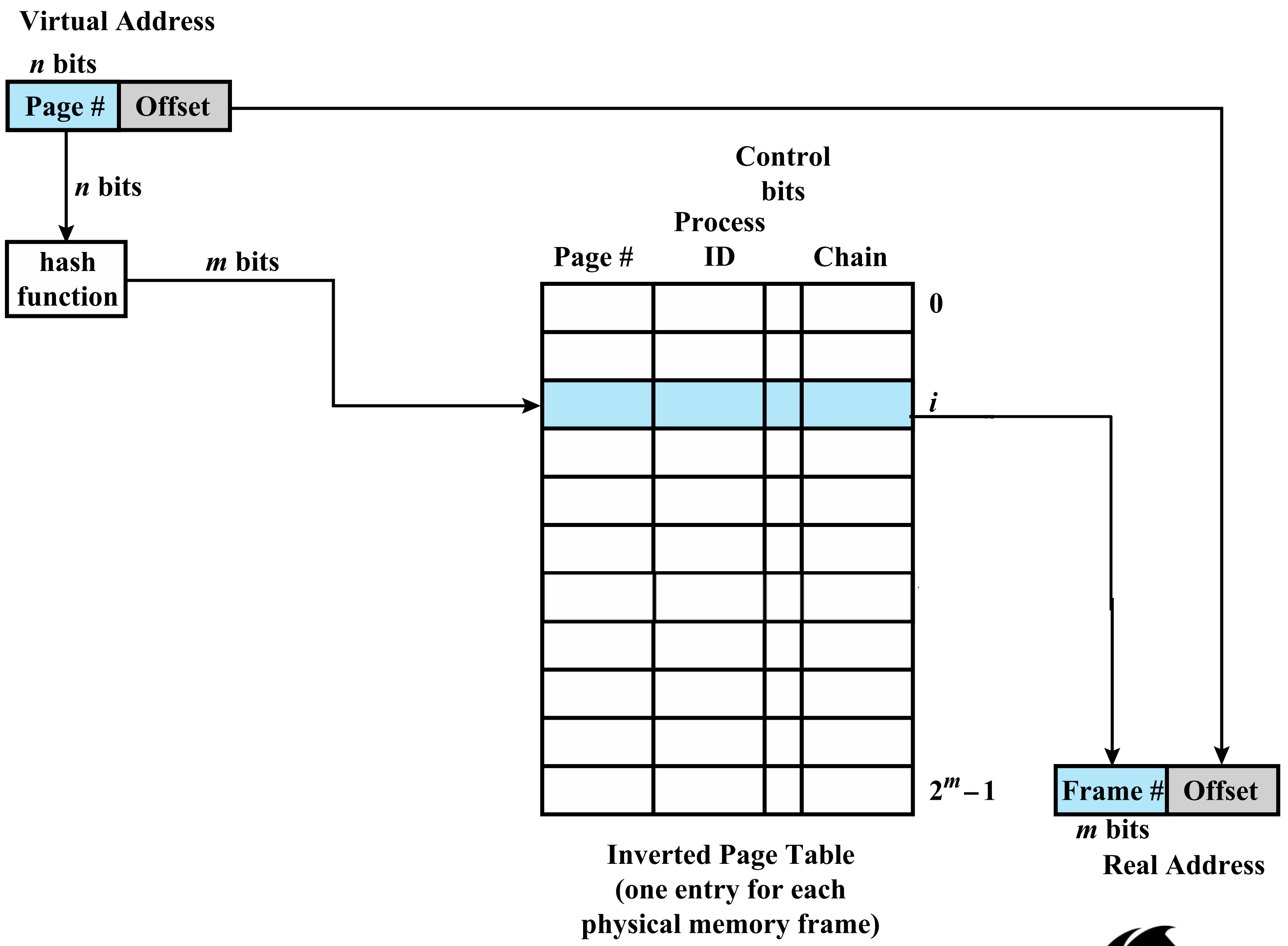


Figure 8.5 Inverted Page Table Structure

Inverted page table

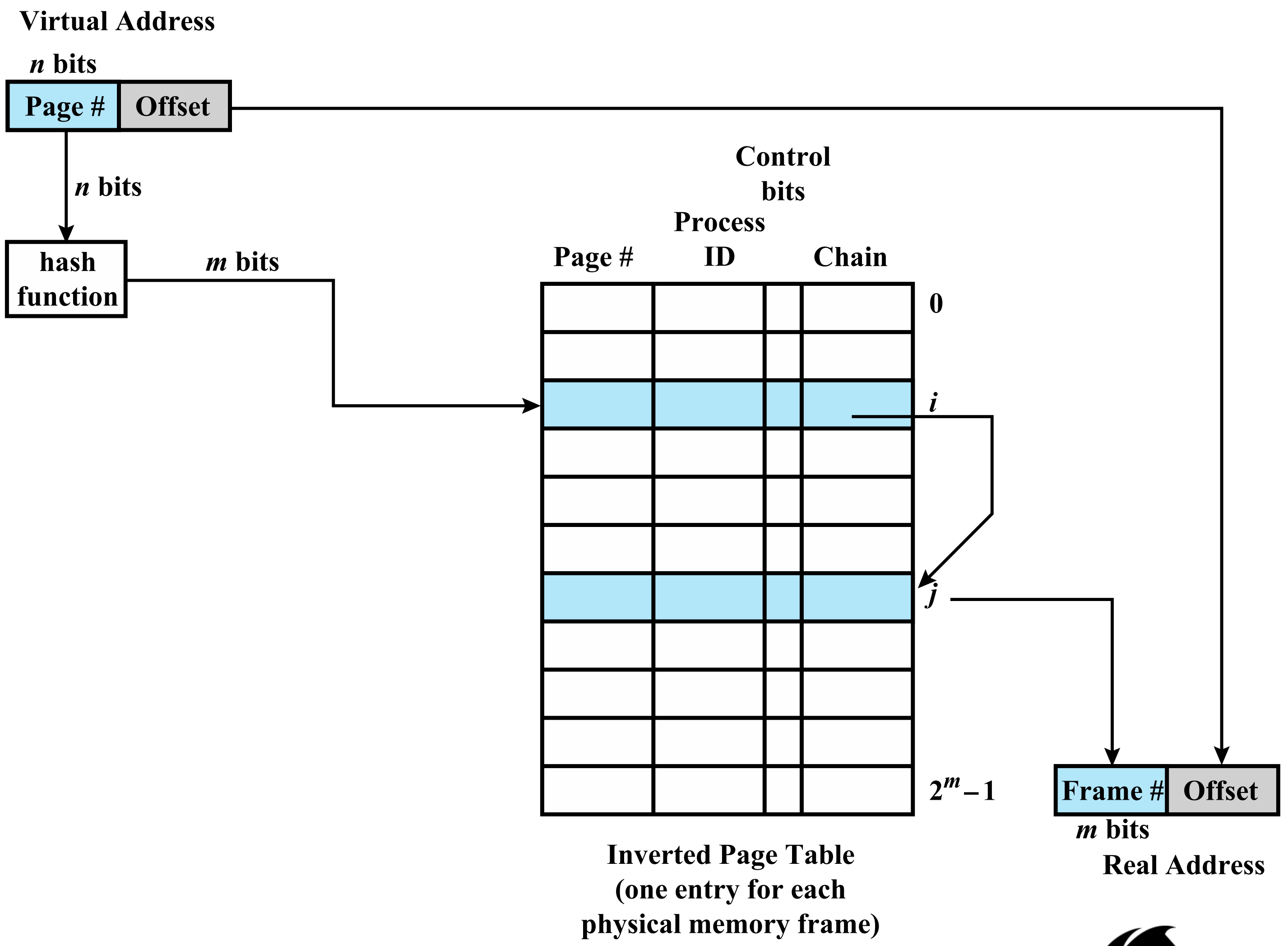


Figure 8.5 Inverted Page Table Structure

Inverted page table

- Each entry in the page table includes:

Page number

Process identifier
- the process that owns this page

Control bits
- includes flags and protection and locking information

Chain pointer
- the index value of the next entry in the chain

Translation lookaside buffer (TLB)

- Each virtual memory reference can cause two physical memory accesses:
  - One to fetch the page table entry
  - One to fetch the data

- To overcome the effect of doubling the memory access time, most virtual memory schemes make use of a special high-speed cache called a *translation lookaside* buffer for page table entries
  - This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used.

Use of a TLB

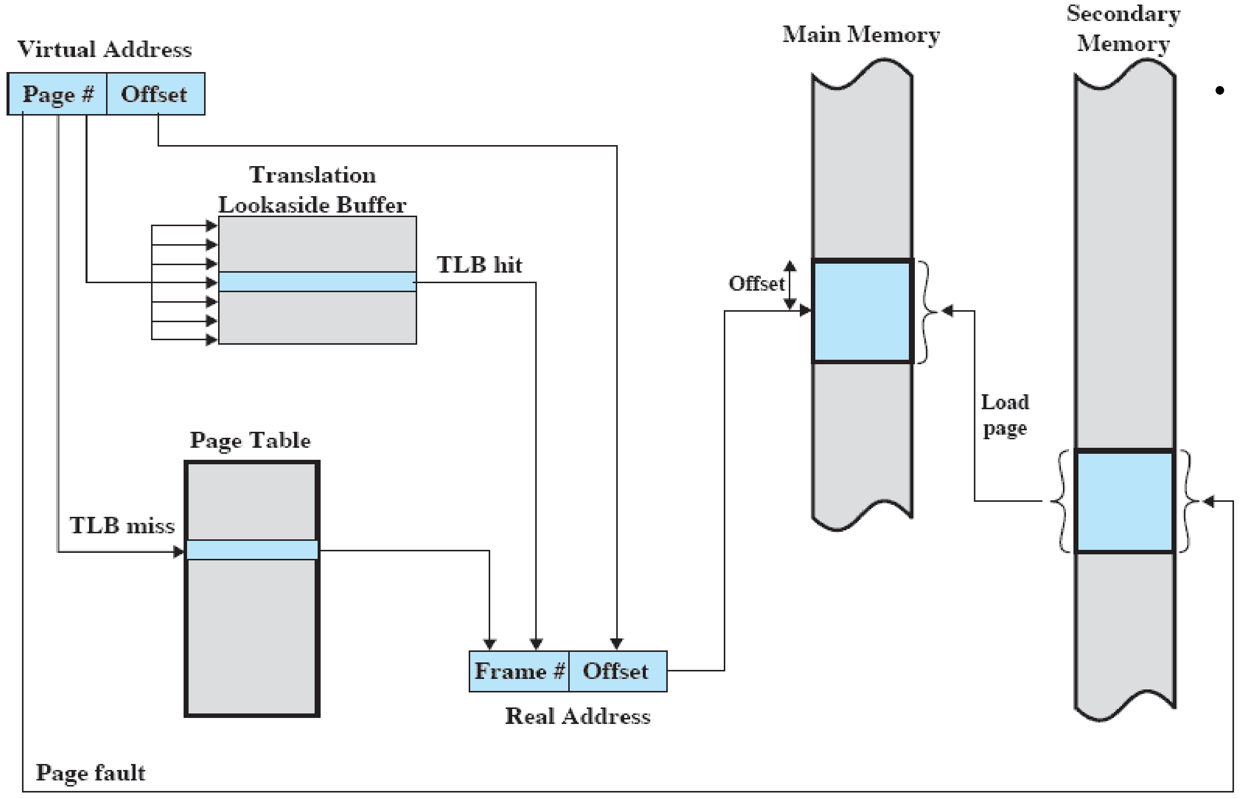


Figure 8.7 Use of a Translation Lookaside Buffer

- Given a virtual address, the processor will first examines the TLB.
- If the desired page table entry is present (**TLB hit**), then the frame number is retrieved and the real address is formed.
- If the desired page table entry is not found (**TLB miss**), then the processor uses the page number to index the process page table and examine the corresponding page table entry.
  - If the "present bit" is set, then the page is in main memory, and the processor can retrieve the frame number from the page table entry to form the real address.
  - The processor also updates the TLB to include this new page table entry.
  - If the "present" bit is not set, then the desired page is not in main memory and a memory access fault, called a **page fault** , is issued. At this point, we leave the realm of hardware and invoke the operating system, which loads the needed page and updates the page table.

TLB operation

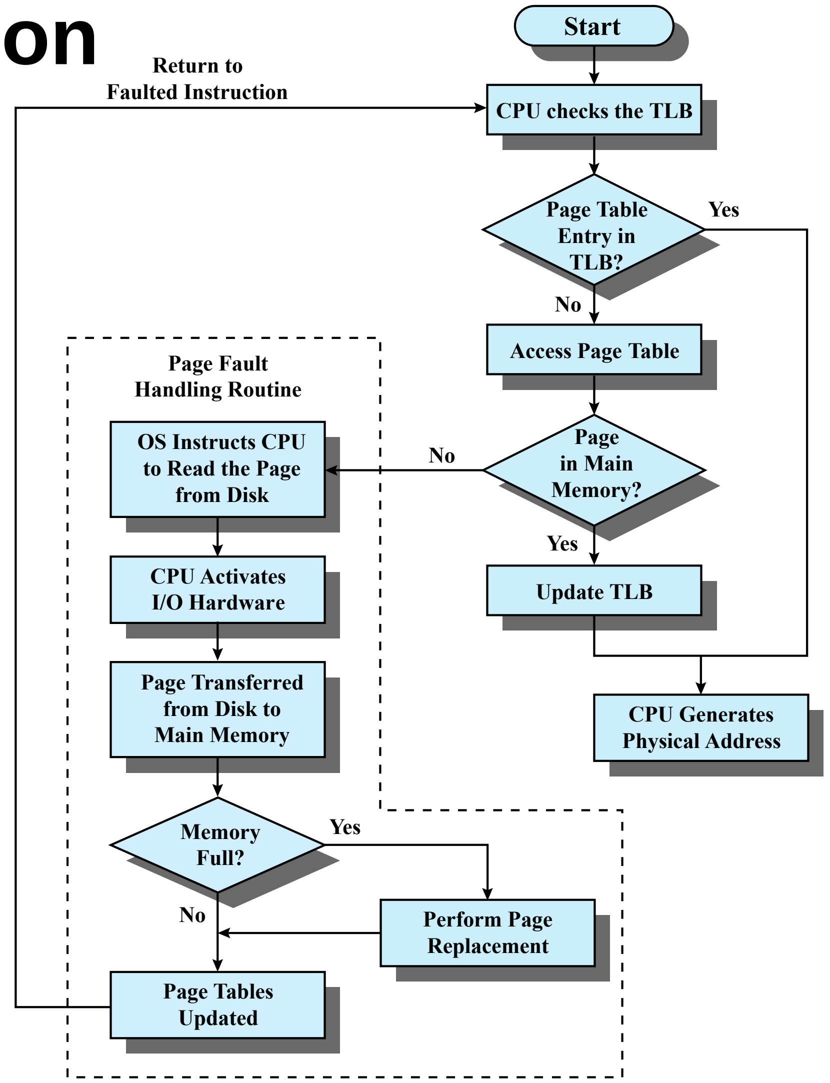
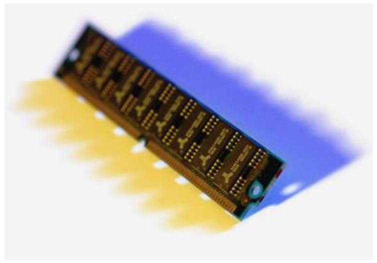

Start
CPU checks the TLB
Return to Faulted Instruction
Page Table Entry in TLB?
Yes
No
Access Page Table
Page in Main Memory?
No
Yes
Update TLB
CPU Generates Physical Address
Page Fault Handling Routine
OS Instructs CPU to Read the Page from Disk
CPU Activates I/O Hardware
Page Transferred from Disk to Main Memory
Memory Full?
Yes
No
Perform Page Replacement
Page Tables Updated

Figure 8.7 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

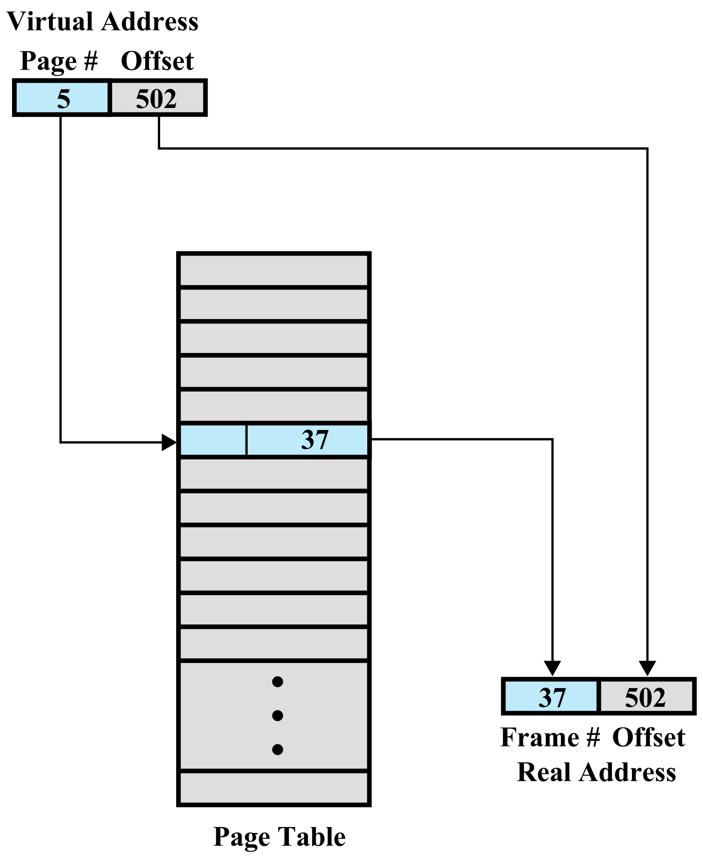
Associative mapping

- The TLB only contains some of the page table entries so we cannot simply index into the TLB based on page number
  - Each TLB entry must include the page number as well as the complete page table entry

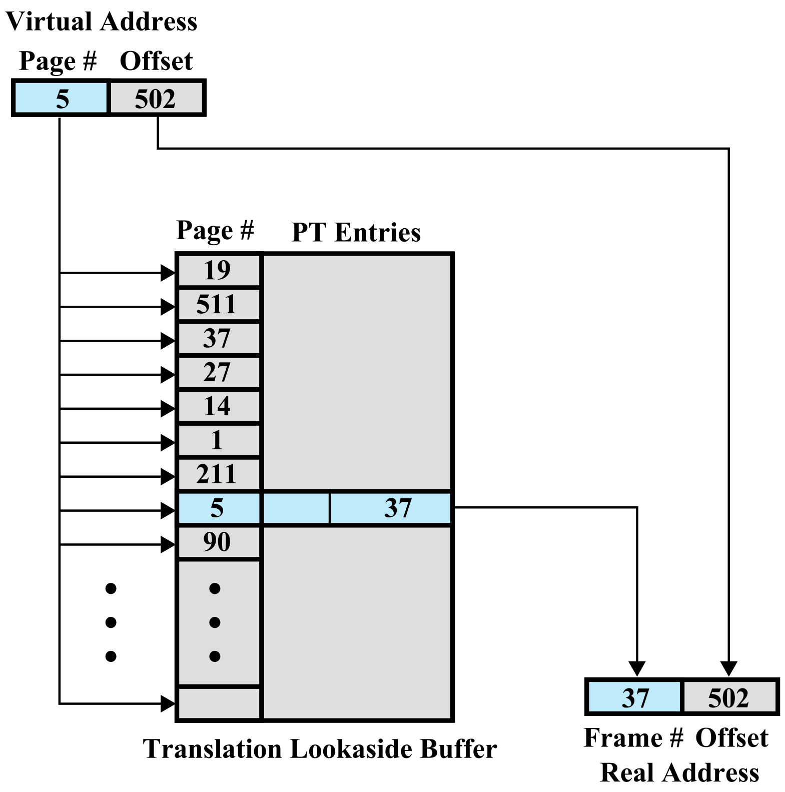
- The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number



Direct vs. associative lookup



(a) Direct mapping



(b) Associative mapping

TLB and memory cache

- The virtual memory mechanism must interact with the cache system
  - i.e. 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.

TLB and cache

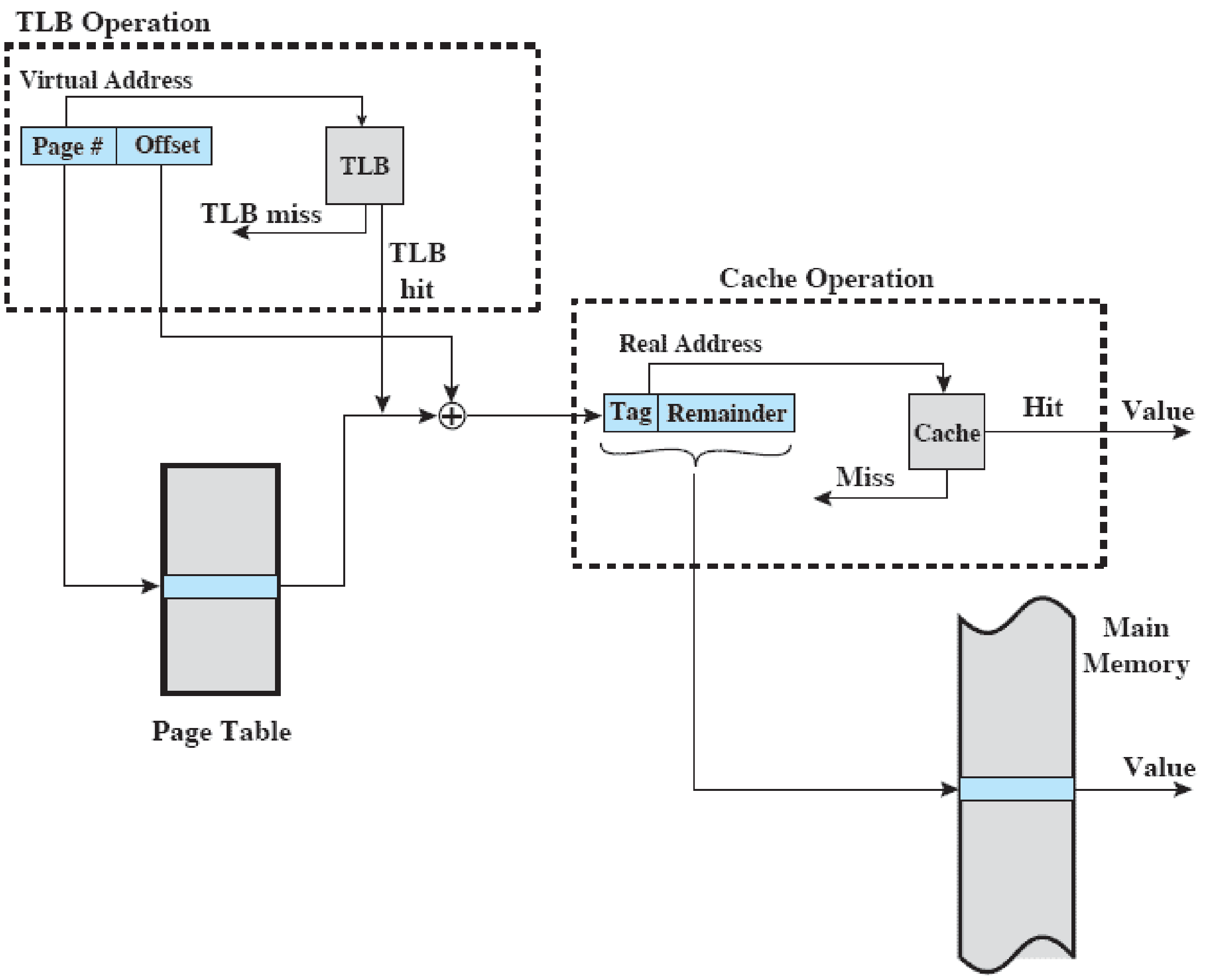
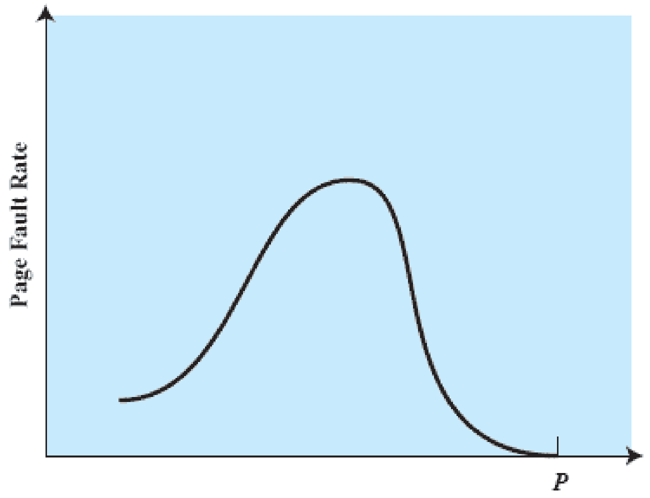


Figure 8.10 Translation Lookaside Buffer and Cache Operation

Page size

- An important hardware design decision is the size of page to be used
- Smaller page sizes give
  - Less internal fragmentation
  - More pages per process – larger page tables
    - Significance: part of page table is not in main memory – may be double page fault for a single memory reference
  - Closer modelling of locality of reference, and thus fewer page faults.
- Larger page size means
  - Fewer pages per process
  - Thus smaller page tables; this avoids double faults
  - Locality of reference weakens and page fault rate increases
  - When a process will fit into a page, thus fewer page faults.

Paging behaviour of a program



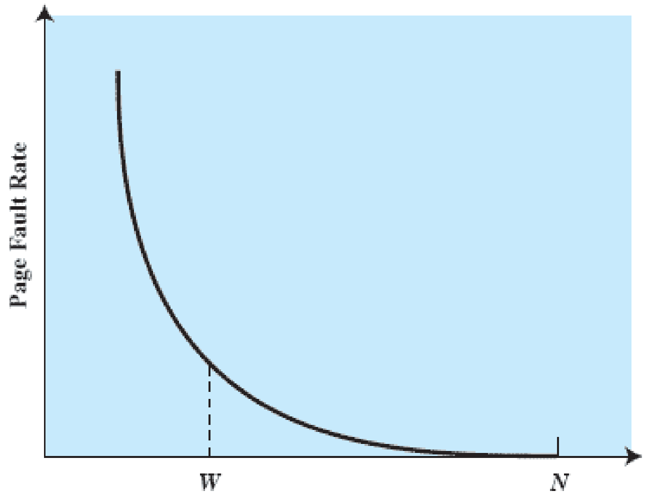
(a) Page Size

P = size of entire process
W = working set size
N = total number of pages in process

- This behaviour 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.

Paging behaviour of a program

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



(b) Number of Page Frames Allocated

P = size of entire process
W = working set size
N = total number of pages in process

Segmentation

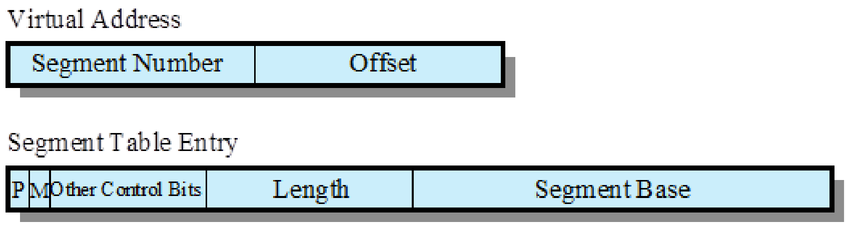
- Segmentation differs from paging in two aspects:
    1. The segments are of **variable length**, pages are fixed size.
    2. Segmentation may be under **programmer control**, paging is invisible to the programmer.

- These two differences lead to some **advantages** of segmented over non-segmented memory are:
    - Data structures of **unpredictable size** can be handled.
    - **Program structure** can be organised around segments, e,g., simplifying separate compilation.
    - Segments can be **shared** among processes.
    - Programs can be organised into segments of the **same protection level**. This is more powerful, and easier to control than with paging.

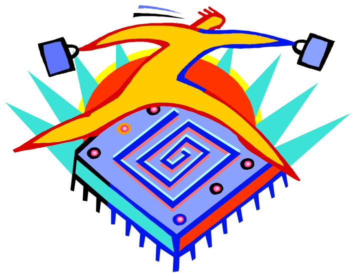
- Structures to control segmentation are very similar to those controlling paging:
- A **segment table** is like a page table with an extra field for the size of the segment

Segment organisation

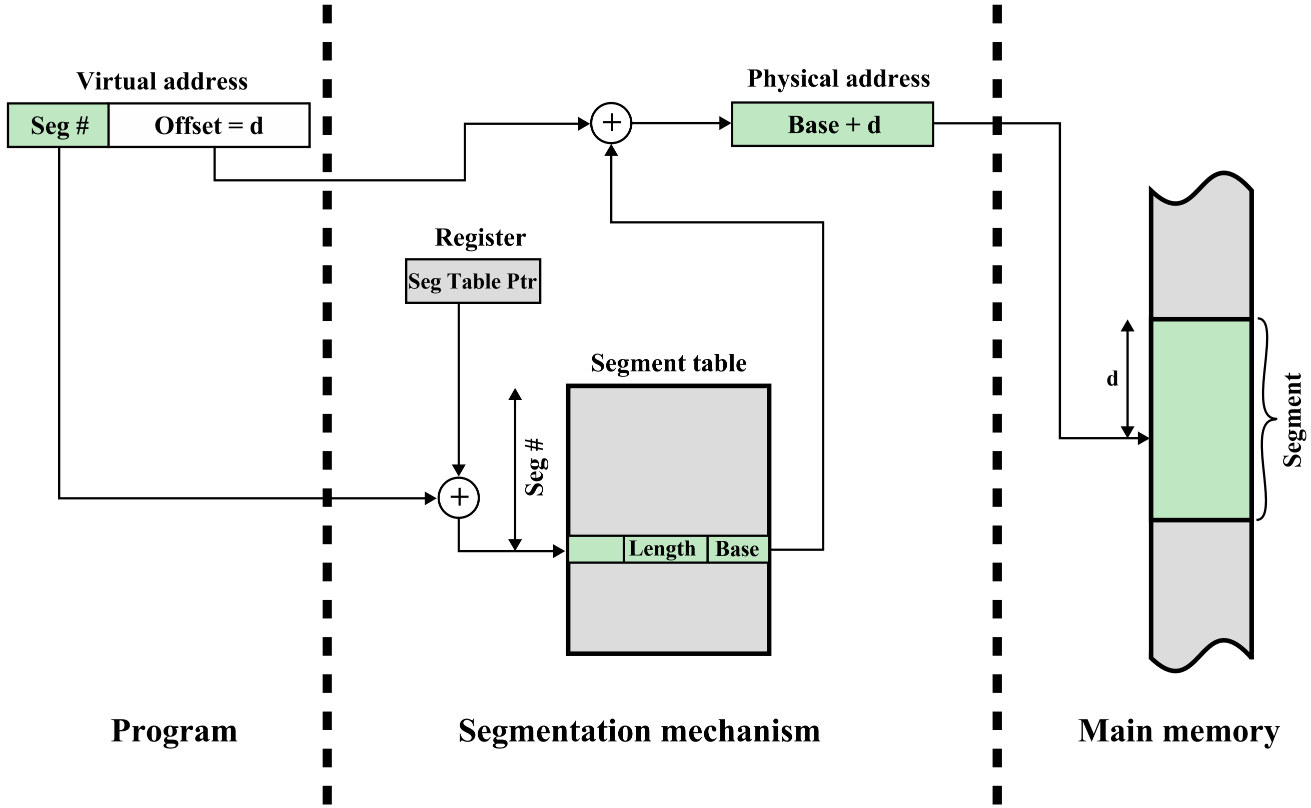
- Each segment table entry contains the starting address of the corresponding segment in main memory and the length of the segment
- A bit is needed to determine if the segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main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



(b) Segmentation only



Address translation

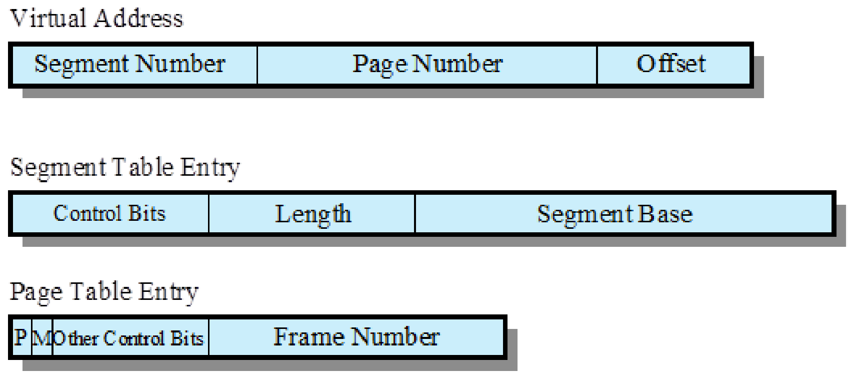


The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of segment number and offset, into a physical address, using a segment table.

Combining paging and segmentation

In a combined paging/segmentation system a user's address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame

- Segmentation is visible to the programmer
- Paging is transparent to the programmer



P= present bit
M = Modified bit

(c) Combined segmentation and paging

Address translation

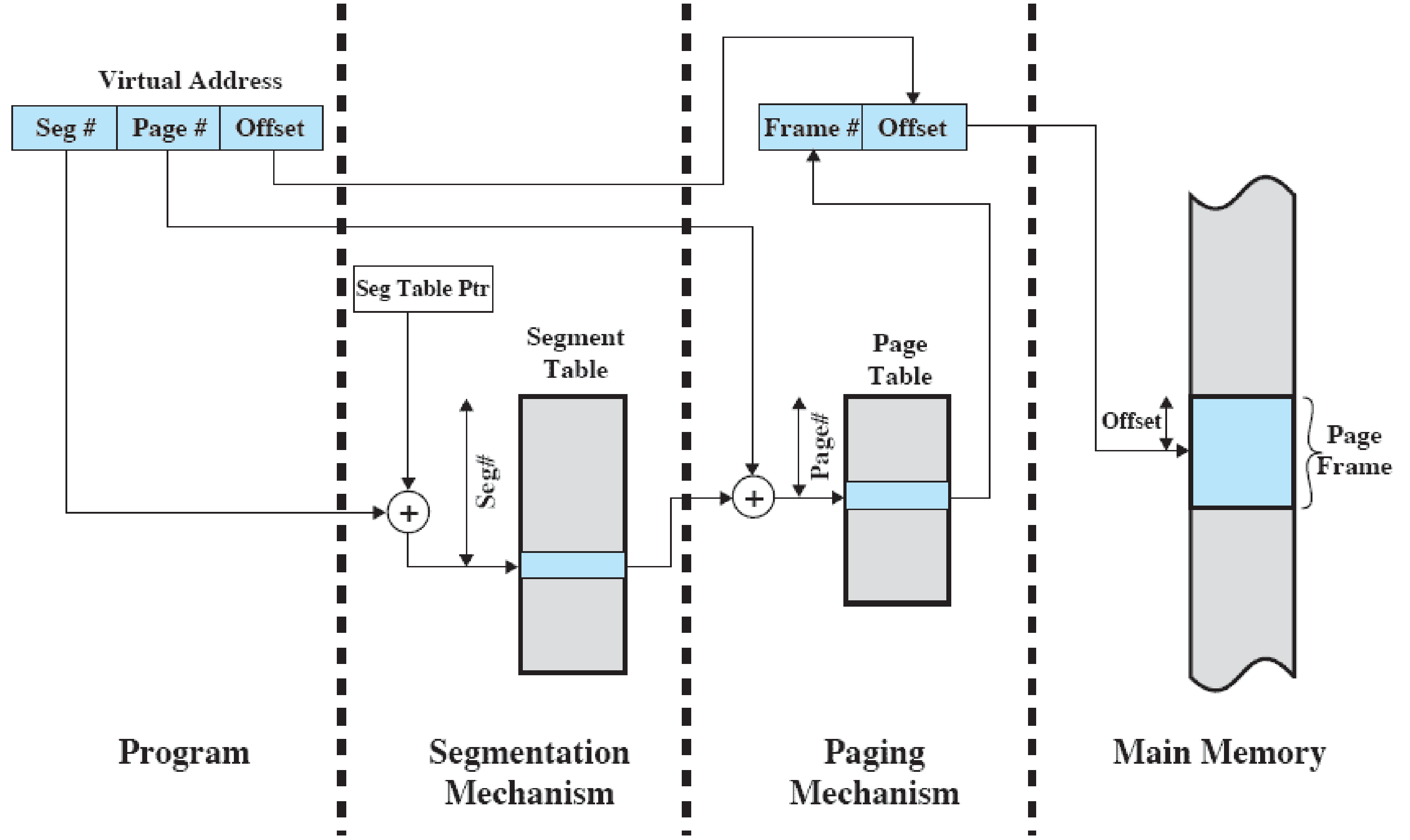
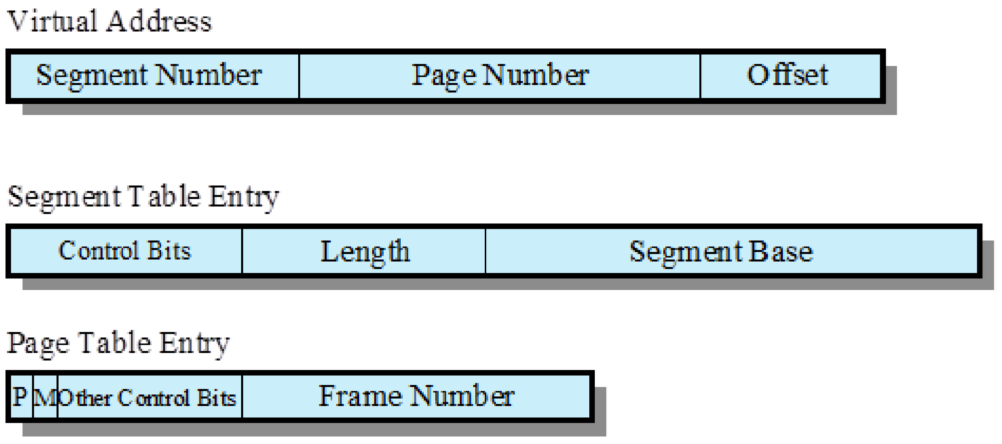


Figure 8.13 Address Translation in a Segmentation/Paging System

Combining segmentation and paging

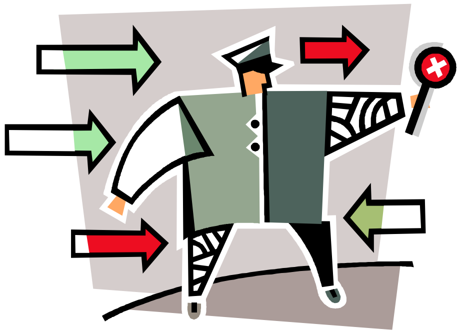


P= present bit
M = Modified bit

(c) Combined segmentation and paging

Protection and sharing

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



Protection relationships

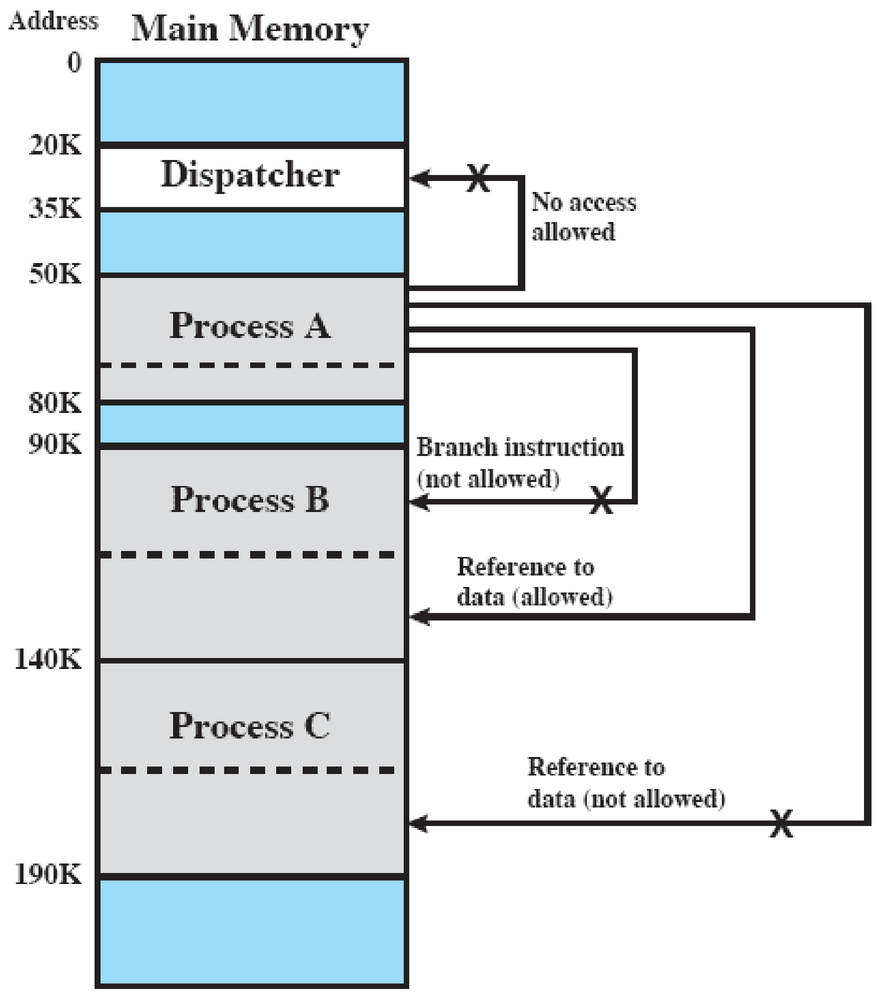


Figure 8.14 Protection Relationships Between Segments

Summary

- Memory management requirements are: relocation, protection, sharing, logical organization and physical organization
- Memory partitioning can be done using
  - Fixed Partitioning (equal sized / unequal sized)
  - Dynamic Partitioning
- In simple paging, main memory is divided into many small equal-sized frames.
- Each process is divided into frame-size pages.
  - Smaller processes require fewer pages; larger processes require more.
- When a process is brought in, all of its pages are loaded into available frames (might not be contiguous frames) and a page table is setup.

Summary

- It is desirable
  - to maintain as many processes in main memory as possible.
  - to free programs from size restrictions in program development
- VM is the way to address both of the above concern
- In VB all address references are logical references that are translated in runtime to physical address
- VM also allows a process to be broken up into pieces and
  - the pieces need not to be contiguously located in main memory
  - It is not even necessary for all of the pieces of the process to be in main memory during execution

Summary

- Two basic approaches to providing VM are
  - Paging
    - Process is divided into relatively small, fixed size blocks called pages
  - Segmentation
    - Segmentation provides for the use of pieces of varying size
- It is possible to combine segmentation and paging in a single memory management scheme
- .

References

- **Operating Systems – Internal and Design Principles**
  - By William Stallings
- Chapter 7, 8