COMPS267F Chapter 7

Memory Management

Dr. Andrew Kwok-Fai LUI

Aim of this Chapter

- Discusses the techniques developed for memory management for multi-programming systems
- Memory plays important role in program execution
  - The main memory is directly addressable by CPU
  - The main memory will store multiple programs and make them ready for execution

Responsibilities of OS in Memory Management

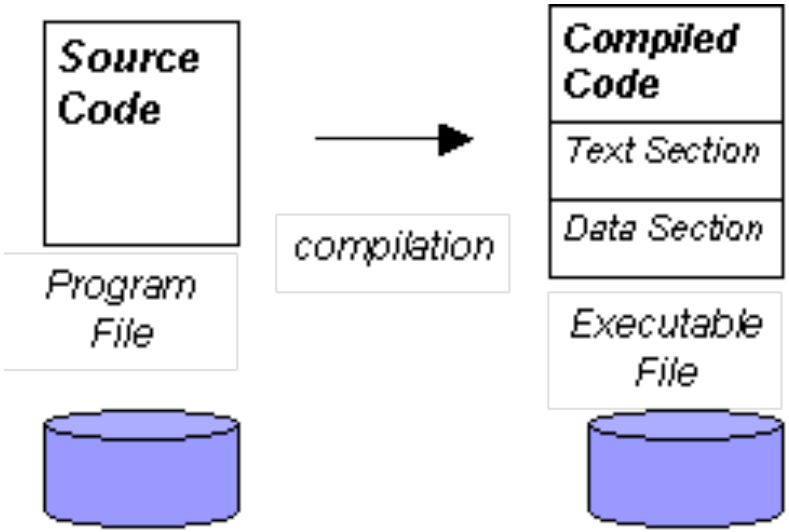
- OS responsibilities
  - Keep record of free memory regions
  - Keep record of ownership of memory allocation
  - Allocate and de-allocate memory
    - Process creation
    - Dynamic memory allocation
    - Find free space of enough size
  - Make space by swapping in and out of the memory space associated with processes
  - Protect memory regions from malicious memory alternation

Do All the Above Efficiently

review of program execution

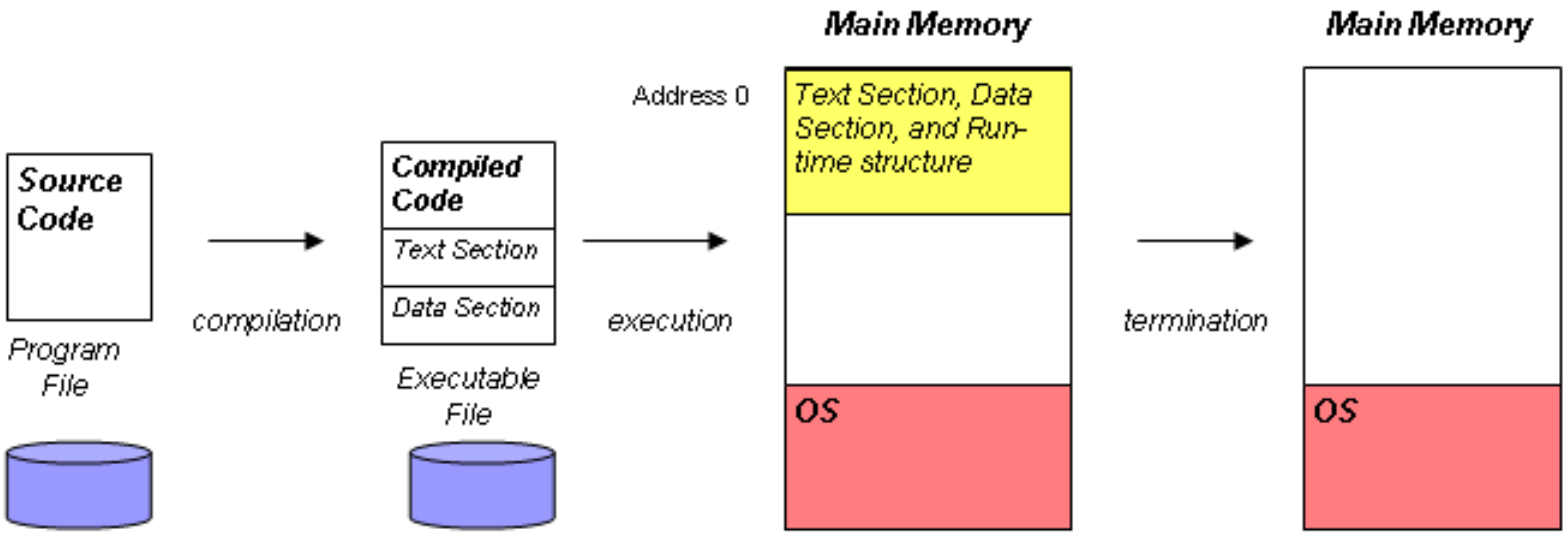
Program Execution

- Program source code and executable files are stored on secondary device
  - The OS must load them into the main memory for execution of instructions



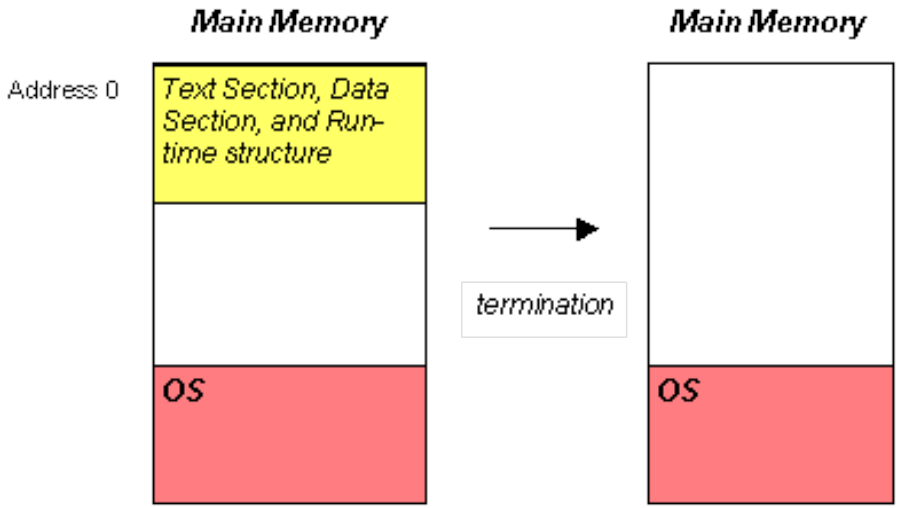
Program Execution

- Task of OS is easy if only one program is in execution at a time.
  - The OS runs in a cycle of loading-execution-termination



Address Zero

- Many instructions refer to addresses
  - LDA, STO, BR instructions in LMC
- The location of the program code in memory is important
  - Compiler often assumes that first address is zero



Example: Program Loading

Example: Loading of LMC Programs

The following is a simple LMC program that reads and prints the sums two numbers.

```
00 901; IN
01 310; STO 10
02 901; IN
03 110; ADD 10
04 902; OUT
05 000; COB
10 000; DAT
```

The program contains both instructions and constant data (address 10). The program assumes that the first instruction is at address 0. The program will be loaded into the main memory at address 0. The program will be executed correctly.

Question: Would the program run correctly if it were loaded to other addresses?

Example: Program Loading

Example: Loading of LMC Programs

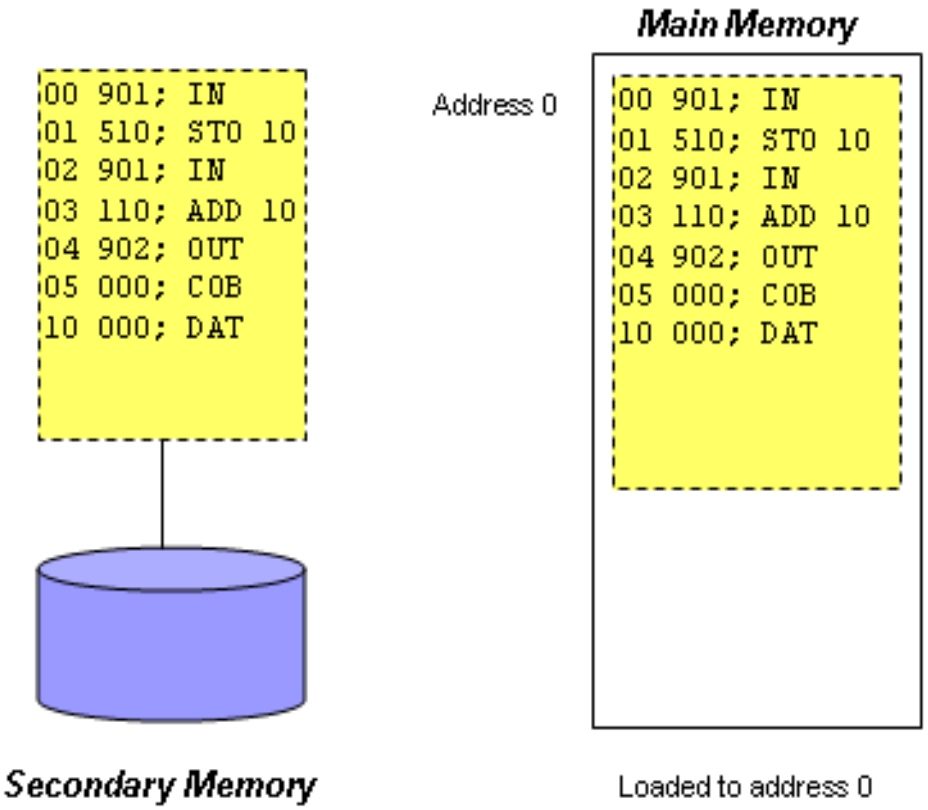
If the program were loaded to address 6, for example, then the program would not run correctly. The same instructions would now locate at different addresses.

06 901; IN
07 310; STO 10
08 901; IN
09 110; ADD 10
10 902; OUT
11 000; COB

The instruction at address 07 would store the input data to address 10. This would overwrite the instruction OUT now located at address 10.

The program could still run correctly if loaded to some addresses, for example, address 20. However, the operating systems have no easy to know whether a particular starting address would render the program erroneous. To be safe from error, the program should always be loaded to address 0.

Example: Program Loading

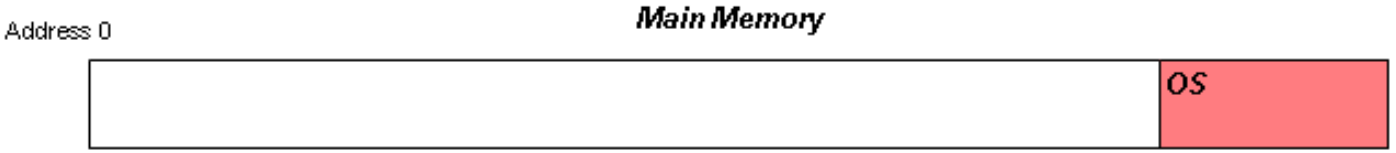


The Problem of Program Loading

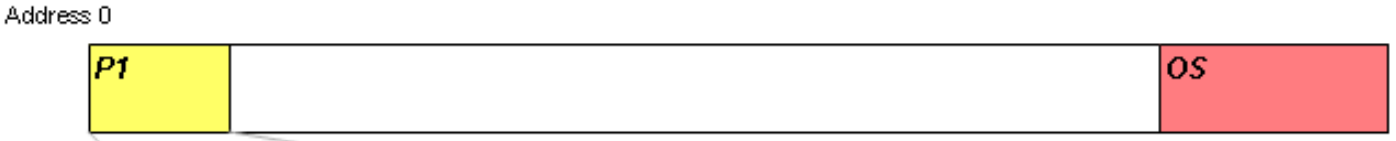
- Unfortunately only one address 0
  - Only one program can be loaded at address 0 at a time
  - Cannot make use of the whole memory space?
  - Cannot load more than one program at a time?
    - Cannot support multi-programming?
- To make use the whole memory space
  - Techniques to allow program loaded at any address and still runnable

Multi-programming and Main Memory

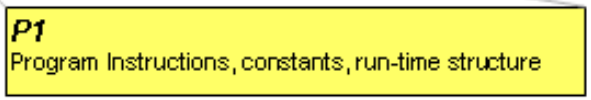
The whole main memory is free except the part already occupied by the OS



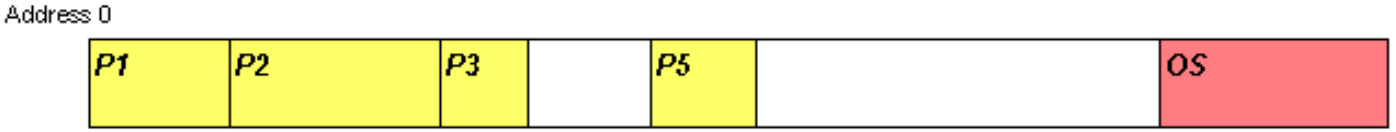
The first process is created and loaded to address 0



The size of P1 depends on the amount of program instructions and constant data



The other processes will occupy other free regions of the main memory

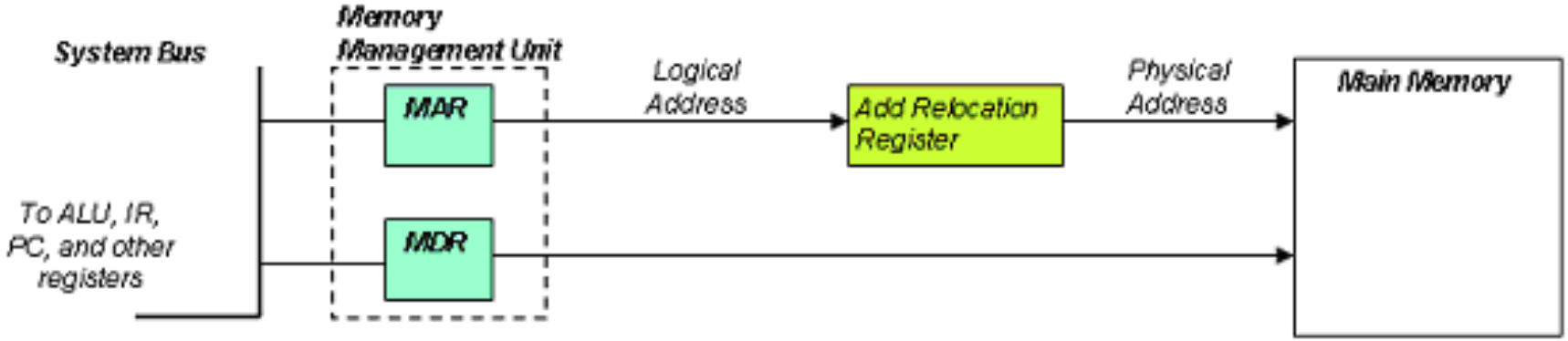


Multi-programming and Main Memory

- Address conversion is a solution
  - To allow programs running from any starting address
- The method
  - At compilation, all programs assumed to start at address 0
  - At execution, programs are loaded at any address A
  - Each reference to an address (e.g. X) in programs must be converted to reflect the new location of the process (A + X).

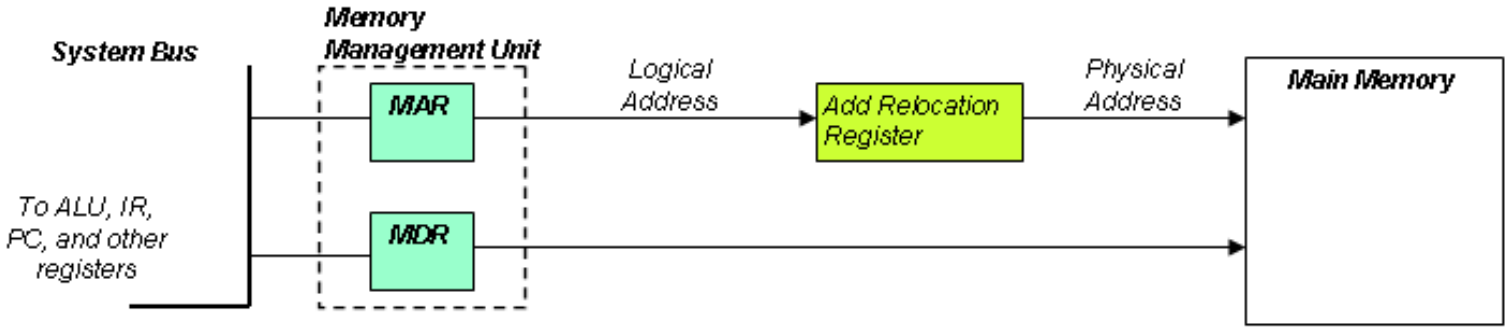
Relocation Register and MMU

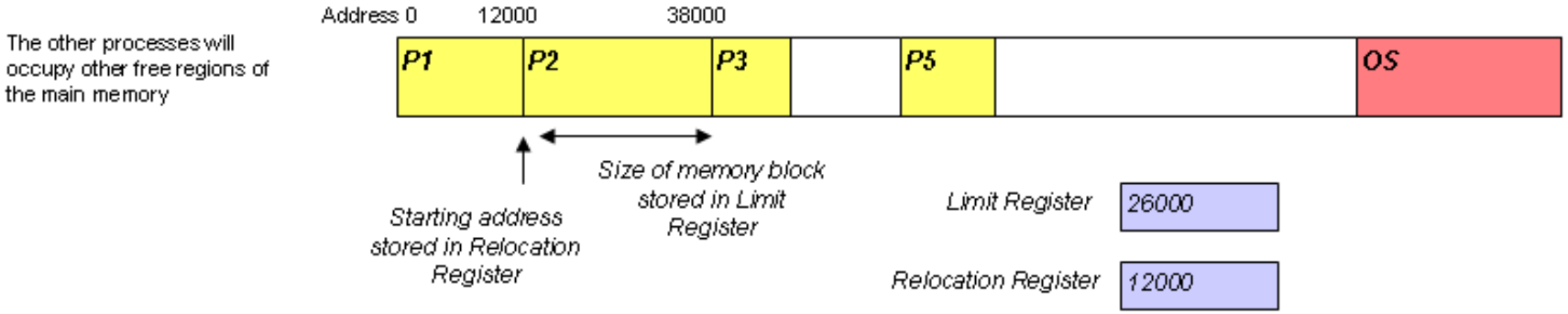
- Relocation register is built into the MMU connection to the main memory
  - Hardware implementation is faster
  - An addition circuitry is used to add MAR to Relocation Register



Relocation Register and MMU

- When a process is in execution, the OS loads the starting address into the Relocation Register
  - All subsequent memory accesses are mapped to the correct address



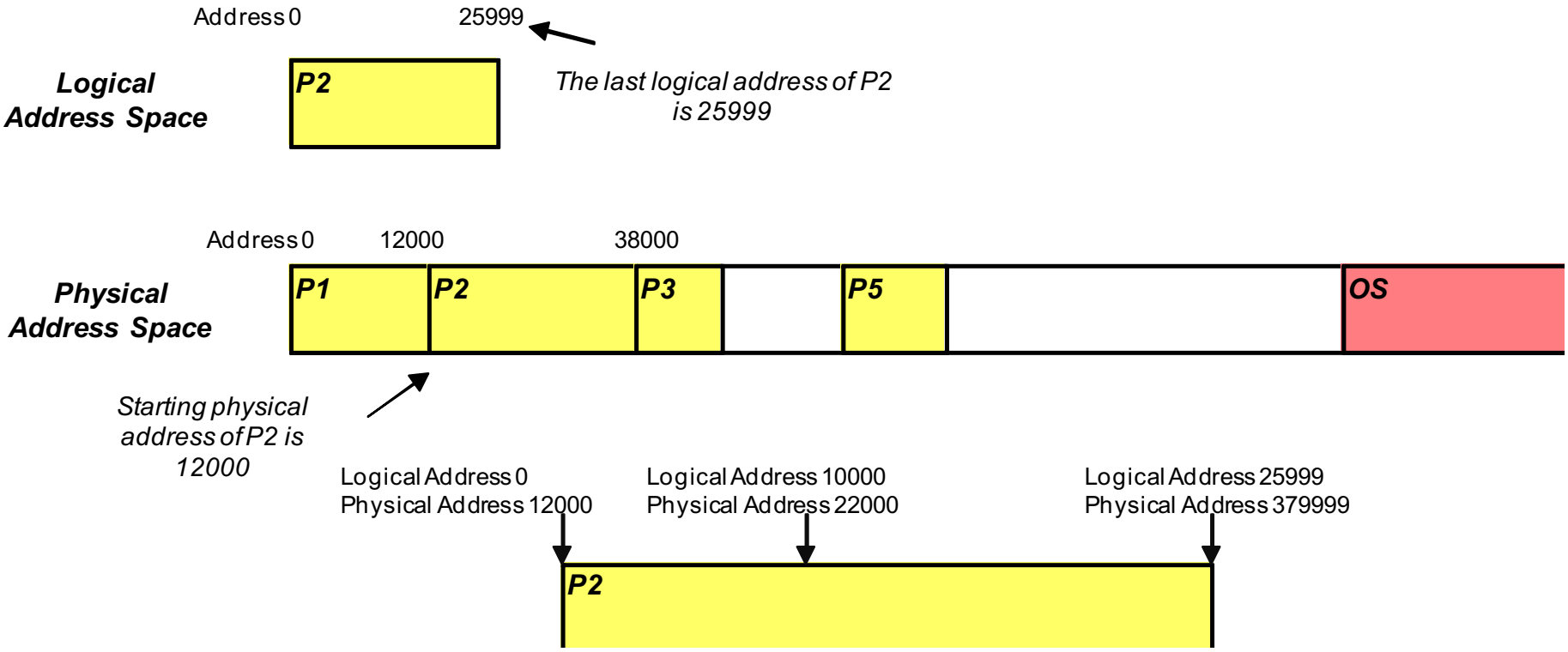


address binding

Logical Address and Physical Address

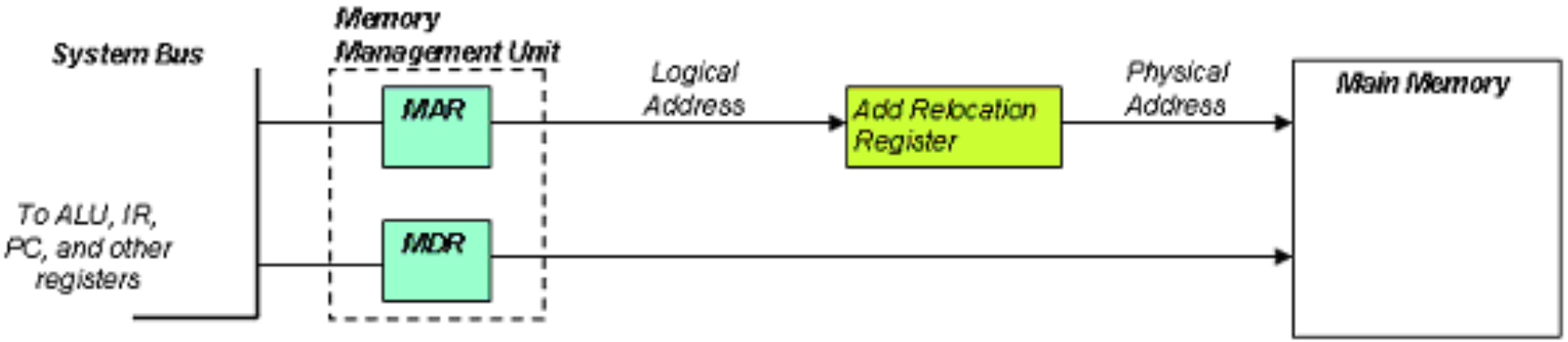
- Logical address: the address from the program's angle
  - The first address of the program starts 0
  - It does not change after compilation
- Physical address: the address where the program is loaded in the main memory
  - The program can actually be loaded at another place
  - It can be different from one execution to another

Logical Address and Physical Address



Logical Address and Physical Address

- Address binding: conversion from logical address to physical address
  - Logical address space
  - Physical address space
  - Binding is the conversion process



Example: Binding

Example: Logical and Physical addresses

Supposed that the program is loaded at address 6. A piece of memory space is allocated to the process from address 6 to address 12. The starting address 6 is stored in the relocation register.

```
06 901; IN
07 310; STO 10
08 901; IN
09 110; ADD 10
10 902; OUT
11 000; COB
```

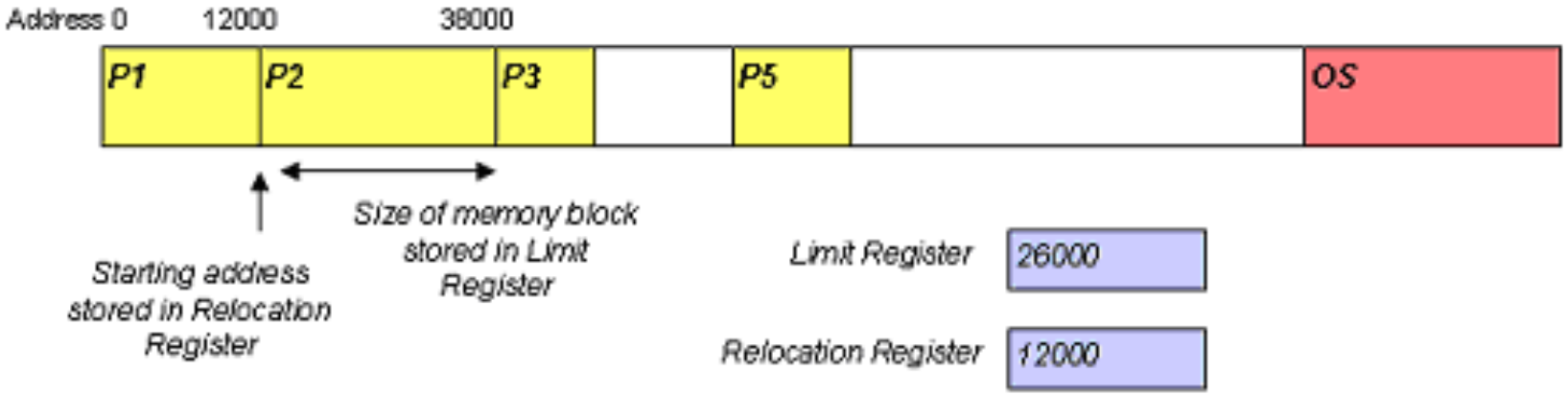
The execution of the instruction at 06 is simple. It involves no memory operation.

The execution of the instruction at 07 involves a memory operation. The address 10 is in the logical address space but this address should be at 10 + 6 = 16 in the physical memory space. At the execution phase, the address 10 is stored in the MAR. The memory management unit then uses the sum of MAR and the relocation register as the actual address to store. In the above program, the input value is stored at address 16.

The program should operate correctly.

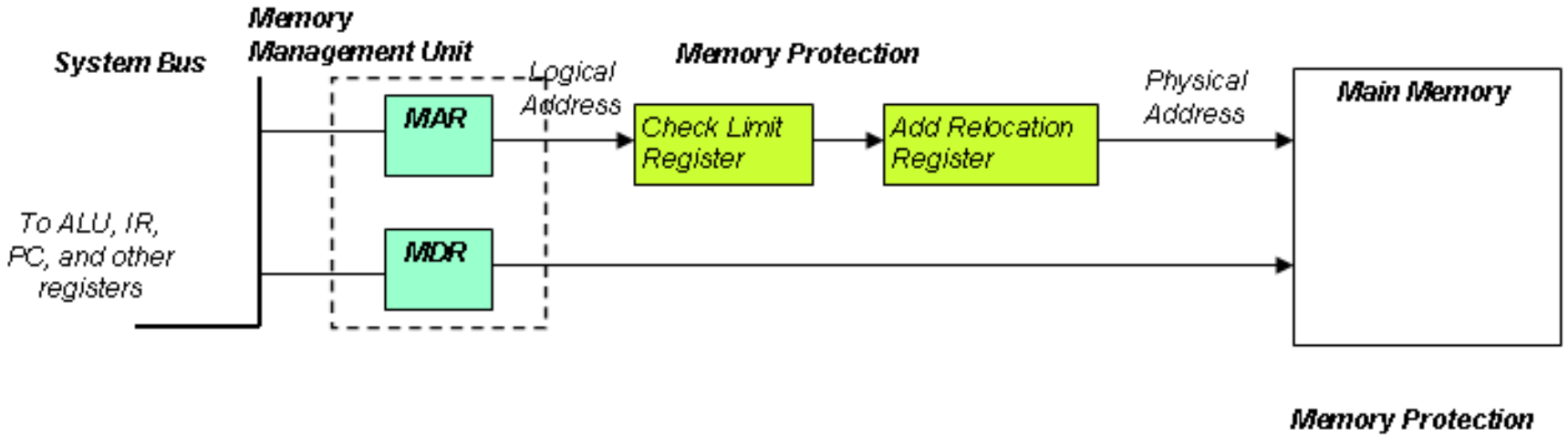
Memory Space Protection

- Memory address of a process must be restricted to the allocated part of the main memory
  - Limit Register: to control the upper-bound
  - Relocation Register: to indicate the lower-bound



Memory Space Protection

- Checking is done when a memory operation occurs
  - Hardware implementation is faster
  - Hazardous memory operation is aborted



Example: Memory Space Protection

Example: Memory Protection

The program contains a STO instruction that stores data to an address way beyond its scope. This might be due to careless programming or a malicious intention.

Supposed that the following program is loaded at address 20. A piece of memory space is allocated to the process from address 20 to address 23. So the relocation register contains 20 and the limit register contains 4.

20 901; IN

21 380; STO 80

22 902; OUT

23 000; COB

In the execution phase of the execution of STO 80, the address 80 is stored in MAR. It is first checked against the limit register. Address 80 is outside the allocated memory space. The operating systems forbid this operation and a system exception would be raised.

Learning Objective @ Chapter 7

- Describe common algorithms for memory space management and allocation
- Analyze the characteristics of the algorithms

Memory Space Management and Allocation

- Performance metrics concerning memory management
  - Level of multi-programming
    - Number of processes that can be loaded into the main memory
  - Utilization rate of main memory
    - Is all memory space usable?
    - Any free space not usable?
  - Efficiency of Data structures used in memory management
    - Keeping of records needs data structures
    - More management results in complex data structures, more time consuming

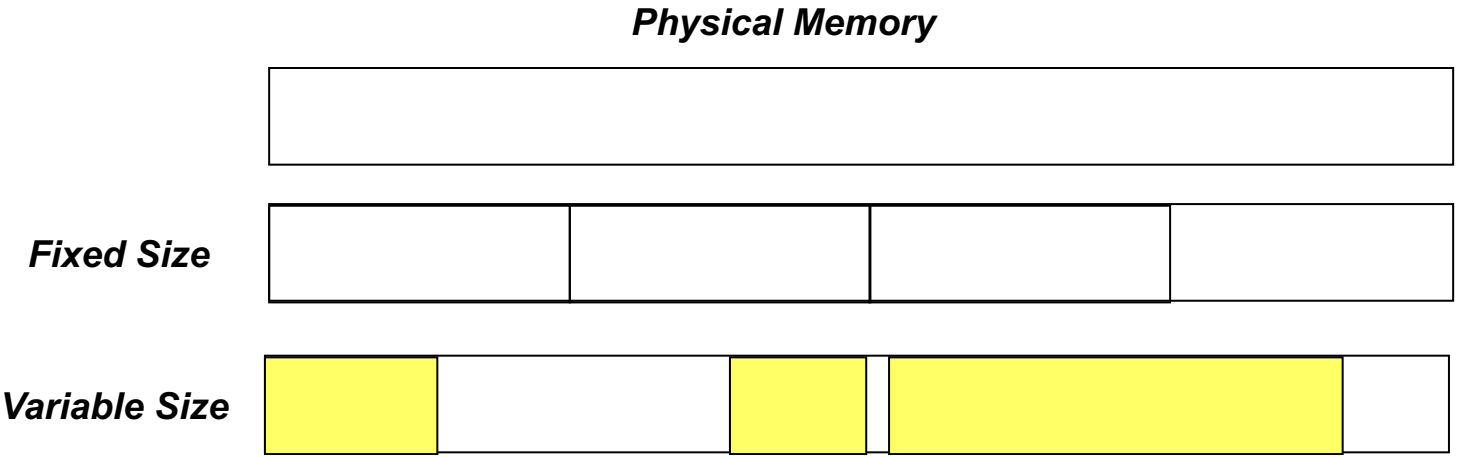
Memory Space Management and Allocation

- Methods of allocating memory space to processes
  - Physical memory space management perspective
    - Partitioning space into fixed size blocks
    - Partitioning space into variable size blocks
  - Process allocation perspective
    - Contiguous allocation: a process' memory is made up of one-piece
    - Non-contiguous allocation: a process' memory is made up of multiple blocks

	Contiguous Allocation	Non-Contiguous Allocation
Fixed Size Partition		
Variable Size Partition		

Memory Space Management and Allocation

	Contiguous Allocation	Non-Contiguous Allocation
Fixed Size Partition		
Variable Size Partition		



Memory Space Management and Allocation

	Contiguous Allocation	Non-Contiguous Allocation
Fixed Size Partition		
Variable Size Partition		

Logical Memory

Contiguous Allocation

P1
Program Instructions, constants, run-time structure

Non-Contiguous Allocation

P1 ***P1*** ***P1***

memory space management by contiguous allocations

	Contiguous Allocation	Non-Contiguous
Fixed Size Partition		
Variable Size Partition		

Process with Contiguous Memory Allocation

- A simple way to allocate memory for a process is contiguous block of memory
  - Everything is packed together
  - An alternative is to split up various parts to different parts in the main memory
    - Advantages of flexibility

Approaches of Memory Management

- Consider a process is a contiguous block of memory, now consider how to manage the physical memory space
- Two approaches of how memory space is allocated to support a process
  - Fixed Size Partition
  - Variable Size Partition

Fixed Size Partitions

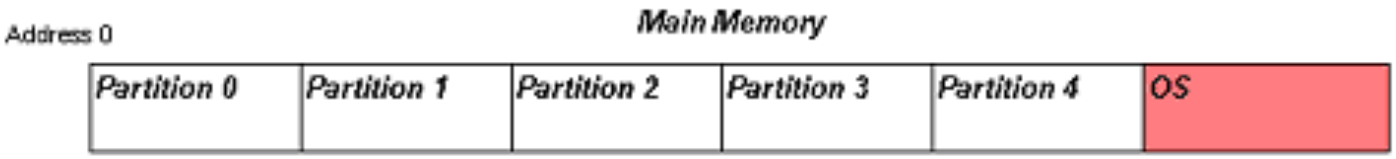
- The physical memory space is split up into a number of fixed size partitions
  - One partition for one process
  - The size of partition is determined by system designers
    - Too small can be insufficient for most processes
    - Too large can waste memory space
  - Data structure support is simple
    - An array indicating whether a partition is allocated or free
  - Memory wastage can be great
    - Small memory requirement processes are still allocated a large partition

Approaches of Memory Management

	Contiguous Allocation	Non-Contiguous
Fixed Size Partition		
Variable Size Partition		

Fixed Size Partitions

The main memory is divided up into fixed size memory partitions



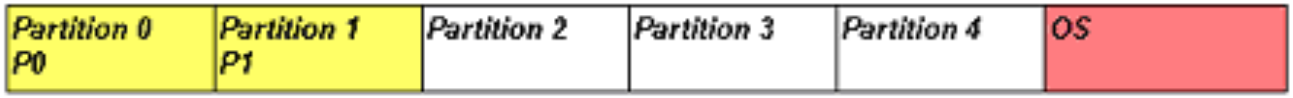
The data structure supporting memory management is simple.

Partition	Occupied
0	free
1	free
2	free
3	free
4	free

Fixed Size Partitions

Process 0 is allocated Partition 0 and Process 1 is allocated Partition 1

Address 0

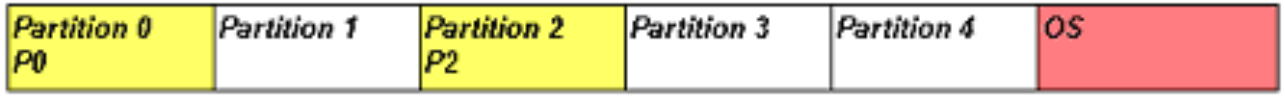


Partition	Occupied
0	P0
1	P1
2	free
3	free
4	free

Fixed Size Partitions

Process 2 is allocated Partition 2.
Process 1 is terminated. Partition 1 is released back.

Address 0



Partition	Occupied
0	P0
1	free
2	P2
3	free
4	free

Variable Size Partitions

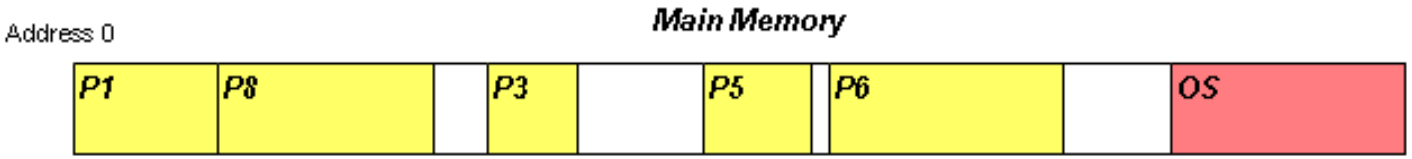
- Physical memory space not partitioned
- Allocation according to the exact requirement of a process
  - Smaller processes are given smaller blocks
    - Satisfying the needs of each process
    - Allocated memory is not wasted
  - Data structure support is complex
    - A linked list is needed to hold records of each allocation
    - Each record include the starting address and size of block
  - Dynamic Memory Allocation Problem
    - Finding suitable free memory for a new process is not straightforward

Variable Size Partitions

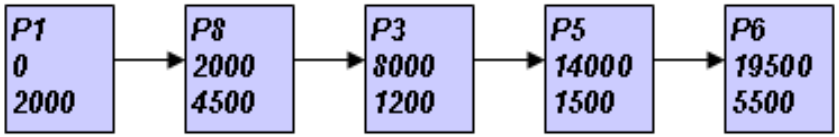
	Contiguous Allocation	Non-Contiguous
Fixed Size Partition		
Variable Size Partition		

Variable Size Partitions

Memory blocks are allocated according to the exact need of each process

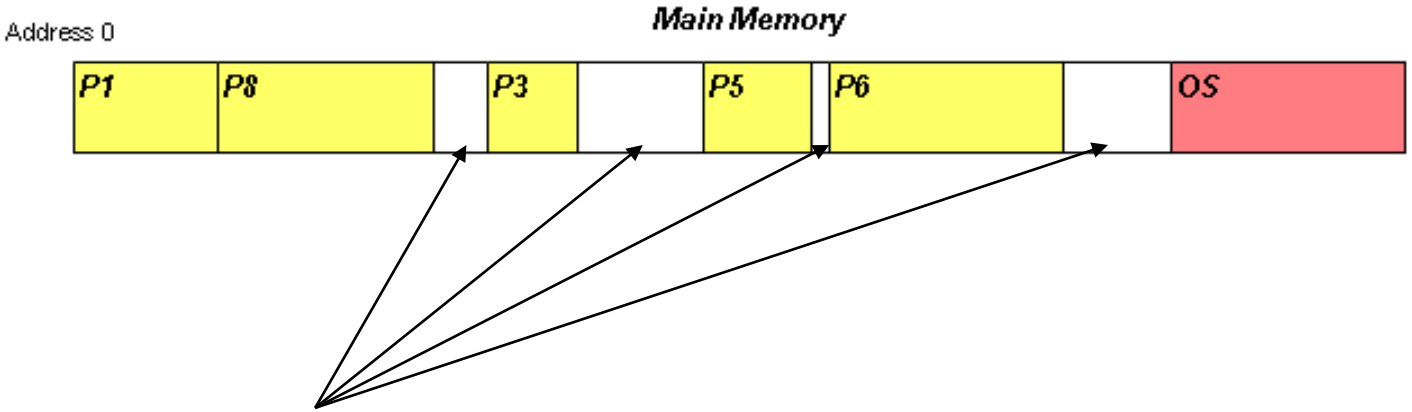


The data structure supporting memory management is a linked list

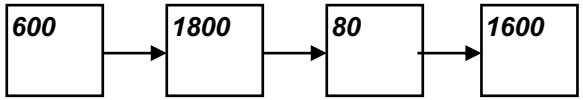


Variable Size Partitions

Memory blocks are allocated according to the exact need of each process



Also a list of "Free Memory Blocks"



Easier for Memory Allocation

Variable Size Partitions

- Three approaches to deal with dynamic memory allocation in variable size partition scheme
  - First-fit
    - Scan and use the first free memory block with sufficient size
  - Best-fit
    - Scan the whole memory space and find the free memory block that can fit the requirement but with minimal difference if over
  - Worst-fit
    - Scan the whole memory space and find the free memory block that can fit the requirement but with maximum difference if over

Example: Variable Size Partitions

Example: Dynamic Storage Allocation Problem

Consider that the free blocks (holes) have sizes 80K, 120K, 40K, and 30K in order from low to high memory space. A memory request of 40K has arrived.

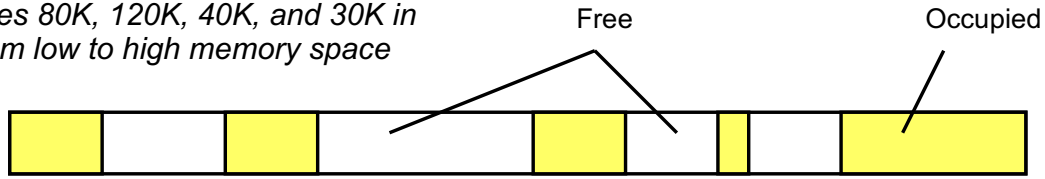
Work out how each of the allocation strategies of first-fit, best-fit, and worst-fit would handle the memory request.

First-fit (from low to high): the 80K free block is selected, remaining 40K after allocation

Best-fit: the 40K free block is selected, remaining 0K

Worst-fit: the 120K free block is selected, remaining 80K

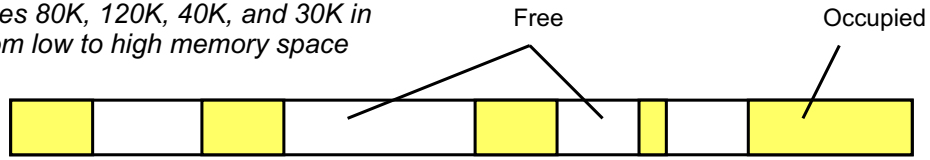
Consider that the free blocks (holes) have sizes 80K, 120K, 40K, and 30K in order from low to high memory space



A memory request of 40K is received

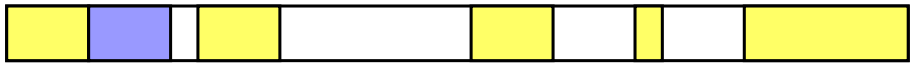
Example: Variable Size Partitions

Consider that the free blocks (holes) have sizes 80K, 120K, 40K, and 30K in order from low to high memory space



A memory request of 40K is received

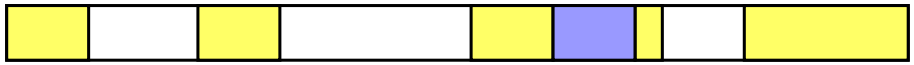
FIRST FIT (Low to High)



The search stops as soon as a suitable free memory block is found

The 80K block is selected

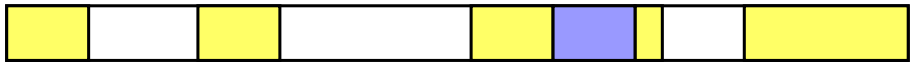
FIRST FIT (High to Low)



The search stops as soon as a suitable free memory block is found

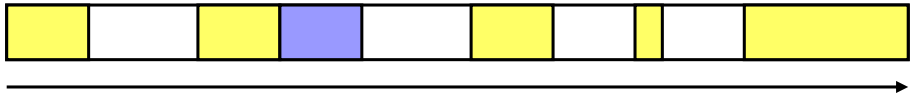
The 40K block is selected

BEST FIT



The 40K block is selected

WORST FIT



The 120K block is selected

Memory Utilization Rate and Fragmentations

- Fragments in the memory are memory space that is not usable or utilizable
  - Affect utilization rate of main memory
  - If much memory space is not usable, then utilization rate is low
- Two types of fragmentations
  - Internal fragmentation
  - External fragmentation

Internal Fragmentation

- Happening in fixed size partition scheme
  - The size of each partition is 200KB
  - But a process requires only 150KB
  - There are 50KB of memory allocated not needed by the process
  - The wastage is known as internal fragmentation

External Fragmentation

- Happening in variable size partition scheme
  - Arbitrary sizes of memory blocks are allocated and released
  - As time goes by, free memory blocks tend to be broken up into smaller and smaller blocks
  - The small blocks can become inadequate for any process
  - The wastage is known as external fragmentation

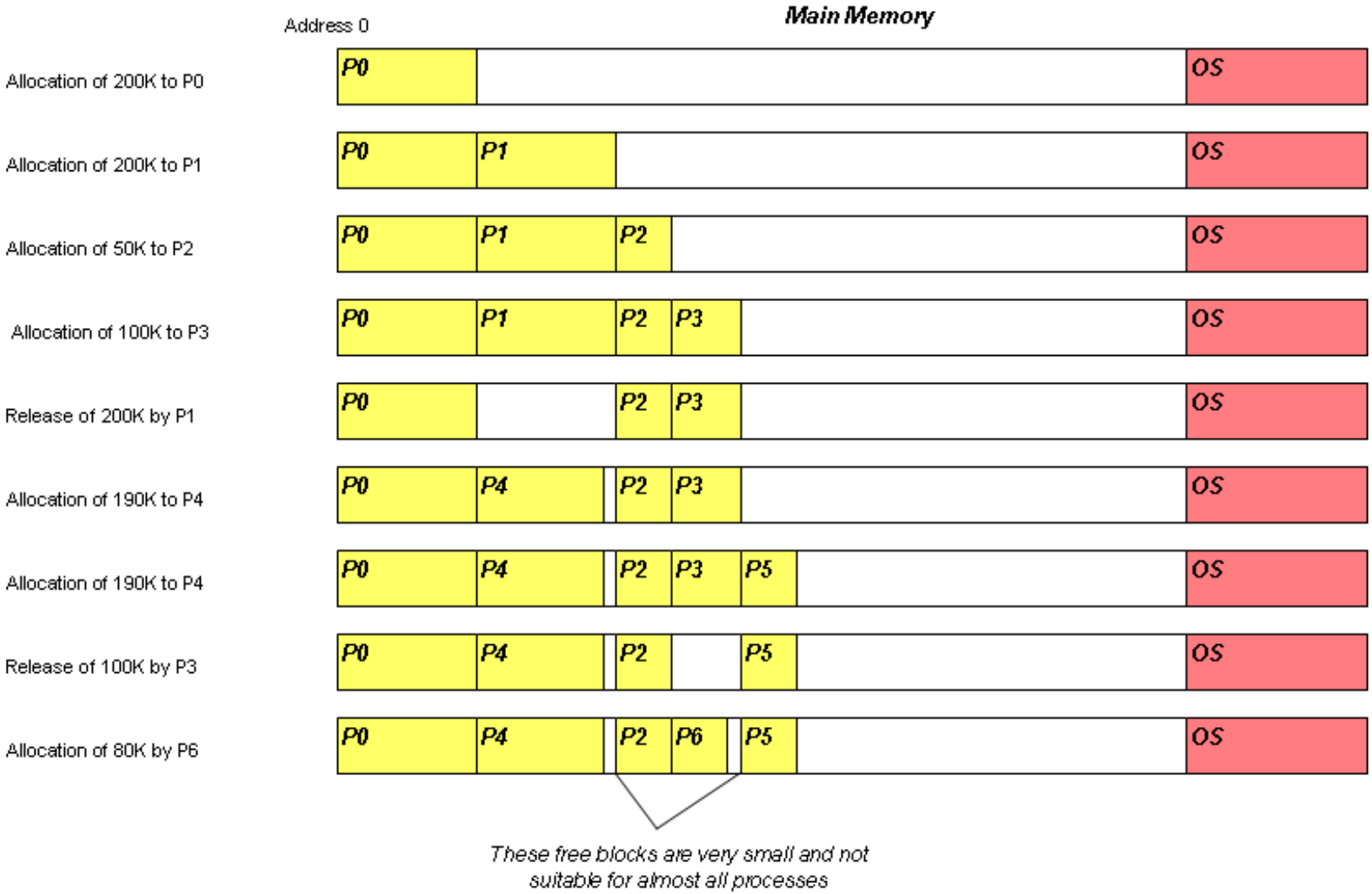
Example: External Fragmentation

Example: External Fragmentation Caused by Repeated Allocation and Release

The following sequence of memory allocation and release would cause external fragmentation with variable-size partition scheme and first-fit allocation strategy.

- Allocation of 200K to P0
- Allocation of 200K to P1
- Allocation of 50K to P2
- Allocation of 100K to P3
- Release of 200K by P1
- Allocation of 190K to P4
- Allocation of 50K to P5
- Release of 100K by P3
- Allocation of 80K by P6

Example: External Fragmentation



Compaction

- Compaction is a solution for external fragmentation
  - Moving allocated blocks together by squeezing out the small and not-utilizable free blocks
  - Costly operation
    - A lot of data copying

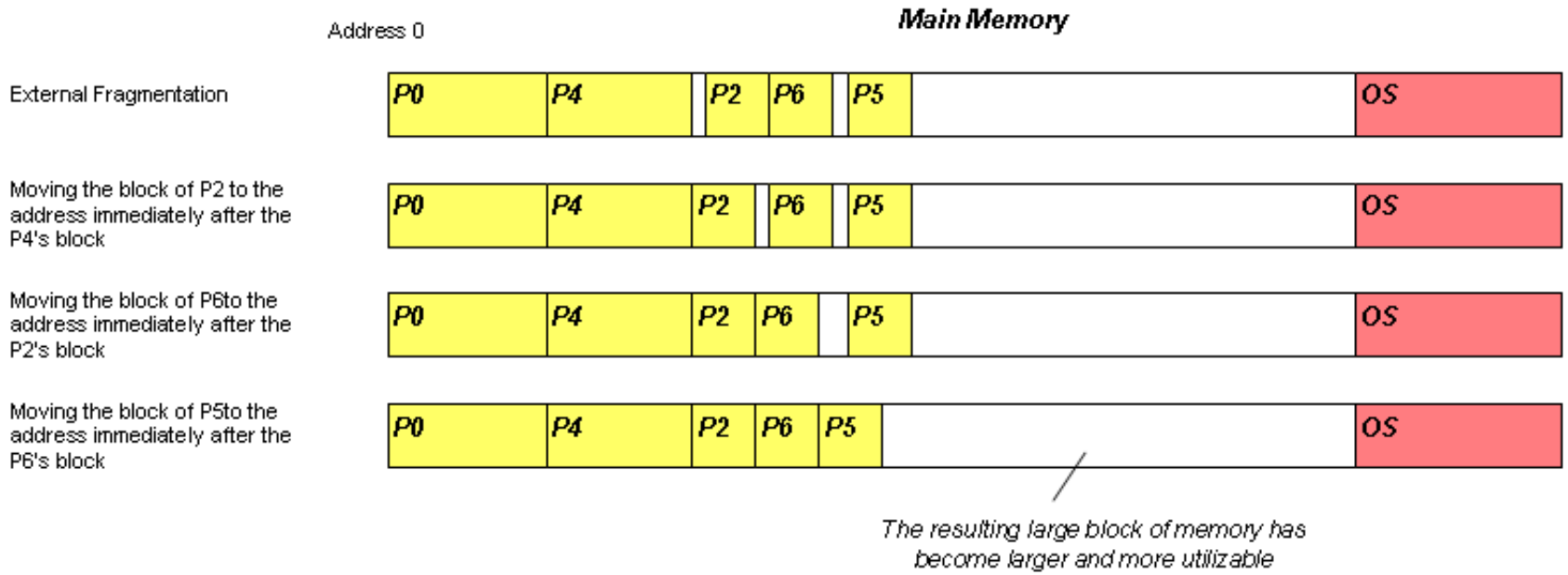
Example: Compaction

Example: External Fragmentation Solved by Compaction

The external fragmentation situation shown in the above example can be solved by compaction. The steps are:

- Checking there is no spare space before P0.
- Checking there is no spare space before P4.
- There is spare space before P2. Copy P2 block to the address immediately after the P4 block.
- There is now spare space before P6. Copy P6 block to the address immediately after the P2 block.
- There is now spare space before P5. Copy P5 block to the address immediately after the P6 block.

Example: Compaction



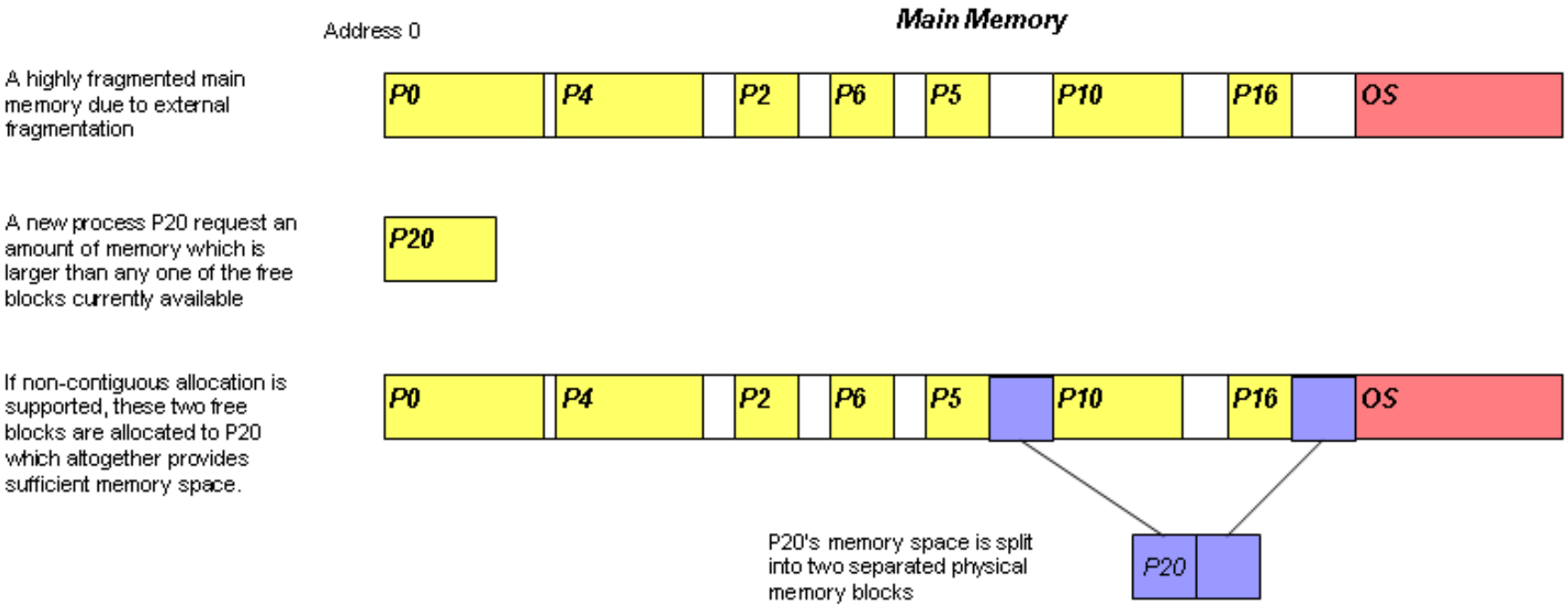
memory management by non-contiguous allocations

	Contiguous Allocation	Non-Contiguous
Fixed Size Partition		
Variable Size Partition		

Non-Contiguous Allocation

- The physical memory space allocation of a process can be split into several memory blocks
  - Reduces wastage
  - Reduces the actual memory demand or requirement by a process
    - Dynamic loading: load program in as needed
    - Some memory parts of a process are not in main memory at all
  - Facilitating sharing of memory between processes
    - A memory block containing a function can be shared between two or more processes

Non-Contiguous Allocation



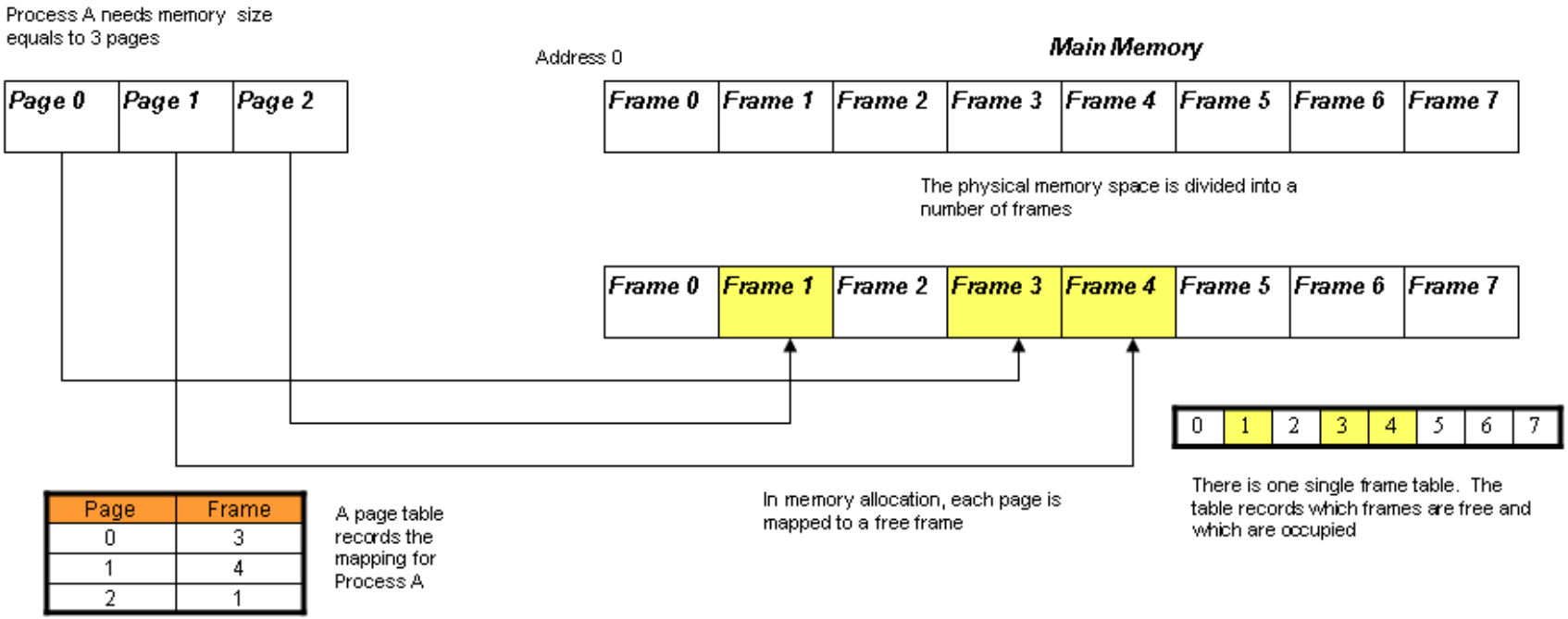
paging

Paging

- A memory management scheme that is based on
  - Non-contiguous memory allocation
  - Fixed-size partition scheme
- Memory space of a process is split up into several parts, loaded into the main memory at different addresses
- The parts are all of the same standard size

	Contiguous Allocation	Non-Contiguous
Fixed Size Partition		
Variable Size Partition		

Paging


Process A needs memory size equals to 3 pages
Page 0
Page 1
Page 2
Address 0
Main Memory
Frame 0
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Frame 6
Frame 7
The physical memory space is divided into a number of frames
Frame 0
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Frame 6
Frame 7
0 1 2 3 4 5 6 7
Page
Frame
0
3
1
4
2
1
A page table records the mapping for Process A
In memory allocation, each page is mapped to a free frame
There is one single frame table. The table records which frames are free and which are occupied

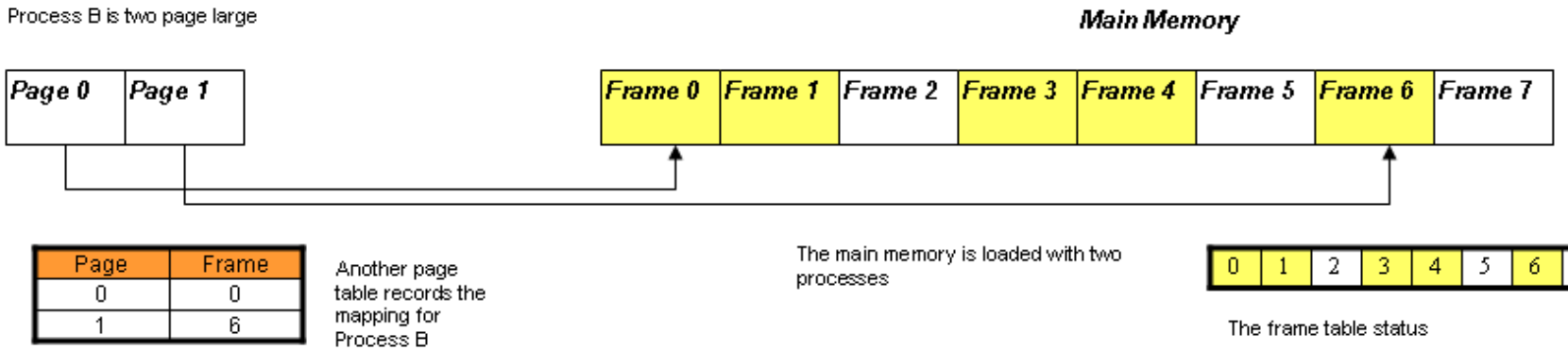
Core Ideas of Paging

- Physical memory space broken up into a number of memory blocks of the same size
  - Each memory block is called a **frame**
  - Each frame has a unique ID, starting from 0
  - Typical frame size is 4KiB (which is 4096 bytes)
- Logical memory space is also broken up into logical memory block of the same size
  - Each memory block is called a **page**
  - Each page has an ID, starting from 0
  - Page size is always equal to frame size in a memory system

Core Ideas of Paging

- Supported by a data structure called page table
  - One page table per process
  - Page table maps the pages to the frames
  - Each entry in a page table is a page ID to frame ID mapping
  - Page table is used in the conversion of logical address to physical address

Paging



Page	Frame
0	0
1	6

Another page table records the mapping for Process B

The main memory is loaded with two processes

0	1	2	3	4	5	6	7

The frame table status

Example: Paging

Example: Number of Frames and Pages

Given a paging system with page size (frame size) of 4096 bytes. The size of the main memory is 1Mbytes. How many frames are there?

Size of main memory is 1Mbytes or 2^{20} bytes.

Size of each frame is 4096 bytes or 2^{12} bytes.

Dividing the main memory up with frames, there are 2^{20} divided by 2^{12} frames.

$= 2^{8}$ frames = 256 frames

The frame ID starts from 0 and the last frame ID is 255.

Example: Paging

Example: Number of Frames and Pages

A process is created with memory requirement of 16000 bytes. How many pages are in its logical address space?

Frame size is the same as page size. Page size is also 4096 bytes or 2^{12} bytes.

The number of pages required is 16000 / 4096 = 3.91 pages

Three pages are not enough.

The actual number of pages in the logical address space is 4.

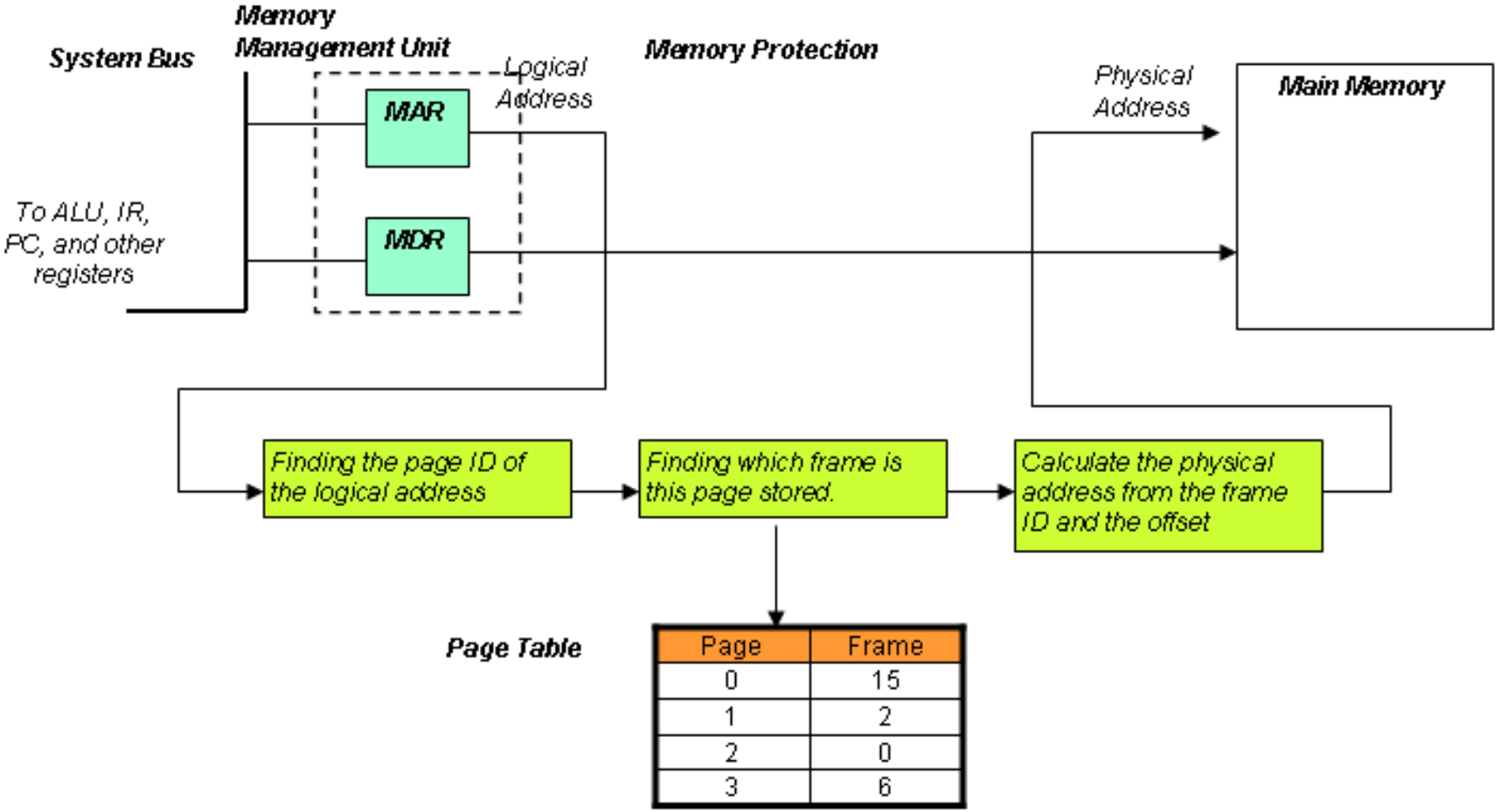
The system is multi-programming and on average each process's logical address space has 6 pages. Estimate how many processes can be loaded into the main memory at a time.

There are 256 frames and on average each process needs 6 frames. So the estimation is 256/6 = 42.67 processes = 42 processes

Address Binding in Paging System

- Every memory operation in a paging system involves accessing the page table for address binding
  - The page table is usually stored in the main memory
- Each memory operation involves actually two memory read/write operations
  - First to access the page table in the main memory
  - Second to access the physical memory location

Address Binding in Paging System



Example: Address Binding in Paging System

Example: Conversion of Logical Address into Physical Address 1

Given a paging system with page size (frame size) of 4096 bytes. The main memory has 1024 frames. So the frame ID ranges from 0 to 1023. The logical address space of a process has 6 pages.

What is the size of the logical address space of the process?

The size of logical address space = 4096 x 6 = 24576

Given the logical address of 12000 and the following page table content, work out the corresponding physical address.

Page	Frame
0	600
1	23
2	8
3	1
4	245
5	246

Example: Conversion from Logical Address to Physical Address in Paging Systems

Example: Conversion of Logical Address into Physical Address 1

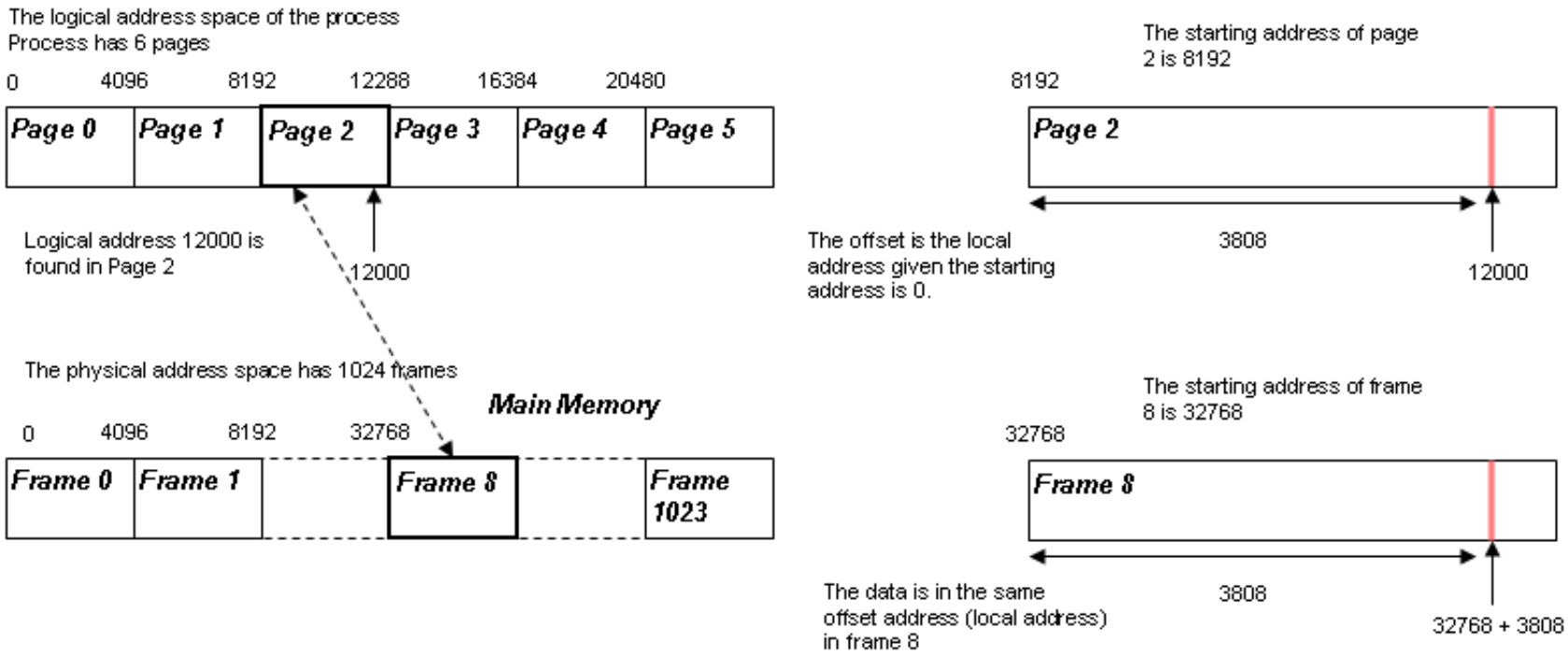
Page	Frame
0	600
1	23
2	8
3	1
4	245
5	246

The steps include the following:

- Work out which page contains logical address 12000. The page containing the address is 12000 / 4096 = 2.92 = Page ID 2.
- Work out the offset of logical address 12000 in page 2. The offset is the local address in page 2. Offset address = 12000 % 4096 = 3808.
- Work out where page 2 is stored in the main memory. The page table says page 2 is mapped to frame 8.
- Work out the starting address of frame 8. Starting address of frame 8 is 4096 x 8 = address 32768.
- Work out the physical address of logical address 12000. It is offset address 3808 in frame 8 = 32768 + 3808 = address 36576.

Logical address 12000 is found in physical address 36576.

Example: Conversion from Logical Address to Physical Address in Paging Systems



Example: Address Binding in Paging System

Example: Conversion of Logical Address into Physical Address 2

Given a paging system with parameters specified above, calculate the physical address of logical address 81200.

- Work out which page contains logical address 81200. The page containing the address is 81200 / 4096 = 19.82 = page ID 19.
- The logical address space of the process has only 6 pages. This address clearly lies beyond the legitimate memory block. This is either due to careless programming or malicious intention. A system exception would be raised.

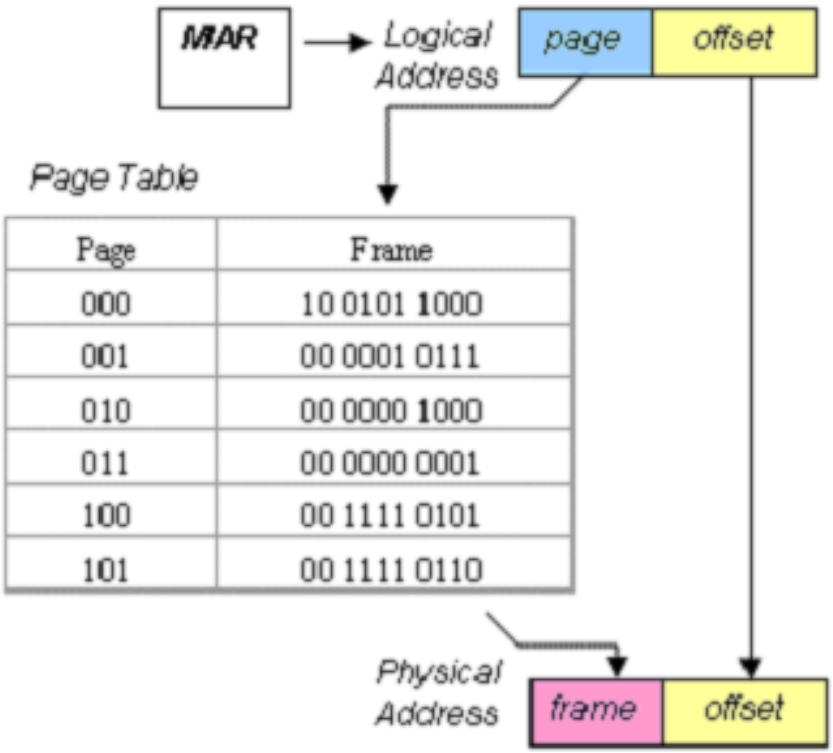
Significance of Page Size

- Logical-address to physical-address conversion and page table must be implemented in a most efficient manner
  - Every memory operation involves these components
- Page size a power of 2 can speed up the conversion from logical address to physical address.
  - No more addition operation is required in the conversion process.
  - Only bit replacement

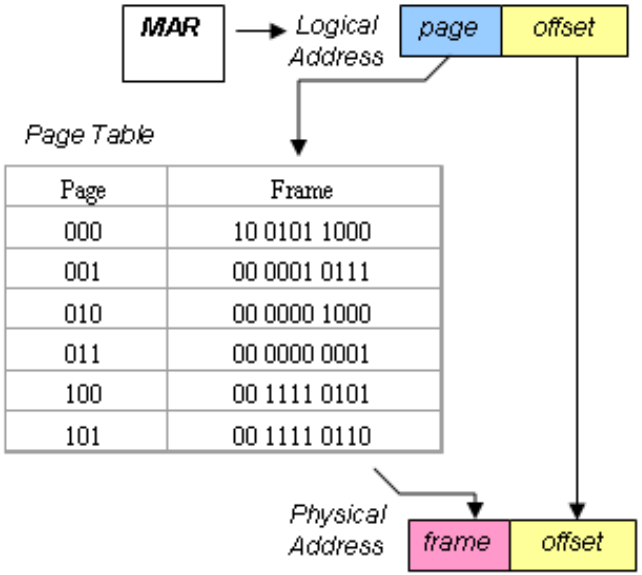
Example: Significance of Page Size

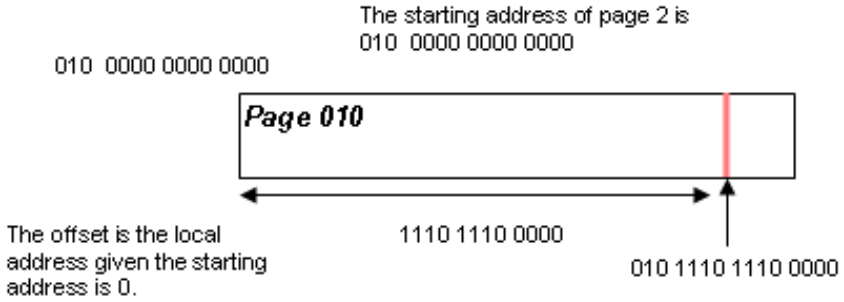
- Page size is a power of 2 allowing logical addresses can be separated in two parts
  - Given that the page size is 2n
  - n number of least significant bits will form the offset address
  - Remaining more significant bits will form the page ID
- Given a page size of 4096, n would be 12.
  - The offset address of a page ranges from 0000 0000 0000 to 1111 1111 1111.
  - Consider the logical address 0011 1101 1111 1000
  - Page ID is 0011
  - Offset address is 1101 1111 1000

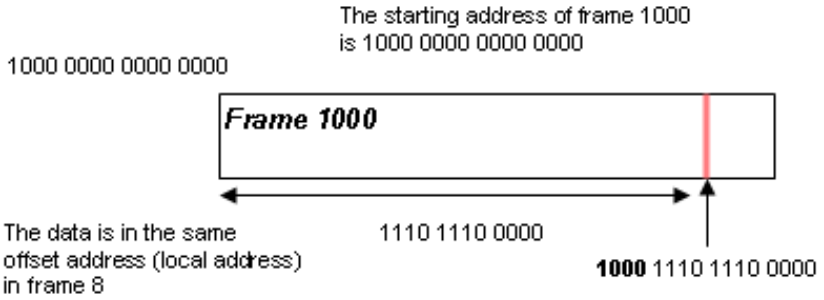
Example: Significance of Page Size



Example: Significance of Page Size







Example: Conversion of Logical Address to Physical Address in Paging Systems

Example: Conversion of Logical Address into Physical Address 3

This example re-do the first example but in binary values.

Given a paging system with page size (frame size) of 4096 bytes (which is 2^{12}). The main memory has 1024 frames (which is 2^{10}). So the frame ID ranges from 00 0000 0000 to 11 1111 1111. The logical address space of a process has 6 pages. The page ID ranges from 000 to 101.

Given the logical address of 12000 (binary 010 1110 1110 0000) and the following page table content, work out the corresponding physical address.

Page	Frame
000	10 0101 1000
001	00 0001 0111
010	00 0000 1000
011	00 0000 0001
100	00 1111 0101
101	00 1111 0110

Example: Conversion of Logical Address to Physical Address in Paging Systems

Example: Conversion of Logical Address into Physical Address 3

Given the logical address of 12000 (binary 010 1110 1110 0000) and the following page table content, work out the corresponding physical address.

Page	Frame
000	10 0101 1000
001	00 0001 0111
010	00 0000 1000
011	00 0000 0001
100	00 1111 0101
101	00 1111 0110

The steps include the following:

The page ID is clear from the logical address. The 12 least significant bits are the offset address. The first 3 bits are the page ID.

Offset address = 1110 1110 0000 and Page ID = 010

The physical address is simply obtained by replacing the page ID with the corresponding frame ID. The frame ID for the page ID 010 is **00 0000 1000**.

The physical address is **00 0000 1000** 1110 1110 0000. In decimal this value is 36576. We have got the same result without the need to add numbers.

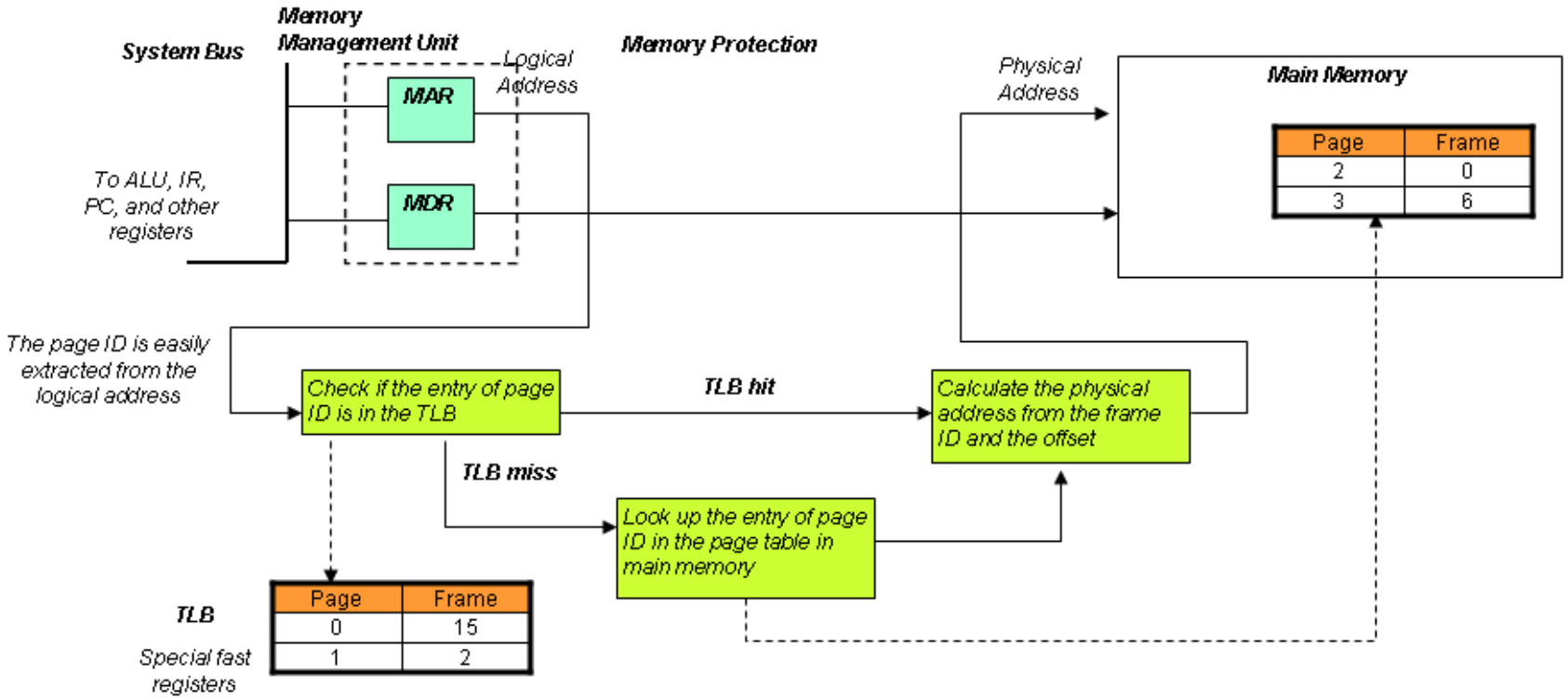
Translation Look-aside Buffer (TLB)

- The conversion of addresses is still an expensive operation
  - Each memory operation involves two memory read/write operations
    - reading the page table
    - actual memory operation
- Page tables are implemented in a set of high-speed registers and dedicated hardware for address mapping
  - Infeasible if the number of entries in the page table is large
  - Too expensive to have a lot of registers
  - More economical to have the page table stored in the main memory

Translation Look-aside Buffer

- Translation Look-aside Buffer (TLB) is an economical solution for speeding up the address binding process
  - Part of a page table stored in high-speed registers, and the other part in the main memory
    - A balanced solution
  - The operating systems guess which part of the page table would be accessed frequently
    - Cached in the TLB for quicker access

Translation Look-aside Buffer



Translation Look-aside Buffer

- Possible to guess correctly
  - Locality of references
    - Most memory address operations usually localized
- A common phenomenon in programming.
  - Executing a program loop repeatedly
  - Contiguous section of instructions being executed repeatedly
  - It is reasonable to assume that this section belong to the same page (or the neighbouring pages)

Translation Look-aside Buffer

- TLB is established when a new process is scheduled to run,
- TLB is flushed when the process is removed.
- Some entries are important
  - Address of the kernel
  - Never removed and called a wired down

Example: Translation Look-aside Buffer

Example: Effective Memory Access Time with TLB

The effective memory access time (EMAT) is the average amount of time taken to perform a memory operation. Assume that each access to the main memory requires 100 ns. However in a paging system, the EMAT is not 100 ns. The conversion of logical to physical address requires an addition access to the page table, which is in the main memory. So the EMAT in a paging system is 100 ns (page table) + 100 ns (actual operation) = 200 ns.

The introduction of TLB in the conversion process can speed up the EMAT considerably. Access to TLB is usually much faster. Let's assume a small figure of 20 ns. If the whole page table can be stored in the TLB, then the EMAT becomes 20 ns (page table in TLB) + 100 ns.

However in real situations, the TLB is large enough to store part of the page table. We can still calculate the TLB if we know the hit rate. The hit-rate is the percentage that a page table entry is found in the TLB.

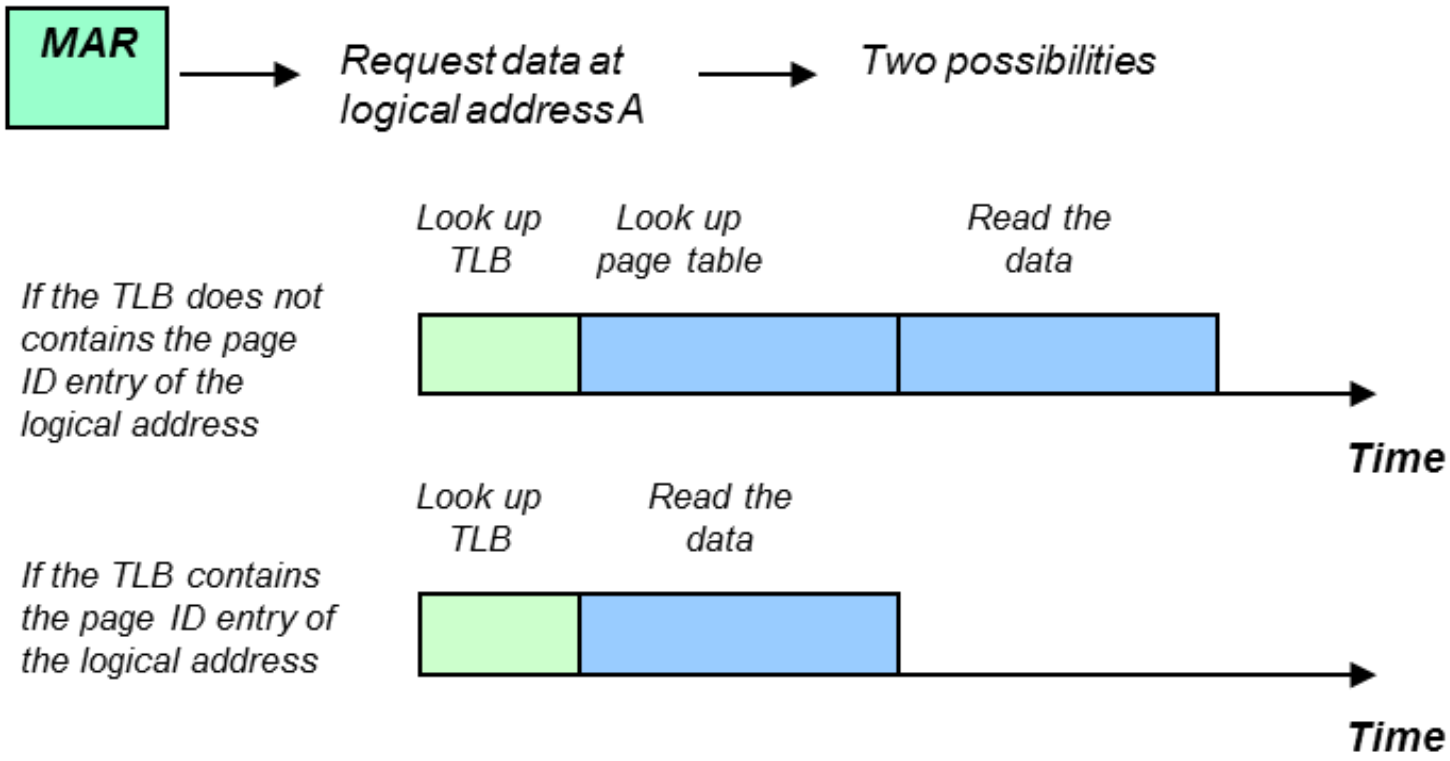
Calculate the EMAT if the hit-rate is 80%.

Time taken if page table entry is found in TLB = 20 ns + 100 ns = 120 ns

Time taken if page table entry is not found in TLB = 20 ns + 100 ns + 100 ns = 220 ns

EMAT = 0.8 x 120 ns + 0.2 x 220 ns = 140 ns

Example: Translation Look-aside Buffer



Fragmentation in Paging Systems

- Paging system is a fixed size partition scheme
  - Suffering from internal fragmentation
- The amount of wastage caused by fragmentation can be estimated using the **50-percent rule**
  - Statistical
- For paging systems the average wastage is half a page size

Paging and Segmentation

	Contiguous Allocation	Non-Contiguous
Fixed Size Partition		Paging
Variable Size Partition		Segmentation

segmentation

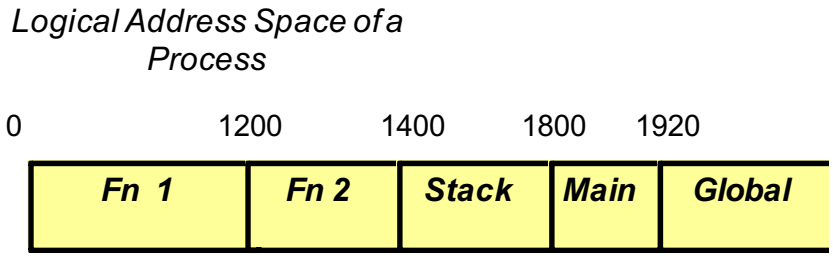
Segmentation

- Segmentation is a memory management scheme
  - Non-contiguous memory allocation
  - Variable-size partition scheme
- Process's logical memory space is split up according to its **logical structure**

	Contiguous Allocation	Non-Contiguous
Fixed Size Partition		
Variable Size Partition		

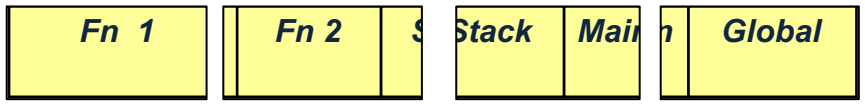
Core Ideas of Segmentation

- Respect the structure of programs
  - There are functional segments in a program
  - These segments are found in different parts in the logical memory
  - Some segments come from different sources (e.g. runtime libraries)
  - Partitioning should be based on this logical structure

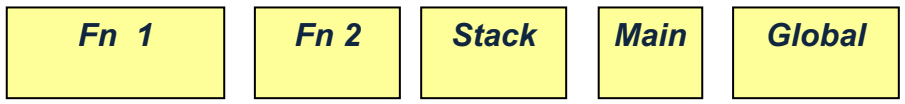


A program has a logical structure, consisting of five parts

Paging



Segmentation

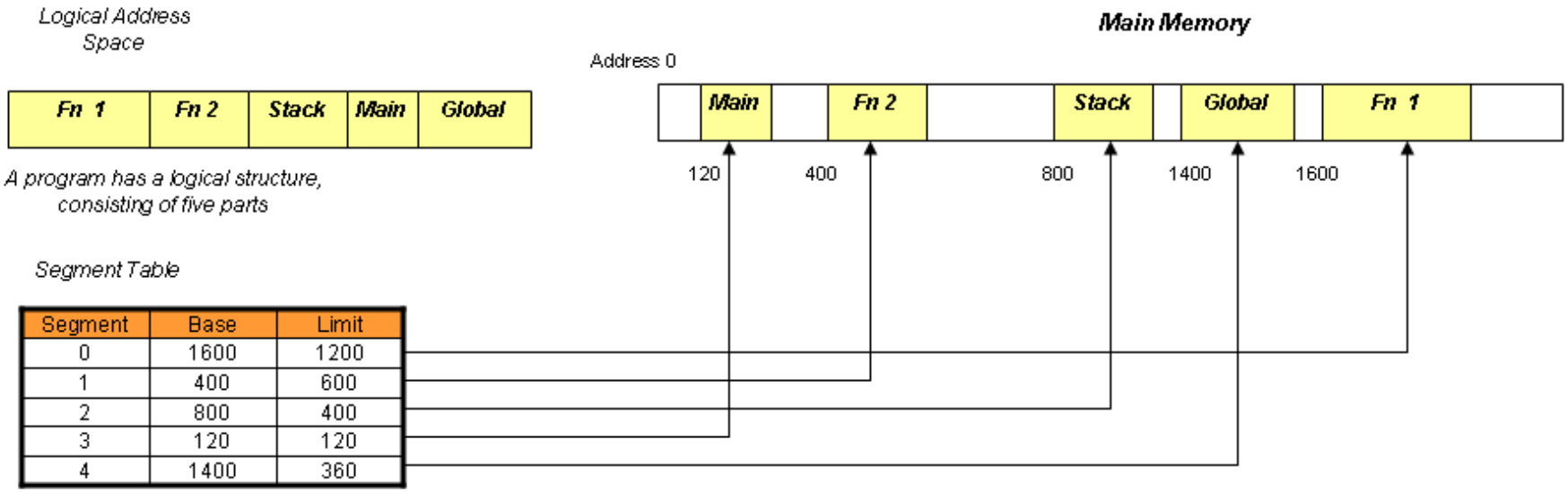


Core Ideas of Segmentation

- Each functional segment is recognized in segmentation
  - A segment has three parameters
    - Segment ID
    - Starting address in the physical memory (the **base address**)
    - Segment size (the **limit**)
  - The first logical address in every segment is 0
  - Logical space split into segments
    - Segment number
    - Address offset to work out segment size

Segment Table

- There is a segment table for each process containing
  - Segment number
  - Base address of the segment in the physical memory
  - Limit that indicates the size of the segment



Issues of Segmentation

- Possibility of sharing segments between processes
  - Library routines that can be used by many processes
  - Loaded into the main memory for all processes to use
  - Save memory space
- Security between processes
  - Read-only or execute-only
  - Hardware to prevent off-limit access
- Prone to external fragmentation
  - Segmentation is based on variable size partition scheme

Address Binding in Segmentation

- Every memory operation involves consultation of the segmentation table and an addition operation
  - The logical address is in the form of (segment ID, offset address)
  - The physical address is calculated:
    - Looking up the base address and the limit of the segment.
    - Detect segmentation error
    - Adding the offset address and the base address

Example: Address Binding in Segmentation

Example: Conversion of Logical Address into Physical Address in Segmentation

Given the following segment table, calculate the following logical addresses.

Segment	Base	Limit
0	1400	1000
1	6300	400
2	4300	400
3	3200	1100
4	4700	1000

- Address (0, 200)
- Address (1, 6)
- Address (2, 500)
- Address (3, 900)

Example: Address Binding in Segmentation

Example: Conversion of Logical Address into Physical Address in Segmentation

Given the following segment table, calculate the following logical addresses.

Segment	Base	Limit
0	1400	1000
1	6300	400
2	4300	400
3	3200	1100
4	4700	1000

- Address (0, 200)
- Address (1, 6)
- Address (2, 500)
- Address (3, 900)

Address (0, 200) means segment 0 and offset address 200. Offset address 200 is within the limit 1000. The physical address is 1400 + 200 = 1600

Address (1, 6) means segment 1 and offset address 6. Offset address 6 is within the limit 400. The physical address is 6300 + 6 = 6306

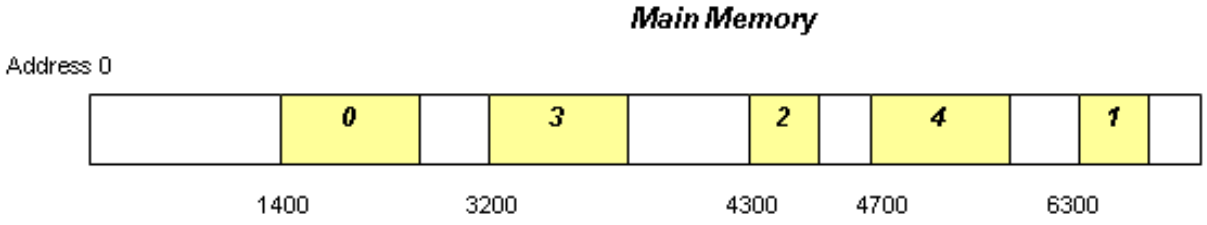
Address (2, 500) means segment 2 and offset address 500. Offset address 500 is outside the limit. Segmentation error would be raised.

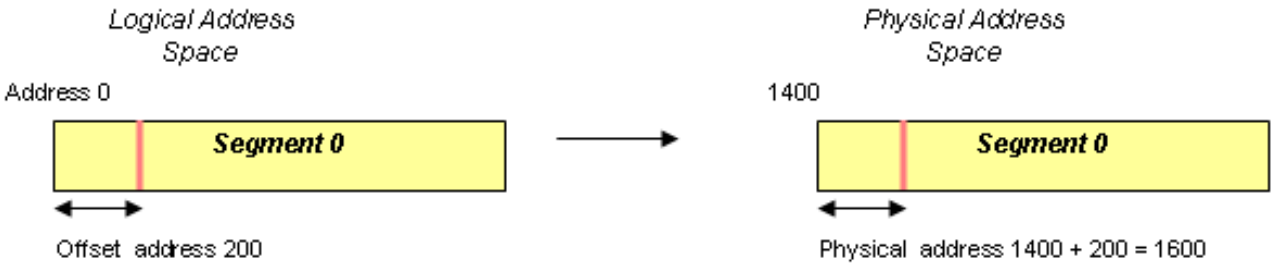
Address (3, 900) means segment 3 and offset address 900. Offset address 900 is within the limit 1100. The physical address is 3200 + 900 = 4100

Example: Address Binding in Segmentation

Segment Table

Segment	Base	Limit
0	1400	1000
1	6300	400
2	4300	400
3	3200	1100
4	4700	1000





Benefits of Segmentation

- Non-contiguous allocation, variable-size partition leads to interesting applications
  - Dynamic loading
    - Delay loading of segments of a program until it is needed
  - Dynamic linking
    - Delay code integration until run-time

Dynamic Loading

- Delaying the loading of a program segment until it is required for program execution
  - Possible with non-contiguous allocation
  - Memory allocation for a process is in multiple stages
  - Better utilization of memory
    - Memory are allocated as needed by program execution
    - The not-needed part can remain in secondary memory
  - Speeding up the starting of a large application
    - Some functions are not loaded until they are needed
    - Most appropriate if a large applications support many functions
  - The selection of program parts to load is the responsibility of the applications

Example: Dynamic Loading

The modules of the word processor

Core	Editor	Drawing	Grammar	Mail	Macros

Address 0

Main Memory

Loading the modules that are immediately needed first

The user has just selected Drawing tools. The Drawing module is now loaded into the main memory

Example: Dynamic Loading

Example: Dynamic Loading of a Word Processor

Word processors are very large applications, providing a large number of functions. However, most of the functions are rarely used. It makes sense to dynamic load them in as and when needed.

A certain word processor is divided into the following modules. The size of each module is indicated in brackets:

- Core (10M)
- Editor (3M)
- Grammar checker (8M)
- Drawing tool (5M)
- Mail merge (6M)
- Macros (12M)

The Core, Editor, and Grammar checker of the word processor may be loaded first. The other modules are loaded when the user invokes the relevant functions. The loading time is significantly reduced. The amount of memory use is also reduced.

Dynamic Linking

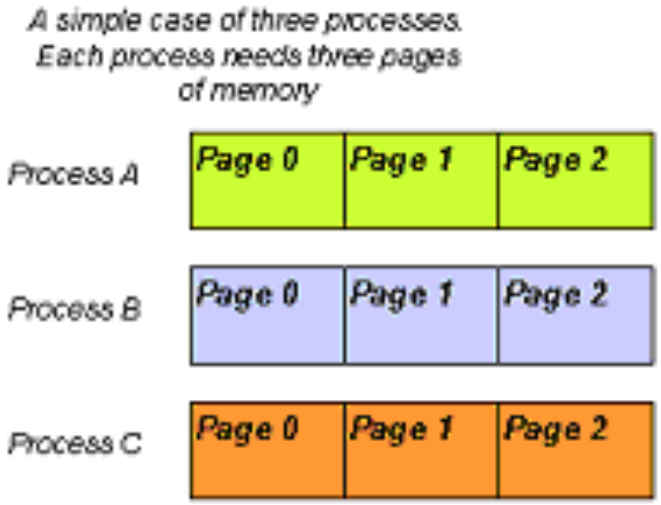
- Allowing the sharing of memory between processes
  - Often the shared memory containing program code
  - Better utilization of memory
    - Instead of having each process loading its own copy of program code of the same function
  - Stubs and Dynamic Linked Libraries (DLL)
    - Stubs are code inserted by compiler to the location of function calls
    - Stubs contain logic to check if the function is loaded
    - It asks the OS to load the function in if not already loaded
    - Functions are stored in Dynamic Linked Libraries
  - Supported by the OS
  - Security hazards
    - Set a block as read-only or execute-only

better memory utilization with swapping

Swapping

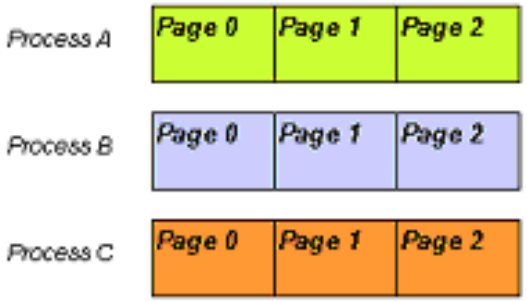
- Consider single-processor system running multi-programming
  - A process is in the Running state at a time
    - Many processes are loaded into the main memory
    - Only one of them is actually running
  - Level of multi-programming is restricted by the size of memory
    - 1G memory can support 10 processes of 100M each
  - Swapping allows more processes to be ready with the same amount of physical memory

Example: Swapping

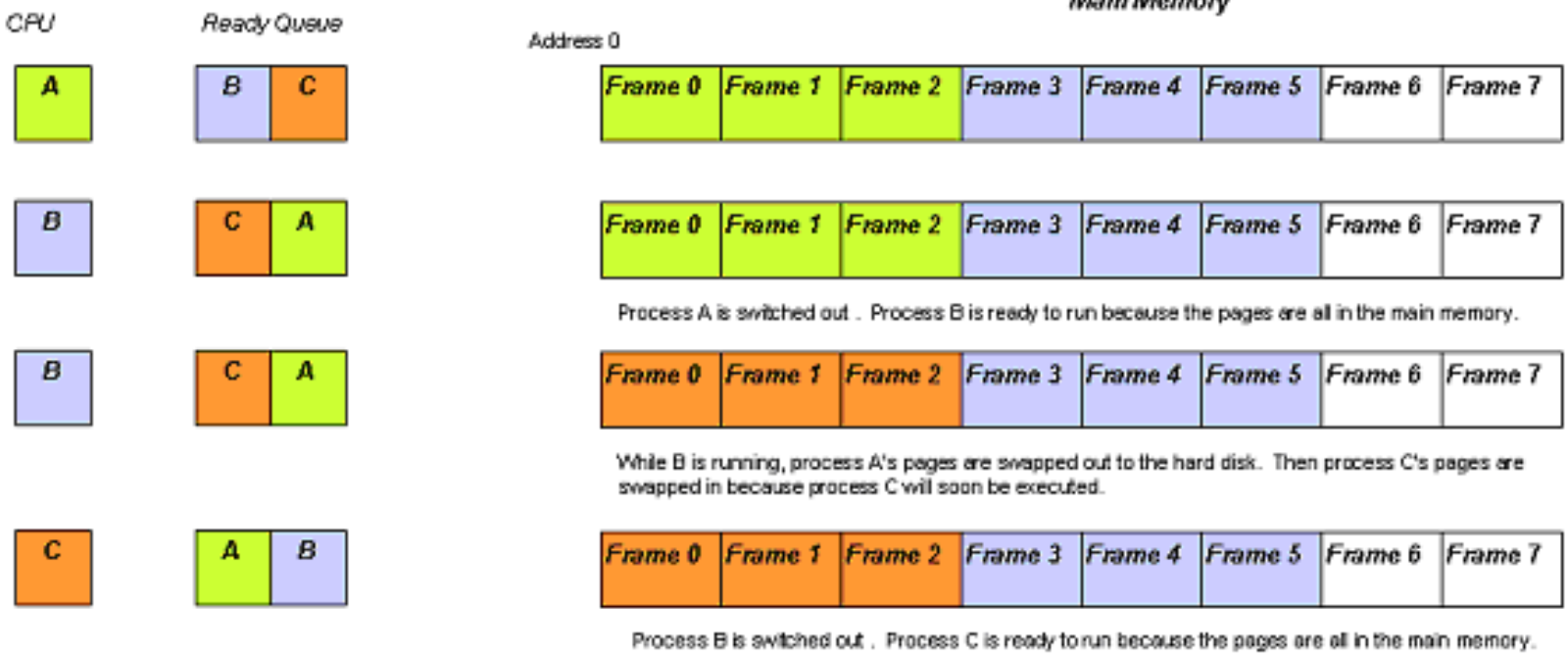


Example: Swapping

A simple case of three processes. Each process needs three pages of memory



Process A is created first, followed by process B and process C. Process A's pages are loaded in and so are process B's pages. There is insufficient physical memory for process C.



Swapping

- Swapping works best with the round-robin process scheduling algorithm
  - **Roll-in, roll-out** concept
    - Swapping-in higher priority process
    - Swapping-out lower priority process
  - When the higher priority process has completed the execution, the lower priority process is swapped-in

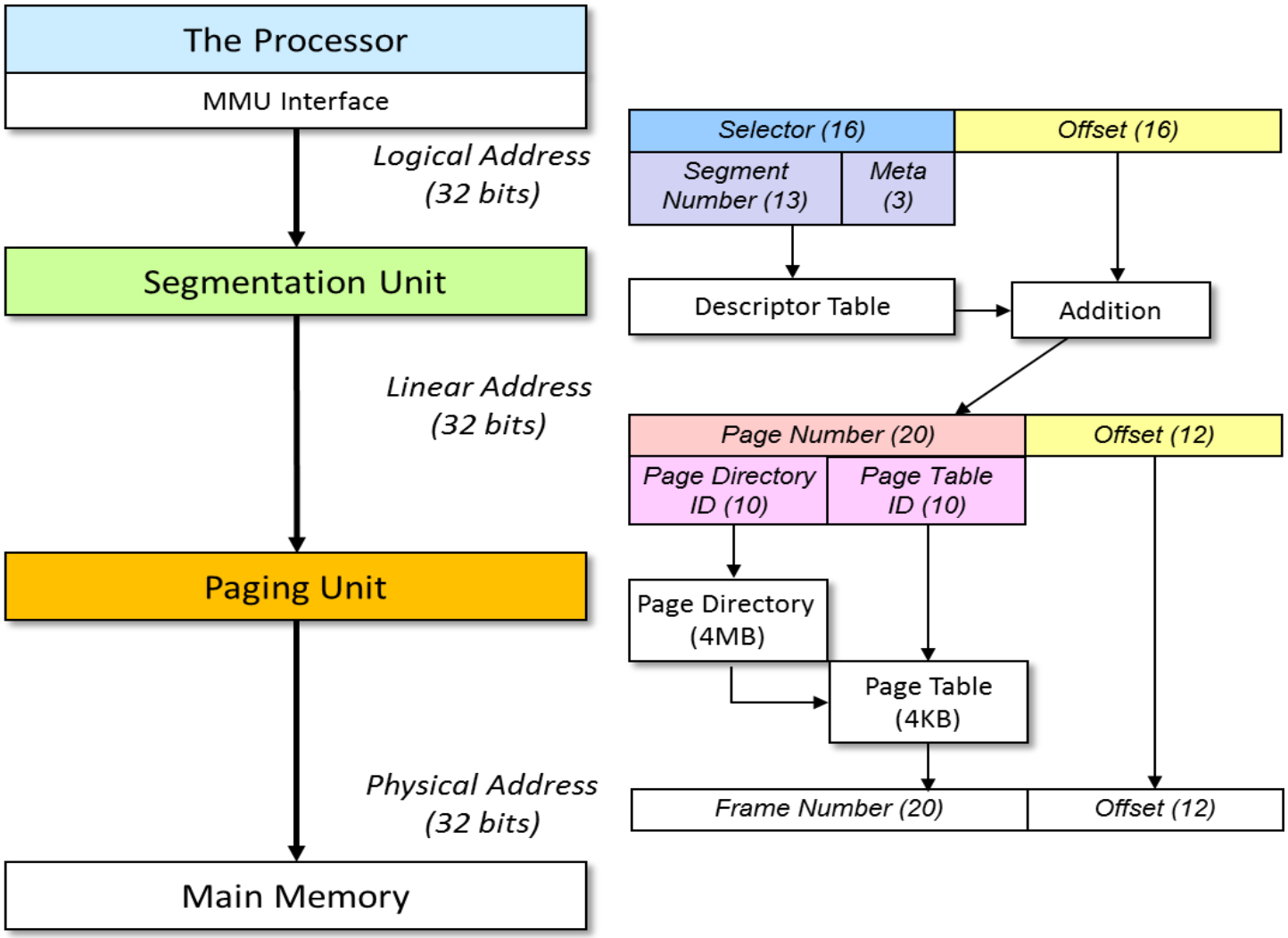
case study: intel ia-32 architecture

Intel CPU and Chipsets

- Intel CPU and chipsets most popular architecture on PC computers
  - IA-32 for the series of 32-bit chips including Pentium
  - IA-32 uses a combination of segmentation and paging
  - IA-64 has removed segmentation due to the 64-bit addressing space scope sufficient for all PC computers

Intel IA-32 Architecture

Intel IA-32 Architecture



Intel IA-32 Architecture

- Logical address
  - Selector part (16 bits) and offset part (16 bits)
  - Two types of segment table (called descriptor tables)
    - Local Descriptor Table (private to the process, 8192 max)
    - Global Descriptor Table (shared, 8192 max)
  - Checked against memory limit registers
  - Converted to linear address
    - Similar to segmentation

Intel IA-32 Architecture

- Linear address
  - Conversion to physical address with paging
  - Two page sizes: 4MB and 4KB
  - Two level paging scheme
    - First 10 bits points to a page directory (4MB or 4KB)
    - For 4KB page, further reference to page table is needed
    - For 4MB page, no further reference is needed

Intel IA-64 Architecture

- Maximum physical memory size of 2^{64} bytes
  - In practice only 48 bits are needed (281,474,976,710,656)
  - Enough for the foreseeable future
  - Three pages sizes: 4KB, 2MB, or 1GB (if supported)
  - Four level of paging hierarchy
  - Segmentation is not there anymore (in long mode)
    - People have been arguing about the loss of segment registers that lead to disable of a number of usages
    - Most x86-64 implementations still provide for backward compatibility