Operating Systems
“Memory Management”

Mathieu Delalandre
University of Tours, Tours city, France
mathieu.delalandre@univ-tours.fr

Lecture available at http://mathieu.delalandre.free.fr/teachings/operating2.html

Operating Systems
“Memory Management”

1. Introduction
2. Contiguous memory allocation
    - 2.1. Partitioning and placement algorithms
    - 2.2. Memory fragmentation and compaction
    - 2.3. Process swapping
    - 2.4. Loading, address binding and protection
3. Simple paging and segmentation
    - 3.1. Paging, basic method
    - 3.2. Segmentation

Introduction (1)

Memory hierarchy: memory is a major component in any computer. Ideally, memory should be extremely fast (faster than executing an instruction on CPU), abundantly large and dirt chip. No current technology satisfies all these goals, so a different approach is taken. The memory system is constructed as a hierarchy of layers.

	Access time (4KB)	**Capacity**
Registers	0.25 - 0.5 ns	< 1 KB
Cache	0.5 - 25 ns	> 16 MB
Main memory	80 - 250 ns	> 16 GB
Disk storage	30 µs - plus	> 100 GB

i.e. from 130.2 $Mb.s^{-1}$ to 15.6 $Gb.s^{-1}$

As one goes down in the hierarchy, the following occurs:

a. decreasing cost per bit,
b. increasing capacity,
c. increasing access time,
d. decreasing frequency of access to the memory by the processor.

Introduction (2)

Memory hierarchy: memory is a major component in any computer. Ideally, memory should be extremely fast (faster than executing an instruction on CPU), abundantly large and dirt chip. No current technology satisfies all these goals, so a different approach is taken. The memory system is constructed as a hierarchy of layers.

	Access time (4KB)	**Capacity**
Registers	0.25 - 0.5 ns	< 1 KB
Cache	0.5 - 25 ns	> 16 MB
Main memory	80 - 250 ns	> 16 GB
Disk storage	30 µs - plus	> 100 GB

i.e. from 130.2 Mb.s^{-1} to 15.6 Gb.s^{-1}

The strategy of using a memory hierarchy works in principle, but only if conditions (a) through (d) in the preceding list apply.

e.g. with a two-level memory hierarchy

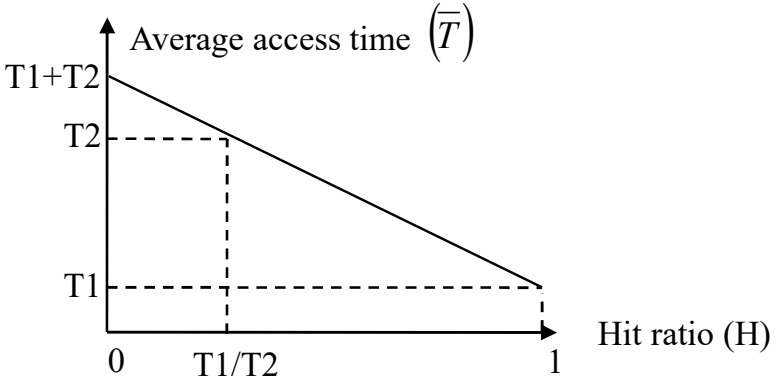
H is the hit ratio, the faction of all memory accesses that are found in the faster memory.

T1 is the access time to level 1.

T2 is the access time to level 2, with *T2*>>*T1*.

$\overline{T}$ is the average access time, computed as:

$$\overline{T} = H \times T_1 + (1 - H) \times (T_1 + T_2)$$



Introduction (3)

Memory hierarchy: memory is a major component in any computer. Ideally, memory should be extremely fast (faster than executing an instruction on CPU), abundantly large and dirt chip. No current technology satisfies all these goals, so a different approach is taken. The memory system is constructed as a hierarchy of layers.

	Access time (4KB)	**Capacity**
Registers	0.25 - 0.5 ns	< 1 KB
Cache	0.5 - 25 ns	> 16 MB
Main memory	80 - 250 ns	> 16 GB
Disk storage	30 µs - plus	> 100 GB

i.e. from 130.2 $Mb.s^{-1}$ to 15.6 $Gb.s^{-1}$

The strategy of using a memory hierarchy works in principle, but only if conditions (a) through (d) in the preceding list apply.

e.g. with a two-level memory hierarchy
Considering 200 ns / 40 µs as access times to the main / disk memory,

$$\overline{T} = H \times 200 + (1 - H) \times 4 \times 10^4$$

For a performance degradation less than 10 percent in main memory,

$$220 = H \times 200 + (1 - H) \times 4 \times 10^4$$

$$H = 0{,}999497$$

then, 1 fault access out of 1990.
additional constraints must be considered, a typical hard disk has:

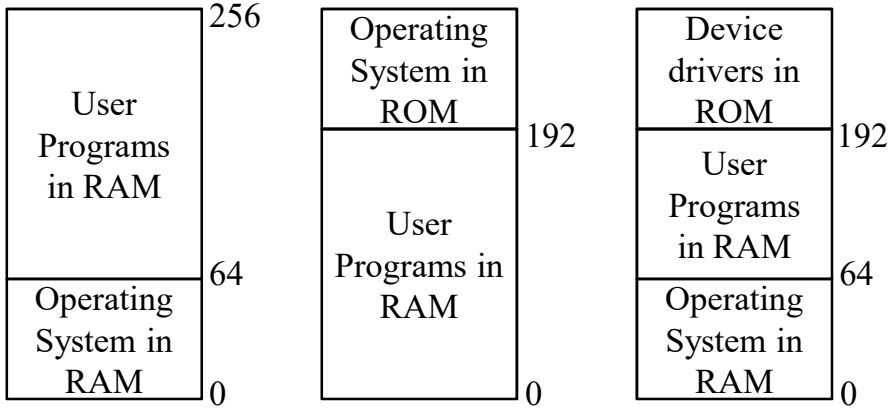
Latency	3 ms
Seek time	5 ms
Access time	0.05 ms
Total	≈8 ms

Introduction (4)

Memory management: managing the lowest level of cache memory is normally done by hardware, the focus of memory management is on the programmer's model of main memory and how it can be managed well.

Memory management without memory abstraction: the simplest memory management is without abstraction. Main memory is generally divided in two parts, one part for the operating system and one part for the program currently executed. The model of memory presented to the programmer was physical memory, a set of addresses belonging to the user's space.

e.g. three simple ways to organize memory with an operating system and user programs:



When a program executed an instruction like

MOV REGISTER1, 80

the computer just moved the content of physical memory location 80 to REGISTER1.

Introduction (5)

Memory management: managing the lowest level of cache memory is normally done by hardware, the focus of memory management is on the programmer's model of main memory and how it can be managed well.

Memory management without memory abstraction: the simplest memory management is without abstraction. Main memory is generally divided in two parts, one part for the operating system and one part for the program currently executed. The model of memory presented to the programmer was physical memory, a set of addresses belonging to the user's space.

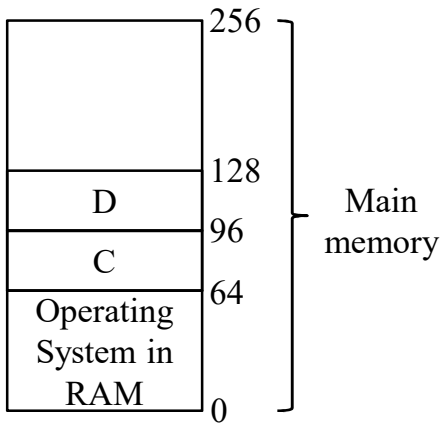
e.g.

programming version of C,D

Program C		Program D	
…	…	…	…
ADD	28	CMP	28
MOV	24		24
	20		20
	16		16
	12		12
	8		8
	4		4
JMP 24	0	JMP 28	0

loadable version of C,D, physical addresses must be directly specified by the programmer in the program itself

Program C		Program D	
…	…	…	…
ADD	92	CMP	124
MOV	88		120
	84		116
	80		112
	76		108
	72		104
	68		100
JMP 88	64	JMP 124	96



Introduction (6)

Memory management: managing the lowest level of cache memory is normally done by hardware, the focus of memory management is on the programmer's model of main memory and how it can be managed well.

Memory management with memory abstraction provides a different view of a memory location depending on the execution context in which the memory access is made. The memory abstractions makes the task of programming much easier, the programmer no longer needs to worry about the memory organization, he can concentrate instead on the problem to be programmed. The memory abstraction covers:

Memory partitioning is interested for managing the available memory into partitions.

contiguous / noncontiguous allocation	assigns a process to consecutive / separated memory blocks.
fixed / dynamic partitioning	manages the available memory into regions with fixed / deformable boundaries.
complete / partial loading	refers to the ability to execute a program that is only fully or partially in memory.

Introduction (7)

Memory management: managing the lowest level of cache memory is normally done by hardware, the focus of memory management is on the programmer's model of main memory and how it can be managed well.

Memory management with memory abstraction provides a different view of a memory location depending on the execution context in which the memory access is made. The memory abstractions makes the task of programming much easier, the programmer no longer needs to worry about the memory organization, he can concentrate instead on the problem to be programmed. The memory abstraction covers:

Placement algorithms: when it is time to load a process into main memory, the OS must decide which memory blocks to allocate.

Fragmentation / compaction: is a phenomenon in which storage space is used inefficiently, reducing capacity or performance and often both. compaction can eliminate, in part, the fragmentation.

Process swapping is a strategy to deal with memory overload, it consists in bringing each process in its entirely, running if for a while, then putting it back on the disk.

Address protection determines the range of legal addresses that the process may access and to ensure that this process can access only these legal addresses.

Address binding: the addresses may be represented in a different way between the disk and main memory spaces. Address binding is a mapping from one address space to another.

Introduction (8)

Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Operating Systems
“Memory Management”

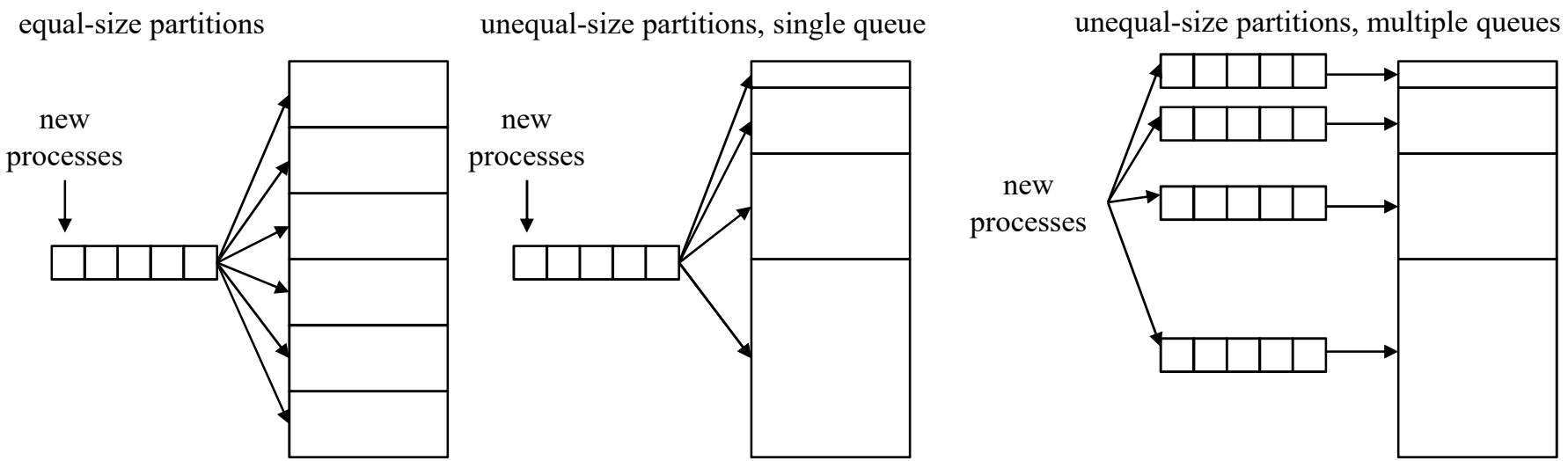
1. Introduction
2. Contiguous memory allocation
    - 2.1. Partitioning and placement algorithms
    - 2.2. Memory fragmentation and compaction
    - 2.3. Process swapping
    - 2.4. Loading, address binding and protection
3. Simple paging and segmentation
    - 3.1. Paging, basic method
    - 3.2. Segmentation

Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Partitioning and placement algorithms
“Fixed partitioning”

Fixed partitioning: the simple scheme for managing available memory is to partition into regions with fixed boundaries. There are two alternatives for fixed partitioning, with equal-size or unequal-size partitions.

	Size of partitions	**Max loading size**	**Memory fragmentation**	**Degree of multi-programming**	**Placement algorithms**
equal-size partitions	M	<M	High	High	Single FIFO queue
unequal-size partitions	[M-N] with M<N	<N	Medium	Medium	Single / multiple FIFO queue(s)

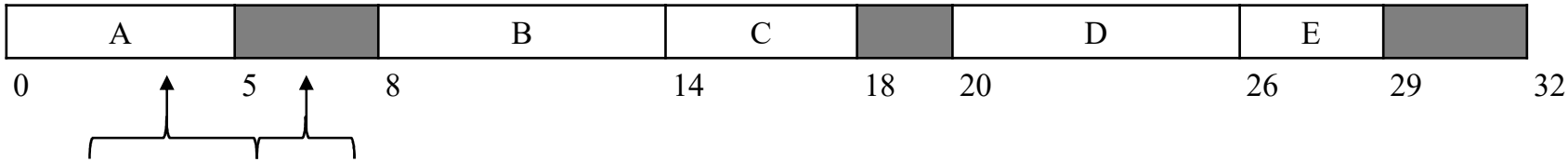


Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Partitioning and placement algorithms
“Memory management with bitmaps”

Memory management with bitmaps: with a bitmap, memory is divided into allocation units as small as a few words and as large as several kilobytes. Corresponding to each allocation units is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).

e.g. a part of memory with five processes and three holes, with the corresponding bitmap.



0-7	1	1	1	1	1	0	0	0
8-15	1	1	1	1	1	1	1	1
16-23	1	1	0	0	1	1	1	1
24-32	1	1	1	1	1	0	0	0

The size of a the allocation unit is an important design issue:

- the smaller the allocation unit, the larger the bitmap.
  e.g. with an allocation unit of 4 bytes (32 bits), the bitmap will take 1/33 of the memory.

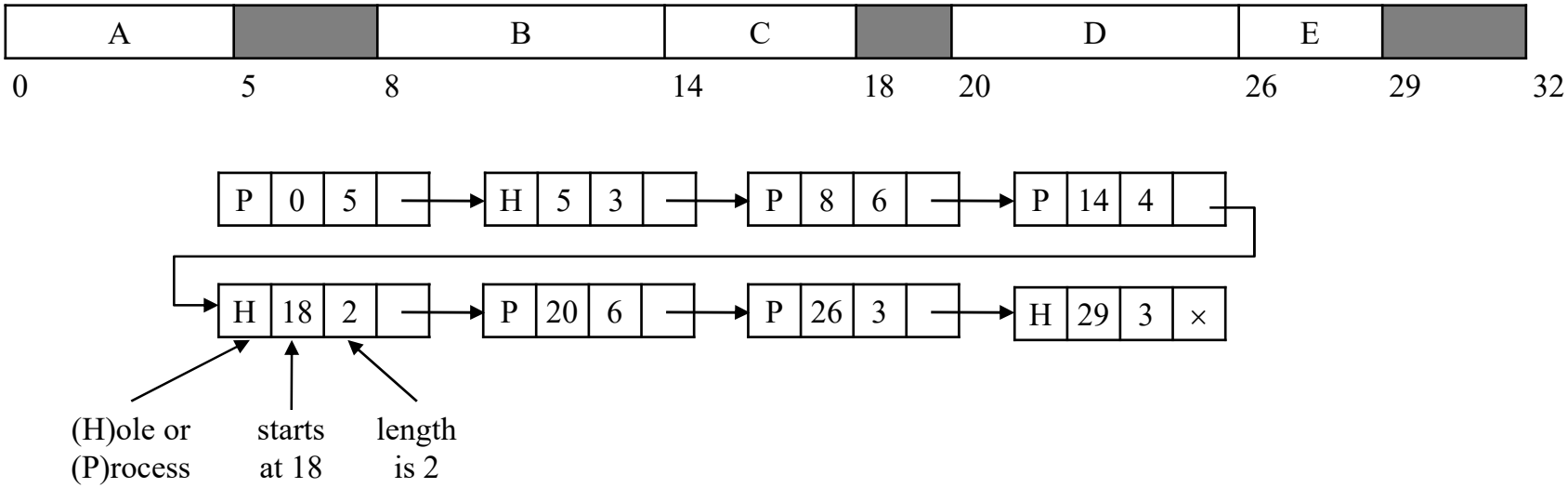
- if the allocation unit is chosen large, the bitmap will be smaller. However, appreciable memory may be wasted in the last unit of process due to the rounding effect.

Another problem is the placement algorithm. When it has been decided to bring k unit process into memory, we must search k consecutive 0 bits in the map, that is a slow operation.

Partitioning and placement algorithms
“Memory management with linked lists” (1)

Memory management with linked lists: another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes.

e.g. a part of memory with five processes and three holes, with the corresponding linked list



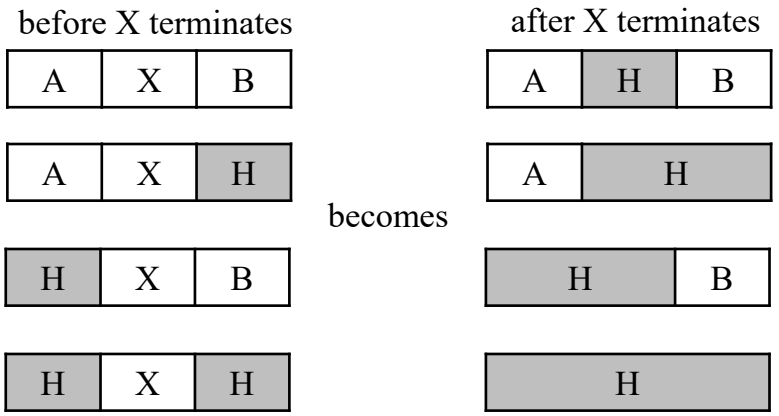
Here, the segment list is sorted by address. Sorting this way has the advantage that when a process terminates, updating the list is straightforward.

Partitioning and placement algorithms
“Memory management with linked lists” (2)

Memory management with linked lists: another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes.

A terminating process has two neighbors (except when it is at the very top or bottom of the memory). These may be either processes or holes, leading to four combinations.

H are holes,
A, B and X processes



Since the process table slot for the terminating process will normally point to the list entry for the process itself, it may be more convenient to have the list as a double-linked list.

Partitioning and placement algorithms
“Memory management with linked lists” (3)

Memory management with linked lists: another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes.

Several algorithms can be used to allocate memory for a created process.

Algorithms	Descriptions
first-fit	It scans along the list of segments from beginning until it finds a hole that is big enough.
next-fit	It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole.
best-fit	It searches the entire list, from beginning to end, and takes the smallest hole that is adequate rather than breaking up a big hole that might be needed later.
worst-fit	To get around the problem, one could think about worst fit, that is, always take the largest available hole.
quick-fit	It maintains separate lists for some of the most common sizes requested, to speed up the best-fist.

Statistical analysis reveals that the first-fit algorithm is not only the simplest one but usually the best and fastest as well.

Partitioning and placement algorithms
“Memory management with linked lists” (4)

Memory management with linked lists: another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes.

e.g. a memory with 8 processes (P1 to P8), P2 is the last allocated. What about P9 (16 MB) with the placement algorithms.

the last allocated block

initial state

Type	Size (MB)
H	8
P4	…
H	12
P1	…
H	22
P3	…
H	18
P5	…
P2	…
H	8
P7	…
H	6
P6	…
H	14
P8	…
H	36

first fit

Type	Size (MB)
H	8
P4	…
H	12
P1	…
P9	16
H	6
P3	…
H	18
P5	…
P2	…
H	8
P7	…
H	6
P6	…
H	14
P8	…
H	36

next fit

Type	Size (MB)
H	8
P4	…
H	12
P1	…
H	22
P3	…
H	18
P5	…
P2	…
H	8
P7	…
H	6
P6	…
H	14
P8	…
P9	16
H	20

best fit

Type	Size (MB)
H	8
P4	…
H	12
P1	…
H	22
P3	…
P9	16
H	2
P5	…
P2	…
H	8
P7	…
H	6
P6	…
H	14
P8	…
H	36

Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Partitioning and placement algorithms
"Buddy memory allocation" (1)

The Buddy memory allocation: fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning limits the number of process and may use space inefficiently. The dynamic partitioning is more complex to maintain and includes the overhead of compaction. An interesting compromise is the Buddy memory allocation, employing an hybrid partitioning.

blocks in a buddy system, memory blocks of holes are available of size 2^K, with $L \le K \le U$ (L, U are the smallest and largest sizes blocks respectively).

init to begin, the entire space is treated as a single block hole of size $2^{K=U}$.

request at any time, the buddy system maintains a list of holes (unallocated blocks) of each size 2^i:
(1) a hole may be removed from the $i+1$ list by splitting it in half to create two buddies of size 2^i in the i list.
(2) whenever a pair of buddies on the i list becomes unallocated, they are removed from that list and coalesced into a single block on the $i+1$ list of size 2^{i+1}.
(3) presented a request for an allocation of size s such that $2^{i-1} \le s \le 2^i$, the following recursive algorithm applies.

```
get hole of size i
  if i equals U+1 or L-1                    } failure case
    failure
  if the list i is empty                    } recursive search
    get hole of size i+1
    split hole into buddies
    put buddies on the i list
  take first hole on the i list             } access case
```

Partitioning and placement algorithms
“Buddy memory allocation” (2)

The Buddy memory allocation: fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning limits the number of process and may use space inefficiently. The dynamic partitioning is more complex to maintain and includes the overhead of compaction. An interesting compromise is the Buddy memory allocation, employing an hybrid partitioning.

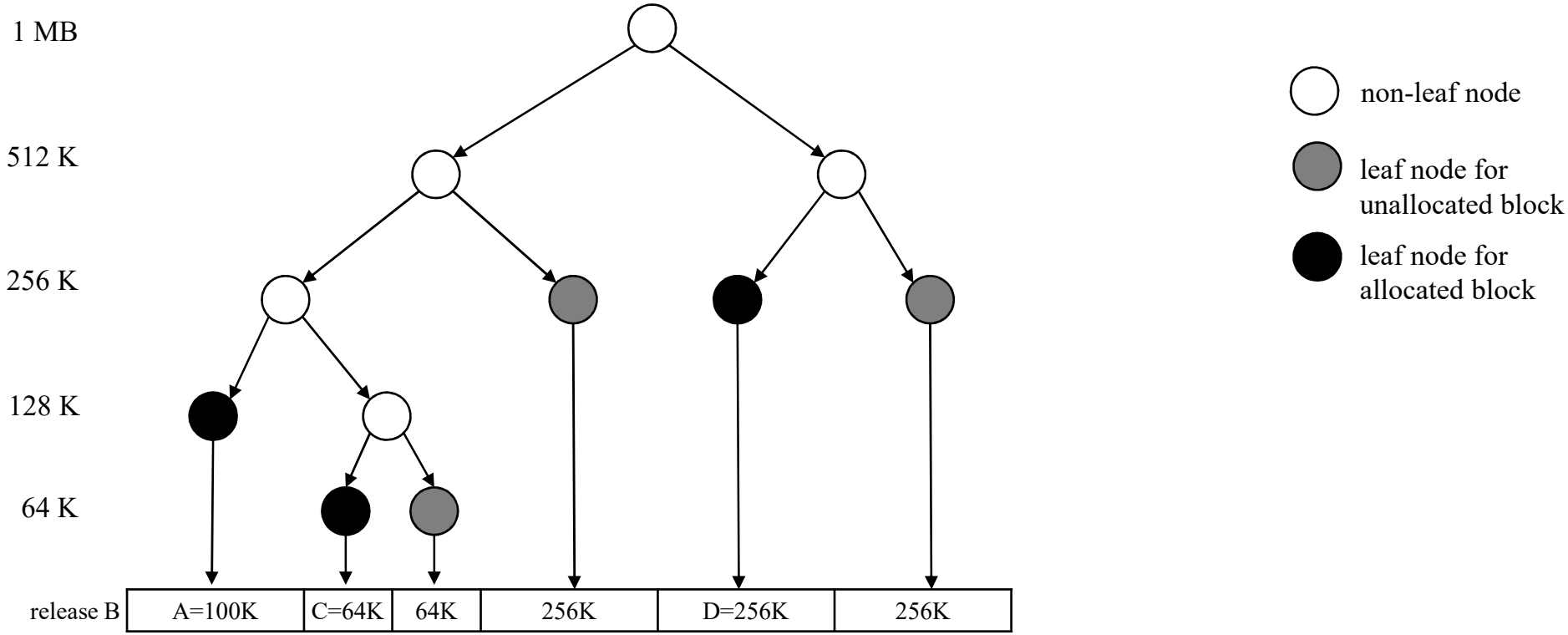
e.g. a 1MB memory with $L=6 \leq K \leq U=10$, considered blocks are 64K, 128K, 256K, 512K, 1MB.

1 MB block	1 MB					
request A = 100K	A=100K	128K		256K	512K	
request B = 240 K	A=100K	128K		B=240K	512K	
request C = 64 K	A=100K	C=64K	64K	B=240K	512K	
request D = 256K	A=100K	C=64K	64K	B=240K	D=256K	256K
release B	A=100K	C=64K	64K	256K	D=256K	256K
release A	128K	C=64K	64K	256K	D=256K	256K
request E = 75K	E=75K	C=64K	64K	256K	D=256K	256K
release C	E=75K	128K		256K	D=256K	256K
release E	512K				D=256K	256K
release D	1MB					

Partitioning and placement algorithms
“Buddy memory allocation” (3)

The Buddy memory allocation: fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning limits the number of process and may use space inefficiently. The dynamic partitioning is more complex to maintain and includes the overhead of compaction. An interesting compromise is the Buddy memory allocation, employing an hybrid partitioning.

e.g. a 1MB memory with $L=6 \leq K \leq U=10$, considered blocks are 64K, 128K, 256K, 512K, 1MB.



Operating Systems
“Memory Management”

1. Introduction
2. Contiguous memory allocation
    - 2.1. Partitioning and placement algorithms
    - 2.2. Memory fragmentation and compaction
    - 2.3. Process swapping
    - 2.4. Loading, address binding and protection
3. Simple paging and segmentation
    - 3.1. Paging, basic method
    - 3.2. Segmentation

Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Memory fragmentation and compaction (1)

External fragmentation: with dynamic partitioning, as processes are loaded and removed from memory, the free memory space could be broken into little pieces. This phenomenon is known as external fragmentation of memory.

e.g. initial state

Type	Size (MB)
H	56
	56

(a) P1 (20MB) loaded

Type	Size (MB)
P1	20
H	36
Total	56
F Rate	0

(b) P2 (14MB) loaded

Type	Size (MB)
P1	20
P2	14
H	22
Total	56
F Rate	0

(c) P3 (18MB) loaded

Type	Size (MB)
P1	20
P2	14
P3	18
H	4
Total	56
F Rate	0

(d) P2 leaves

Type	Size (MB)
P1	20
H	14
P3	18
H	4
Total	56
F Rate	0,22

(e) P4 (8MB) loaded

Type	Size (MB)
P1	20
P4	8
H	6
P3	18
H	4
Total	56
F Rate	0,4

(f) P1 leaves

Type	Size (MB)
H	20
P4	8
H	6
P3	18
H	4
Total	56
F Rate	0,33

(g) P5 (14MB) loaded

Type	Size (MB)
P1	14
H	6
P4	8
H	6
P3	18
H	4
Total	56
F Rate	0,62

One way to compute fragmentation rate of memory F Rate is

$$F\ Rate = 1 - \frac{\max_{\forall i}(H_i)}{\sum_{\forall i} H_i}$$

A F Rate close to 1 corresponds to a strong fragmentation

Memory fragmentation and compaction (2)

External fragmentation: with dynamic partitioning, as processes are loaded and removed from memory, the free memory space could be broken into little pieces. This phenomenon is known as external fragmentation of memory.

50-percent rule: memory fragmentation depends of the exact sequence of incoming processes, the considered system and placement algorithms (e.g. first, next and best fit). Statistical analysis reveals that even with some optimization, given N allocated blocks 0,5N block will be lost to fragmentation. That is, one third of memory $\frac{0{,}5N}{1N + 0{,}5N}$ may be unusable. This property is known as the 50-percent rule.

Memory fragmentation and compaction (3)

External fragmentation: with dynamic partitioning, as processes are loaded and removed from memory, the free memory space could be broken into little pieces. This phenomenon is known as external fragmentation of memory.

Memory compaction: when internal fragmentation results in multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is knows as memory compaction. It requires lot of CPU time.

e.g. on a 1GB machine that can copy 4 bytes in 20 ns, it would take 5 s to compact all memory.

Memory fragmentation and compaction (4)

Internal fragmentation: memory allocated to a process may be larger than the request memory, two cases occur.

Track overhead: the general approach is to break the physical memory into fixed size blocks and allocate memory in units based on block size. The difference between these two numbers is internal fragmentation of memory.

e.g.

initial state

Type	Size (bytes)
P1	…
H	16384
P2	…

P3 (16382 bytes)

Type	Size (bytes)
P1	…
P3	16382
H	2
P2	…

case without allocation of blocks of fixed-size, a hole of size 2 bytes is produced

P3 (16382 bytes)

Type	Size (bytes)
P1	…
P3	16384
P2	…

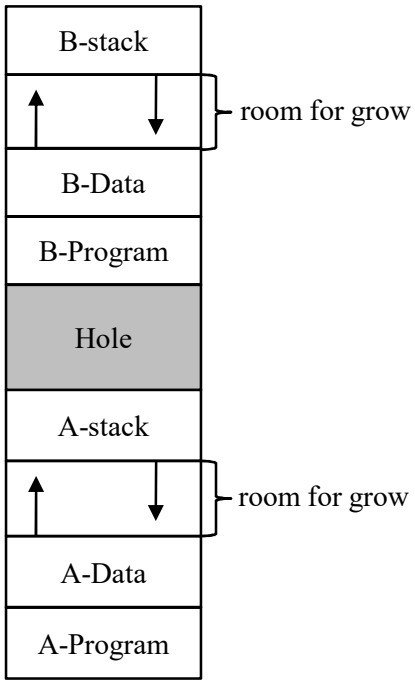
case with allocation of blocks of fixed-size 256 bytes, 64 blocks are allocated to P3 internal fragmentation is then 16384-16382 = 2 bytes

Memory fragmentation and compaction (5)

Internal fragmentation: memory allocated to a process may be larger than the request memory, two cases occur.

Extra memory allocation: when it is expected that the most processes will grow as they run, it is probably a good idea to allocate a little extra memory.

e.g. a process with two growing segments, a data segment to being used as a heap for variables that are dynamically allocated and released, and a stack segment for the local normal variables and return addresses. An arrangement is to place the stack at the top that is growing downward, and the data segment just beyond the program text that is growing upward.



Operating Systems
“Memory Management”

1. Introduction
2. Contiguous memory allocation
    - 2.1. Partitioning and placement algorithms
    - 2.2. Memory fragmentation and compaction
    - 2.3. Process swapping
    - 2.4. Loading, address binding and protection
3. Simple paging and segmentation
    - 3.1. Paging, basic method
    - 3.2. Segmentation

Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Process swapping (1)

Swapping is a strategy to deal with memory overload, it consists in bringing each process in its entirely, running if for a while, then putting it back on the disk.

e.g. a 88MB memory system, we consider the following events with swapping.

(a) P1 (32MB) loaded

Type	Size (MB)
P1	32
H	56
Total	88

(b) P2 (16MB) loaded

Type	Size (MB)
P1	32
P2	16
H	40
Total	88

(c) P3 (32MB) loaded

Type	Size (MB)
P1	32
P2	16
P3	32
H	8
Total	88

(d) P1 swapped out from disk

Type	Size (MB)
H	32
P2	16
P3	32
H	8
Total	88

(e) P4 (16MB) loaded

Type	Size (MB)
P4	16
H	16
P2	16
P3	32
H	8
Total	88

(f) P2 (16MB) leaves

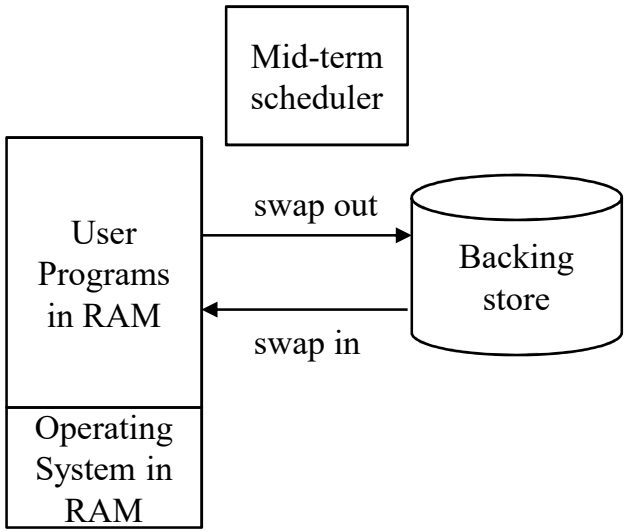
Type	Size (MB)
P4	16
H	32
P3	32
H	8
Total	88

(g) P1 swapped in at a different location

Type	Size (MB)
P4	16
P1	32
P3	32
H	8
Total	88

Process swapping (2)

Swapping is a strategy to deal with memory overload, it consists in bringing each process in its entirely, running if for a while, then putting it back on the disk.



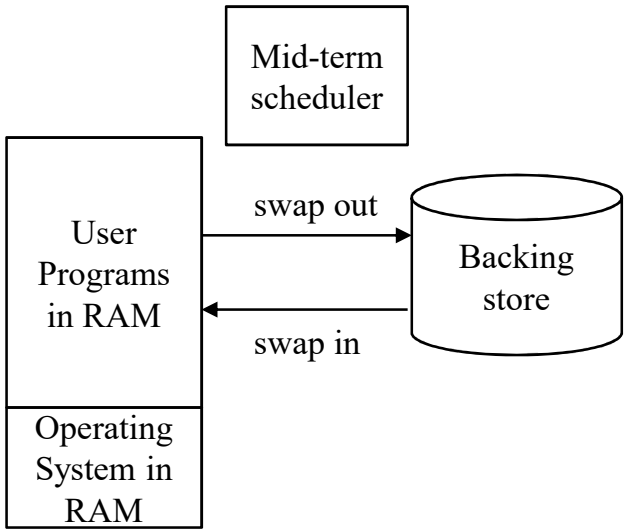
Mid-term scheduler removes processes from main memory (if full) and places them on secondary memory (such as a disk drive) and vice versa.

	criterion
swap out	I/O blocked processes, huge memory processes, ready processes with low-level priorities, etc.
swap in	no ready process in the ready queue, processes in the ready suspend queue with priorities higher to the ones in the ready queue, processes for which the blocking event will occur soon, etc.

Backing store is a fast disk that must be large enough to accommodate copies of all memory images. The system maintains suspend queues consisting in all processes whose are on the backing store. The context switch time in such a swapping system is fairly high.

Process swapping (3)

Swapping is a strategy to deal with memory overload, it consists in bringing each process in its entirely, running if for a while, then putting it back on the disk.



The context switch time in such a swapping system is fairly high.

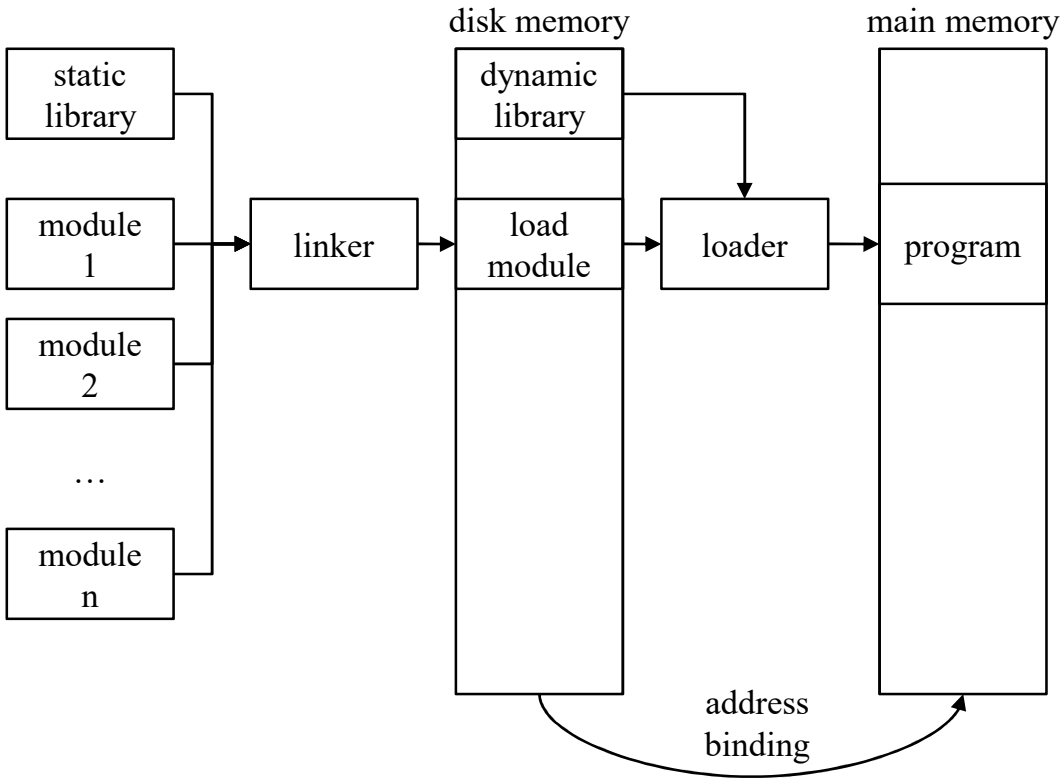
e.g. a user process of 10 MB and backing store with a transfer rate of 40 MB.s^{-1}
- The standard swap operation is 10/40 = ¼ s = 250 ms.
- Assuming no head seeks are necessary, and a latency of 8 ms, the swap time is 258 ms.
- The total swap time (in and out) is then 516 ms.

Operating Systems
“Memory Management”

1. Introduction
2. Contiguous memory allocation
    - 2.1. Partitioning and placement algorithms
    - 2.2. Memory fragmentation and compaction
    - 2.3. Process swapping
    - 2.4. Loading, address binding and protection
3. Simple paging and segmentation
    - 3.1. Paging, basic method
    - 3.2. Segmentation

Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Loading, address binding and protection
“Loading and address binding” (1)

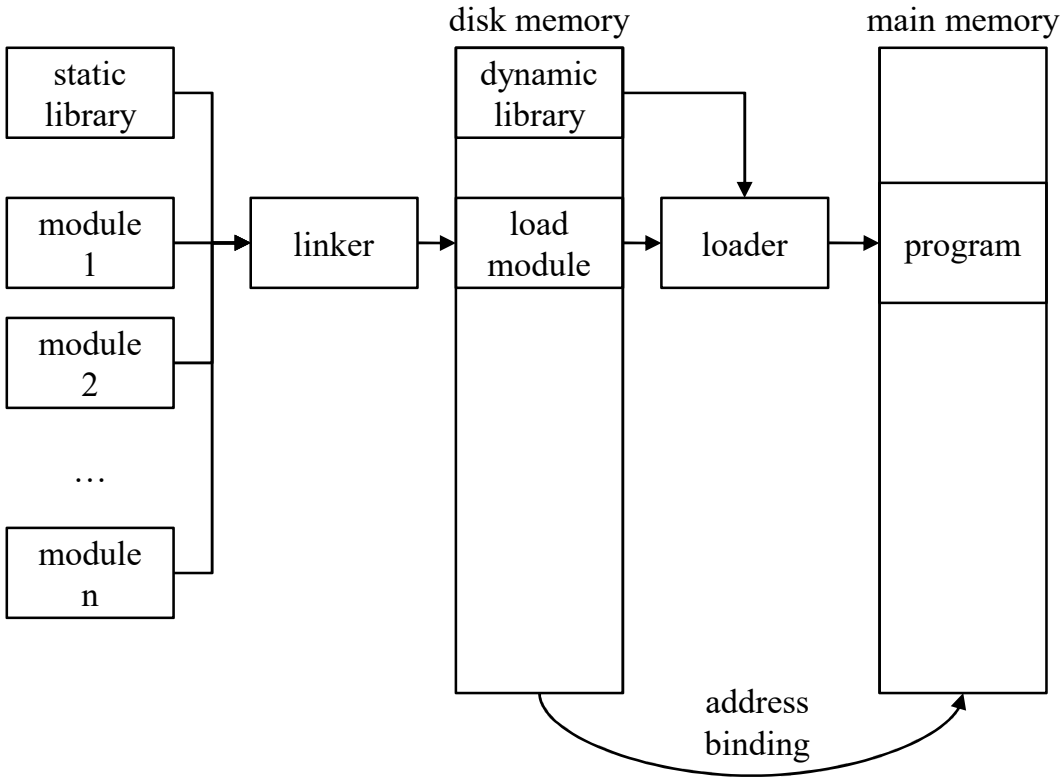


Loading: the first step is to load a program into the main memory and to create a process image. Any program consists in compiled modules in an object-code form linked together or to library routines. The library routines can be incorporated into the program or referenced as a shared code to be call at the run time.

Address space is the set of addresses that a process can use to address memory. Each process has its own address space, independent of those belonging to other processes.

Address binding: addresses may be represented in different way between the disk and main memory spaces. Address binding is a mapping from one address space to another.

Loading, address binding and protection
“Loading and address binding” (2)

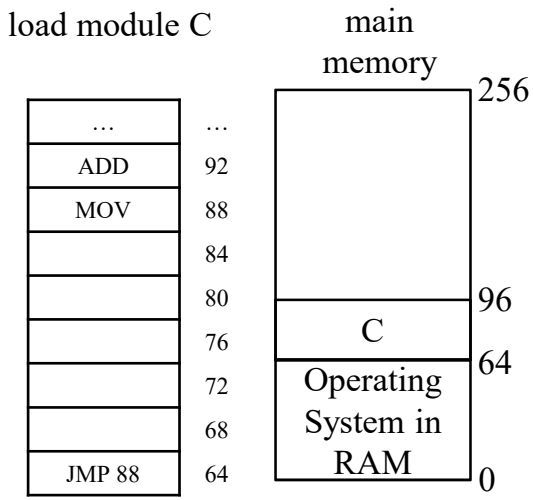


Loading mode	**Binding time**
Absolute loading	At the programming time
	At the compile or assembly time
Relocatable loading	at the load time
Dynamic loading	at the run time

Loading, address binding and protection
“Loading and address binding” (3)

Absolute loading: an absolute loader requires that a given load module always be loaded into the same location in main memory. Thus, in the load module presented to the loader, all address references must be specific to main memory addresses.

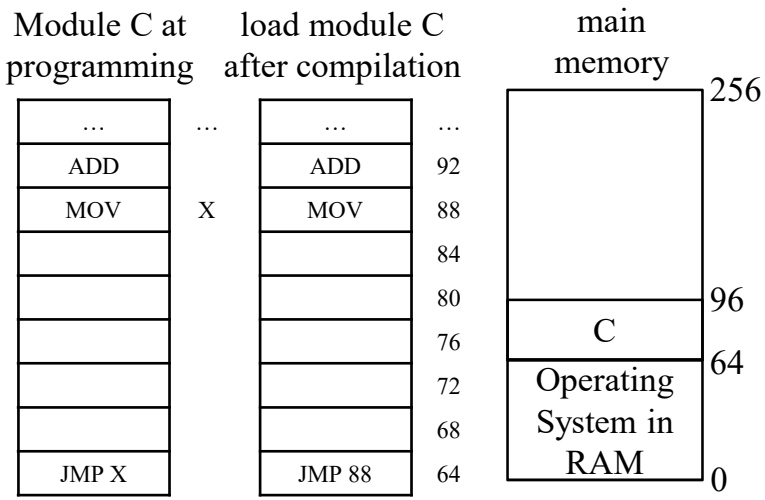
Binding time	**Function**
programming time	All actual physical addresses are directly specified by the programmer in the program itself.



Loading, address binding and protection
“Loading and address binding” (4)

Absolute loading: an absolute loader requires that a given load module always be loaded into the same location in main memory. Thus, in the load module presented to the loader, all address references must be specific to main memory addresses.

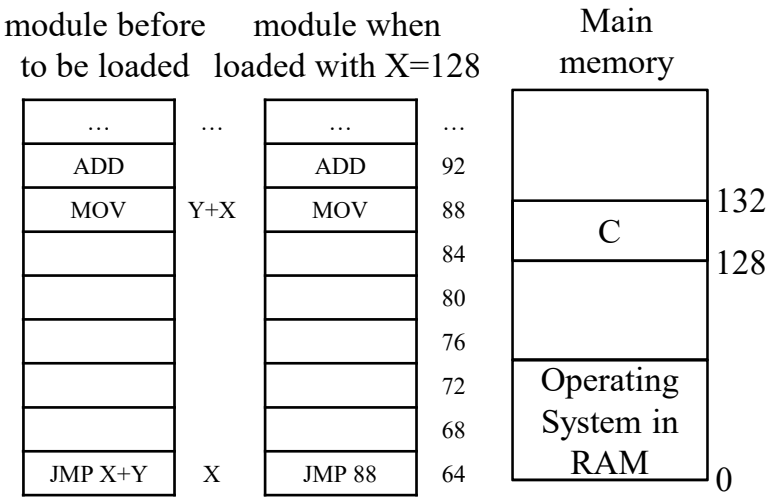
Binding time	**Function**
compile or assembly time	The program contains symbolic address references, and these are converted to actual physical addresses by compiler or assembler.



Loading, address binding and protection
“Loading and address binding” (5)

Relocatable loading: when many programs share a main memory, it may not desirable to decide ahead of time into which region of memory a particular module should be loaded. It is better to make that decision at load time, thus we need a load module that can be located anywhere in main memory.

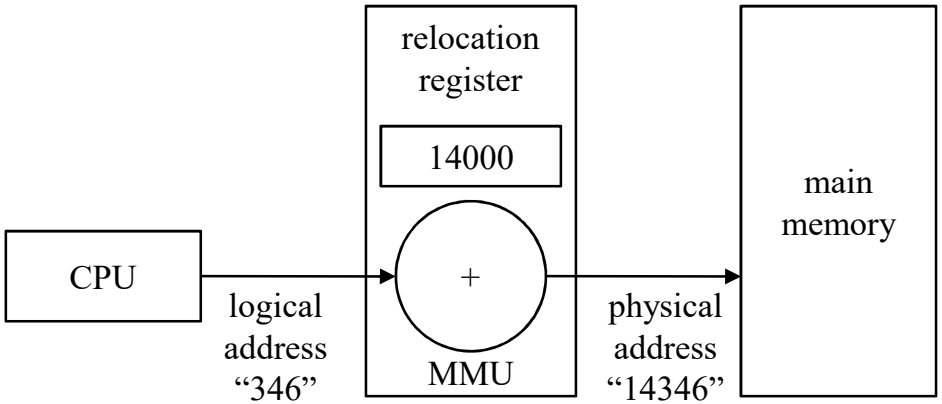
Binding time	**Function**
load time	The compiler or assembler produces relative addresses. The loader translates these to absolute addresses at the time of program loading. The load module must include information about that tells the loader where the addresses references are. This set of information is prepared by the compiler and referred as the relocation directory.



Loading, address binding and protection
“Loading and address binding” (6)

Dynamic runtime loading: to maximize memory utilization, we would like to be able to swap the process image back into different locations at different times, that involves to update the load module at every swap. The relocatable loaders cannot support this scheme. The alternative is to defer the calculation of the absolute address until it is actually needed at run time.

Binding time	**Function**
run time	The loaded program retains relative addresses, these are converted dynamically to absolute addresses by processor hardware.



Logical address space is the set of all logical addresses generated by a program, in the range *[0,max]*.

Relocation register: the value *R* in the relocation register is added to every logical address to obtain the corresponding physical address.

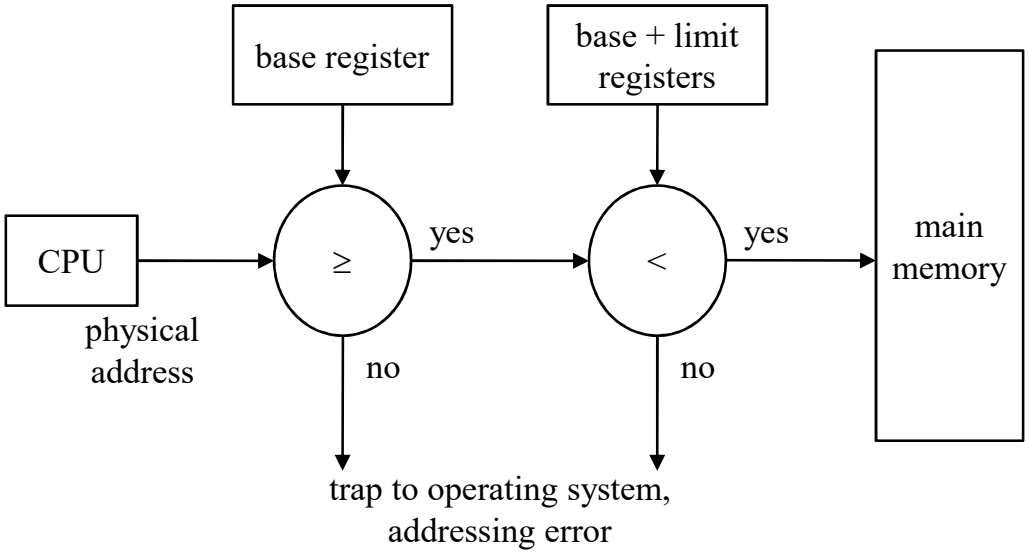
Physical address space: is the set of all physical addresses corresponding to the logical addresses, in the range *[R+0,R+max]*.

Memory Management Unit (MMU): the run time mapping from logical to physical addresses is done by an hardware device called MMU.

Loading, address binding and protection
"Address protection" (1)

Address protection: we need to make sure that every process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. Protection of memory space is accomplished by having a special CPU hardware.

Non-dynamic address protection (absolute and relocatable loading)



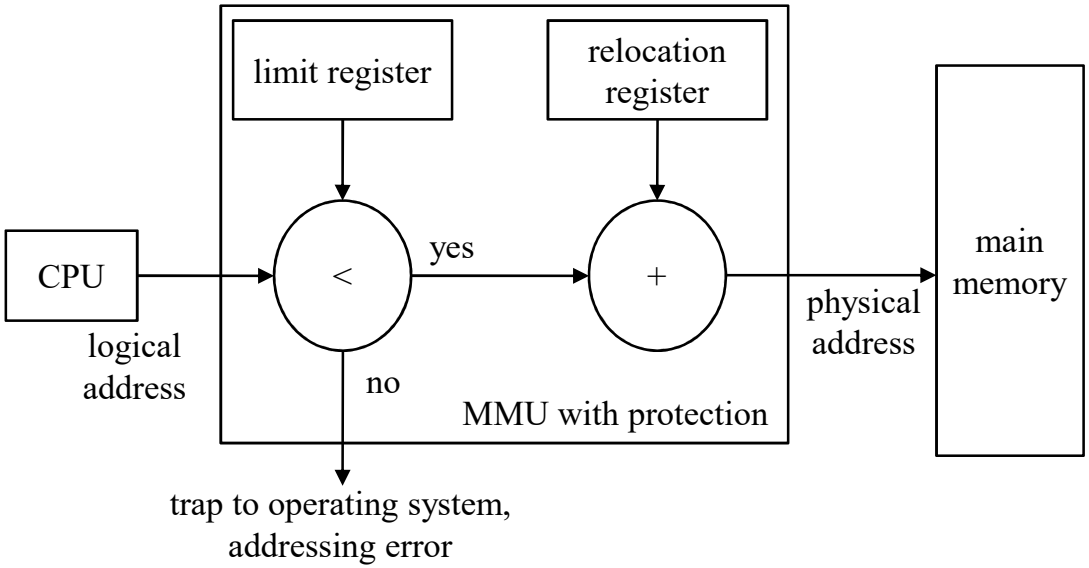
Base register holds the smallest legal physical memory address.

Limit (or bound) register specifies the size of the range e.g. if the base register holds 300 040 and limit register is 120 900, then the program can legally access all addresses from 300 040 to 420 940.

Loading, address binding and protection
“Address protection” (2)

Address protection: we need to make sure that every process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. Protection of memory space is accomplished by having a special CPU hardware.

Dynamic address protection (dynamic runtime loading)



The protection must be tuned with the relocation register in the case of MMU.

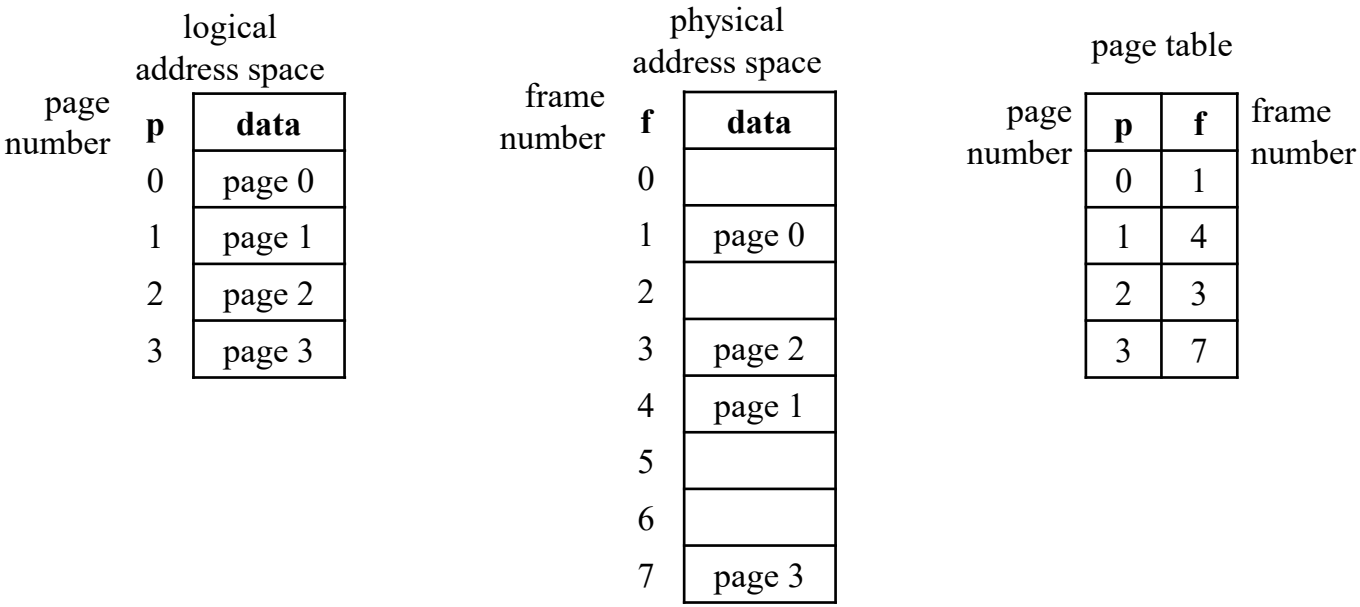
Operating Systems
“Memory Management”

1. Introduction
2. Contiguous memory allocation
    - 2.1. Partitioning and placement algorithms
    - 2.2. Memory fragmentation and compaction
    - 2.3. Process swapping
    - 2.4. Loading, address binding and protection
3. Simple paging and segmentation
    - 3.1. Paging, basic method
    - 3.2. Segmentation

Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Simple paging and segmentation
“Paging, basic method” (1)

Paging: is a memory-management scheme that permits the physical address space of a process to be noncontiguous. The basic method for implementing paging involves breaking logical memory into fixed-sized blocks called **pages** and breaking physical memory into blocks of the same size called **frame**. A **page table** contains the base address of each page in physical memory. When a process is to be executed, its pages are loaded into any available memory frames from the backing store.



Simple paging and segmentation
“Paging, basic method” (2)

Paging: is a memory-management scheme that permits the physical address space of a process to be noncontiguous. The basic method for implementing paging involves breaking logical memory into fixed-sized blocks called **pages** and breaking physical memory into blocks of the same size called **frame**. A **page table** contains the base address of each page in physical memory. When a process is to be executed, its pages are loaded into any available memory frames from the backing store.

e.g. 4 processes are loaded in main memory using paging: A, C (4 pages), B (3 pages) and D (5 pages).

frame number	(t_0) initial state	(t_1) A loaded	(t_2) B loaded	(t_3) C loaded	(t_4) B swapped out	(t_5) D loaded
0		A.0	A.0	A.0	A.0	A.0
1		A.1	A.1	A.1	A.1	A.1
2		A.2	A.2	A.2	A.2	A.2
3		A.3	A.3	A.3	A.3	A.3
4			B.0	B.0		D.0
5			B.1	B.1		D.1
6			B.2	B.2		D.2
7				C.0	C.0	C.0
8				C.1	C.1	C.1
9				C.2	C.2	C.2
10				C.3	C.3	C.3
11						D.3
12						D.4
13						
14						

Simple paging and segmentation
“Paging, basic method” (3)

Paging: is a memory-management scheme that permits the physical address space of a process to be noncontiguous. The basic method for implementing paging involves breaking logical memory into fixed-sized blocks called **pages** and breaking physical memory into blocks of the same size called **frame**. A **page table** contains the base address of each page in physical memory. When a process is to be executed, its pages are loaded into any available memory frames from the backing store.

e.g. 4 processes are loaded in main memory using paging with A, C (4 pages), B (3 pages) and D (5 pages).

(t_5) D loaded

0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

data structure at t_5

page table A

p	f
0	0
1	1
2	2
3	3

page table B

p	f
0	-
1	-
2	-

page table C

p	f
0	7
1	8
2	9
3	10

page table D

p	f
0	4
1	5
2	6
3	11
4	12

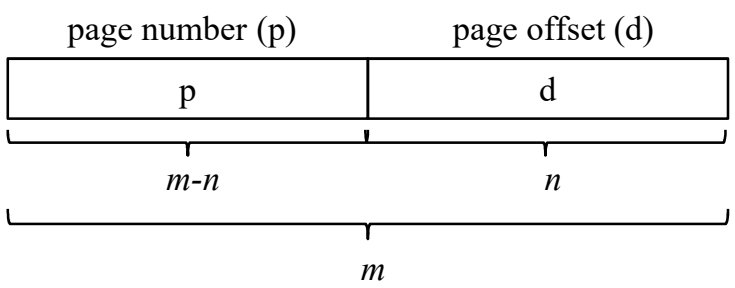
free frame list

f
13
14

Simple paging and segmentation
“Paging, basic method” (4)

Address binding: every address generated by the CPU is dived in two parts, a page number (p) and a page offset (d). The page number is used as index into the page table, and the page table contains the base address of each page in physical memory. This base address is combined with the page offset (d) to define the physical memory address that is sent to the memory unit.

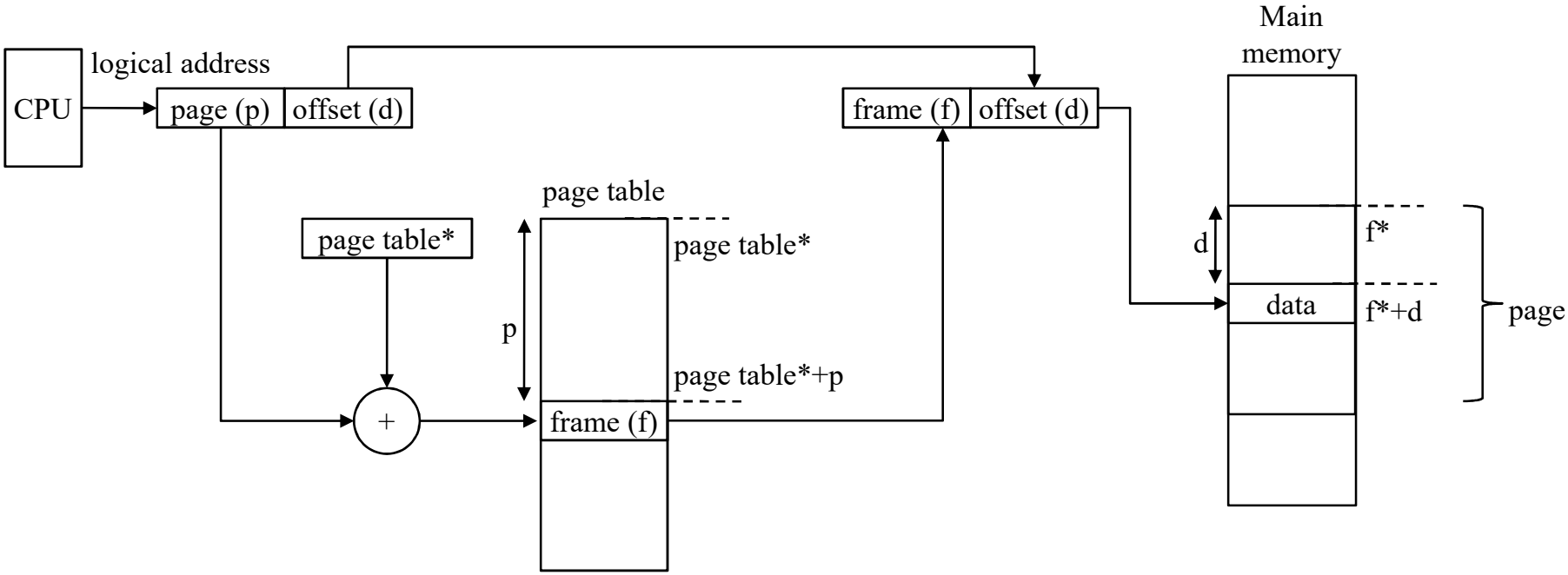
The size of a page is typically a power of 2 that makes the translation of a logical address into a page number and page offset particularly easy e.g. if the size of logical address space is 2^m, and a page size is 2^n addressing units, then the high order m-n bits designates the page number. As a result, the addressing scheme is transparent to programmer, assembler and linker.



m	number of bits to encode the logical address
n	number of bit to encode the page offset
m-n	number of bits to encode the page number

Simple paging and segmentation
“Paging, basic method” (5)

Address binding: every address generated by the CPU is dived in two parts, a page number (p) and a page offset (d). The page number is used as index into the page table, and the page table contains the base address of each page in physical memory. This base address is combined with the page offset (d) to define the physical memory address that is sent to the memory unit.



Simple paging and segmentation
“Paging, basic method” (6)

Address binding: every address generated by the CPU is dived in two parts, a page number (p) and a page offset (d). The page number is used as index into the page table, and the page table contains the base address of each page in physical memory. This base address is combined with the page offset (d) to define the physical memory address that is sent to the memory unit.

e.g. a memory of 16 bytes with a page size of 4 bytes, consider the logical address 0101.

logical address space

address	p	d	data
0000	0	0	a
0001		1	b
0010		2	c
0011		3	d
0100	1	0	e
0101		1	f
0110		2	g
0111		3	h
1000	2	0	i
1001		1	j
1010		2	k
1011		3	l
1100	3	0	m
1101		1	n
1110		2	o
1111		3	p

p	f
0	1
1	3
2	0
3	2

page table

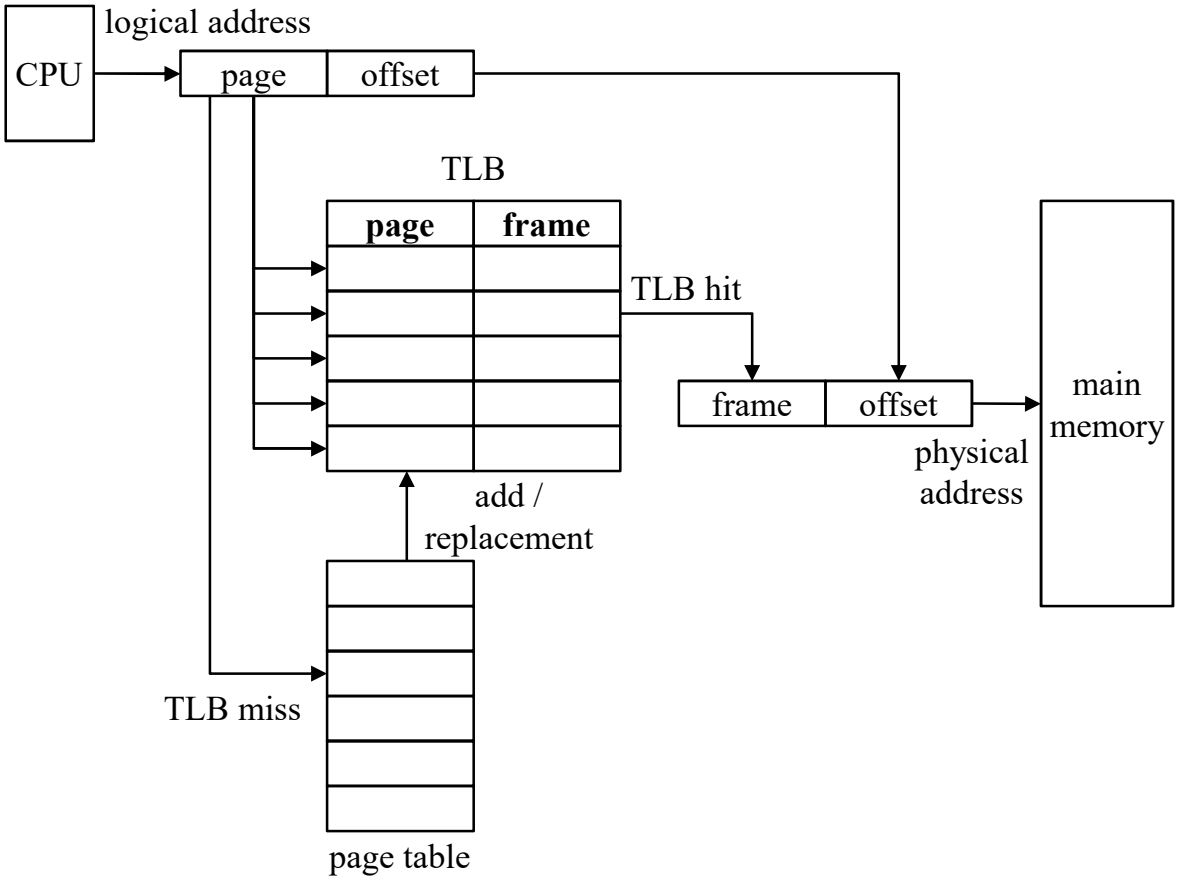
the logical address
0101 (page 1, offset 1)
corresponds to the physical address
1101 (frame 3, offset 1)

physical address space

address	f	d	data
0000	0	0	i
0001		1	j
0010		2	k
0011		3	l
0100	1	0	a
0101		1	b
0110		2	c
0111		3	d
1000	2	0	m
1001		1	n
1010		2	o
1011		3	p
1100	3	0	e
1101		1	f
1110		2	g
1111		3	h

Simple paging and segmentation
“Paging, basic method” (7)

Transaction look-aside buffer (TLB): in principle every binding from logical address space to physical address space using paging causes two physical memory accesses (1) to fetch the appropriate page table entry (2) to fetch the desired data. This will have the effect of doubling the memory access time. To overcome this problem, we use a special high speed cache for page table entries called Transaction look-aside buffer (TLB).



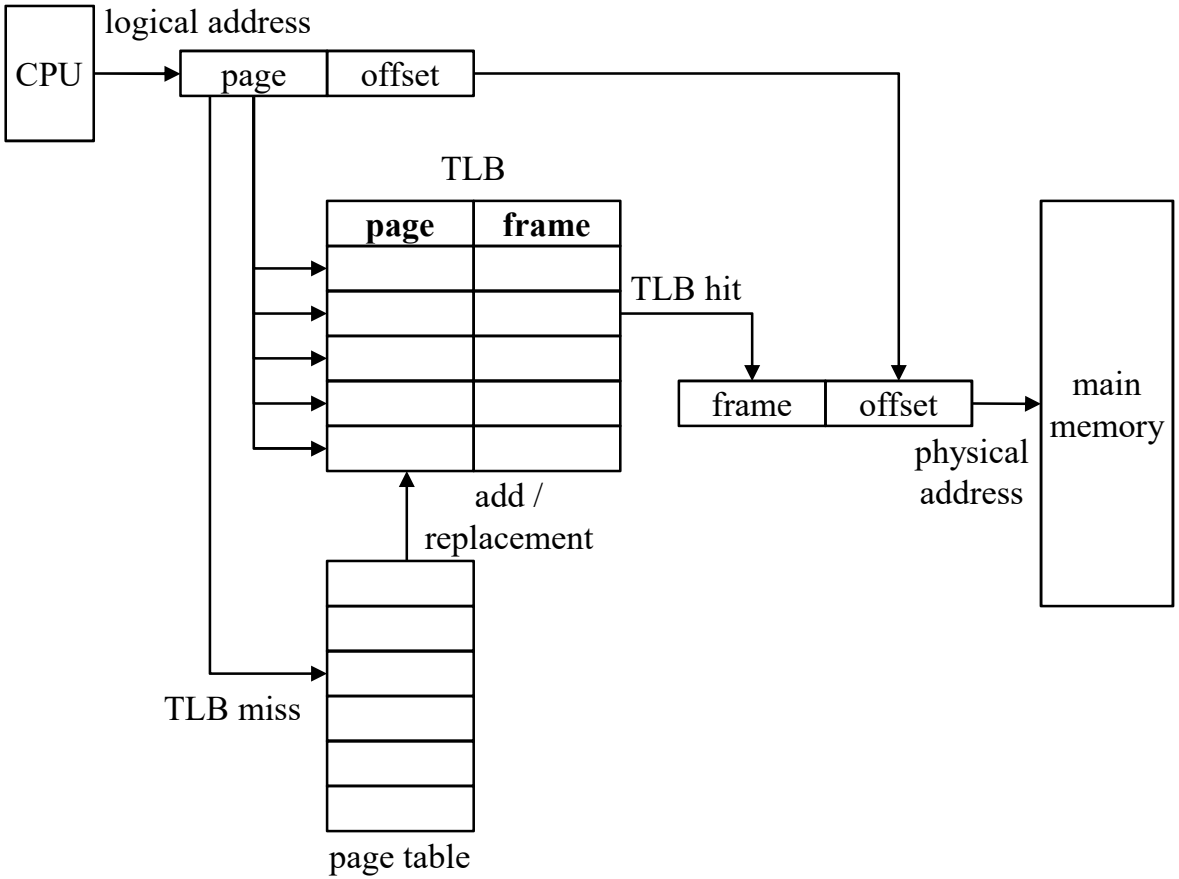
TLB is an associative high speed memory. Each entry in the TLB consists of two parts **{key; value}**.

When an **item** (i.e. a page number) is presented, it is compared with all the **keys** simultaneously. This technique is referred as **associative mapping**. If **item == key**, the corresponding **value** (i.e. offset) is returned.

The search is fast and hardware, however, is expensive. Typically, the number of entries in a TLB is small e.g. 64 to 1024 entries.

Simple paging and segmentation
“Paging, basic method” (8)

Transaction look-aside buffer (TLB): in principle every binding from logical address space to physical address space using paging causes two physical memory accesses (1) to fetch the appropriate page table entry (2) to fetch the desired data. This will have the effect of doubling the memory access time. To overcome this problem, we use a special high speed cache for page table entries called Transaction look-aside buffer (TLB).



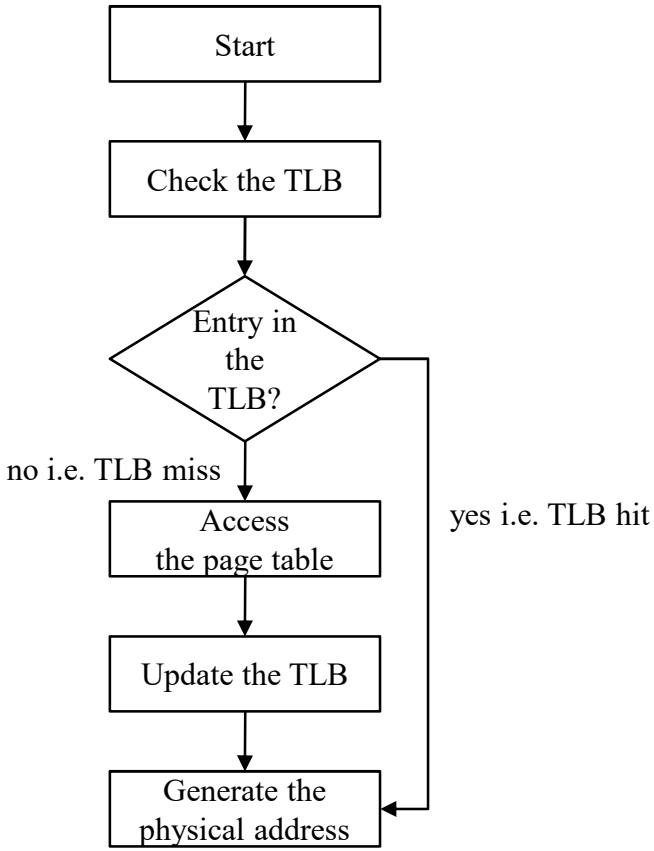
TLB hit: it occurs when the page number is in the TLB.

TLB miss: if the page number is not in the TLB, we add the page number and frame so that they will be found quickly on the next reference. If the TLB is already full of entries, OS must select one entry for replacement (LRU or FIFO policy, etc).

wired down: some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB.

Simple paging and segmentation
"Paging, basic method" (9)

Transaction look-aside buffer (TLB): in principle every binding from logical address space to physical address space using paging causes two physical memory accesses (1) to fetch the appropriate page table entry (2) to fetch the desired data. This will have the effect of doubling the memory access time. To overcome this problem, we use a special high speed cache for page table entries called Transaction look-aside buffer (TLB).



TLB hit: it occurs when the page number is in the TLB.

TLB miss: if the page number is not in the TLB, we add the page number and frame so that they will be found quickly on the next reference. If the TLB is already full of entries, OS must select one entry for replacement (LRU or FIFO policy, etc).

wired down: some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB.

Simple paging and segmentation
“Paging, basic method” (10)

Shared pages: an advantage of paging is the possibility of sharing common code. This can appear with reentrant code (or pure code) that is a non-self-modifying code i.e. it never changes during execution.

e.g. consider a system that supports three users, each of whom executes a text editor. It the text editor consists of 150 KB of code and 50 KB of data space, we need 600 KB to support the three users. If the code is reentrant code, it can be shared as:

Process 1

page number p	Data
0	editor 1
1	editor 2
2	editor 3
3	data 1

p	f
0	3
1	4
2	6
3	1

frame number

Process 2

page number p	Data
0	editor 1
1	editor 2
2	editor 3
3	data 2

p	f
0	3
1	4
2	6
3	7

frame number

Process 3

page number p	Data
0	editor 1
1	editor 2
2	editor 3
3	data 3

p	f
0	3
1	4
2	6
3	2

frame number

Process 4

page number p	Data
0	editor 1
1	editor 2
2	editor 3
3	data 4

p	f
0	3
1	4
2	6
3	9

frame number

	loaded
0	
1	data 1
2	data3
3	editor 1
4	editor 2
5	
6	editor 3
7	data 2
8	
9	data 4
10	
11	

Simple paging and segmentation
"Paging, basic method" (11)

Address protection: address (or memory) protection in a paged environment is accomplished by protection bits associated to each frame. Normally, these bits are kept in the page table.

valid bit (v): when this bit is set to valid, the associated page in the process's logical address space is thus a valid page. When the bit is set to invalid then the page is not in the process's logical address space.

e.g. in a system with m=14 bits address space (0 to 16383), using a page size of 2 KB, then n=11, m-n=3 (i.e. 8 pages). We have a process P of size 10438 bytes.

logical address space

p	data
0	page 0
1	page 1
2	page 2
3	page 3
4	page 4
5	page 5

physical address space

f	data
0	page 4
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3
8	page 5

page table P

p	f	v
0	1	true
1	4	true
2	3	true
3	7	true
4	0	true
5	8	true
6	×	false
7	×	false

valid bit

Accesses to addresses up to 12287 (6×2^{11}) are valid, only the addresses from 12288 to 16383 are not valid.

Because the program extends to the address from 10468 to 12288, only the references beyond that address are illegal. This problem results of the 2KB page size and reflects the internal fragmentation of paging.

Simple paging and segmentation
“Paging, basic method” (12)

Address protection: address (or memory) protection in a paged environment is accomplished by protection bits associated to each frame. Normally, these bits are kept in the page table.

> **r/w bit:** defines a page to be read-write or read only; we can easily expand this approach to provide a finer level of protection by considering execute-only sate.

Operating Systems
“Memory Management”

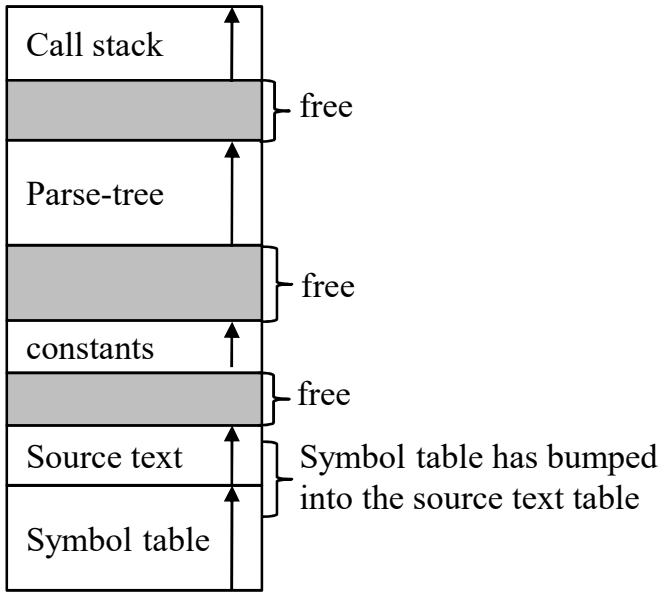
1. Introduction
2. Contiguous memory allocation
    - 2.1. Partitioning and placement algorithms
    - 2.2. Memory fragmentation and compaction
    - 2.3. Process swapping
    - 2.4. Loading, address binding and protection
3. Simple paging and segmentation
    - 3.1. Paging, basic method
    - 3.2. Segmentation

Methods	Partitioning	Placement algorithms	Fragmentation / compaction	Swapping	Address binding & protection	Layer
Fixed partitioning	contiguous / fixed / complete	searching algorithms	yes / no	yes	no / yes (MMU)	OS kernel
Memory management with bitmap	contiguous / dynamic / complete		yes / yes			
Memory management with linked lists	contiguous / dynamic / complete					
Buddy memory allocation	contiguous / hybrid / complete		yes / no			
Simple paging and segmentation	noncontiguous / dynamic / complete		yes / no		yes (TLB)	programs / services

Simple paging and segmentation
"Segmentation" (1)

One vs. separated address spaces: logical addressing discussed so far is one-dimensional because the logical addresses go from zero to some maximum. For many problems, having separate logical address space may be much more better.

e.g. a compiler has tables that are built up as compilation proceeds, and each of them grows continuously. In a one dimensional memory, these tables are allocated contiguous chunks of logical addresses. Consider what happens if a program has a much larger than usual number of variables; the chunk of address space allocated for a table may fill up.



Some approaches to deal with are:

(1) the compiler could simply issue a message saying that the compilation cannot continue.

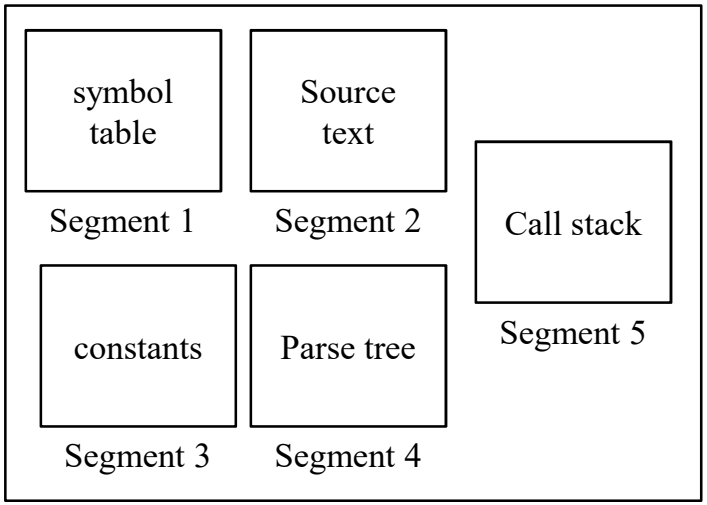
(2) to play "Robin Hood", tacking space from the tables with an excess of room and giving it to the tables with little room. This is a nuisance at best and a great deal of tedious, unrewarding work, at worst.

Simple paging and segmentation
"Segmentation" (2)

Segmentation is a memory management scheme that supports independent address spaces called **segment**.

-Each segment consists in linear sequence of addresses, from 0 to some maximum (usually very large).
-The length of each segment may be anything from 0 to the maximum allowed.
-Different segments may have different lengths.
-Segment lengths may change during execution.
-With segments, a logical address consists of a two tuple <segment number, offset>.

e.g. a compiler has tables that are built up as compilation proceeds, and each of them grows continuously. Because each segment constitutes a separate address space, the different segments can grow independently without affecting each other.

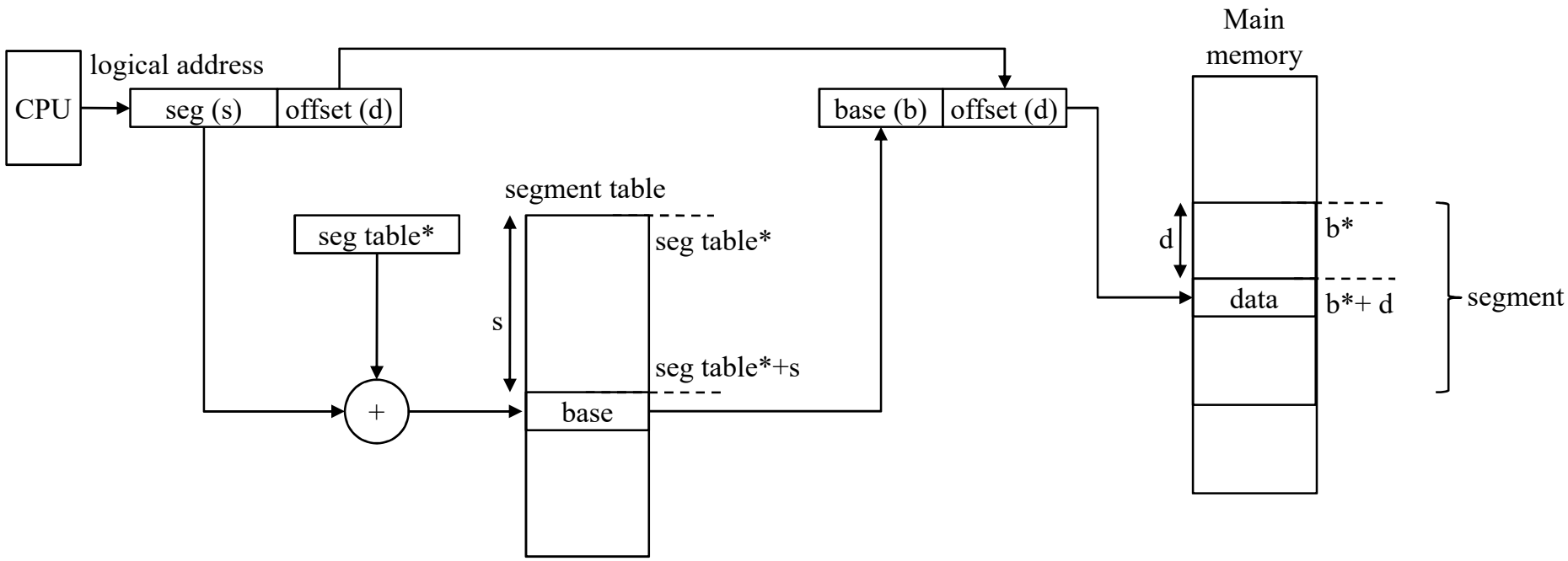


Logical address space

Simple paging and segmentation
"Segmentation" (3)

Segmentation: is a memory management scheme that supports independent address spaces called **segment**.

-Each segment consists in linear sequence of addresses, from 0 to some maximum (usually very large).
-The length of each segment may be anything from 0 to the maximum allowed.
-Different segments may have different lengths.
-Segment lengths may change during execution.
-With segments, a logical address consists of a two tuple <segment number, offset>.



Simple paging and segmentation
"Segmentation" (4)

Combined paging and segmentation: both paging and segmentation have their strengths. In combined paging / segmentation system, the user's address space is broken up into a number of segments, at the discretion of the programmer. Each segment is, in turn, broken up into a number of fixed-size page.

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

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