

Operating Systems

Processes & Resource management

Processes

- Program: a file stored on disk - static
 - Make source code
 - Compile it to an object file
 - Link object file(s) + libraries with run-time system = executable file
- Process: the live invocation of a program
 - Each has a unique PID (a number).

Execution of program

- Create necessary data structures
- Find the executable code on the disk
- Load it into memory
- Start execution
 - load starting address into program counter
 - run the program one instruction at a time

What if the same program is executed again

- Create a new process
 - Memory protection needed if multi-user
- No need to load the program as already there the new process⁴ has its own program counter
- The two processes will compete for resources.

“Process”

- Abstraction
 - Self-contained entity
 - Separate different activities from each other
- Each process has its own memory area
- A process cannot affect the state of another

Process states

- Definition of Process

An execution stream in the context of a process state

- Process Control Block

- the information that must be saved so execution can be resumed later as if nothing had happened

Process State

- As a process executes, it changes *state*
 - **new**: The process is being created.
 - **running**: Instructions are being executed.
 - **waiting**: The process is waiting for some event to occur.
 - **ready**: The process is waiting to be assigned to a process.
 - **terminated**: The process has finished execution.

Process Control Block (PCB)

- Information associated with each process.
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)

pointer	Process state
Process number	
Program counter	
Registers	
Memory limits	
List of open files	
...	

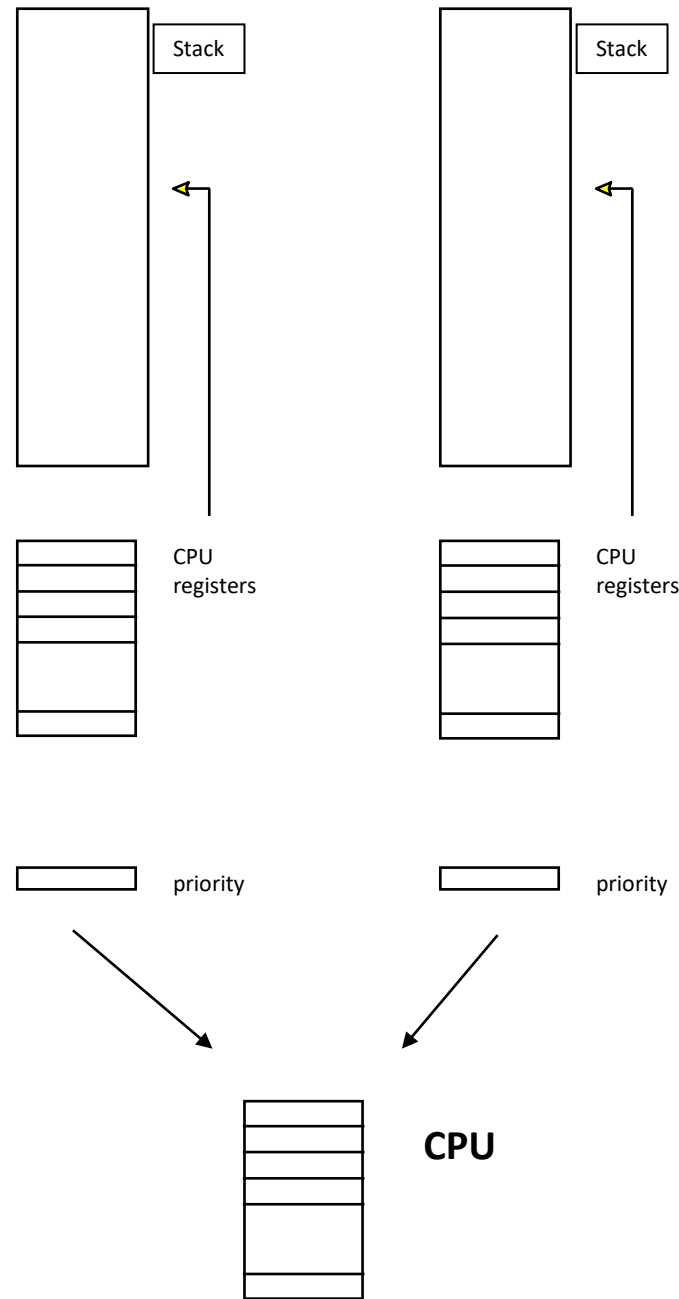
Context Switch

- The switch between two processes
e.g. to perform I/O or run another process
 - Save the context block (process state)
 - When exec instruction: PC, registers, flag changes
 - CPU is reused by each process in turn
 - Each process has its own area of RAM
 - text, data, stack

Virtual CPU

- Multiprocessing on one CPU =>
 - Each process has a virtual CPU
the processes appear to be executing simultaneously
 - execute a process a few ms
 - save the context (registers (incl. Flag and PC))
 - switch to another process
 - execute that a few ms

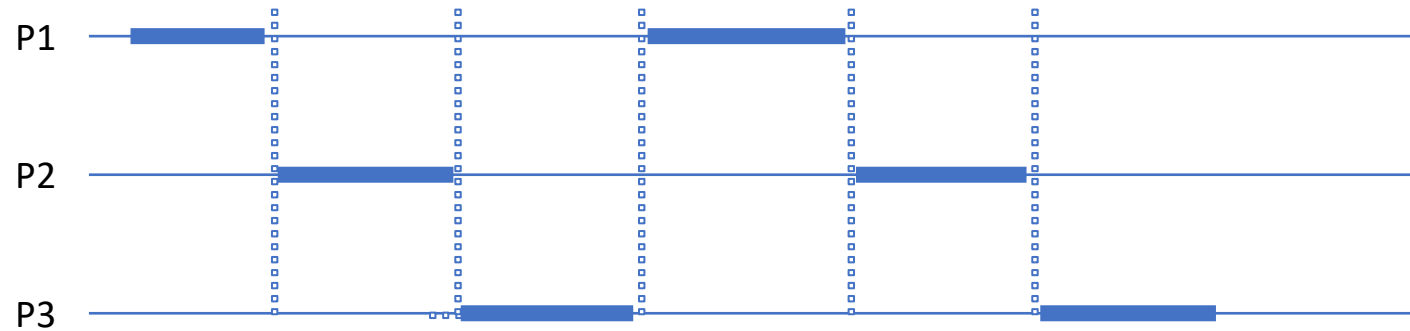
Processes relation to CPU



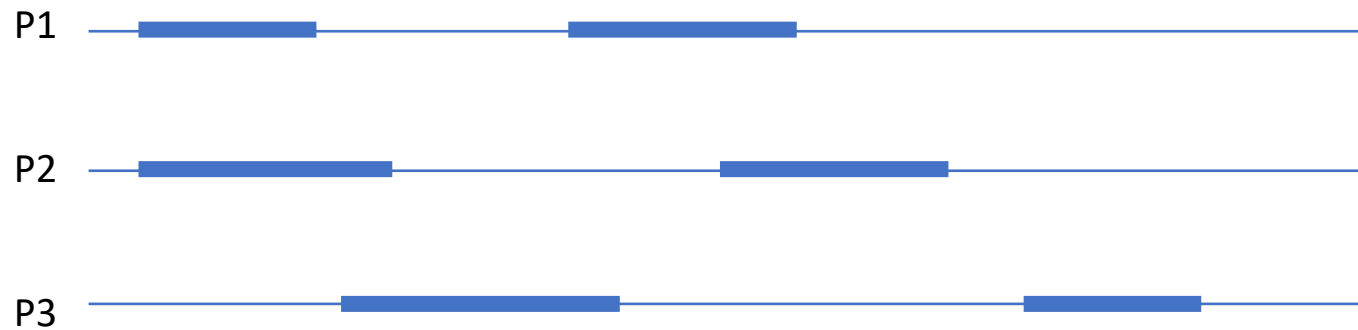
Each process has
its own stack

The CPU is shared

Apparent and true concurrency

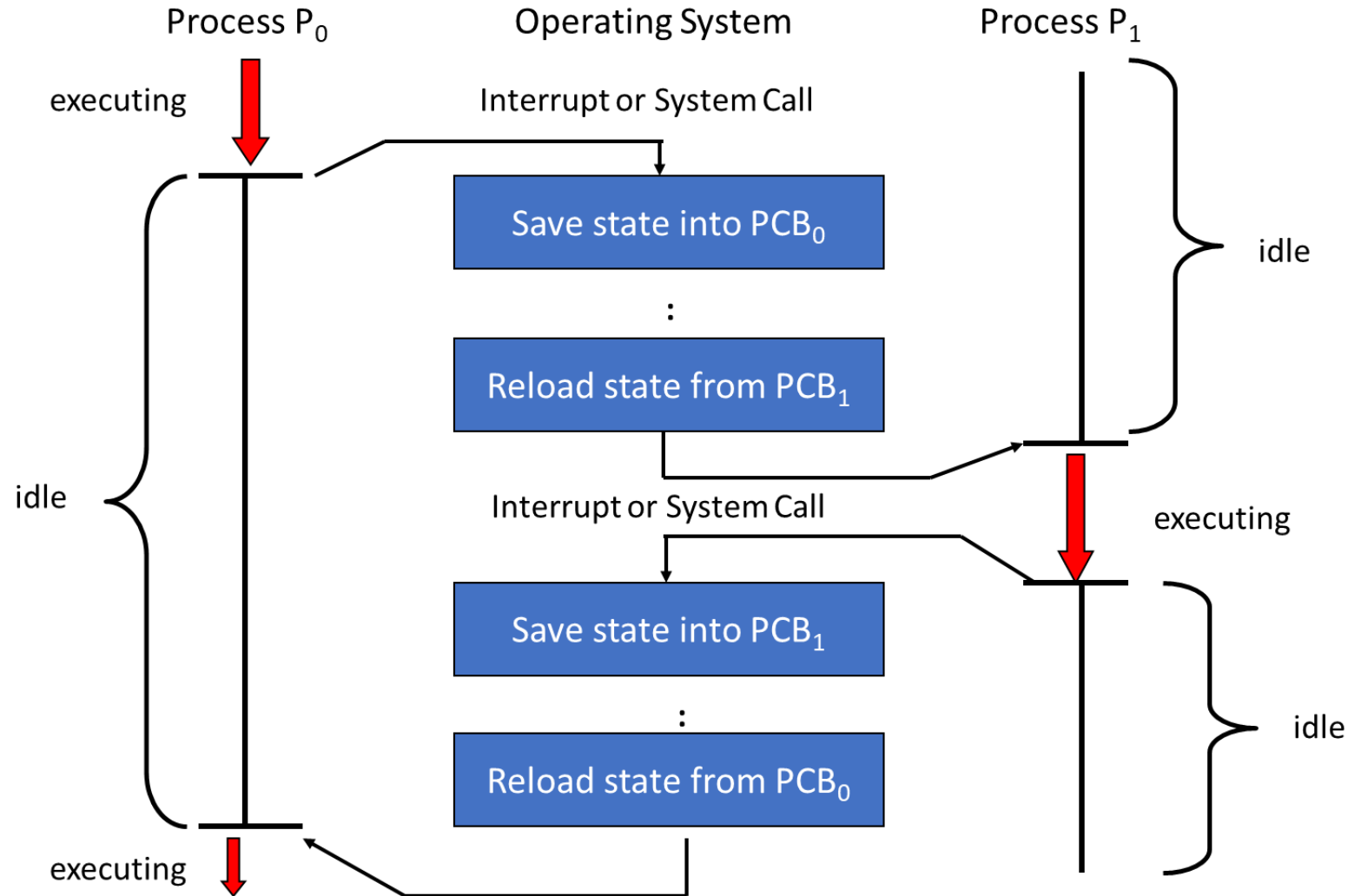


Apparent concurrency



True concurrency (overlapping)

CPU Switch From Process to Process



Pre-emptive systems

- Pre-emptive OS
 - execution can be interrupted to schedule another process
- Non-pre-emptive OS
 - cannot be interrupted
 - Also called “co-operative multitasking”
the running process must relinquish the CPU voluntarily.

An OS is non-deterministic

- Not possible to predict the order in which interrupts occur
- Therefore not possible to synchronise processes using timing
- Today's O/S's are often event-driven

Event-driven OS

- MS Windows OS are all event driven
 - wait for an asynchronous event to happen
main() = short loop waiting for events
 - responds to the event
 - keyboard event: echo on screen
 - mouse interrupt: moving mouse, button click
 - timer interrupt: schedule another process
 - read/write disk interrupt: execute system call
 - disk controller sends interrupt when finished r/w

Resources

- A process can be seen as
 - a unit of dispatching (scheduling)
 - a unit of resource ownership
 - competing for resources
 - RAM
 - disk space
 - CPU
 - printers
 - ...

Essential facilities for multitasking

- what do we need?
- 1. Memory protection features
 - Check every time a location is accessed
 - Usually need hardware support for speed

Essential facilities cont.

- 2. Privileged instruction set

- user mode (runs higher level functions)

- kernel mode (runs system calls)

- privileged instructions running in kernel mode:

- enable/disable interrupts
 - switch processes on CPU
 - access memory protection hardware registers
 - perform I/O
 - halt the CPU

Essential facilities cont.

- 3. Interrupt handler
 - ability to interrupt CPU
 - save PC + regs on stack
 - run interrupt handler
 - determine source of interrupt
 - action
- 4. Real-time clock
 - time execution
 - wait(ms) e.g. timeouts

Process states

- Process = a unit of dispatching
 - within this: 5 (or 7) execution states
 - = the life cycles of a process from birth to death

The state of a process are:

- New: Process created, waiting to be admitted
- Ready to run
- Running (being executed)
- Blocked (waiting for I/O to finish)
- Exit: stopped voluntarily or forced.

Boot procedure

- Process 0 forks to create process 1
 - PID 0 is called the swapper process
swaps processes in/out of memory for scheduler
 - PID 1 is called the init process
= ancestor to all other processes

ps

FLAGS	UID	PID	PPID	PRI	NI	SIZE	RSS	STA	TTY	COMMAND
100	0	1	0	0	0	776	388	S	?	init [3]
40	0	2	1	0	0	0	0	SW	?	(kflushd)
40	0	3	1	-12	-12	0	0	SW<	?	(kswapd)
140	0	47	1	0	0	748	356	S	?	/sbin/kernelld
140	0	181	1	0	0	804	440	S	?	syslogd
100140	0	190	1	0	0	788	388	S	?	klogd
100040	2	201	1	0	0	792	420	S	?	/usr/sbin
100140	0	212	1	0	0	860	488	S	?	crond
100140	1	223	1	0	0	764	332	S	?	portmap
140	0	234	1	0	0	1248	768	S	?	/usr/sbin/snmpd -f
100140	0	246	1	0	0	784	404	S	?	inetd
100140	0	257	1	0	0	832	412	S	?	lpd
140	0	272	1	0	0	1396	872	S	?	sendmail:
100140	0	284	1	0	0	756	340	S	?	gpm -t ms
100140	0	297	1	0	0	1480	744	S	?	smbd -D
100140	0	306	1	0	0	1316	724	S	?	nmbd -D
100100	500	319	1	0	0	1496	940	S	1	/bin/login -- root
100100	0	320	1	0	0	740	308	S	2	/sbin/mingetty tty2
100100	0	321	1	0	0	740	308	S	3	/sbin/mingetty tty3
100100	0	322	1	0	0	740	308	S	4	/sbin/mingetty tty4
100100	0	323	1	0	0	740	308	S	5	/sbin/mingetty tty5
100100	0	324	1	0	0	740	308	S	6	/sbin/mingetty tty6
100140	0	326	1	0	0	732	244	S	?	update (bdflush)
0	500	327	319	17	0	1200	780	S	1	-bash
100000	500	417	327	10	0	860	492	R	1	ps lax

Process Creation

- `int fork(void);`
 - Create a clone of the calling process:
 - the new process is almost an exact copy of the old process. The child inherits UID, environment, signal settings etc. It has access to the same files.
 - Return values from `fork()`
 - The parent receives the child's PID
 - The child receives a zero.
 - -1 would indicate an error

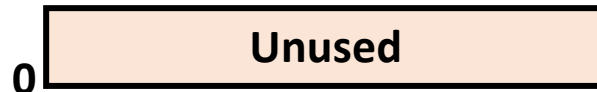
Process Creation

- `fork()`
 - creates (clones) a child process
 - returns the child's pid
- The child
 - inherits its parents characteristics
 - access to the same resources and open files
 - has same user id, environment, signal settings ...
 - begins execution where the parent left off

Process Address Space

A process provides each program with its own private address space. This space cannot be read or written to by any other process.

On Linux systems the code segment starts at address 0x080400



The Read Only segment.

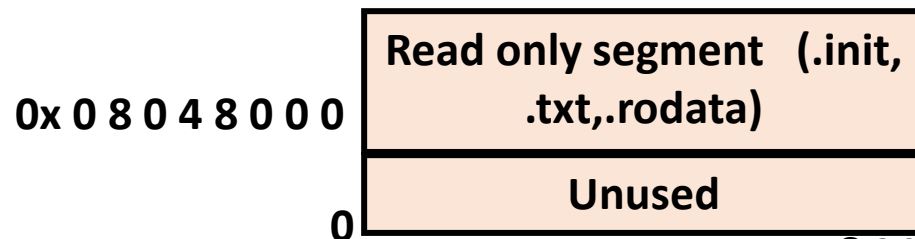
This is always started at address 0x08048000

Into which we store

.init – Defines a small function `_init` that is called by the program's initialisation code

.txt – The machine code of the compiled program

.rodata – The read only data, such as format strings in `printf` statements, and jump tables for switch statements

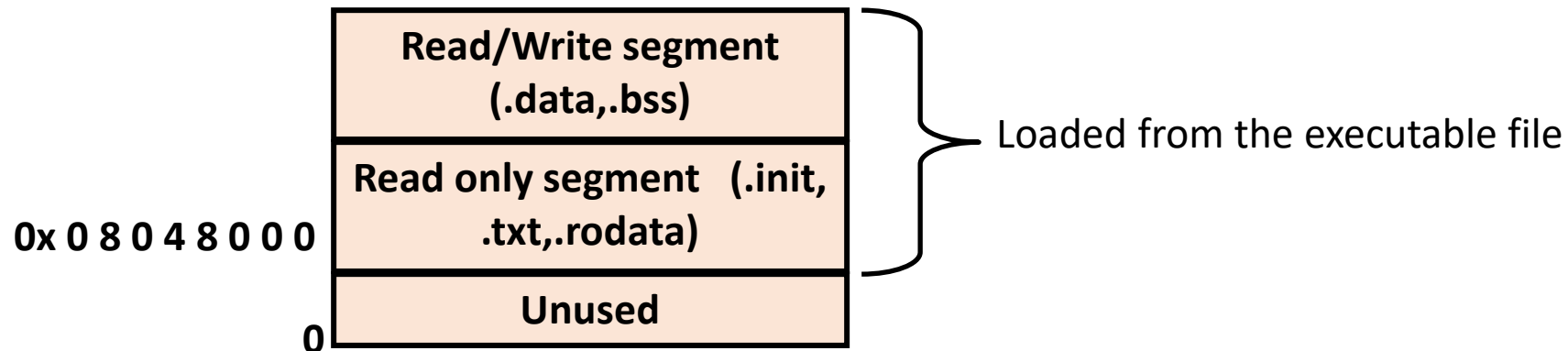


The Read/Write segment.

Contains:-

.data – Initialised global C variables. Rem. That local C variables are maintained at run time on the run time stack

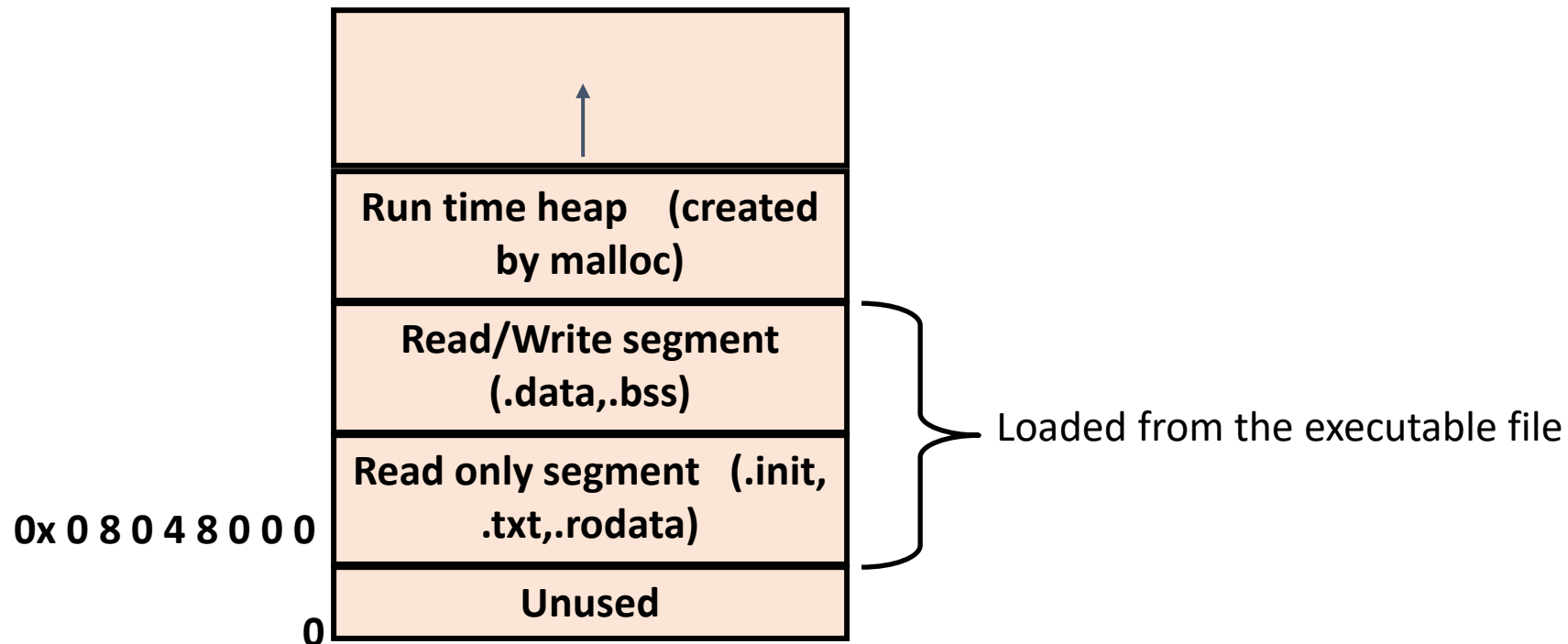
.bss – (Block Storage Start); Uninitialised global C variables. This holds no space but is a place holder. Object file formats distinguish between initialised and uninitialised variables. The uninitialised variables don't get allocated any space in the object file.



The Run time heap

This is a pool of memory which is used for storing data structures and these are allocated using either the functions malloc or calloc

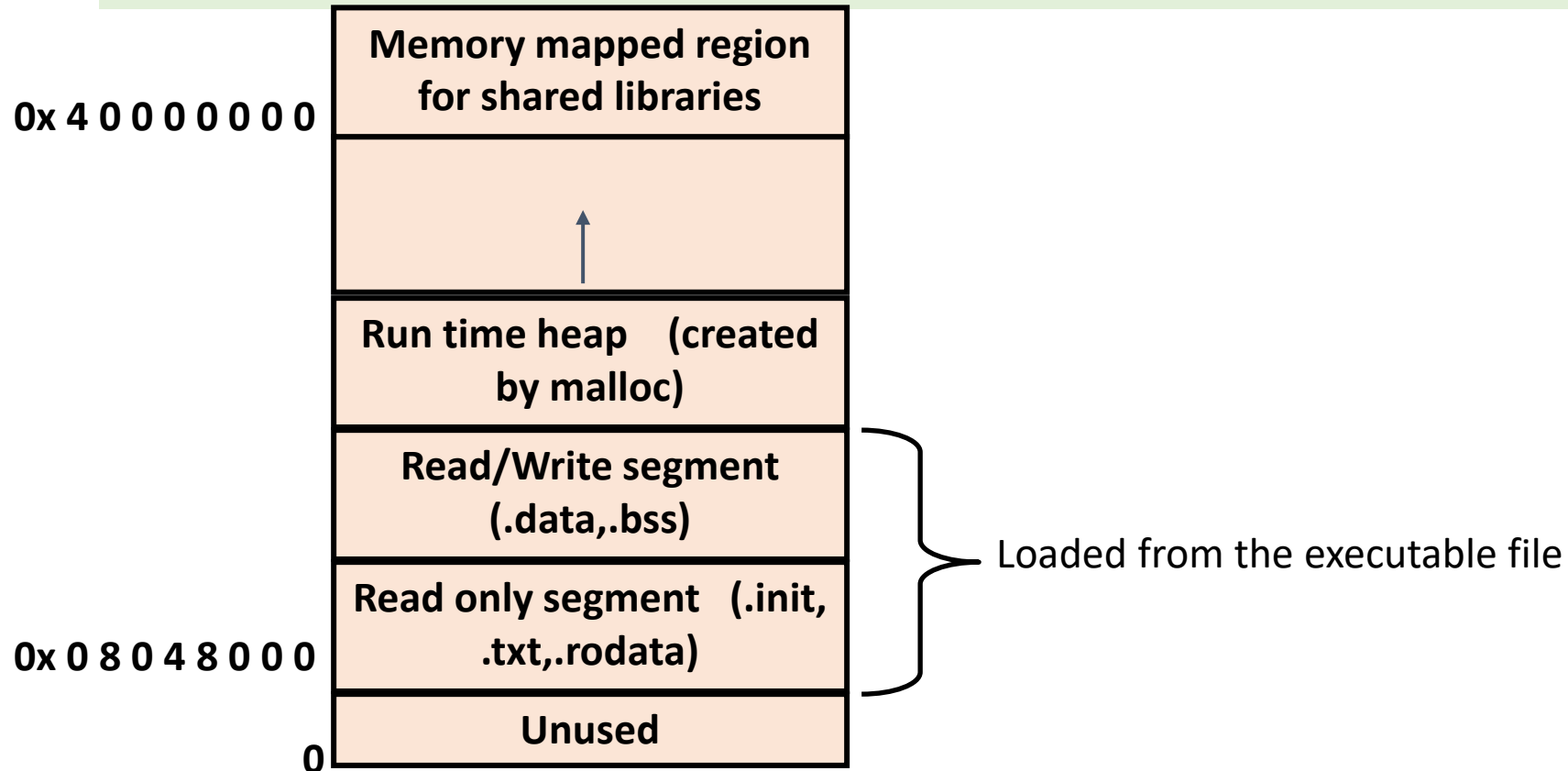
Note that there is free space so to allow the storage of the data structures



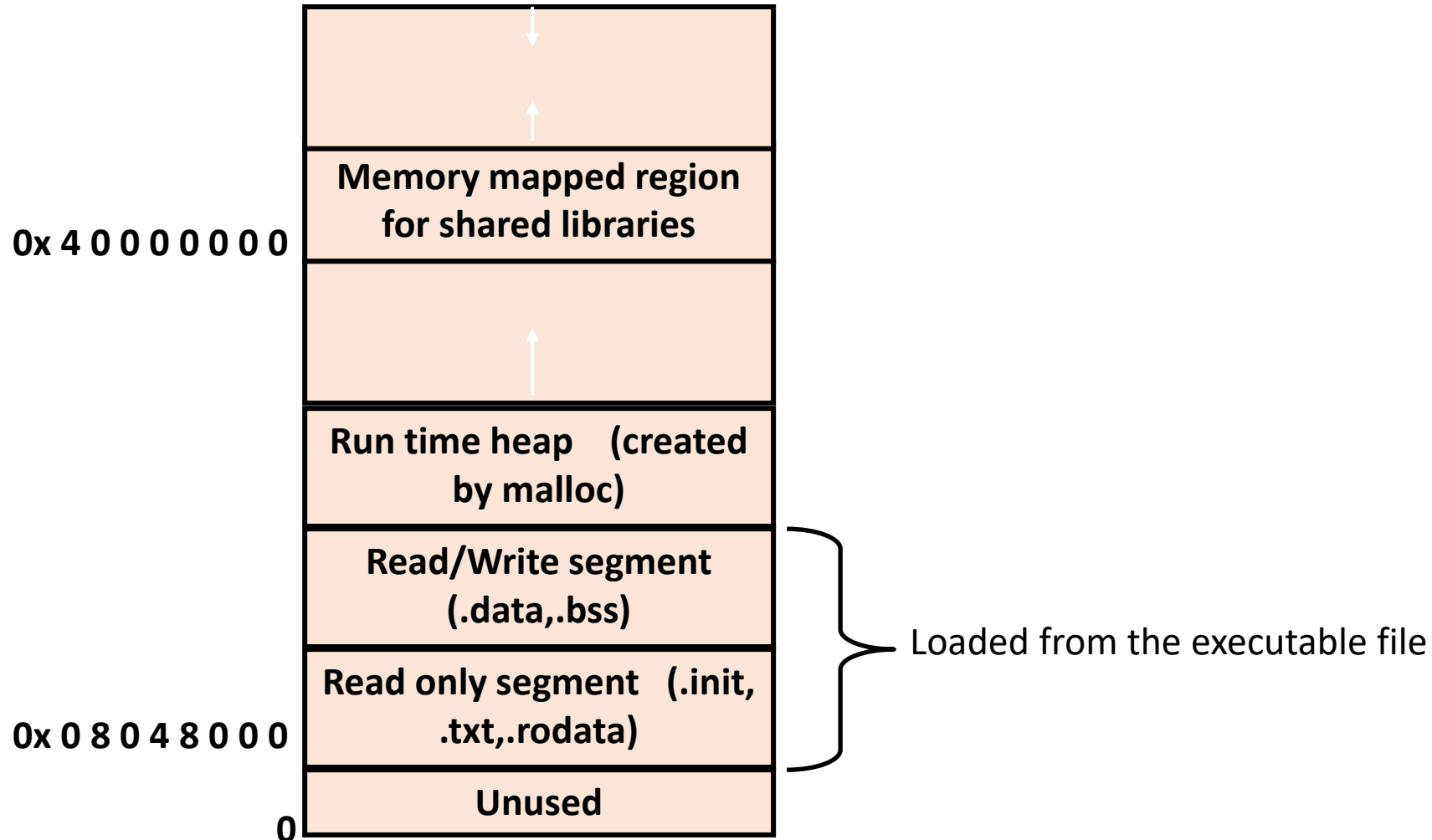
The Memory Mapped region for shared libraries

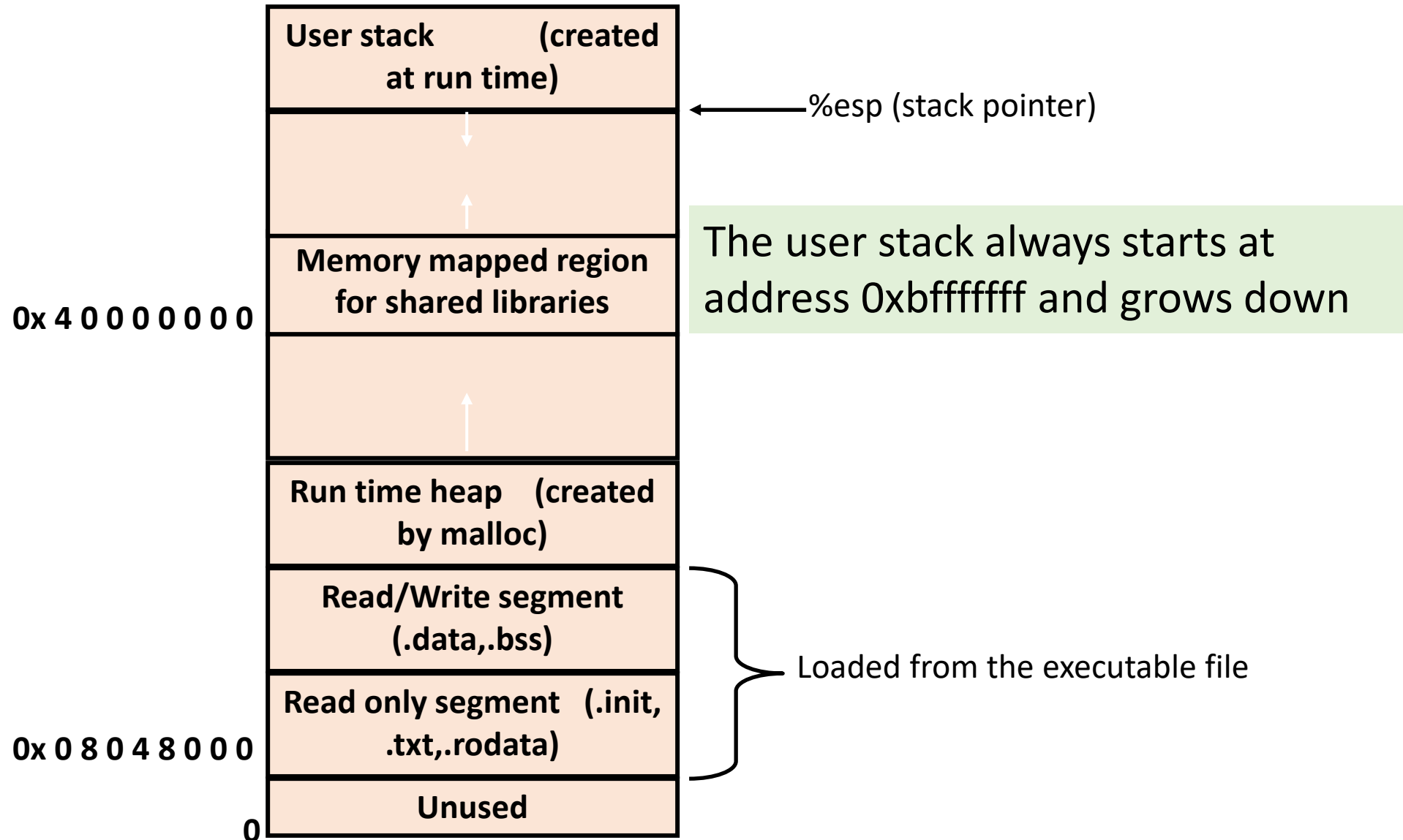
This segment starting at address 0x40000000 is reserved for the shared libraries.

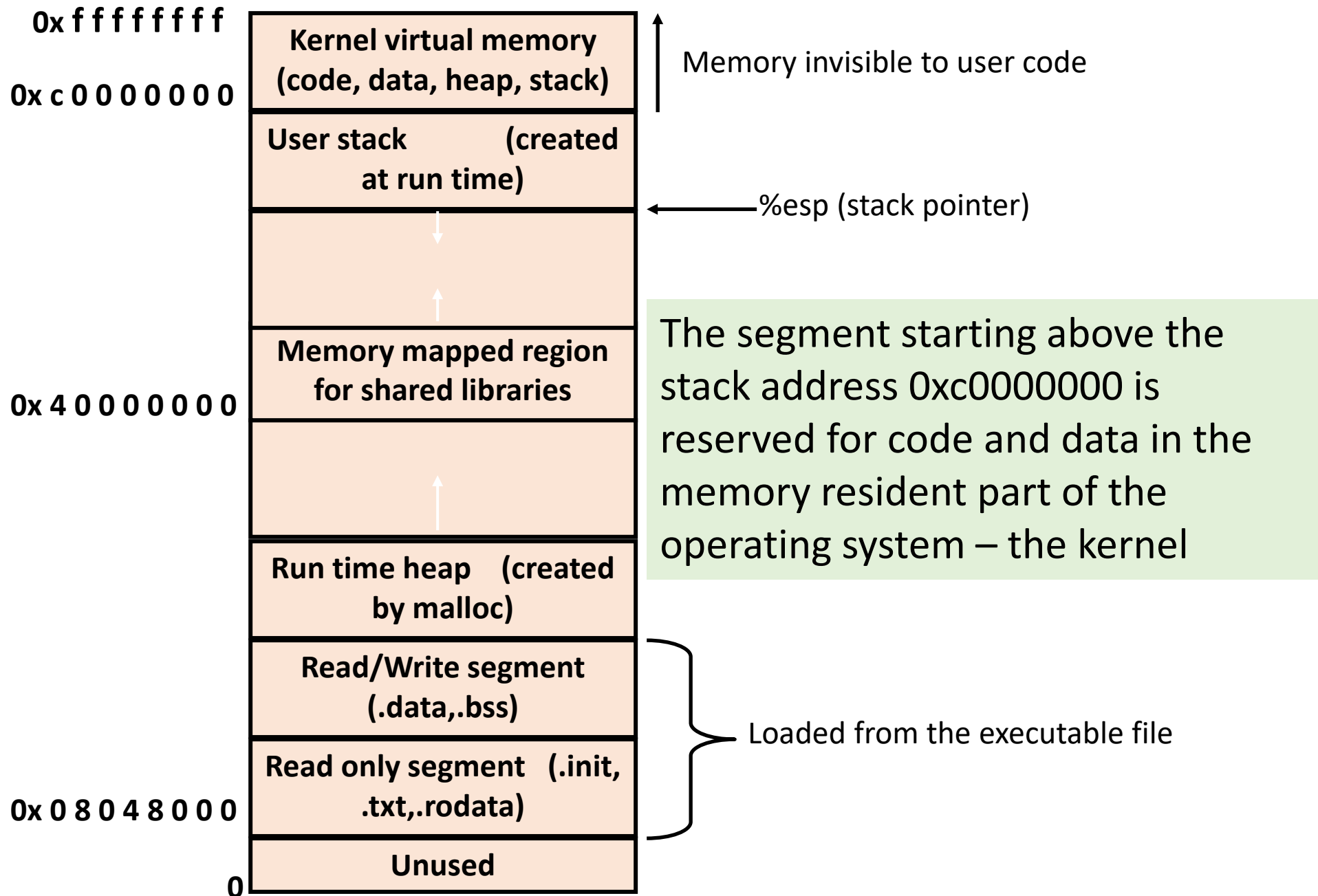
Where a shared library is an object module that at run time can be loaded and linked with the program



Padding allowing space to grow for shared libraries and the user stack at run time







Resource Allocation -Deadlock

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes **in order to proceed**.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_0 is waiting for a resource that is held by P_0 .

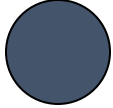

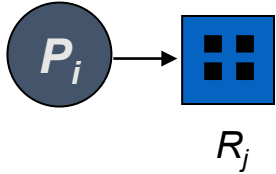
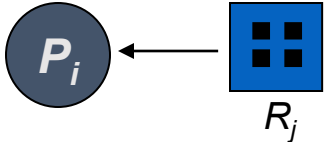
These four conditions are not independent. The first three are necessary condition, and the fourth is necessary and sufficient. The fourth condition incorporates the first three.

Resource-Allocation Graph

A set of vertices V and a set of edges E .

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource **types** in the system.
There are one or more *instances* of a resource *type*.
A request is made for an *instance* of a resource *type*
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

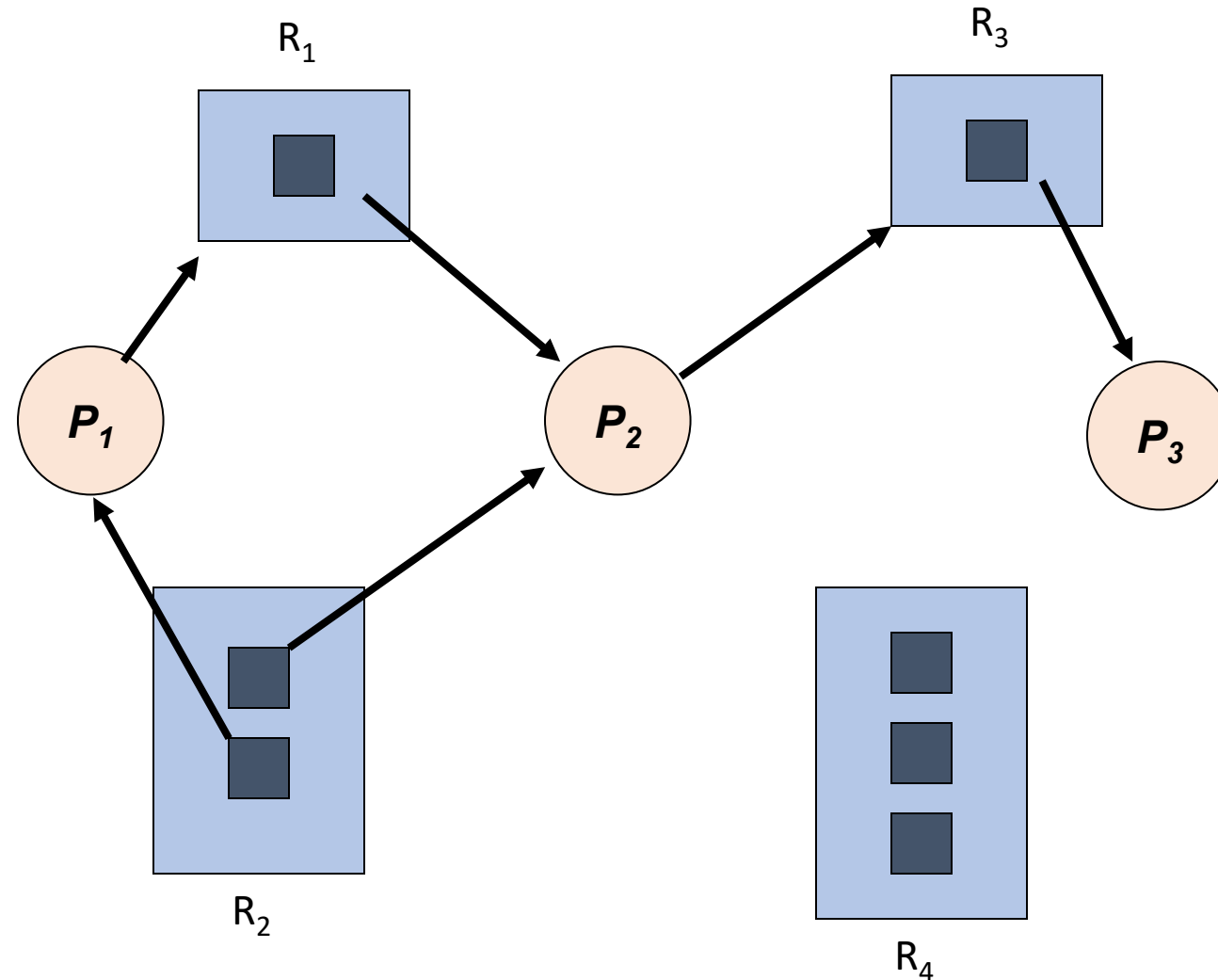
Resource-Allocation Graph

- Process 
- Resource Type with 4 instances 
- P_i requests an instance of R_j 
 R_j
- P_i is holding an instance of R_j 
 R_j

Example of a Resource Allocation Graph

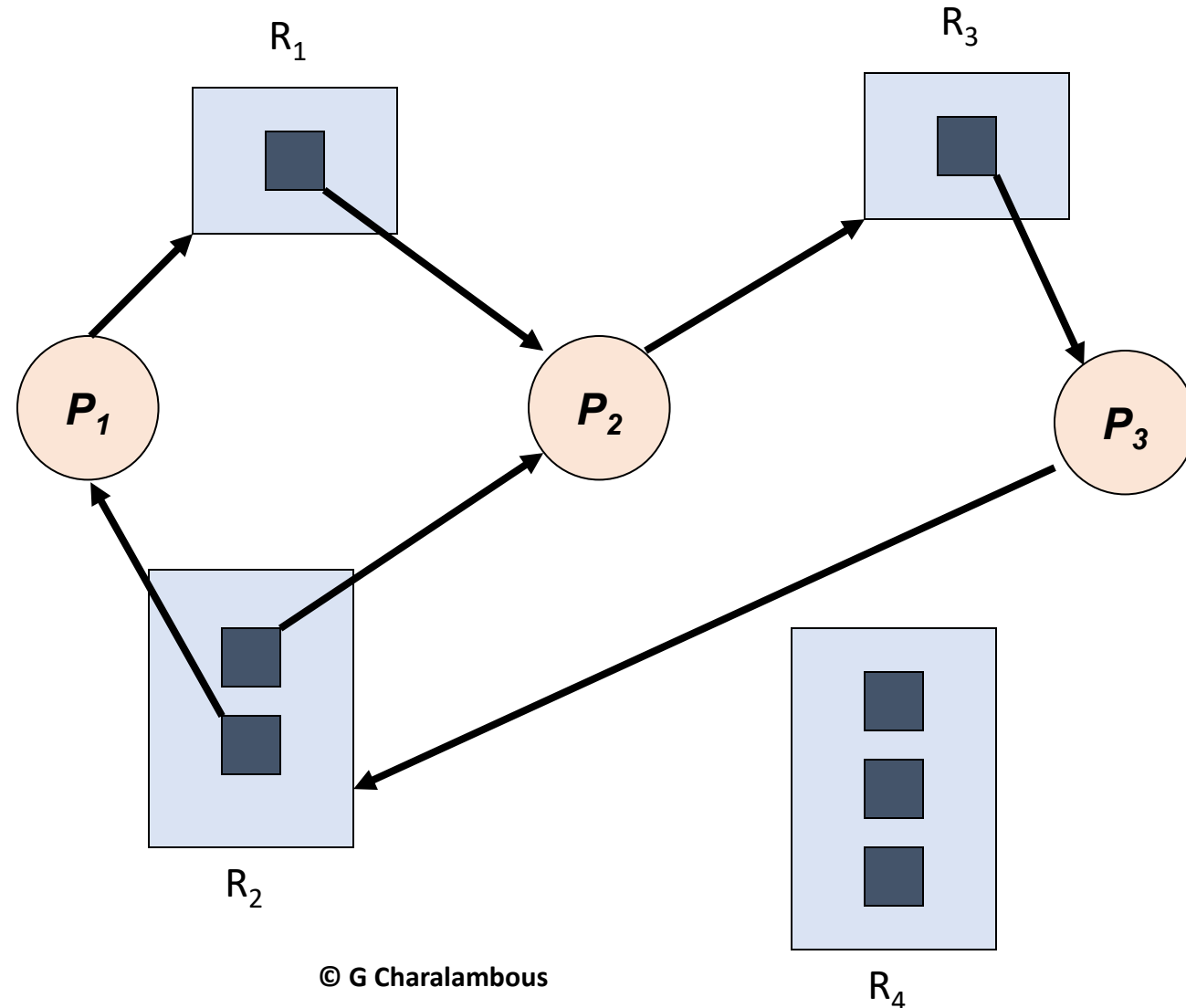
Example:

P_2 holds an instance of R_1 and R_2 , and is waiting for an instance of R_3 ... P_2 will execute when P_3 releases R_3



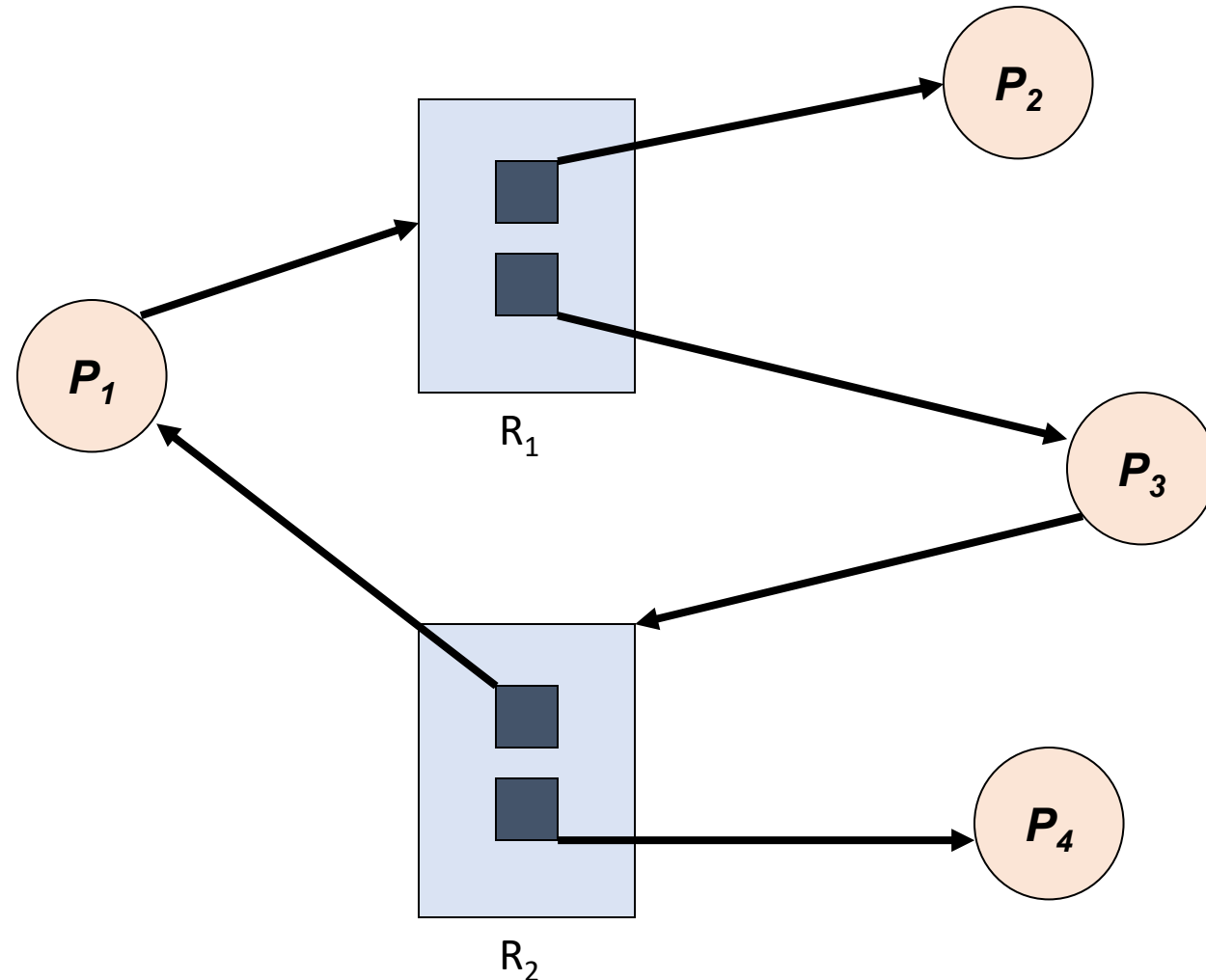
Resource Allocation Graph With A Deadlock

Circular wait: all three Processes are mutually waiting for each other to release a resource.



Resource Allocation Graph With A Cycle But No Deadlock

When P_4 releases its instance of R_2 , then R_2 can be allocated to P_3 , breaking the cycle.



Basic Facts

- If graph contains **no cycles** \Rightarrow **no deadlock**.
- If graph contains a cycle \Rightarrow
 - if only **one instance** per resource type, **then deadlock**.
 - if **several instances** per resource type, **possibility of deadlock**.

... a necessary, but not a sufficient condition for deadlock

Distinguish between a cycle and a circular wait - the former is a necessary condition, the latter is a necessary and sufficient condition.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state - **Deadlock prevention or avoidance.**
- Allow the system to enter a deadlock state and then recover - **Deadlock detection.**
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
... ignore the problem and maybe it will go away! If it doesn't - re-boot the machine!

Deadlock Detection

- Grant resources freely and test for deadlock
- If deadlock
 - Kill all deadlocked processes
 - Kill random processes until cycle broken
 - Take resources from selected processes
 - Restart the process and remove all resources
 - Rollback processes (will probably deadlock again!)

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
 - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
 - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption** –
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
 - Preempted resources are added to the list of resources for which the process is waiting.
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number of resources* of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the **resource-allocation state** to ensure that there can never be a circular-wait condition.
- *Resource-allocation state* is defined by the number of available and allocated resources, and the **maximum demands** of the processes.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.
- **System is in safe state if there exists a safe sequence (sequence of execution) of all processes.**
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$.

Safe State continued

- If P_i resource needs are not immediately available, then P_i can wait until all *previous* P_j have finished.
- When *all* P_j have finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.
- **BASIC CONCEPT SUMMARY:**

As each P_i terminates, the total amount of the resource in question gets a net increase by the amount of what P_i *was holding*

Before it was granted the extra amount it needed - the loaning and returning of this “extra amount” has no net effect on the total resource - only the held amount contributes.

Basic Facts

- If a system is in safe state \Rightarrow no deadlocks.
- If a system is in unsafe state \Rightarrow possibility of deadlock.
- **Avoidance** \Rightarrow ensure that a system will never enter an **unsafe state**.
- Example of Deadlock Avoidance is the Banker's Algorithm by Dijkstra (Uses as an analogy a bank which offers loans and receives payments to/from customers. The banker will only grant a loan if the needs of future needs of its customers can be met.

Resource: Memory Management

- Memory needs to be allocated / de-allocated
 - before loading a new process
 - during execution
 - allocated, reallocated, de-allocated
- Issues
 - Paging
 - Protection of process memory
 - Relocation
 - Sharing
 - Memory allocation.

Paging

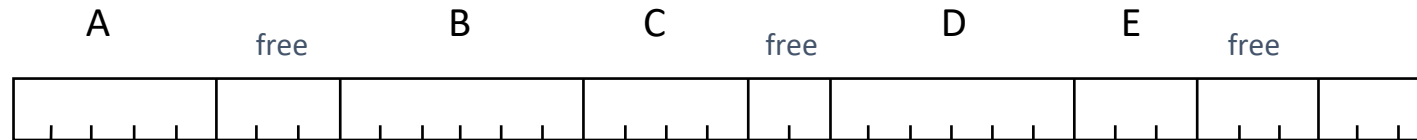
- RAM divided into pages
- The pages are swapped to and from disk
- Advantages
 - Little waste of memory
 - on average: 50% of one page per process
 - Allocation/de-allocation is simple and relatively fast
- Disadvantages
 - Protection of read-only pages difficult to implement
 - Address translation: high overhead
 - Risk of “thrashing” where spend time swapping processes and not executing them!

Memory map table (MMT)

- Keeps track of page frames in memory
- Translates virtual addresses into physical addresses
 - if virtual address found in MMT: use this address
 - if not: generate "page fault"
 - find a little-used page
 - write it to disk if changed
 - load page with wanted virtual address
 - update data structures and tables

Bit Maps

- Memory is divided into small units. A bit map keeps tracks of which pages are allocated and which are free



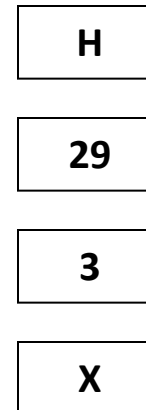
11111000
11111111
11001111
11111000

Linked Lists

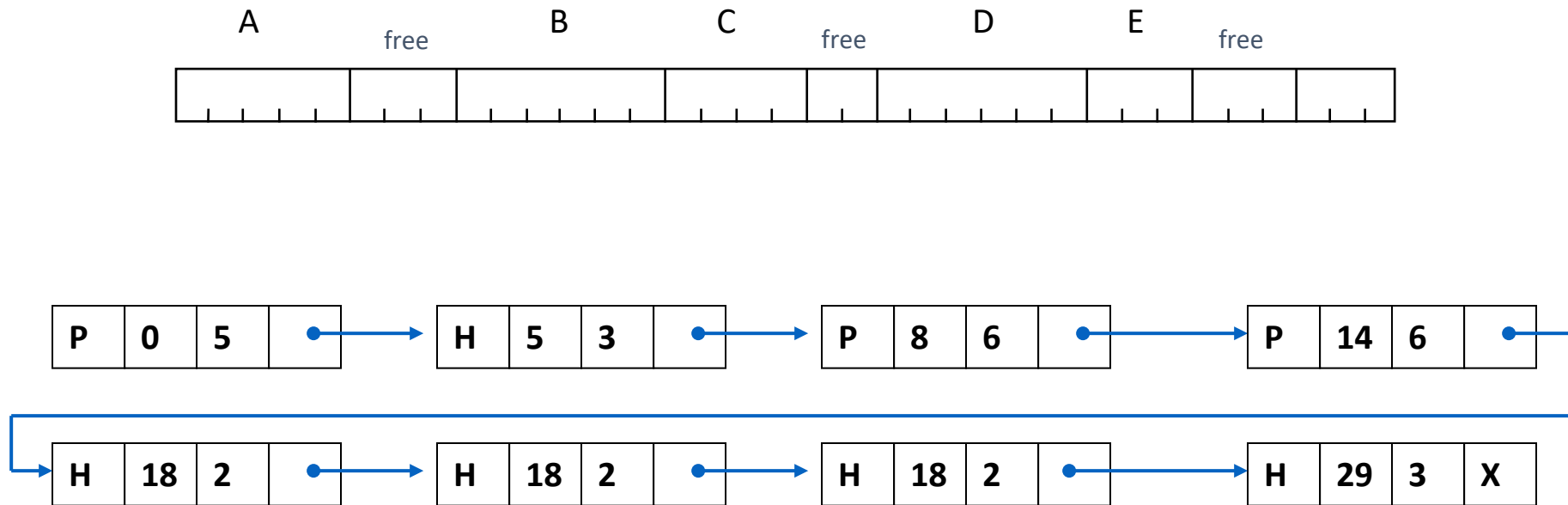
- Memory is divided into variable sized blocks
 - free blocks
 - occupied blocks (used by a process)

Define a node which contains four fields

- char --- H or P (H --- hole / P --- process)
- Base address
- Length
- Pointer to the next node



Linked Lists Example



Linked Lists cont.

- Typically two linked lists
 - **allocated list**: all used memory blocks
 - **free list**: all available free blocks (“holes”)
 - can be implemented inside the free block
 - store address of next free hole at start of each hole

Linked Lists cont.

- Allocate memory
 - scan free list for suitable block
may break large block in two smaller blocks
 - take off free list
 - add to allocated list
- Deallocate memory
 - take off allocated list
 - insert in free list
 - combine it with neighbouring free block if possible

Algorithms

- First fit: choose the first big enough
 - results in cluster of tiny blocks at the front
- Next fit: start where prev. left off to spread
 - memory compaction difficult and expensive
- Best fit: search whole list, chose best fit
 - results in large linked list of tiny unusable holes!
- Worst fit: search whole list, split the worst fit
- Quick fit: maintains separate list of common sizes
(sorted by size)

First fit

Process 1 212K
 Process 2 417K
 Process 3 112K
 Process 4 426K

100	500K	200K	300K	600K
-----	------	------	------	------

The underlined number is the modified block.

First fit: start from the first element and take the first free area large enough is used

	100	500	200	300	600
212	100	<u>288</u>	200	300	600
417	100	288	200	300	<u>183</u>
112	100	<u>176</u>	200	300	183
426	garbage collection or compaction necessary				

Next fit

Process 1 212K
 Process 2 417K
 Process 3 112K
 Process 4 426K

100	500K	200K	300K	600K
-----	------	------	------	------

Next fit: start from current element and take the first area large enough

	100	500	200	300	600
212	100	<u>288</u>	200	300	600
417	100	288	200	300	<u>183</u>
112	100	288	200	300	<u>71</u>
426	garbage collection or compaction necessary				

Best fit

Process 1 212K
Process 2 417K
Process 3 112K
Process 4 426K

100	500K	200K	300K	600K
-----	------	------	------	------

Best fit: scan the whole list and choose the one that fits best

	100	500	200	300	600
212	100	500	200	<u>88</u>	600
417	100	<u>83</u>	200	88	600
112	100	83	<u>88</u>	88	600
426	100	83	88	88	<u>174</u>

Worst fit

Process 1 212K
 Process 2 417K
 Process 3 112K
 Process 4 426K

100	500K	200K	300K	600K
-----	------	------	------	------

Worst fit: scan the whole list and choose the hole that fits worst

	100	500	200	300	600
212	100	500	200	300	<u>388</u>
417	100	<u>83</u>	200	300	388
112	100	83	200	300	<u>276</u>
426	garbage collection or compaction necessary				

Algorithms cont.

- All the “fit”-algorithms results in external fragmentation
 - = waste between allocated blocks
- If no free memory?
 - swap processes to disk
 - compact memory
 - garbage collection

Buddy Systems

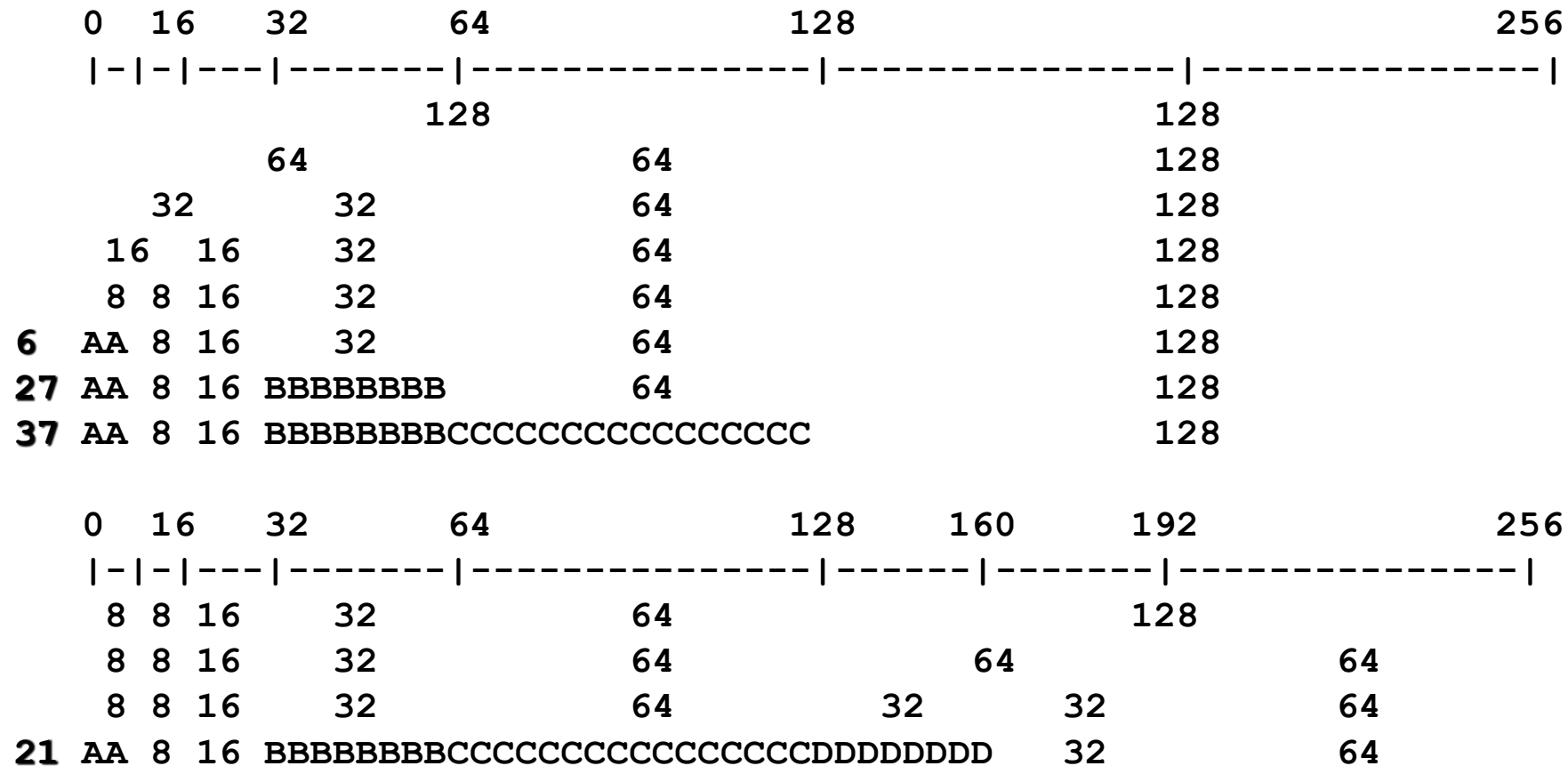
- MMU maintains blocks of powers of 2
1, 2, 4, 8, 16 ...
- All requests are rounded up to power of 2
- e.g. 1M = 1024K memory
 - Allocate 70K
 - split 1M into 2 * 512K
 - split first 512K into 2 * 256K
 - split first 256K into 2 * 128K
 - take the first

Buddy System cont.

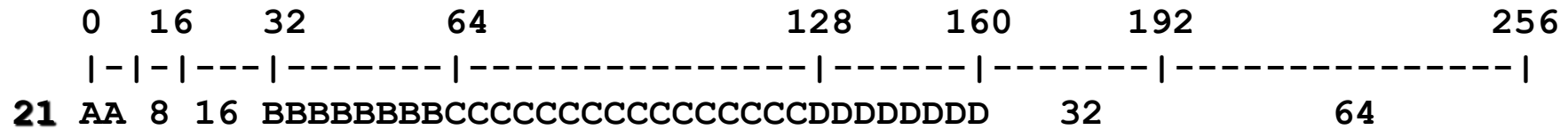
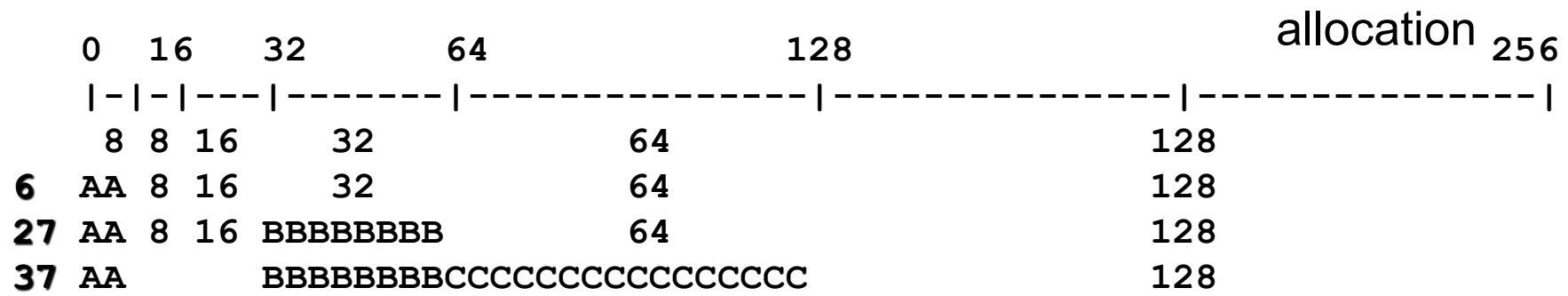
- Note that each block has a “buddy”
= the block of the same size next to it
- Only buddy’s can later be combined to one large block!
- Very fast but inefficient utilisation of memory
 - suffers from internal fragmentation (waste inside blocks)

Example

- Assume 256K memory, starting at address 0.
Initially all memory is free and requests arrive in this order: 6K, 27K, 37K, 21K



Free blocks can be found as 8K at 8K, 16K at 16K, 32K at 160K, 64K at 192K



Free blocks can be found as 8K at 8K, 16K at 16K, 32K at 160K, 64K at 192K

deallocation

Now let first B, then A be de-allocated



Segmentation

This provides an alternative method for dividing a process, unlike that of paging where we have fixed sizes of memory now we can use variable length portions of memory --- called segments

Its similar to the variable partition method so that a process can be loaded in several portions (segments)

Segments can be any length, upto a maximum determined by the design of the system

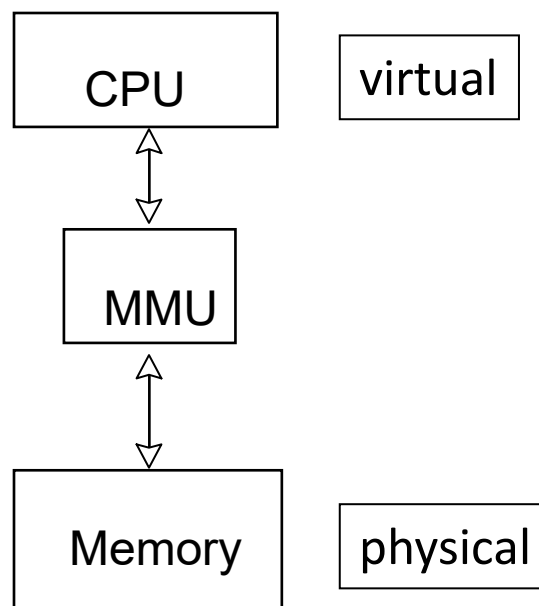
Paging is impressed upon the physical form of the process and is independent of the programs structure unlike that of segmentation which reflects on the logical structure of the process.

Memory Organisation

- Paged virtual memory
- Segmented virtual memory
- Both paged and segmented

Paged Virtual Memory

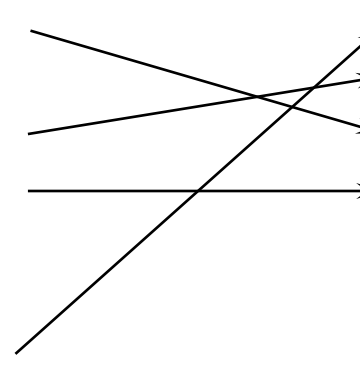
- Address translation
 - Physical memory divided into fixed sized pages
 - CPU deals with virtual addresses
 - MMU translates to physical addresses



Example of Memory map table (MMT)

PAGE TABLE:	
Virt Page	Page Frame
0	2
1	not in main memory
2	1
3	3
4	not in main memory
5	not in main memory
6	0
7	not in main memory

MEMORY MAP:			
pf	Virtual Page	Page frame	Physical
0	0000-1023	2	0 0000-1023
1	1024-2047	-	1 1024-2047
2	2048-3071	1	2 2048-3071
3	3072-4095	3	3 3072-4095
4	4096-5119	-	
5	5120-6143	-	
6	6144-7167	0	
7	7168-8191	-	



Translate: 0000, 5363, 3071, 3072, 3073, 2048, 4196

Result

Virtual	Physical
0000	2048
5363	page fault
3071	2047
3072	3072
3073	3073
2048	1024
4196	page fault

Paging Algorithms

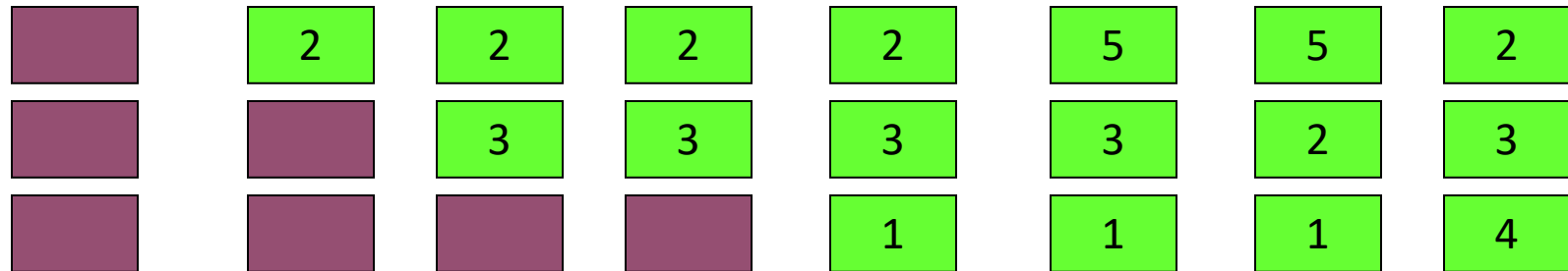
- First-In, First Out (FIFO)
 - Oldest page swapped out
 - No guarantee that this is the least active page!
- Least Recently Used (LRU)
 - Least active page discarded
 - Implementation: linked list sorted on usage
- Not Frequently Used (NFU)
 - Counter used for each page when page referenced add 1 else add 0. Page with the lowest counter is swapped

- First-In, First Out (FIFO) Example
 - Oldest page swapped out

page address stream
formed by executing
process

2 3 2 1 5 2 4

frames



F F F

Page fault – page not in main memory needs to be
retrieved from secondary storage

•Least Recently Used (LRU) Example

–Least active page discarded

page address stream
formed by executing
process

2 3 2 1 5 2 4

frames

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

F

F

Page fault – page not in main memory needs to be
retrieved from secondary storage

If frequency of >2 frames is identical then the oldest is replaced

•Not Frequently Used (NFU) Example

- Counter used for each page when page referenced add 1 else add 0. Page with the lowest counter is swapped

page address stream
formed by executing
process

2 3 2 1 5 2 4

Frames 2nd
no counter

		2;0	2;0	2;1	2;1	2;1	2;2	2;2
			3;0	3;0	3;0	5;0	5;0	5;0
					1;0	1;0	1;0	4;0

F

F

Page fault – page not in main memory needs to be retrieved from secondary storage and swapped with frame with the lowest counter

If two or more frames have identical counters then the oldest is replaced

Advantages of paging

- Each process has own virtual memory
- Virtual memory goes from 0-max
=> no relocation problems
- Supports processes larger than physical mem.
- External fragmentation eliminated
- Internal fragmentation depends on page size

Comparing a page and a frame on a computer system

- A page in a paging system is a virtual memory block with a fixed length.
- A frame in a paging system is a main memory block with fixed length.
- A page has virtual address, and it is transferred as a unit between main memory and secondary memory.
- A frame is mapped to one page of virtual memory.
- Programs use virtual memory.
- A page address is a virtual address, but a frame address is an address of physical location in memory.