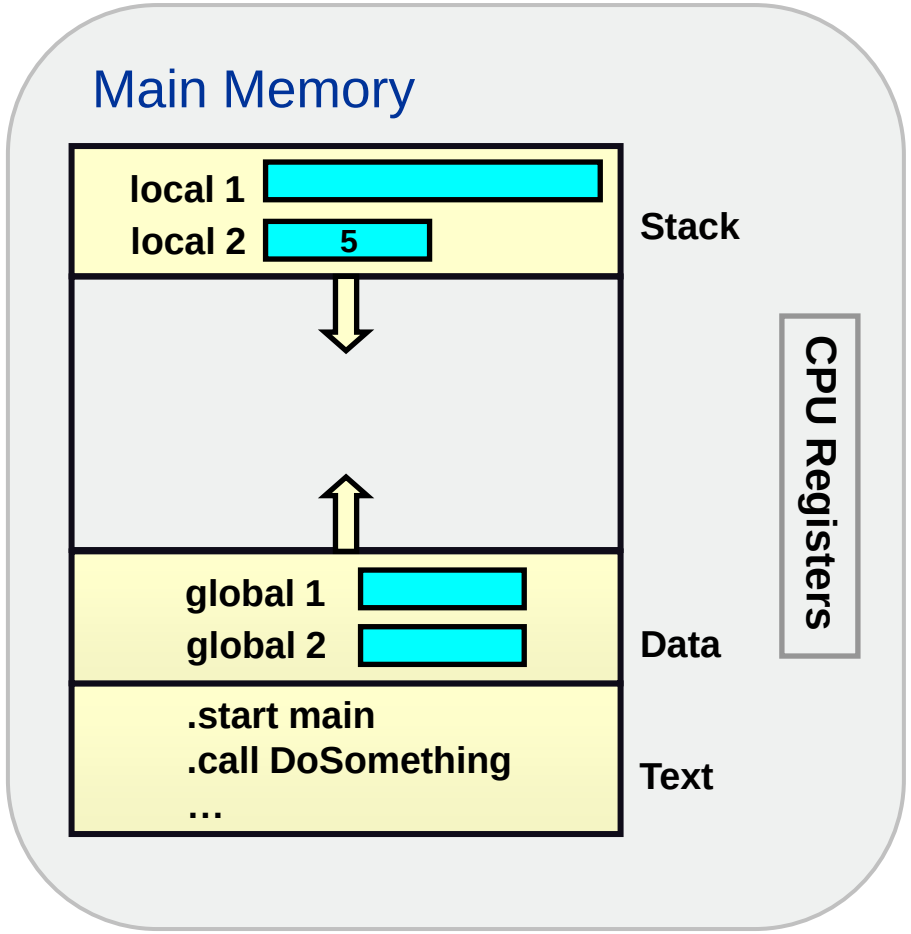
- **What is a process**
  - a program in execution
  - process execution progress in sequential fashion

- **program vs process**
  - Program - Passive and on disk
  - Process - Active and in memory

Process contains

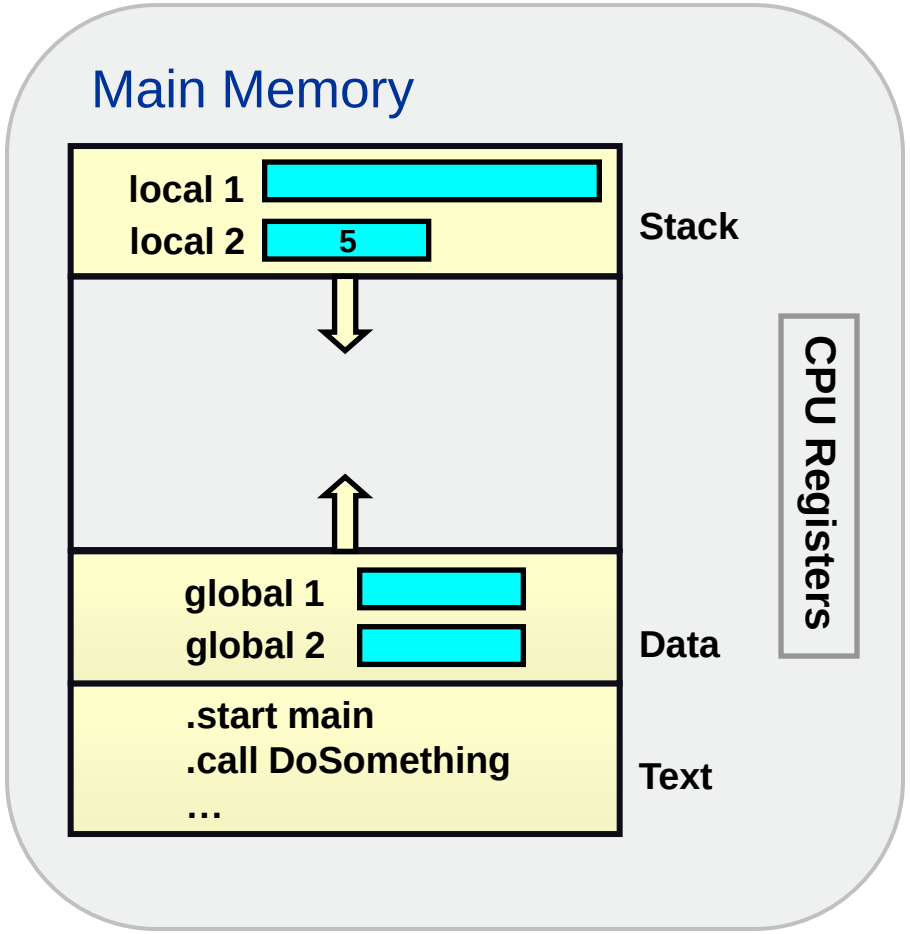
```
int global1 = 0;
int global2 = 0;

void DoSomething()
{
   int local2 = 5;

   local2 = local2 + 1;
   ...
 }

int main()
{
   char local1[10];

   DoSomething();
                ...
}
```

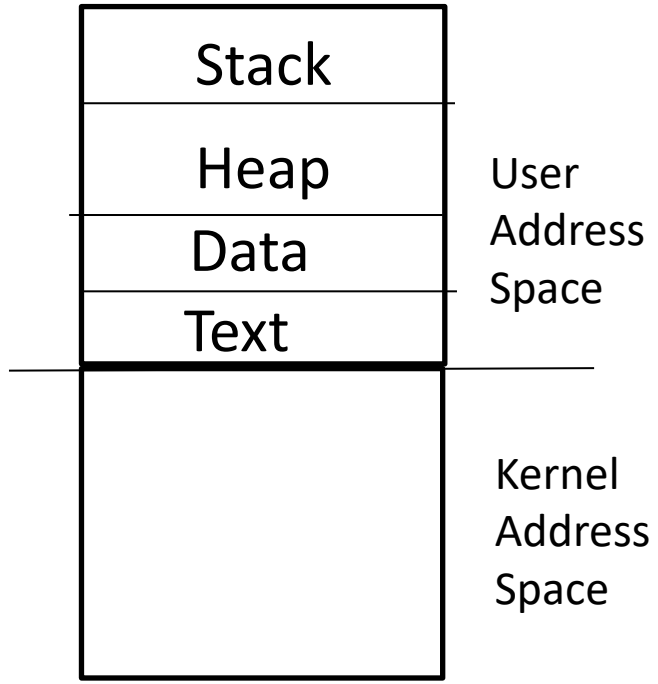


Process contains

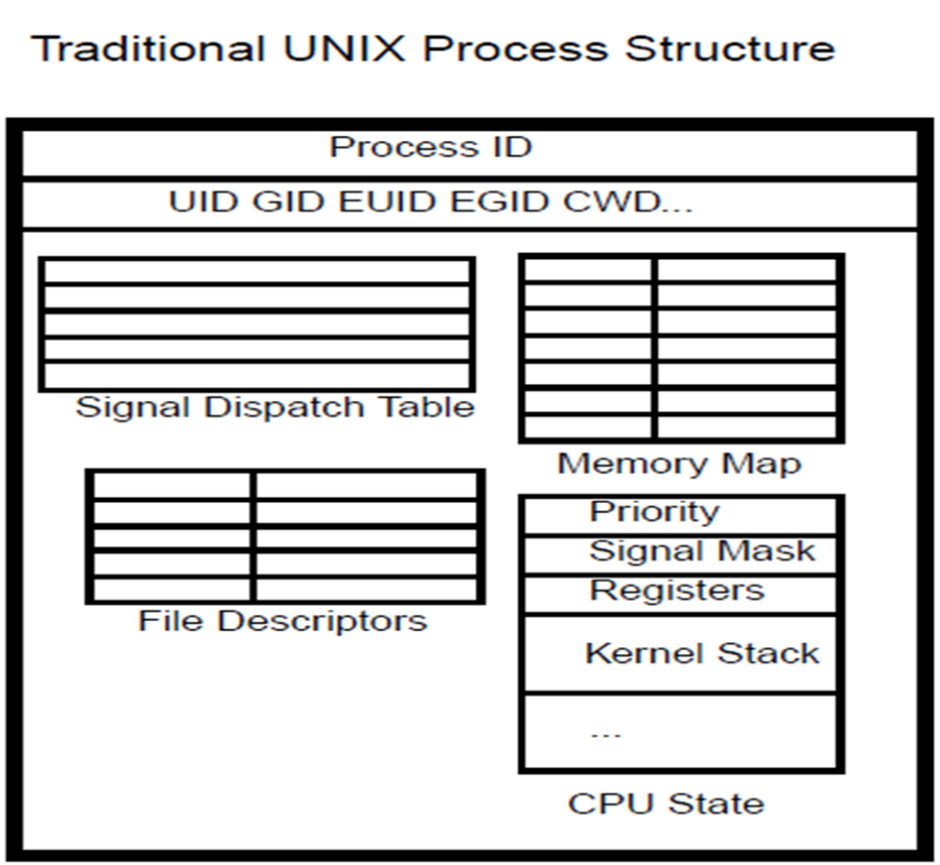
- **User Address Space**

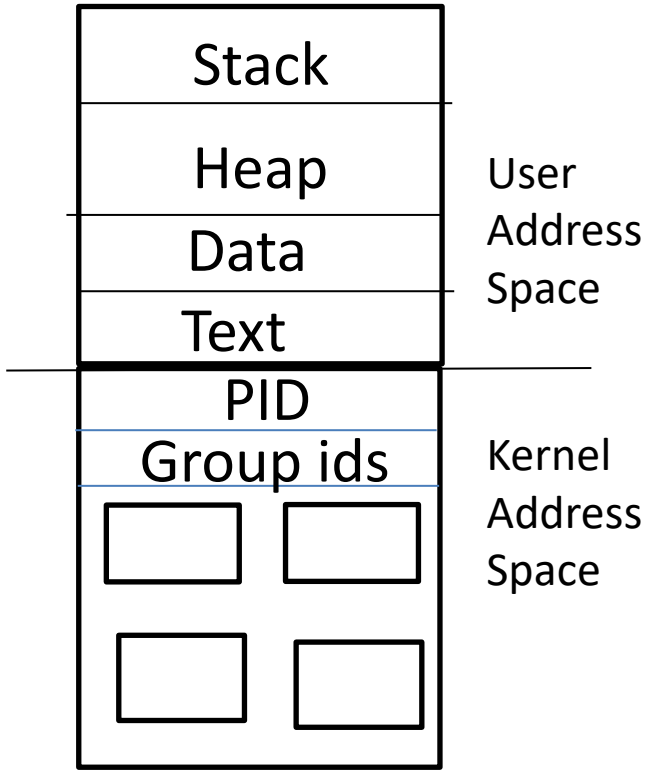
```
int global1 = 0;
int global2 = 0;

void DoSomething()
{
   int local2 = 5;

   local2 = local2 + 1;
   ...
 }

int main()
{
   char local1[10];

   DoSomething();
                ...
}
```





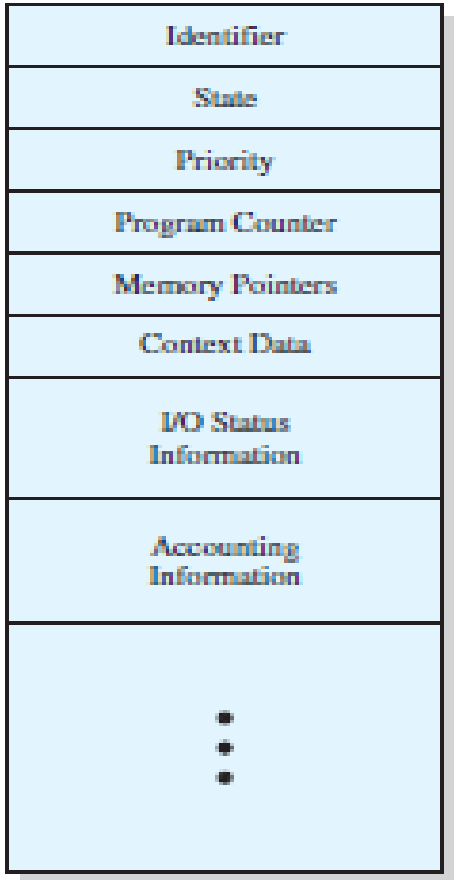
- **Kernel Address Space**



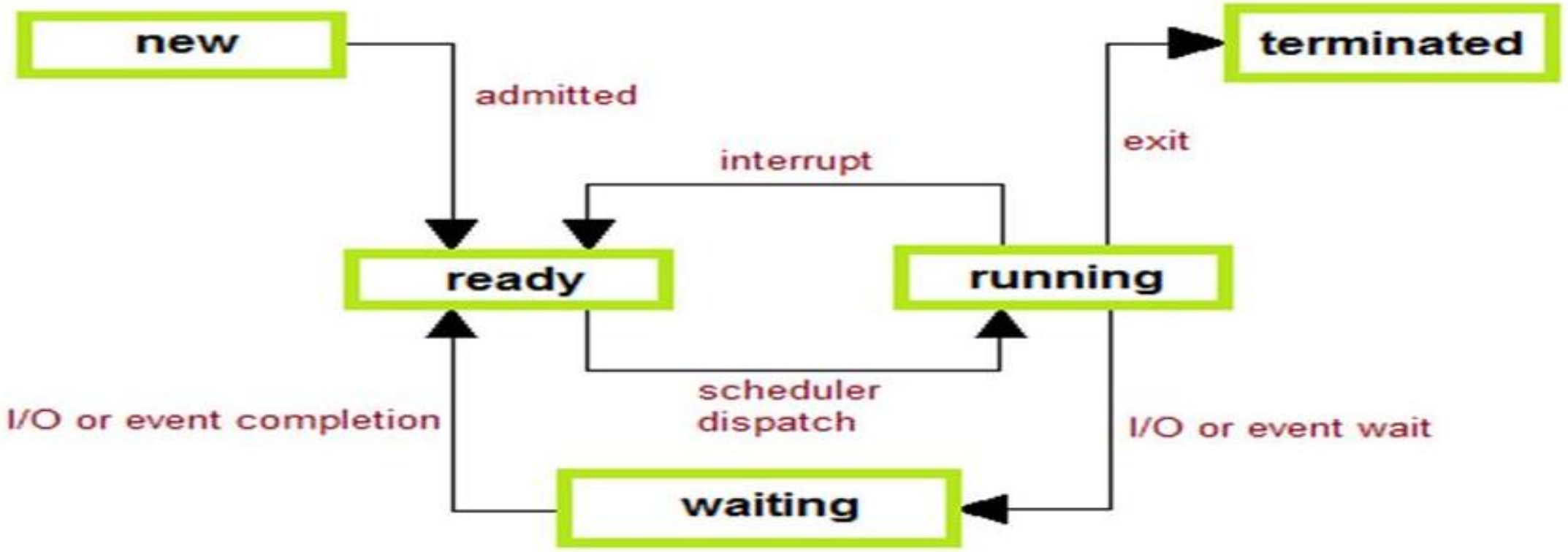


PCB

- **Process can be uniquely characterized by a number of elements**
  - Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers:
    - pointers to code and data
    - memory blocks shared with other processes
  - Context data
  - I/O status information
    - outstanding I/O requests
    - I/O devices assigned to the process
  - Accounting information
    - list of files in use by processor …



Process Life Cycle



Process Life Cycle

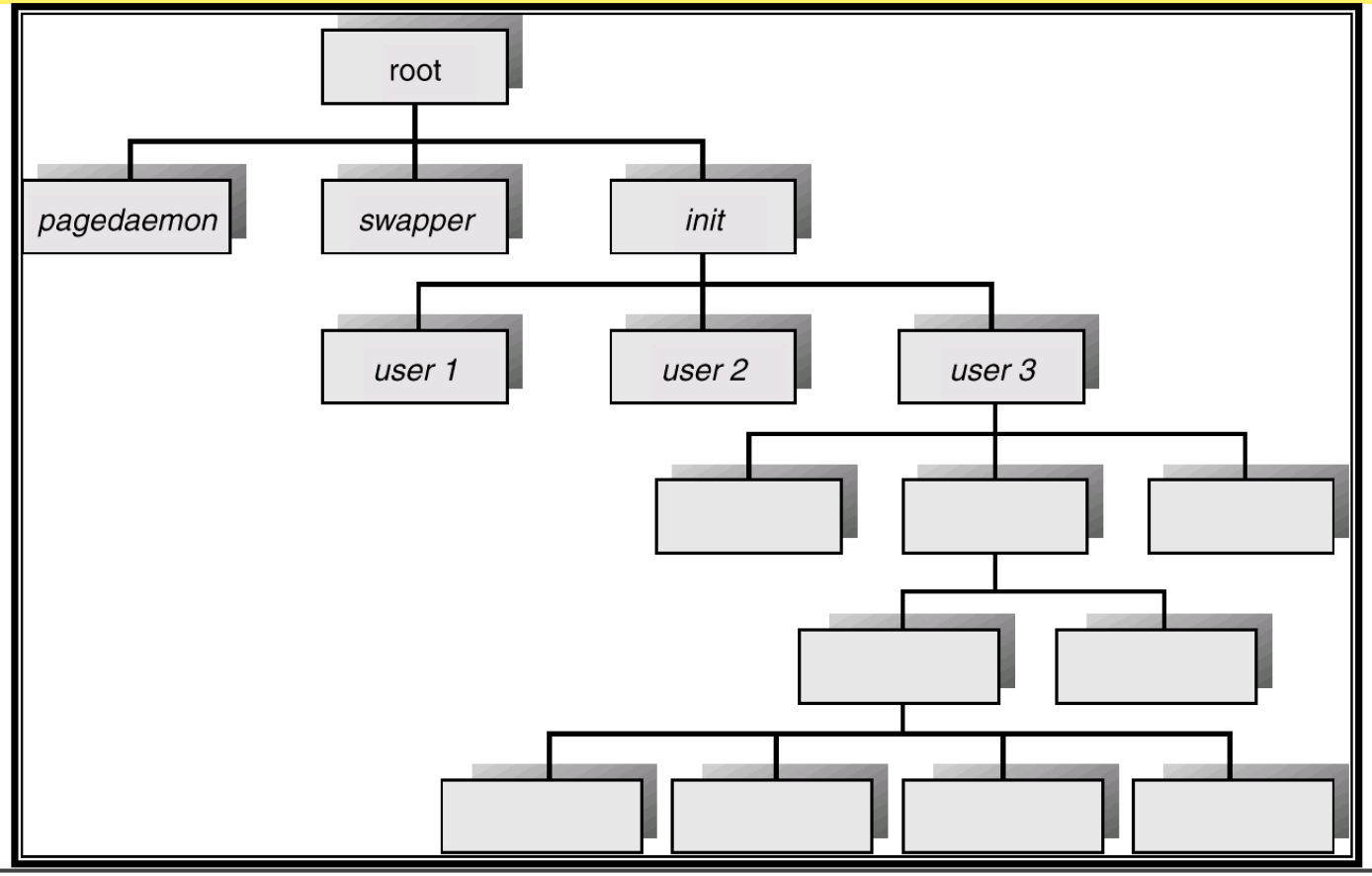
- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running** - The CPU is working on this process's instructions.
- **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur.
- **Terminated** - The process has completed.

Process Identifier

- Identified by unique id – pid (int)
- Variable type pid_t
- getpid() and getppid()

Process Creation

- Processes are identified using unique process identifier (pid) : integer number
- each process has one parent but zero, one, two, or more children

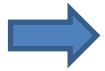


- **Resource sharing**
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.

- **Execution : 2 possibilities**
  - Parent and children execute concurrently.
  - Parent waits until some or all its children terminate.

- **Address space : two possibilities**
  - Child is a duplicate of parent. (program and data same)
  - Child has a new program loaded into it.

- **UNIX examples**
  - fork system call creates new process
  - exec system call used after a fork to replace the process' memory space with a new program.
  - ps command can be used to list processes



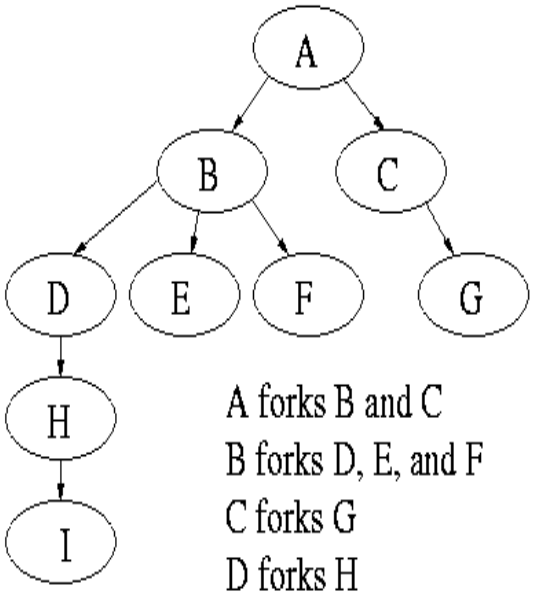
Reasons for process creation

- **Events that cause processes to be created**
  - System initialization.
  - Execution of a process creation system call by a running process
  - A user request to create a new process.
  - Initiation of a batch job.

- **When the OS creates a process at the explicit request of another process, the action is referred to as process spawning**

Create a Process

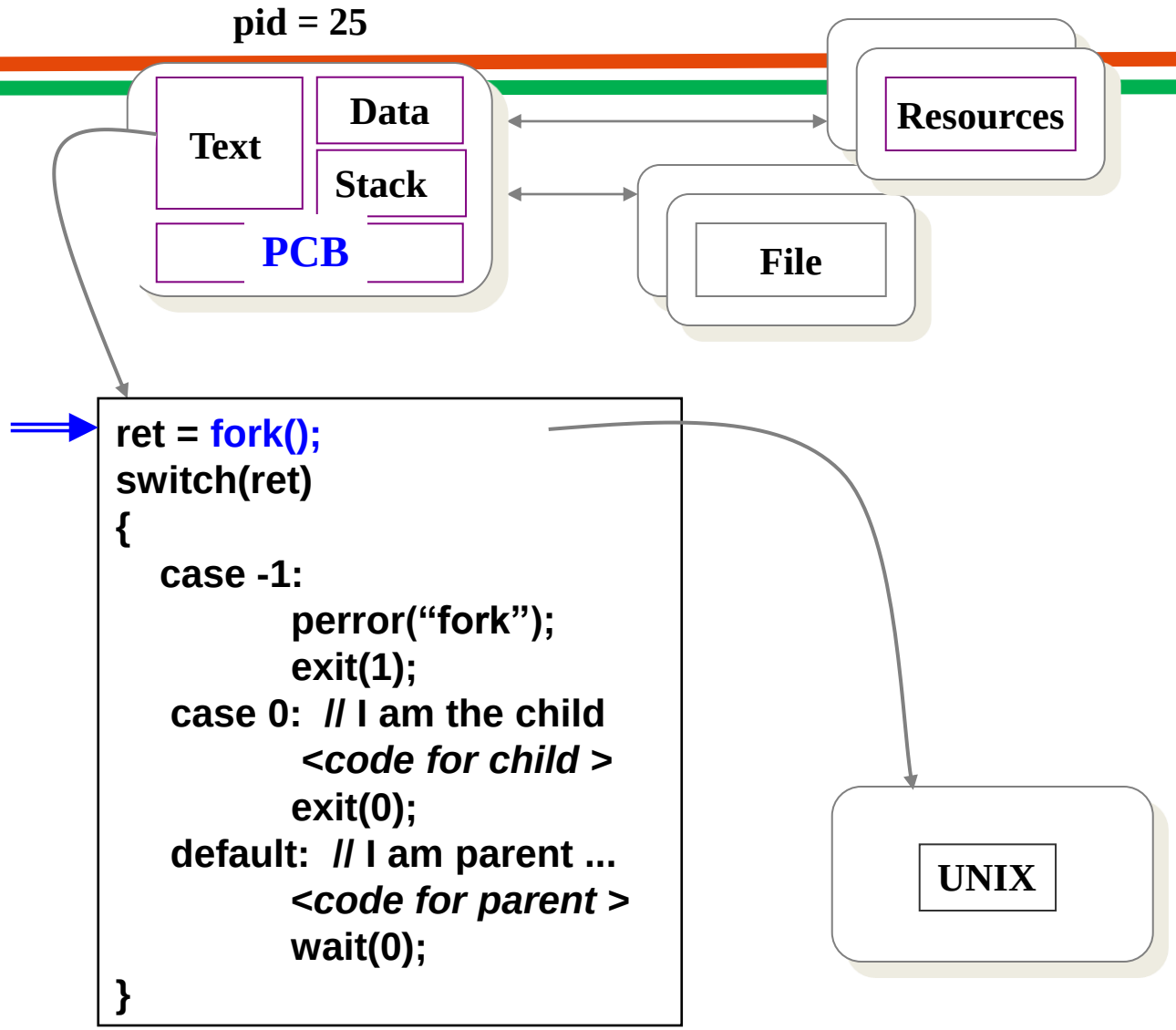
- Linux – fork system call is used to create a process
- Creates a child process by making a copy of the parent process --- an exact duplicate.
- Implicitly specifies code, registers, stack, data, files

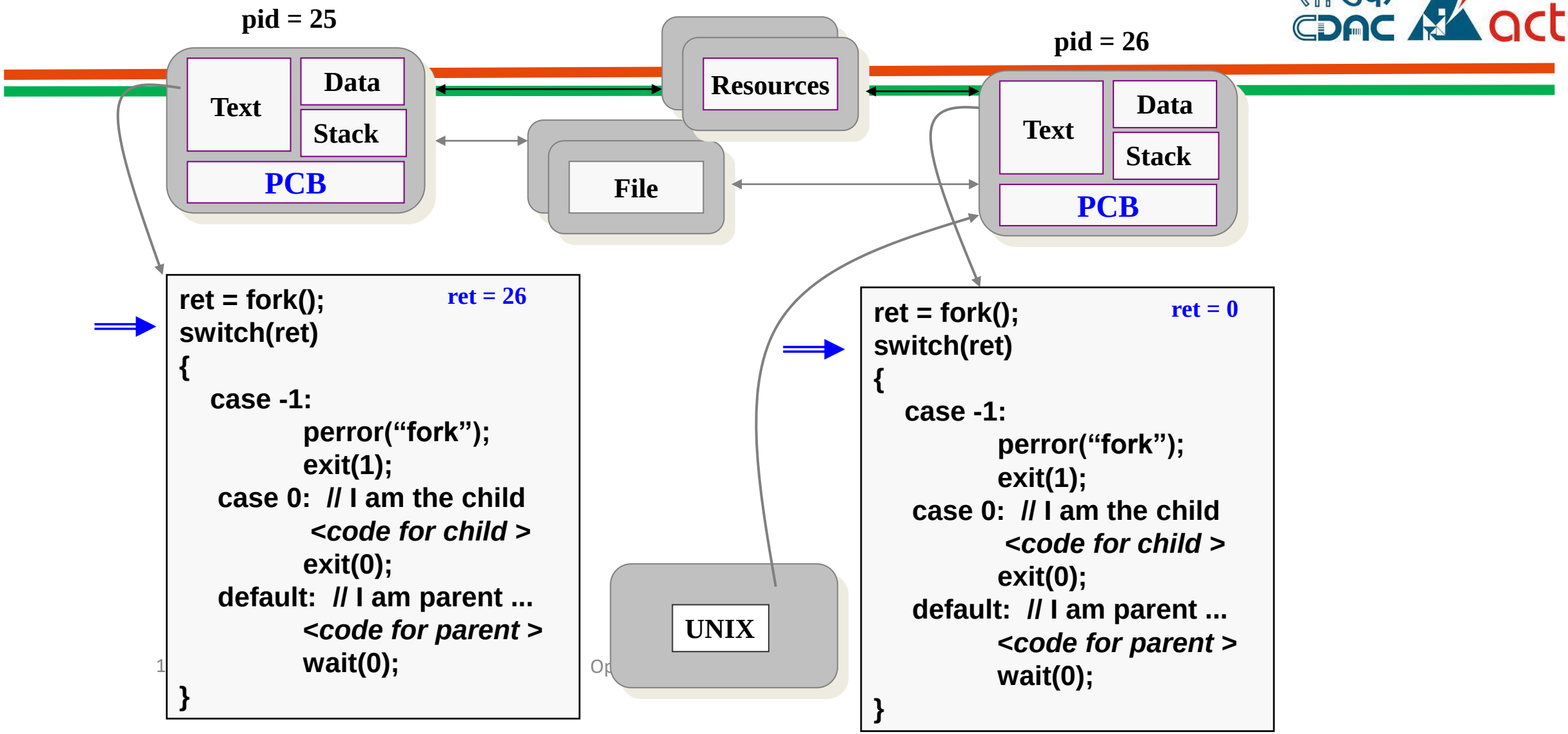


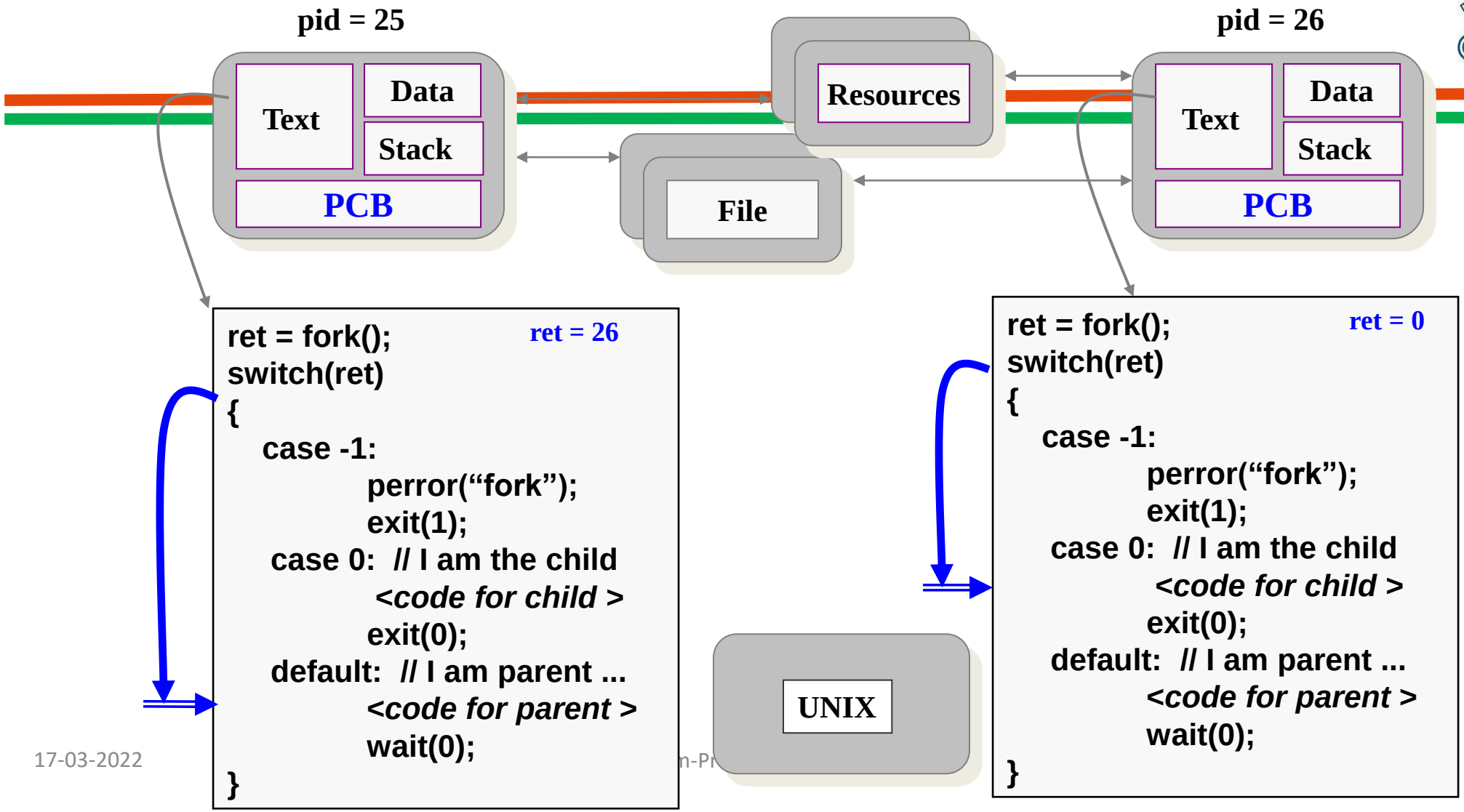
- **Fork()**

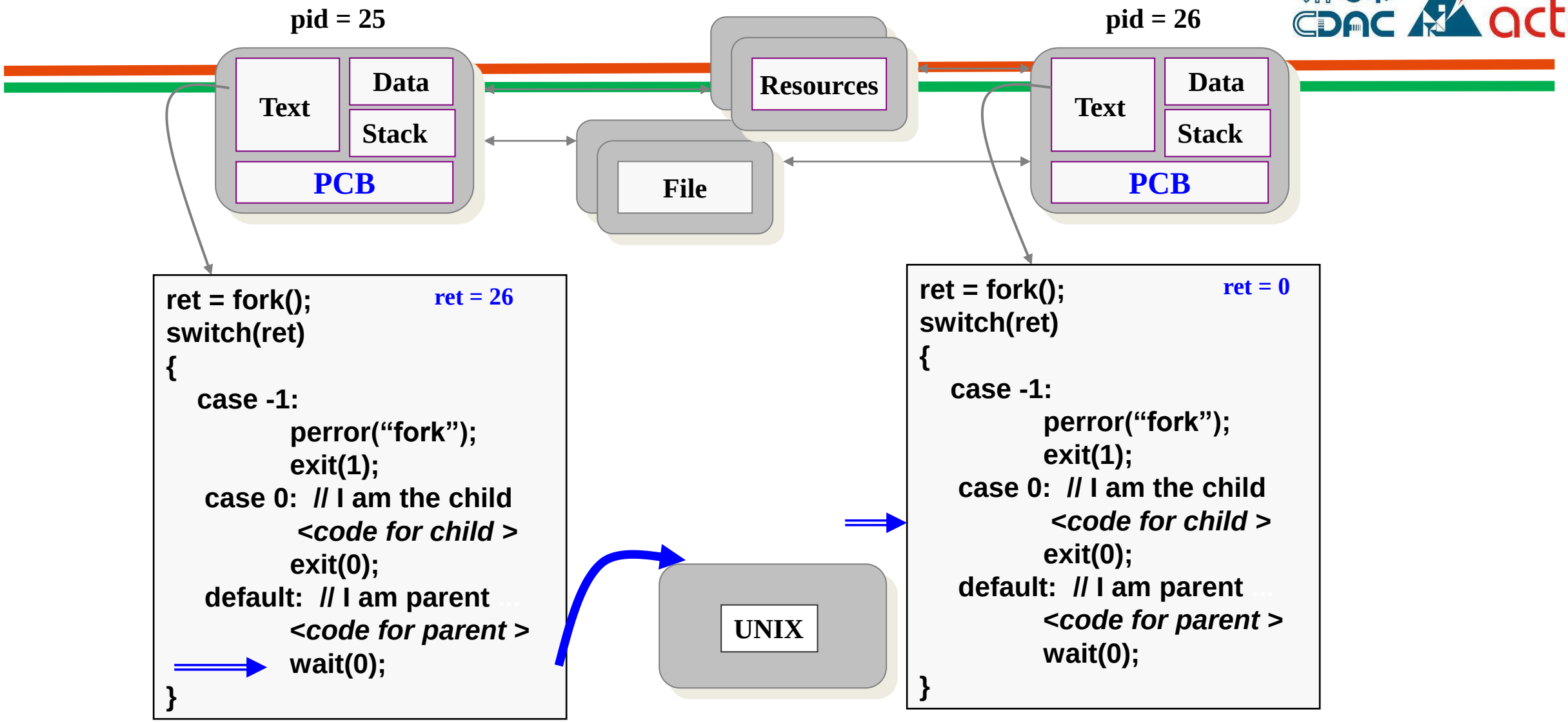
```
#include <sys/types.h>
#include <unistd.h>
pid_t fork( void );
```

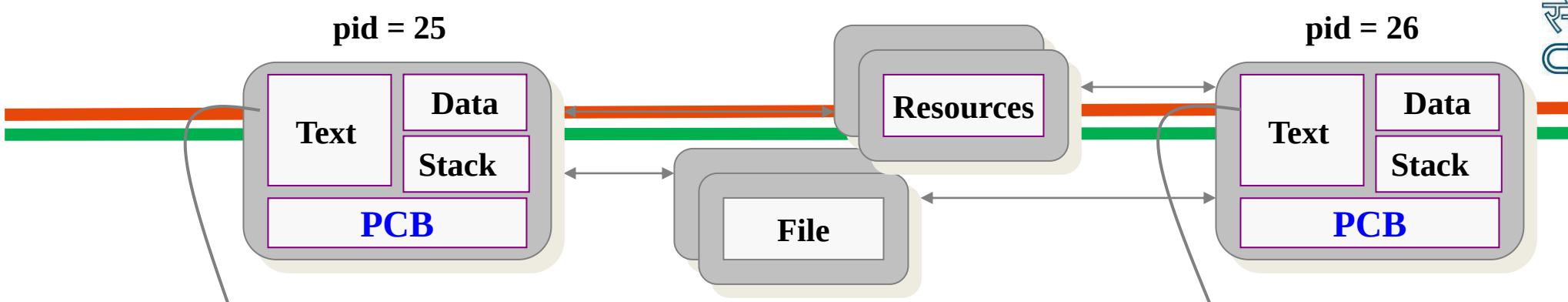
- **Both parent and child continue to execute**
  - In the child: ret == 0;
  - In the parent: ret == the process ID of the child.


pid = 25
Text
Data
Stack
PCB
Resources
File
ret = fork();
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(0);
}
UNIX


pid = 25
Text
Data
Stack
PCB
Resources
File
pid = 26
Text
Data
Stack
PCB
ret = fork();
ret = 26
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(0);
}
UNIX
ret = fork();
ret = 0
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(0);
}


pid = 25
Text
Data
Stack
PCB
Resources
File
pid = 26
Text
Data
Stack
PCB
ret = fork();
ret = 26
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(0);
}
UNIX
ret = fork();
ret = 0
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(0);
}

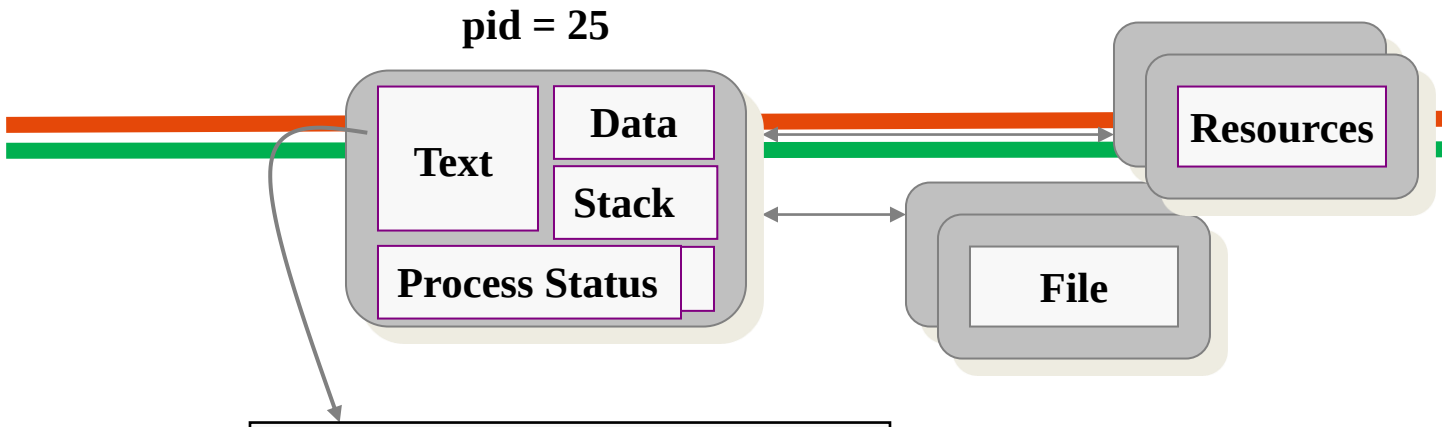

pid = 25
Text
Data
Stack
PCB
Resources
File
pid = 26
Text
Data
Stack
PCB
ret = fork();
ret = 26
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent
<code for parent >
wait(0);
}
UNIX
ret = fork();
ret = 0
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent
<code for parent >
wait(0);
}




```
ret = fork();                    ret = 26
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
             <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

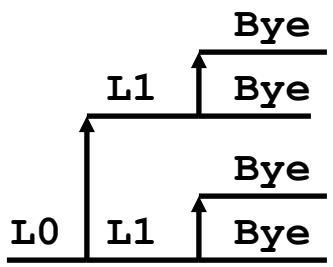
UNIX

```
ret = fork();                    ret = 0
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
             <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

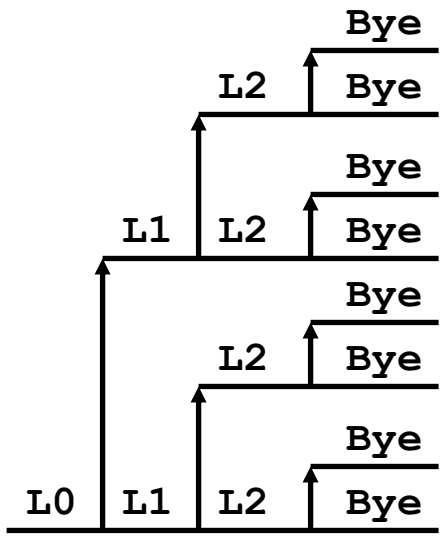



```
ret = fork();                    ret = 26
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
             <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
            < … >
```

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```




```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Process Termination

- **Process executes last statement and asks the operating system to decide it (exit).**
  - Output data from child to parent (via wait).
  - Process' resources are deallocated by operating system.

- **Parent may terminate execution of children processes (abort).**
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.

Exit()

- **To terminate child may call exit(number)**
- **The system call**
  - Saves result = argument of exit
  - Closes all open files, connections and deallocates memory
  - Checks if parent is alive
    - If parent is alive, holds the result value until the parent requests it (with wait);
    - in this case, the child process does not really die, but it enters a zombie/defunct state
    - If parent is not alive, the child terminates (dies)

Wait()

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

- **Waiting for any child to terminate: wait()**
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- **Example**
  - a shell waiting for operations to complete

Waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

- **By default, waitpid() waits only for terminated children, but this behaviour is modifiable via the options argument, as described below.**
- **The value of pid can be:**
  - < -1 : Wait for any child process whose process group ID is equal to the absolute value of pid.
  - -1 : Wait for any child process.
  - 0 : Wait for any child process whose process group ID is equal to that of the calling process.
  - > 0 : Wait for the child whose process ID is equal to the value of pid.

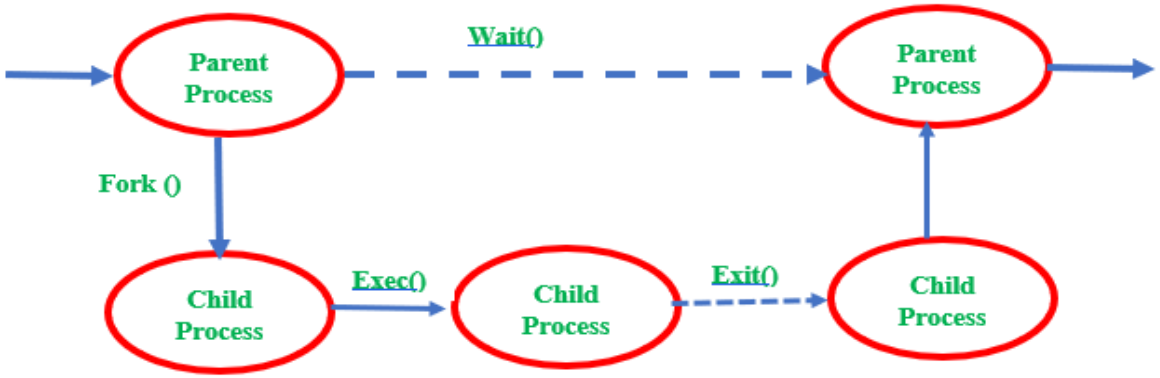
Types of Process

- Parent Process
- Child process

- Foreground process
- Background process

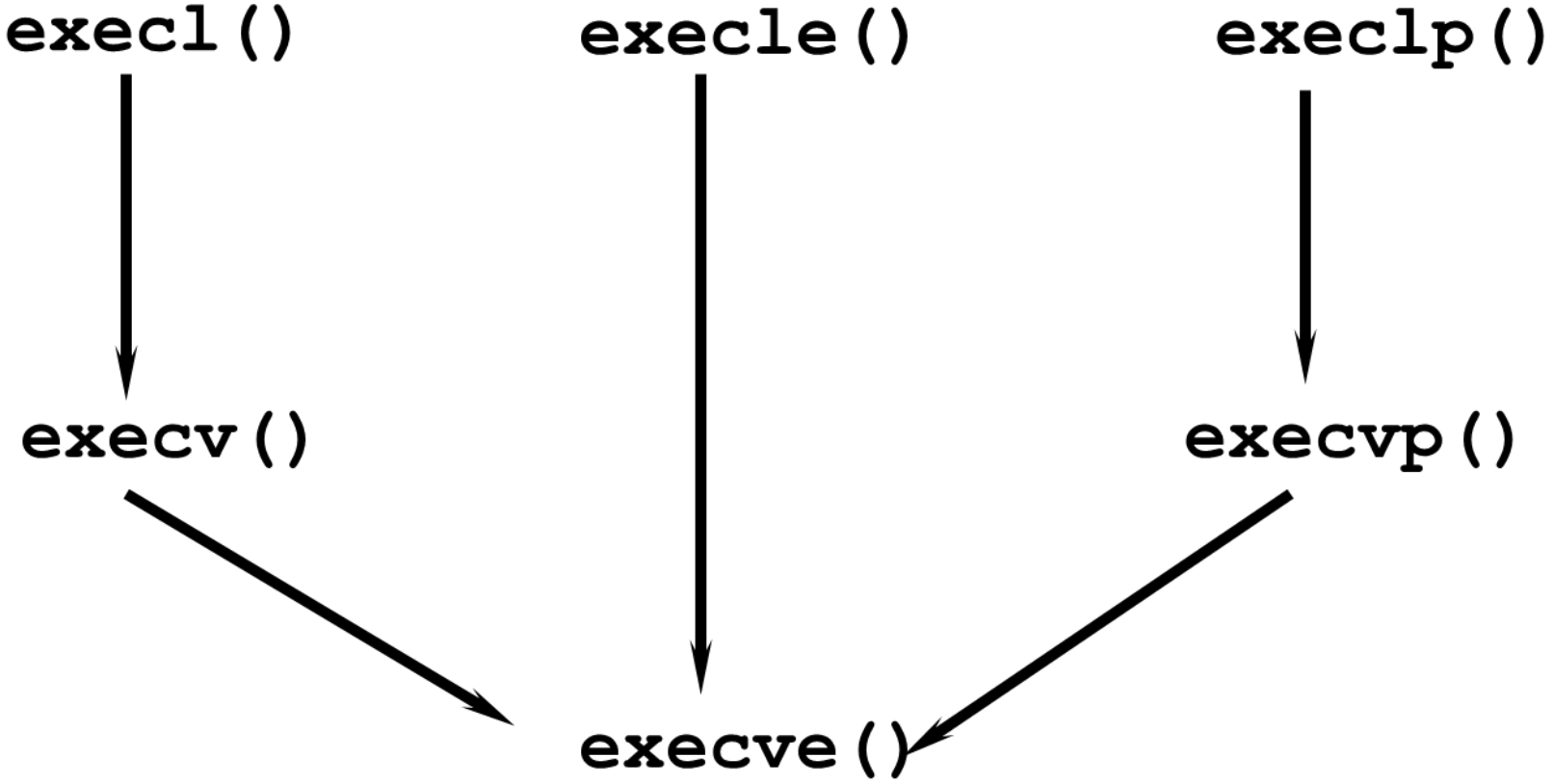
- Interactive process
- Non-interactive process

- Zombie process
- Orphan process
- Daemon Process



- **Zombie process**
  - process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state"
- **Orphan process**
  - An orphan process is a computer process whose parent process has finished or terminated, though it remains running itself.
- **Daemon Process**
  - It runs in background there is no direct control of a user. it remains in the background all the time and exits only when the system shuts down

Exec..() family



Execv()

- **The system call execv executes a file, transforming the calling process into a new process.**

execv(const char * path, char * const argv[])

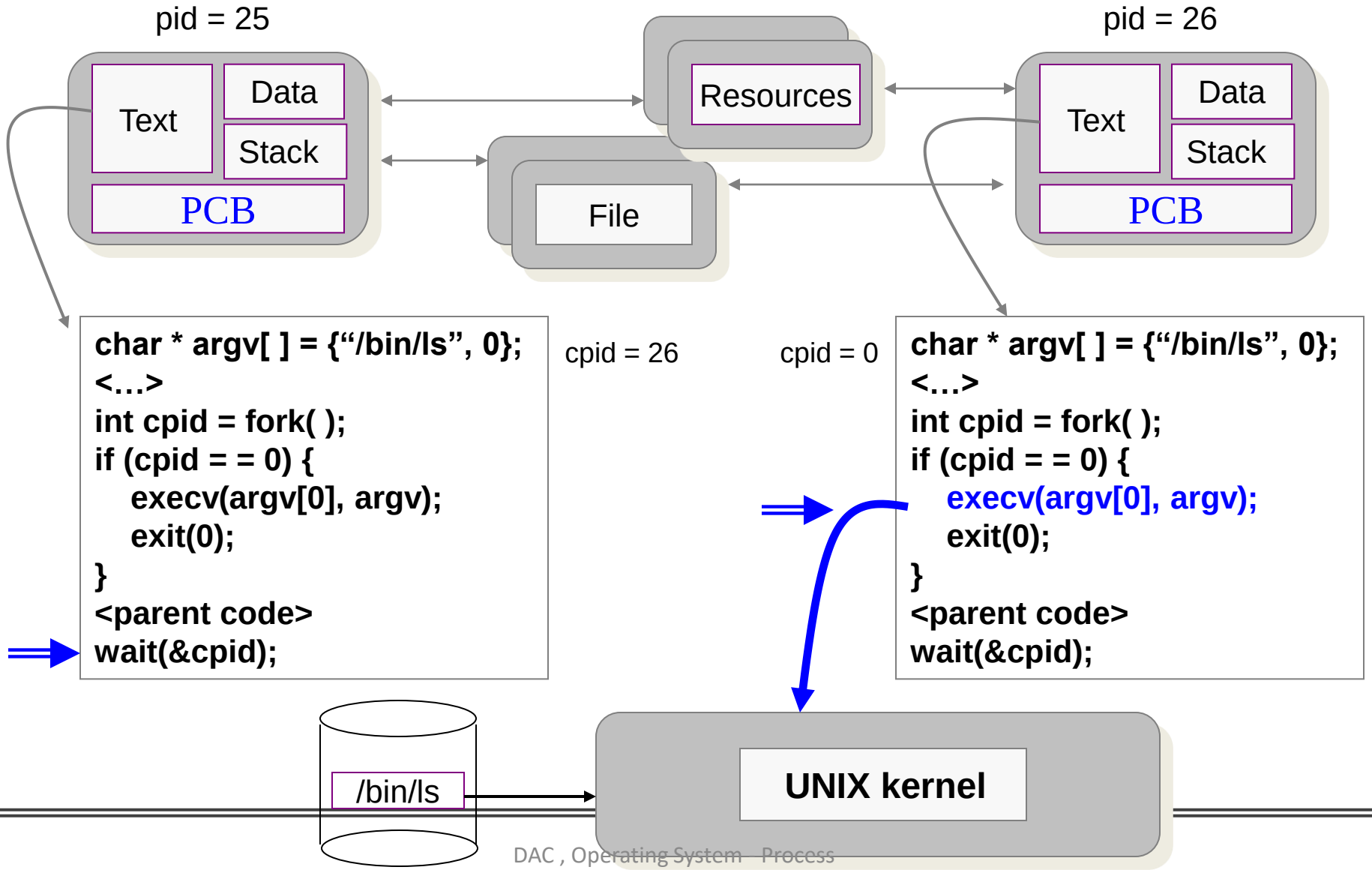
- path is the full path for the file to be executed
- argv is the array of arguments for the program to execute
  - each argument is a null-terminated string
  - the first argument is the name of the program
  - the last entry in argv is NULL
- `

execv Example

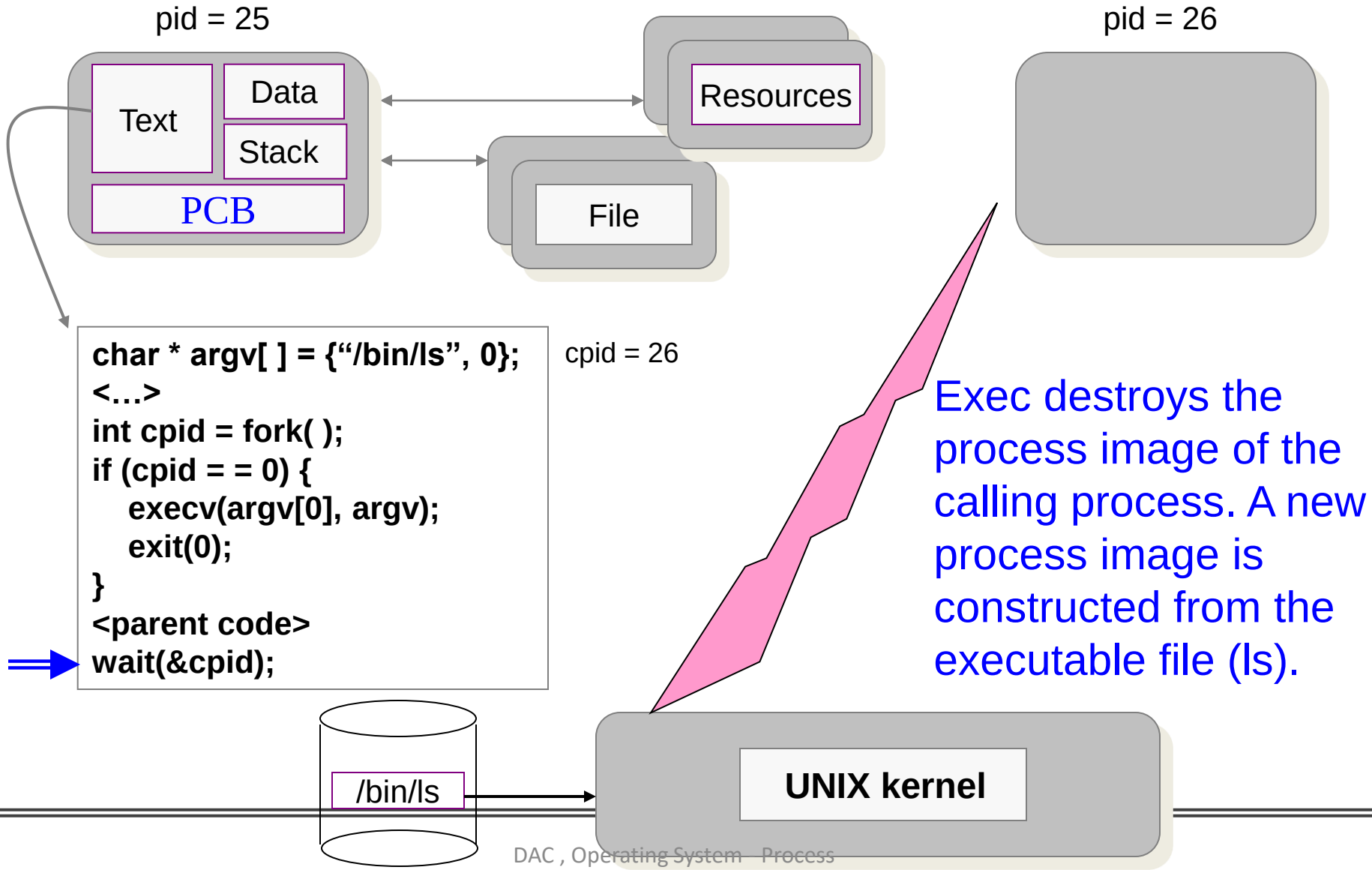
```
#include <stdio.h>
#include <unistd.h>

char * argv[] = {"/bin/ls", "-l", 0};
int main()
{
    int pid, status;

    if ( (pid = fork() ) < 0 )
    {
       printf("Fork error \n");
       exit(1);
    }
    if(pid == 0)        { /* Child executes here */
       execv(argv[0], argv);
    } else    /* Parent executes here */
       wait(&status);
    printf("Hello there! \n");
    return 0;
}
```

Note: the NULL string at the end

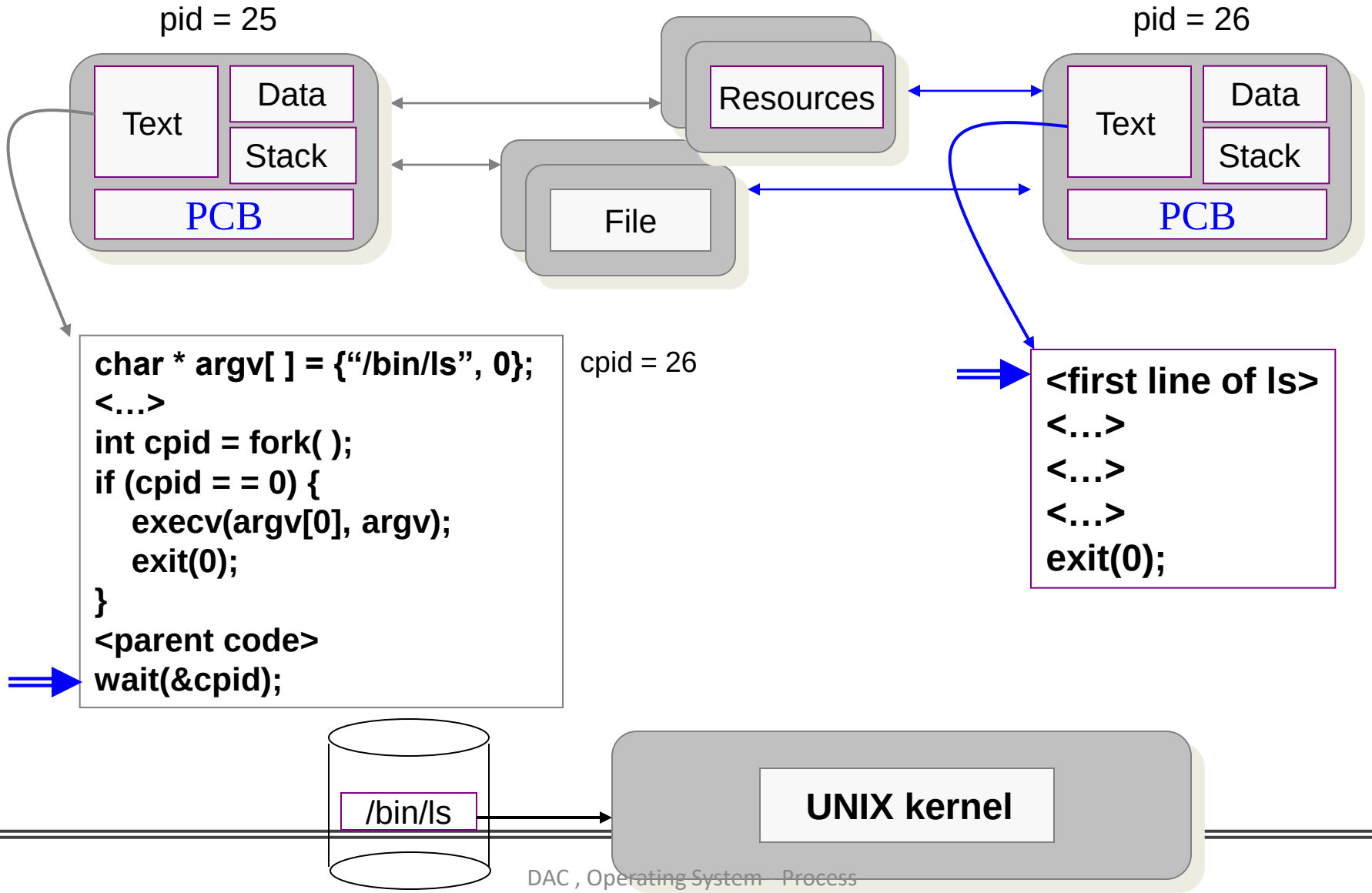
How execv Works (1)



How execv Works (2)



How execv Works (3)



execv Example

```
#include <stdio.h>
#include <unistd.h>

char * argv[] = {"/bin/ls", "-l", 0};
int main()
{
    int pid, status;

    if ( (pid = fork() ) < 0 )
    {
       printf("Fork error \n");
       exit(1);
    }
    if(pid == 0)        { /* Child executes here */
       execv(argv[0], argv);
       printf("Exec error \n");
       exit(1);
    } else    /* Parent executes here */
       wait(&status);
    printf("Hello there! \n");
    return 0;
}
```

Note: the NULL string at the end

execl

- Same as execv, but takes the arguments of the new program as a list, not a vector:

- Example:

```
execl("/bin/ls", "/bin/ls", "-l", 0);
```

- Is equivalent to

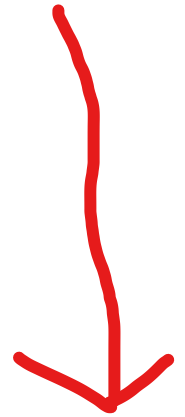
```
char * argv[] = {"/bin/ls", "-l", 0};
execv(argv[0], argv);
```

Note the NULL string at the end

- execl is mainly used when the number of arguments is known in advance

Variations of execv()

- **execv**
  - Program arguments passed as an array of strings
- **execvp**
  - Searches for the program name in the PATH environment
- **execl**
  - Program arguments passed directly as a list
- **execlp**
  - Searches for the program name in the PATH environment



Example : execvp

```
int spawn (char* program, char** arg_list)
{
pid_t child_pid;
child_pid = fork ();
if (child_pid != 0) { /* This is the parent process. */    return child_pid; }
else {
        execvp (program, arg_list);
        fprintf (stderr, "an error occurred in execvp\n");
        abort ();}}
int main (){
char* arg_list[] = { "ls", "-l","/",NULL };
spawn ("ls", arg_list);
printf ("done with main program\n");
return 0;
}
```

exec variations

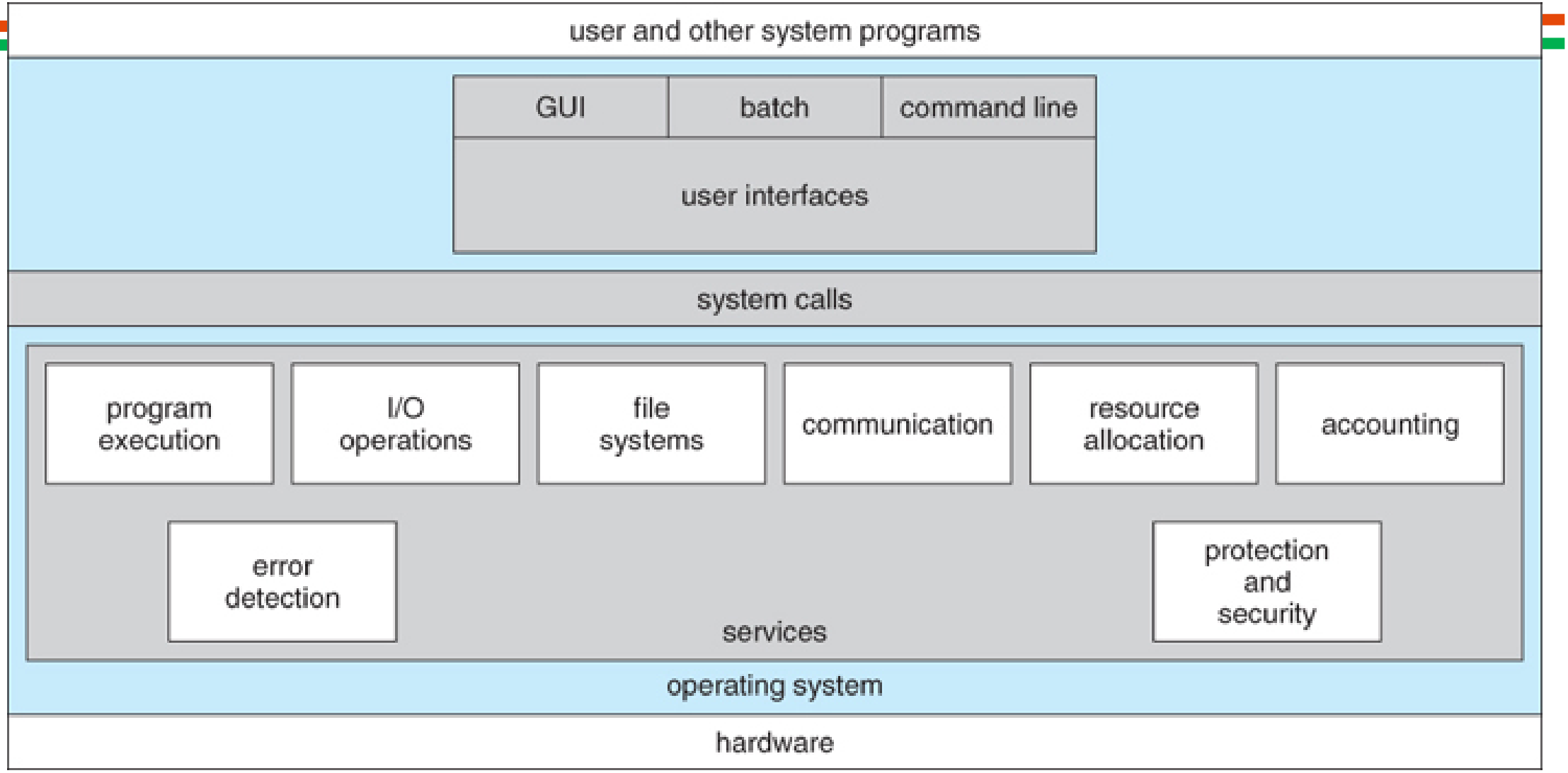
- int execl(const char *path, const char *arg, ...);
- int execlp(const char *file, const char *arg, ...);
- int execle(const char *path, const char *arg, ..., char *const envp[]);
- int execv(const char *path, char *const argv[]);
- int execvp(const char *file, char *const argv[]);
- int execve(const char *filename, char *const argv [], char *const envp[]);

Combine fork,exec,wait

- **Single call that combines all three**
  - int system(const char *cmd);

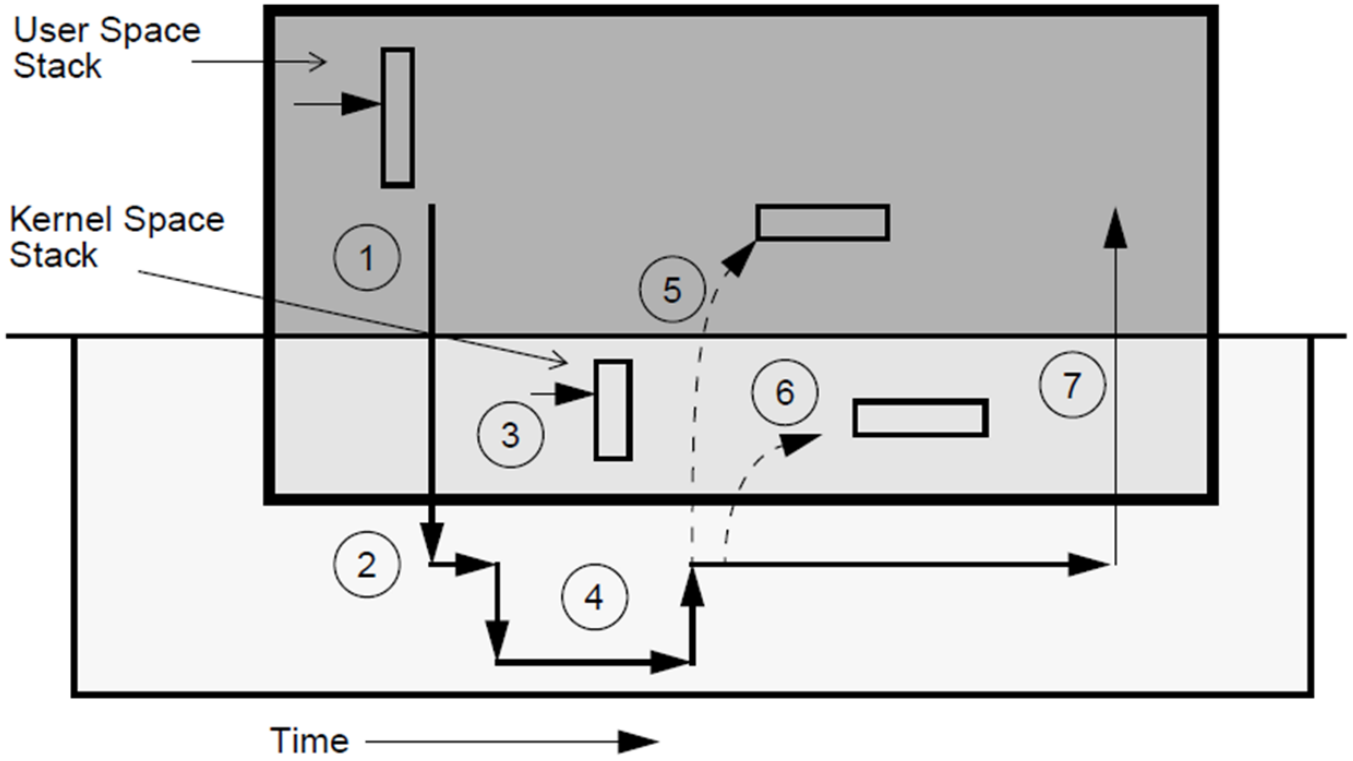
- **Example**

```
int main()
{
   system("echo Hello world");
}
```


user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

System Call

- **The system call is the means by which a process requests a specific operating system service.**



- **When a process makes a system call, the following events occur:**

1. The process traps to the kernel.
2. The trap handler runs in kernel mode, and saves all of the registers.
3. It sets the stack pointer to the process structure's kernel stack.
4. The kernel runs the system call.
5. The kernel places any requested data into the user-space structure that the programmer provided.
6. The kernel changes any process structure values affected.
7. The process returns to user mode, replacing the registers and stack pointer, and returns the appropriate value from the system call.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

System API

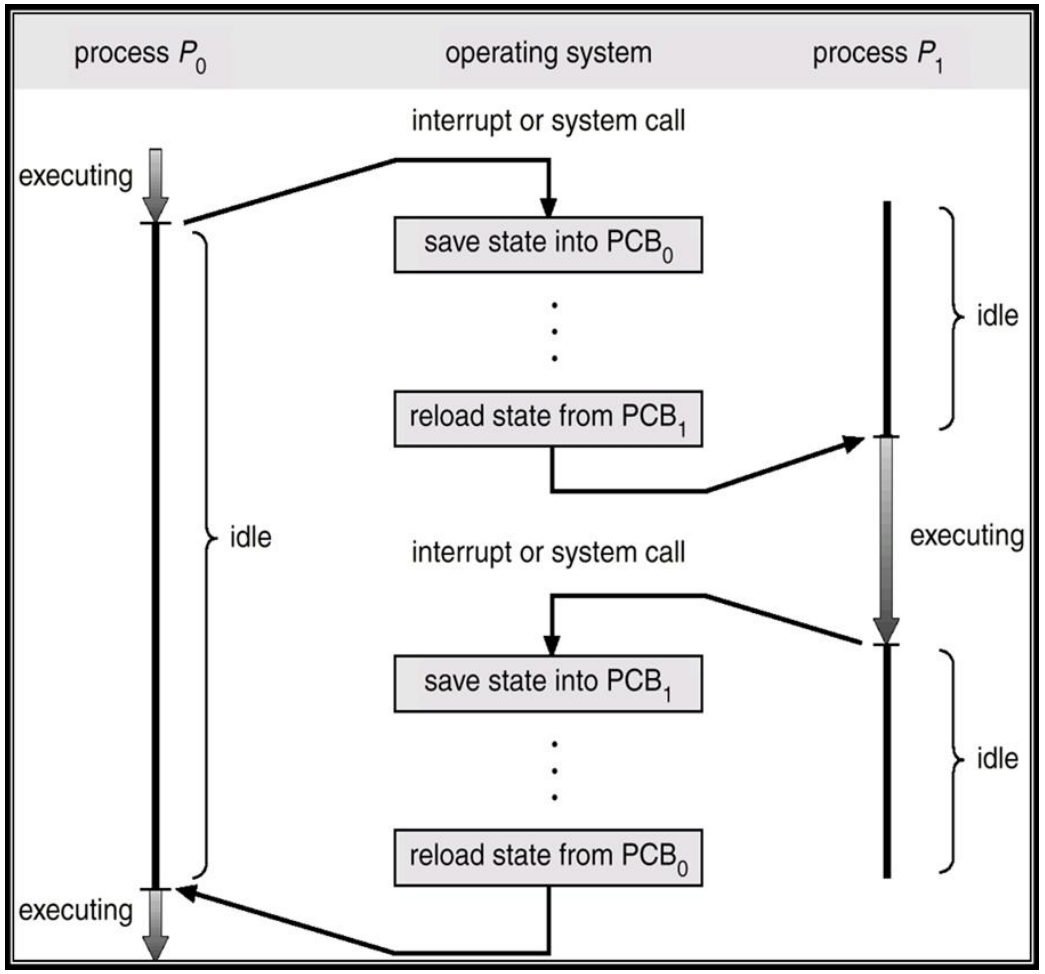
- **Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use**

- **Three most common APIs are**
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
int main(void) {
 long ID1, ID2;
 /*--------------------------*/
 /* direct system call */
 /* SYS_getpid (func no. is 20) */
 /*--------------------------*/
 ID1 = syscall(SYS_getpid);
 printf ("syscall(SYS_getpid)=%ld\n", ID1);
 /*--------------------------*/
 /* "libc" wrapped system call */
 /* SYS_getpid (Func No. is 20) */
 /*--------------------------*/
 ID2 = getpid();
 printf ("getpid()=%ld\n", ID2);
 return(0);
}
```

- **Dispatcher & swap**
- **Preemptive and non-preemptive Scheduling**
- **Scheduling Algorithms**
  - FCFS
  - SJF & SRTF
  - RR

CPU Switch



1	5000	27	12004
2	5001	28	12005
3	5002	----------Timeout	
4	5003	29	100
5	5004	30	101
6	5005	31	102
----------Timeout		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----------Timeout	
16	8003	41	100
----------I/O Request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		----------Timeout	

Dispatcher

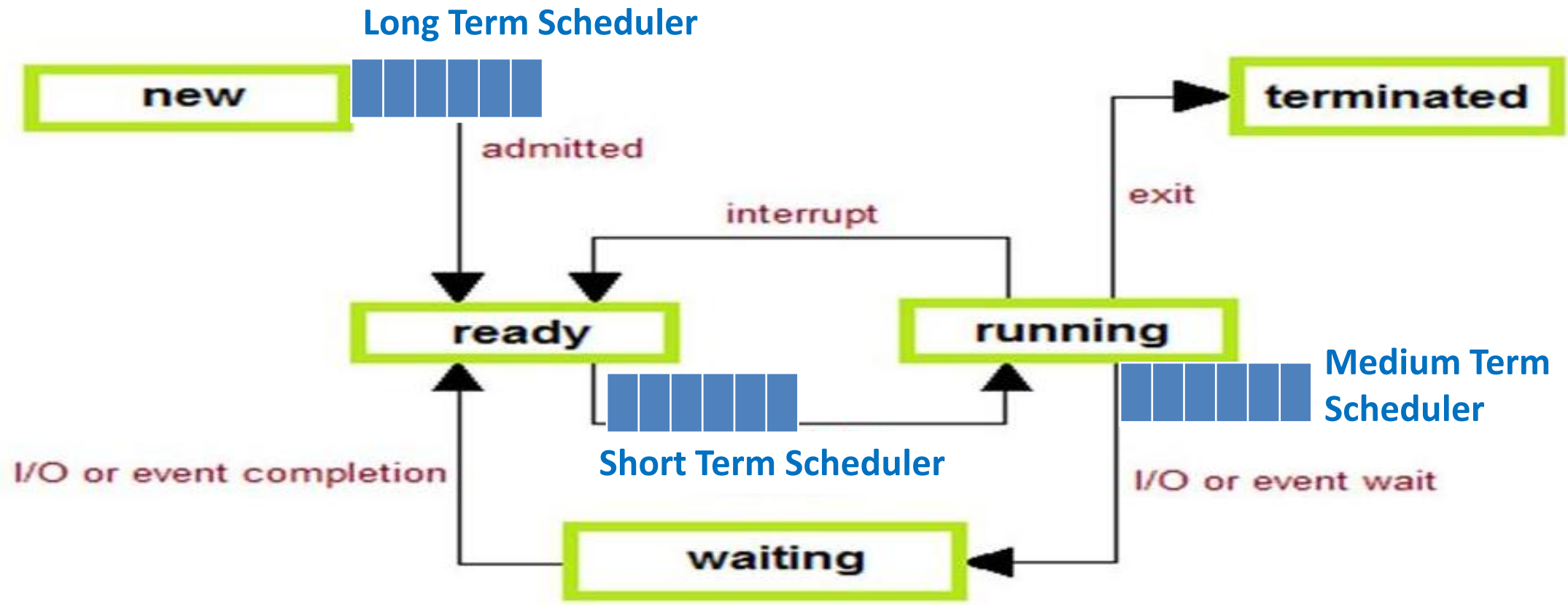
- **Module which gives control of CPU to the process selected by short term scheduler**
  - Switch context
  - Switching to user mode
  - Jump to exact location in the program to restart the process

- **Dispatch latency**
  - Time taken by dispatcher to stop on process and start execution of the other process

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----------	Timeout
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
----------	I/O Request
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
----------	Timeout
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----------	Timeout
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----------	Timeout

Scheduler

- **Long term (Job Scheduler)**
  - selects which processes should be brought into the ready queue
  - Degree of multiprogramming
  - I/o bound or CPU bound
- **Medium term (swapping scheduler)**
  - Swapping is necessary to improve the process mix. Swap-in and swap-out of processes.
- **Short term (CPU scheduler)**
  - selects which process should be executed next and allocates CPU


Long Term Scheduler
new
admitted
terminated
exit
interrupt
ready
running
Medium Term Scheduler
Short Term Scheduler
I/O or event completion
I/O or event wait
waiting

- **Non-preemptive scheduling**
  - when a process terminates or
  - when an explicit system request causes a wait state

- **Preemptive scheduling**
  - An interrupt occurs
  - When new processes become ready with higher priority

Acronyms

- **CPU utilization –** keep the CPU as busy as possible
- **Throughput –** # of processes, complete their execution per time unit
- **Turnaround time –** amount of time taken between submission of program to execute and return of the output.
- **Waiting time –** amount of time a process waits in the ready queue
- **Response time –** amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- **Arrival Time -** time when a process enters into the ready state and is ready for its execution

Algorithms

- **First-come, First-Served (FCFS)**
  - Complete the jobs in order of arrival
- **Shortest Job First (SJF)**
  - Complete the job with shortest next CPU requirement (e.g., burst)
  - Provably optimal w.r.t. average waiting time
- **Priority**
  - Processes have a priority number
  - Allocate CPU to process with highest priority
- **Round-Robin (RR)**
  - time quantum or time slice based
  - For now, assume a FIFO queue of processes

FCFS

- **Draw Gantt chart and compute average wait time**

Process	Burst time
P1	7
P2	3
P3	4
P4	6

0 7 10 14 20

Process	Burst time	Waiting Time
P1	7	0
P2	3	7
P3	4	10
P4	6	14
	Average	7.75

SJF

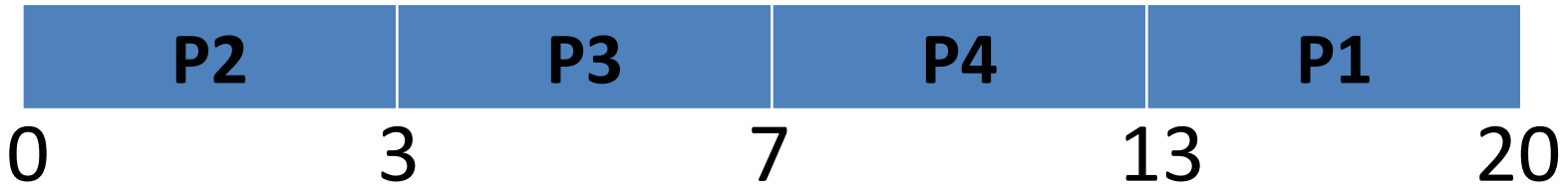
- **Non pre-emptive**
  - once CPU given to the process it cannot be preempted until completes its CPU burst

- **Pre-emptive**
  - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)

SJF

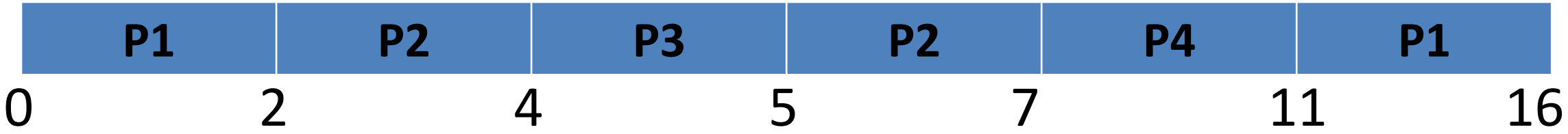
Process	Burst time	Waiting Time
P1	7	13
P2	3	0
P3	4	3
P4	6	7
		Avg: 5.75

Process	Arrival Time	Burst time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4



SRTF

Process	Arrival Time	Burst time	Waiting Time
P1	0.0	7	9
P2	2.0	4	1
P3	4.0	1	0
P4	5.0	4	2
			Avg : 3

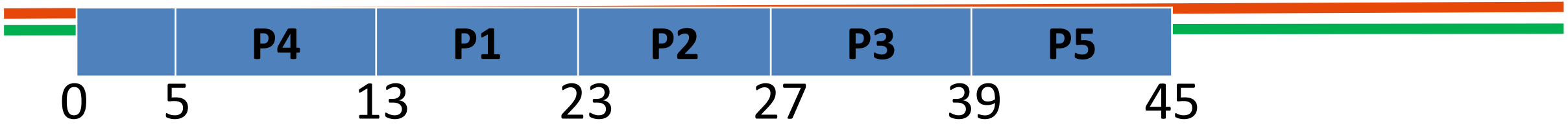


Scheduling with Priority

- **Calculate TAT and waiting time**
  - Turnaround time = Process finish time – Arrival time
  - Waiting time = Turnaround time – CPU Burst time

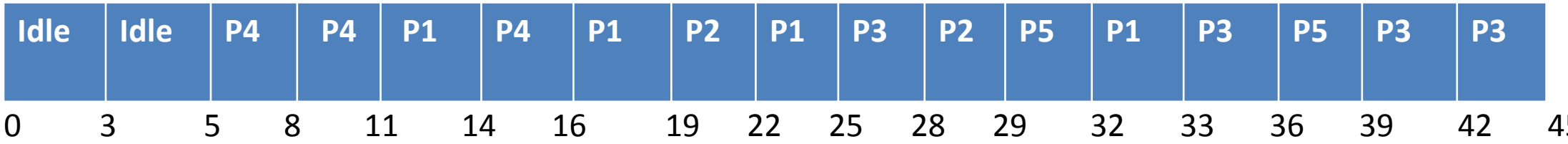
Process	Arrival Time	Burst time	Priority
P1	10	10	2
P2	15	4	1
P3	20	12	4
P4	5	8	3
P5	25	6	5

Priority



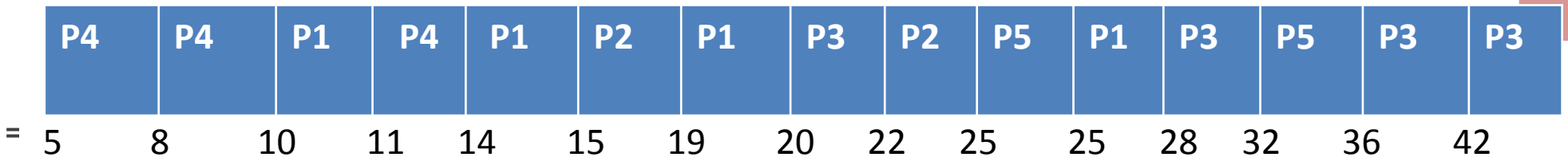
Process	Arrival Time	Burst time	Priority	Finish Time	Turnaround time	Waiting time
P1	10	10	2	23	13	3
P2	15	4	1	27	12	8
P3	20	12	4	39	19	7
P4	5	8	3	13	8	0
P5	25	6	5	45	20	14

- **Average Turnaround time : 14.4 and average waiting time is 6.4**



Process	Arrival Time	Burst time	Finish Time	Turnaround time	Waiting time
P1	10	10	33	23	13
P2	15	4	29	14	10
P3	20	12	45	25	13
P4	5	8	16	11	3
P5	25	6	39	14	8
				Avg: 87/5 =17.4	Avg: 47/5 =9.4

Request Queue



Part 1 -Topics

- Basics of computer memory
- Need for Memory Management
- Concepts
  - Address, Address space
  - Address Translation
  - Address Binding
  - Loading, Linking
  - Swapping
- Memory Allocation
  - Continuous/Contiguous Allocation
    - Fixed/Static Partitioning
    - Internal and External Fragmentation, Compaction
    - Variable/Dynamic Partitioning
    - Memory Allocation Techniques
      - First Fit
      - Best Fit
      - Worst Fit

Basics of Memory

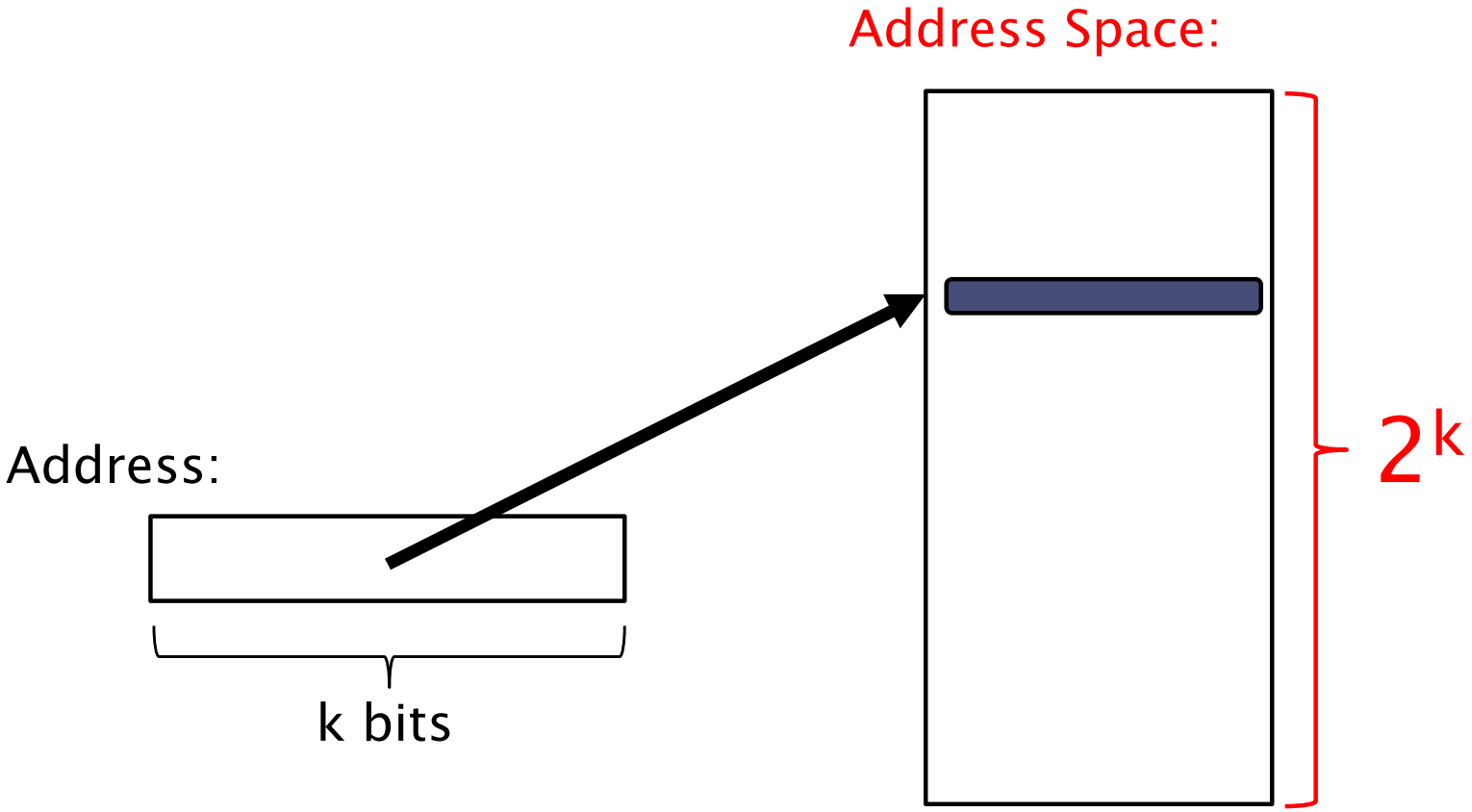
- Basic unit of memory is bit.
- A bit is a single unit of digital information and is represented either as 0 or a 1.
- Everything in a computer's memory takes the form of bits i.e. 0's and 1's.
- Example you are watching a movie, opening a text file etc , to the computer they are all 0's and 1's.
- Each of these are stored in a memory cell which can switch between two states 0 and 1.
- Files and programs consists of millions of these bits.
- These are all processed in the Central Processing Unit (CPU).
- CPU acts as the computer's brain.

Since a computer operates only on bits ie. 0s and 1s, hence the units are represented as power of 2s.

Multiples of Bits		
Unit (Symbol)	**Value (SI)**	**Value (Binary)**
Kilobit (Kb) (Kbit)	10^3	2^{10}
Megabit (Mb) (Mbit)	10^6	2^{20}
Gigabit (Gb) (Gbit)	10^9	2^{30}
Terabit (Tb) (Tbit)	10^{12}	2^{40}
Petabit (Pb) (Pbit)	10^{15}	2^{50}
Exabit (Eb) (Ebit)	10^{18}	2^{60}
Zettabit (Zb) (Zbit)	10^{21}	2^{70}
Yottabit (Yb) (Ybit)	10^{24}	2^{80}

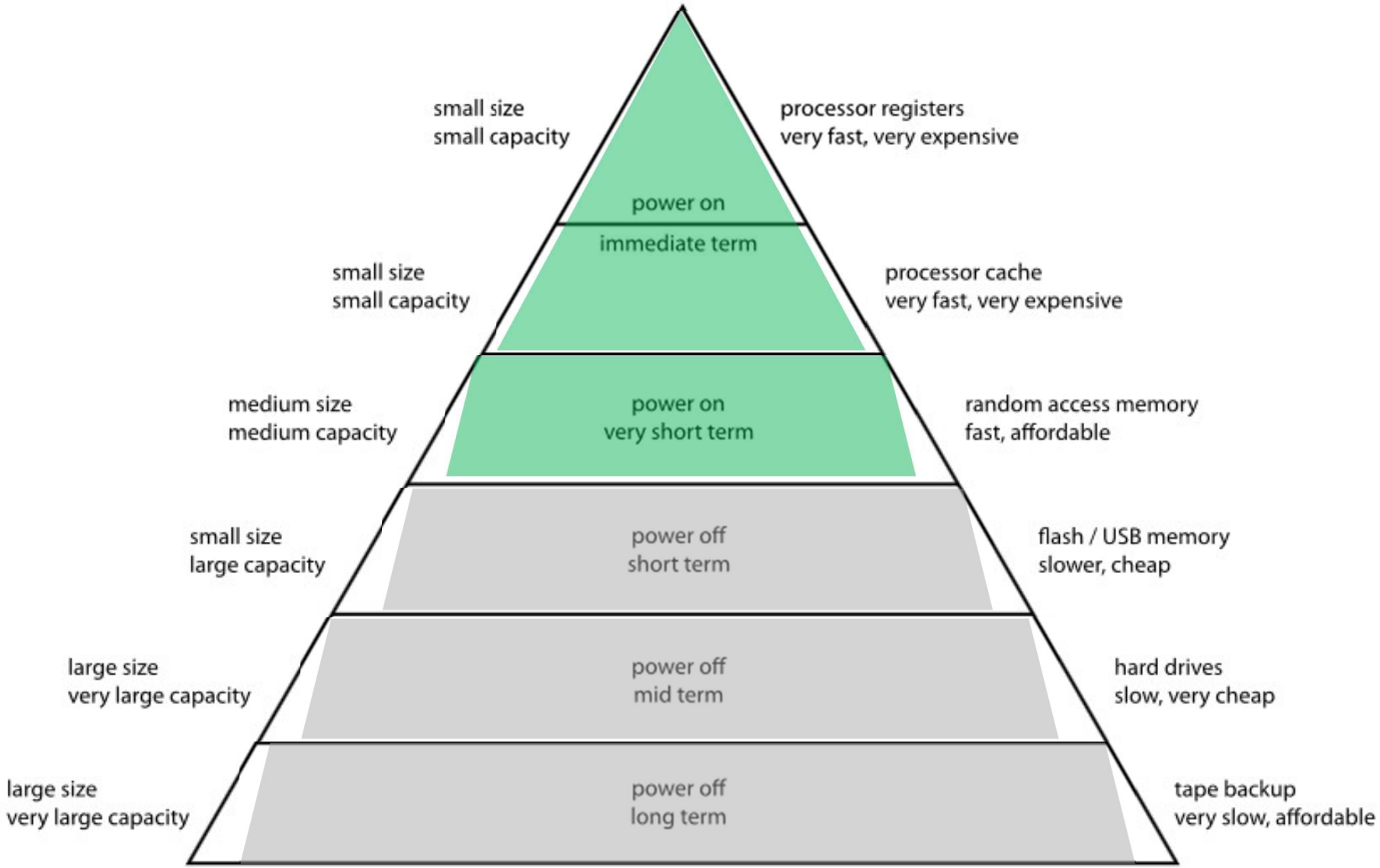
Multiples of Bytes		
Unit (Symbol)	**Value (SI)**	**Value (Binary)**
Kilobyte (kB)	10^3	2^{10}
Megabyte (MB)	10^6	2^{20}
Gigabyte (GB)	10^9	2^{30}
Terabyte (TB)	10^{12}	2^{40}
Petabyte (PB)	10^{15}	2^{50}
Exabyte (EB)	10^{18}	2^{60}
Zettabyte (ZB)	10^{21}	2^{70}
Yottabyte (YB)	10^{24}	2^{80}

Address/Address Space



- What is 2^{10} bytes (where a byte is abbreviated as "B")?
  - 2^{10} B = 1024b = 1 KB (for memory, 1K = 1024, *not* 1000)
- How many bits to address each byte of 4KB page?
  - 4KB = 4×1KB = 4× 2^{10}= 2^{12}⇒ 12 bits (or $\log_2 4$)
- How much memory can be addressed with 20 bits? 32 bits? 64 bits?

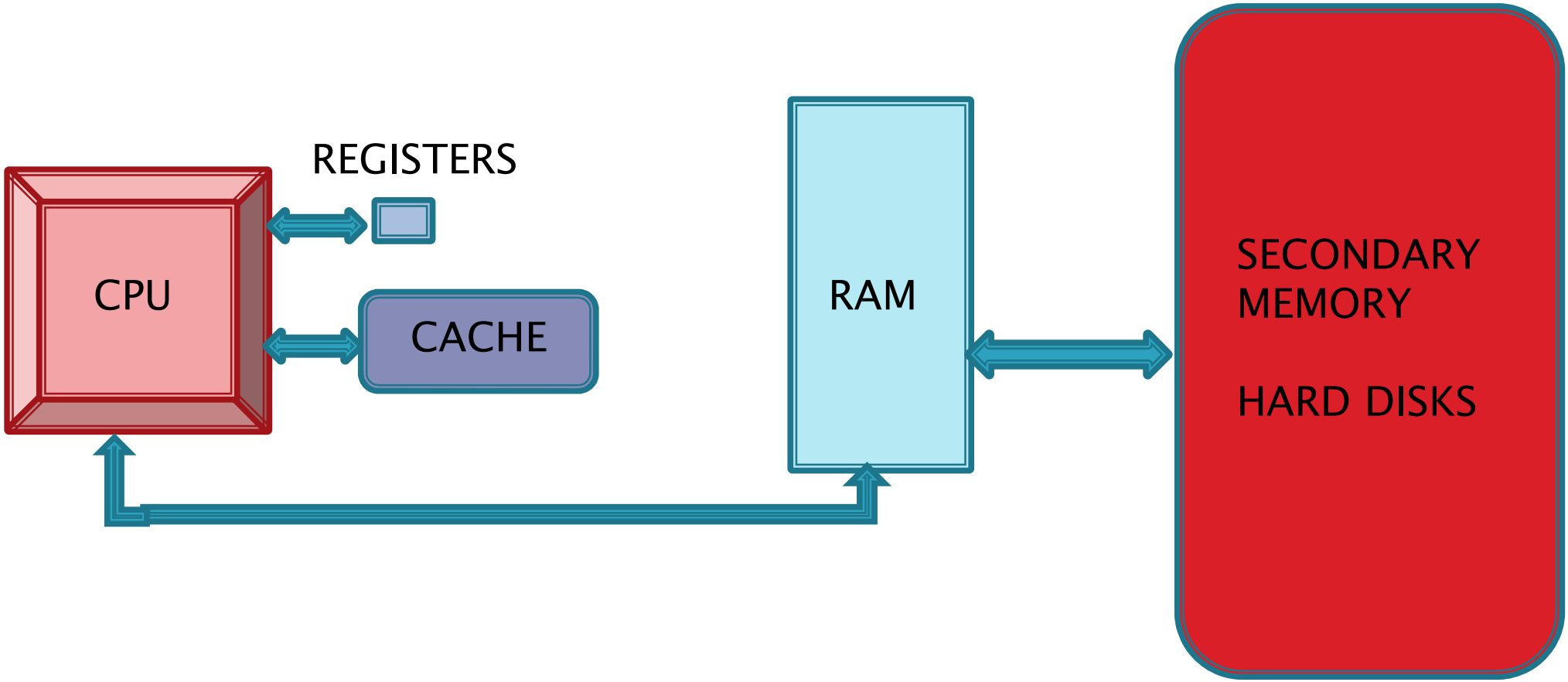
Computer Memory Hierarchy



Need for Memory Management

- As the number of bits to be processed grows exponentially, the need for memory management becomes essential.
- The following factors provide the basis of memory management:
  - Size – More the better
  - Cost – Less the better
  - Speed – More the better
- If we try to increase the size, the speed of access to data will decrease.
- We need to find an optimal solution which will provide the desired results.
- Memory management is one of the most important features of the operating system because it affects the execution time of process directly

Memory Hardware



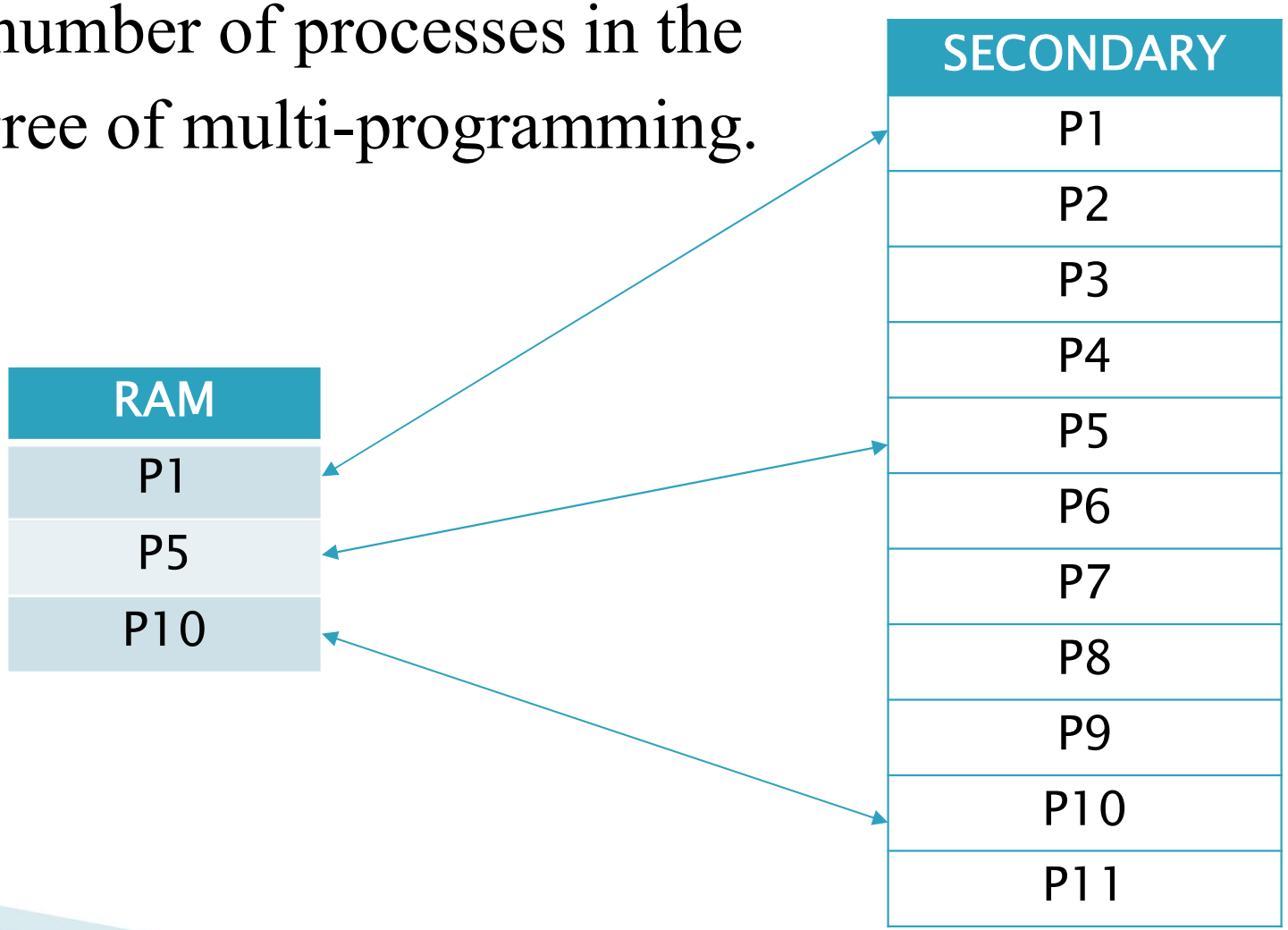
CPU does not have access to Secondary memory directly.

Concepts

- Program must be brought (from disk) into memory for it to be executed
- Main memory and registers are the only storage which CPU can access directly
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory is required to ensure correct operation

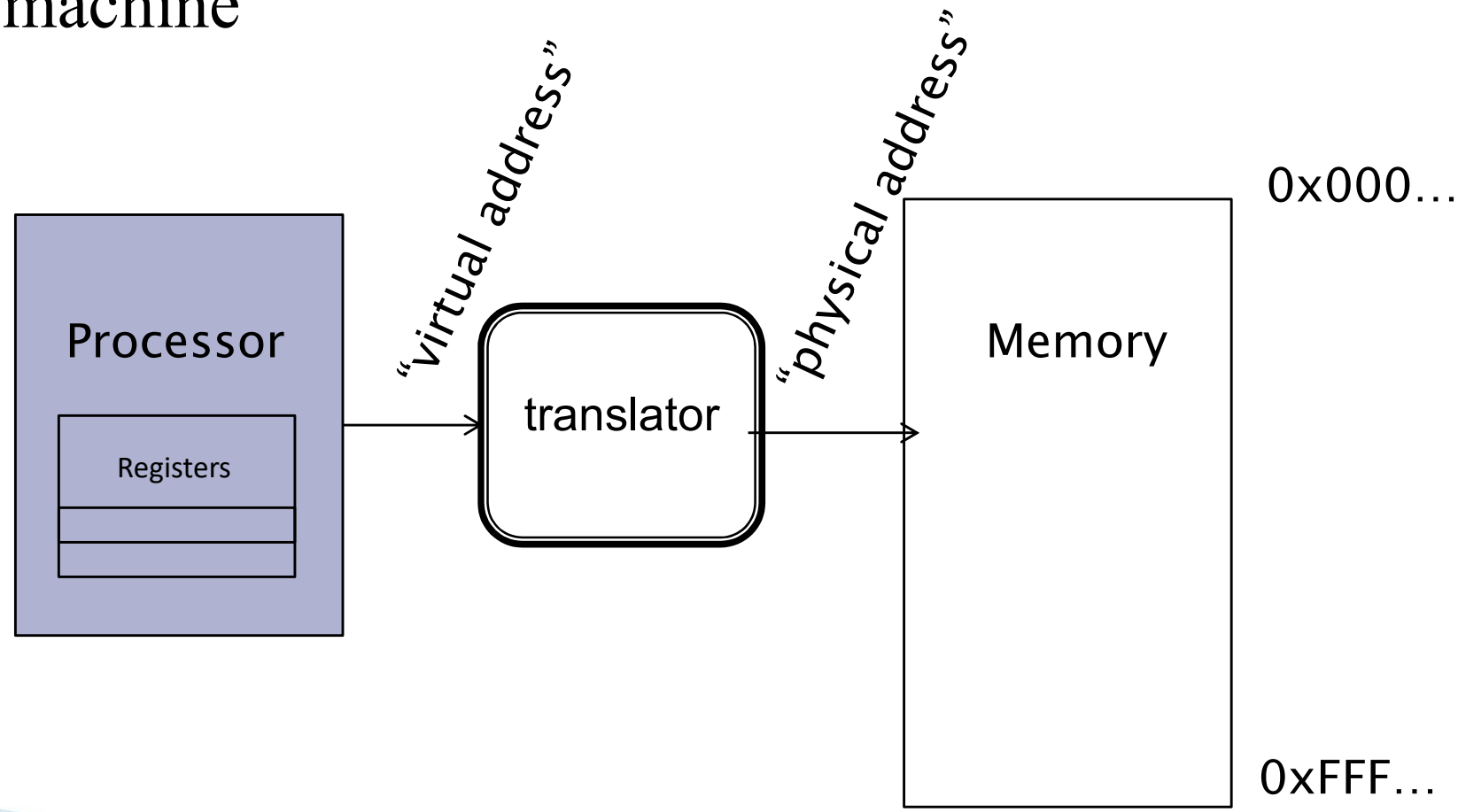
Degree of Multi-programming

- All programs are stored in secondary memory/storage.
- To execute the programs, we need to load them into RAM
- The maximum number of processes in the RAM is the degree of multi-programming.



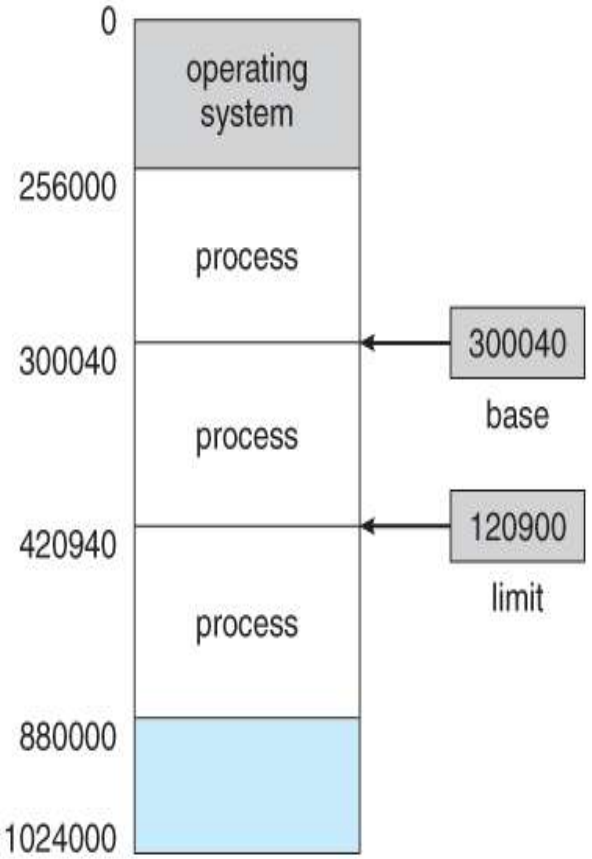
Address Translation

- Program operates in an address space that is distinct from the physical memory space of the machine

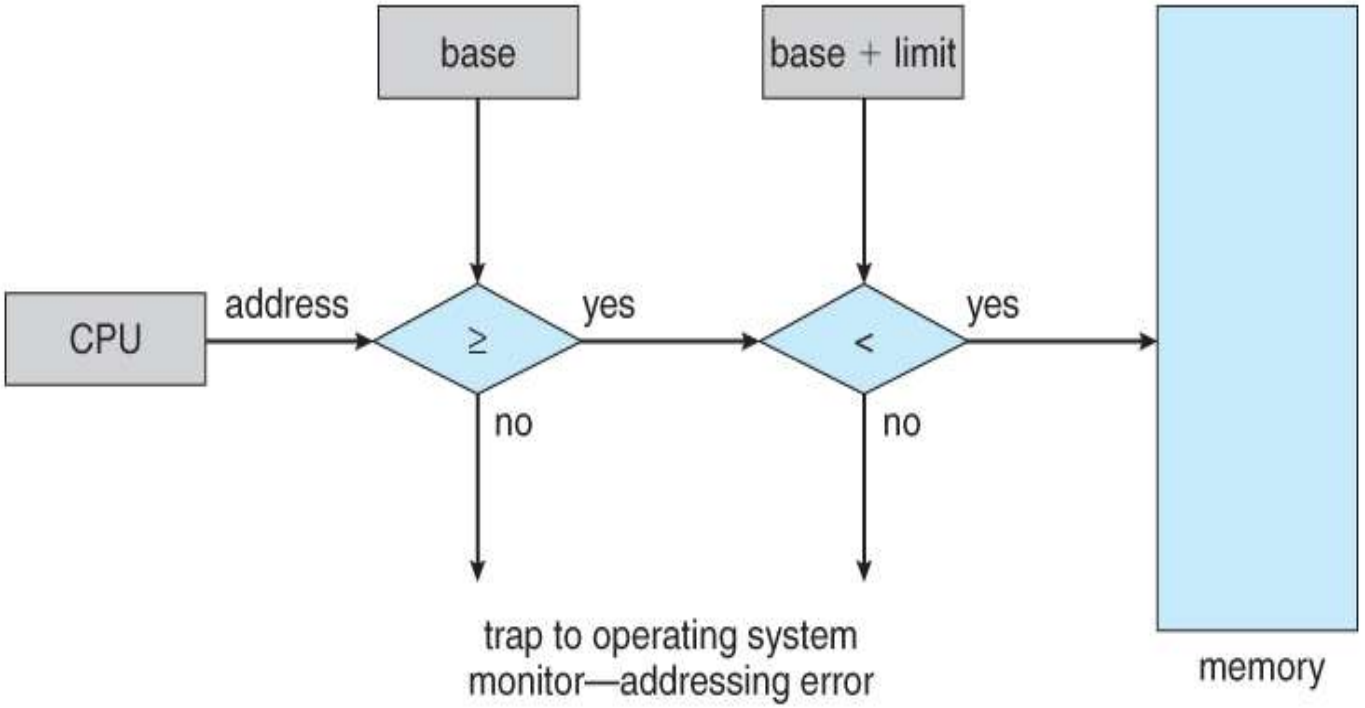


Address Translation

- The memory hardware doesn't know what a particular part of memory is being used for
- User processes must be restricted so that they only access memory locations that "belong" to that particular process.
- This is usually implemented using a **base register** and a **limit register** for **each process.**
- ***Every*** memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated.
- The OS has access to all existing memory locations, as this is necessary to swap users' code and data in and out of memory. It should also be obvious that changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.



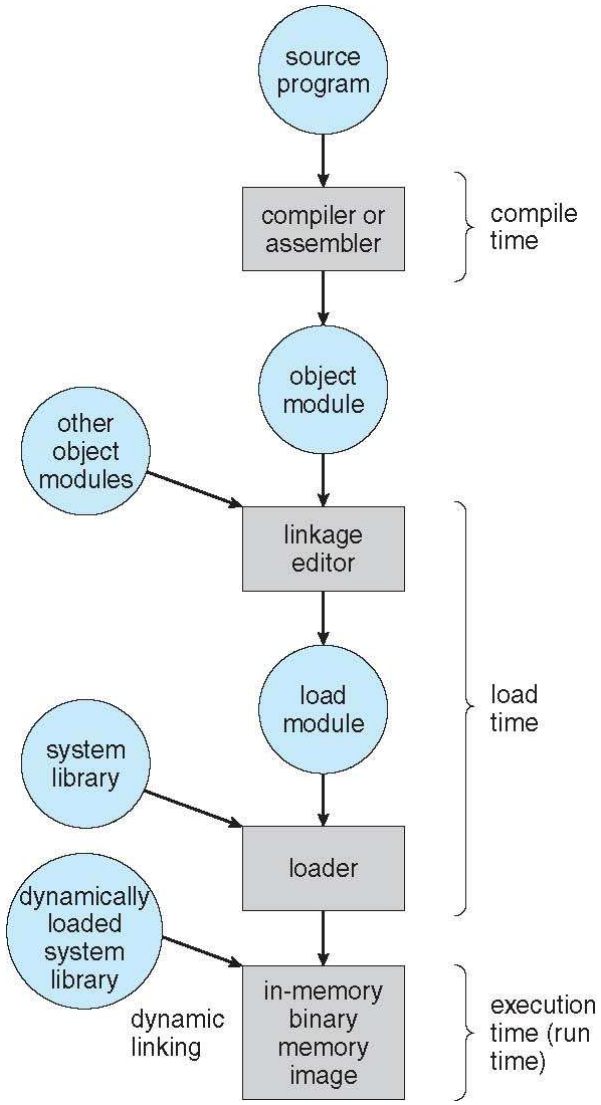
Address Translation Hardware



Address Binding

- **Compile Time** - If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled. (DOS)
- **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
- **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time.

Multistep Processing of a User Program



Dynamic Loading

- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

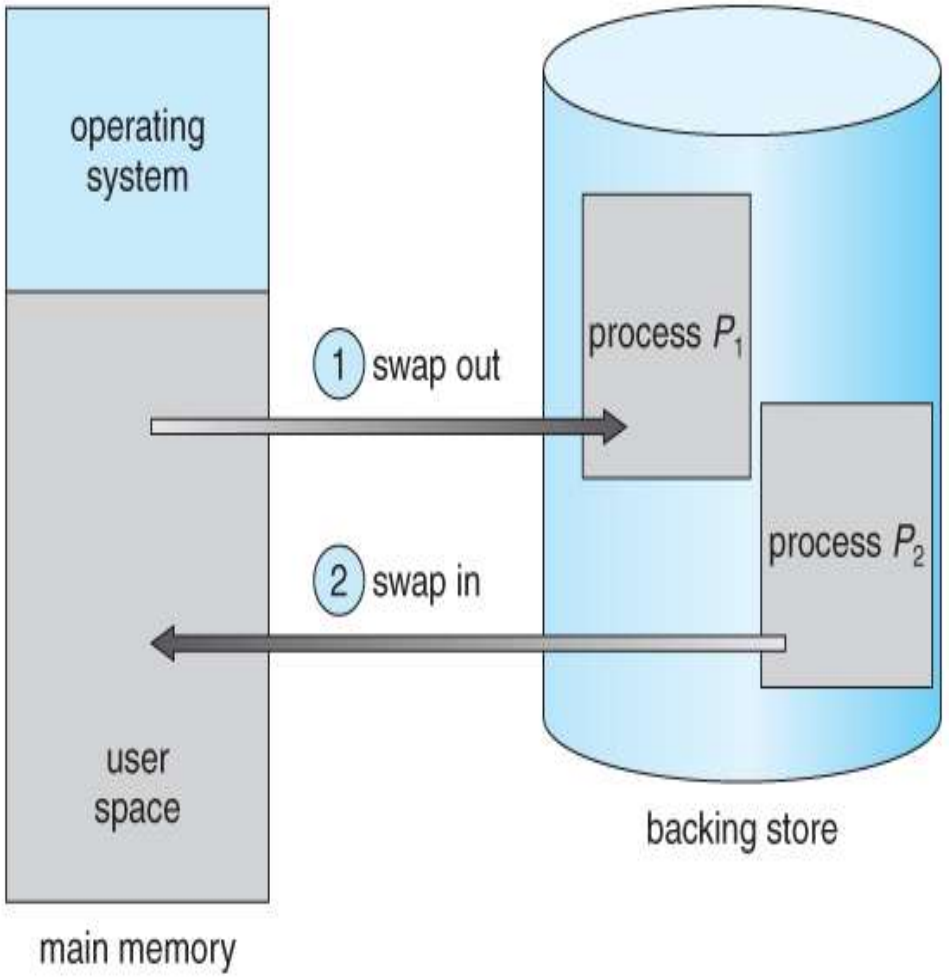
Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** – linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

Swapping

- A process must be loaded into memory in order to execute.

- If there is not enough memory available to keep all running processes at the same time, some processes who are not currently using the CPU may have their memory swapped out local disk called ***backing store.***
- If compile-time/ load-time address binding is used, then processes must be swapped back into the same memory location from which they were swapped out. If runtime binding is used, then processes can be swapped back into any available location.

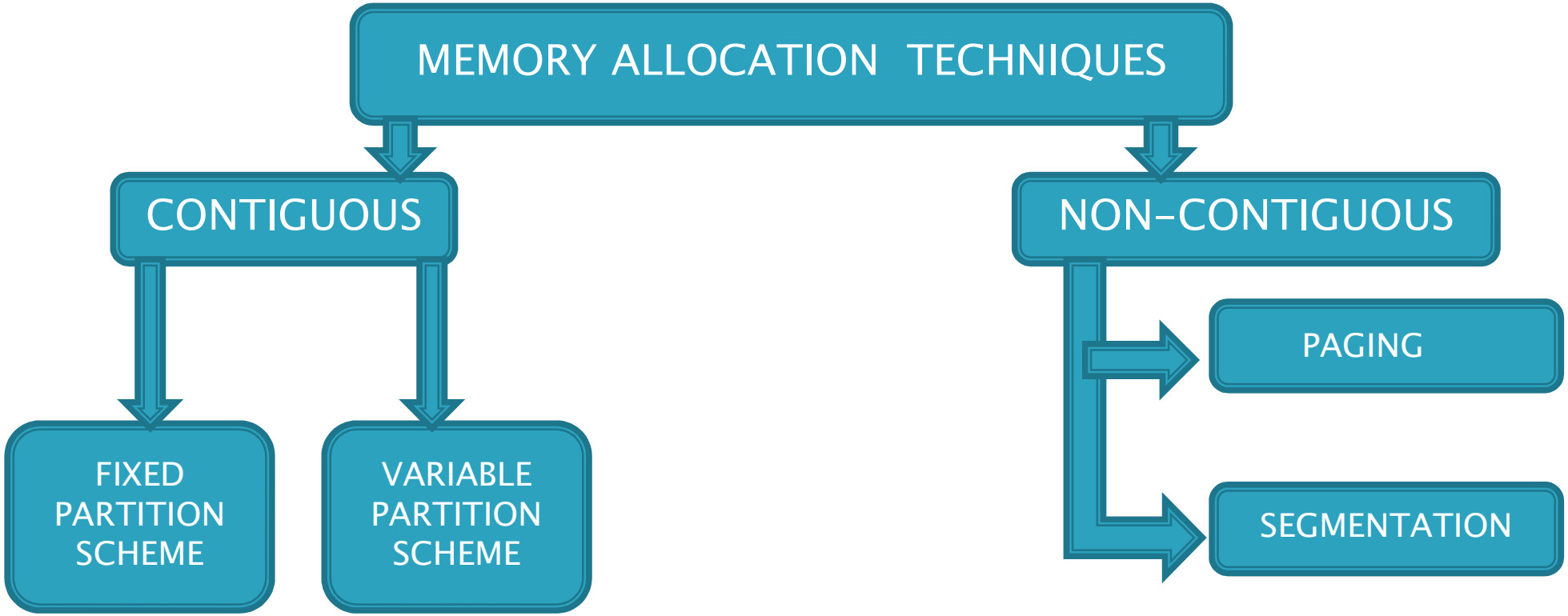


Memory Management functionalities

- The problem the OS has is how best to manage the memory resource, that is how and when to divide the memory into blocks and how and when to allocate the blocks to processes.

- Challenges:
  - Memory Allocation
  - Memory Protection
  - Garbage Collection
  - IO Support
  - Swapping, fragmentation and compaction

Memory Allocation Techniques



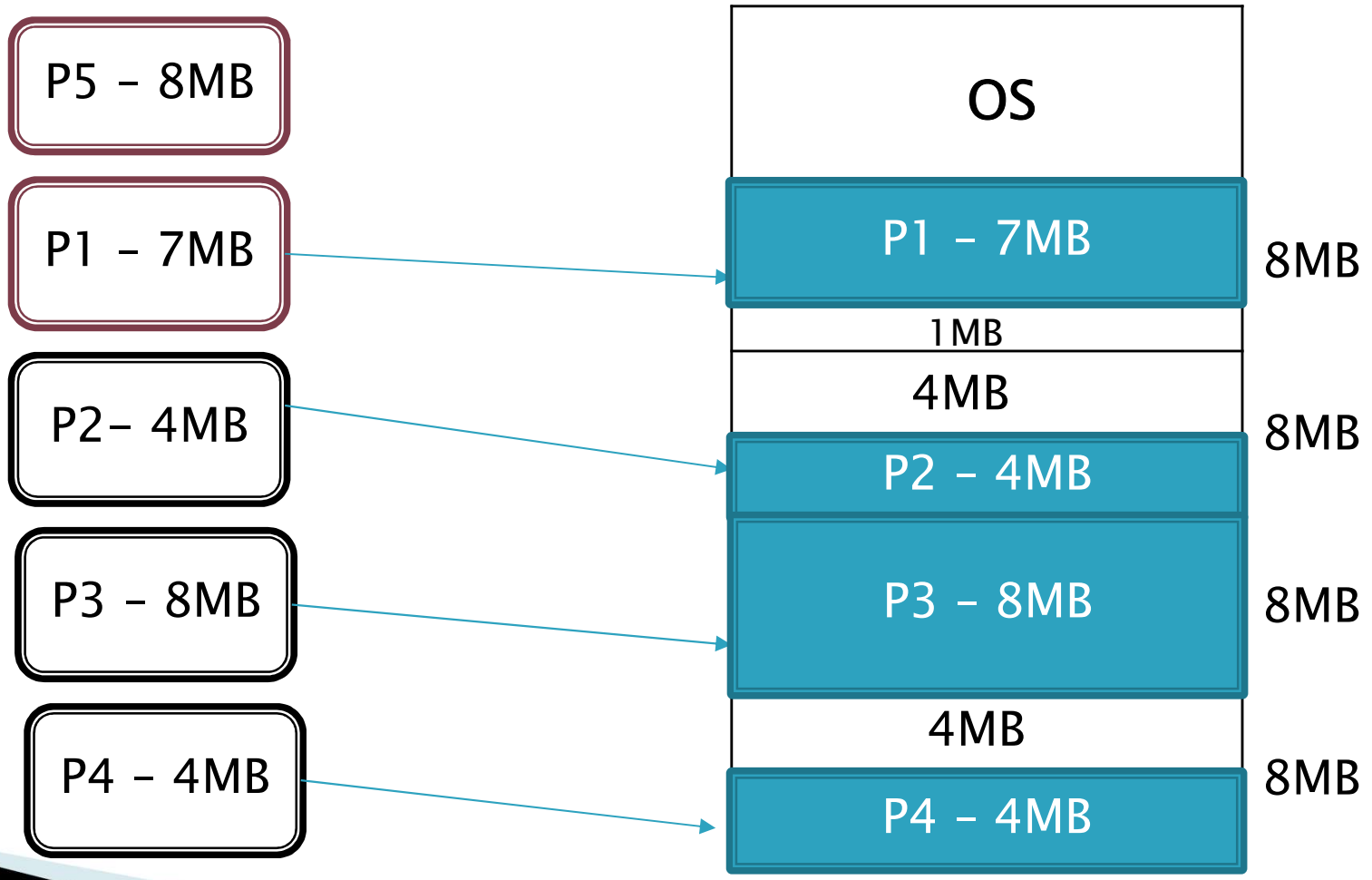
Fixed (static) Partitioning Scheme

- Number of partitions are fixed.
- Partitions are made before the execution or during system configure.
- Size of each partition may or may not be same.
- Hence, we have two types to fixed partitioning:
  - Equal size partitions
  - Unequal size partitions
- Contiguous allocation of memory is done and hence spanning is not allowed.

Fixed Partitioning – Equal size

External Fragmentation

Internal Fragmentation



Fixed Partitioning – Un-equal size

- Better memory utilization than equal size partition.
- Processes of different sizes can be accommodated.
- Here also we see that 16Mb blocks have 8MB unused space causing Internal Fragmentation
- We have 20 MB space available if we add all un-used spaces.
- A new process P6 with size 16MB cannot be allocated space causing External Fragmentation.

Fixed (static) Partitioning Scheme

- **Equal size partitions** - It divides the main memory into equal number of fixed sized partitions.
  - Memory use is inefficient, i.e., block of data loaded into memory may be smaller than the partition causing **internal fragmentation**.
  - Any process whose size is bigger than partition size cannot be loaded.

- **Un-equal Size Partitions** - It divides the main memory into un-equal sized partitions.
  - Efficient memory usage than equal size partitioning.
  - Still suffers from internal fragmentation.

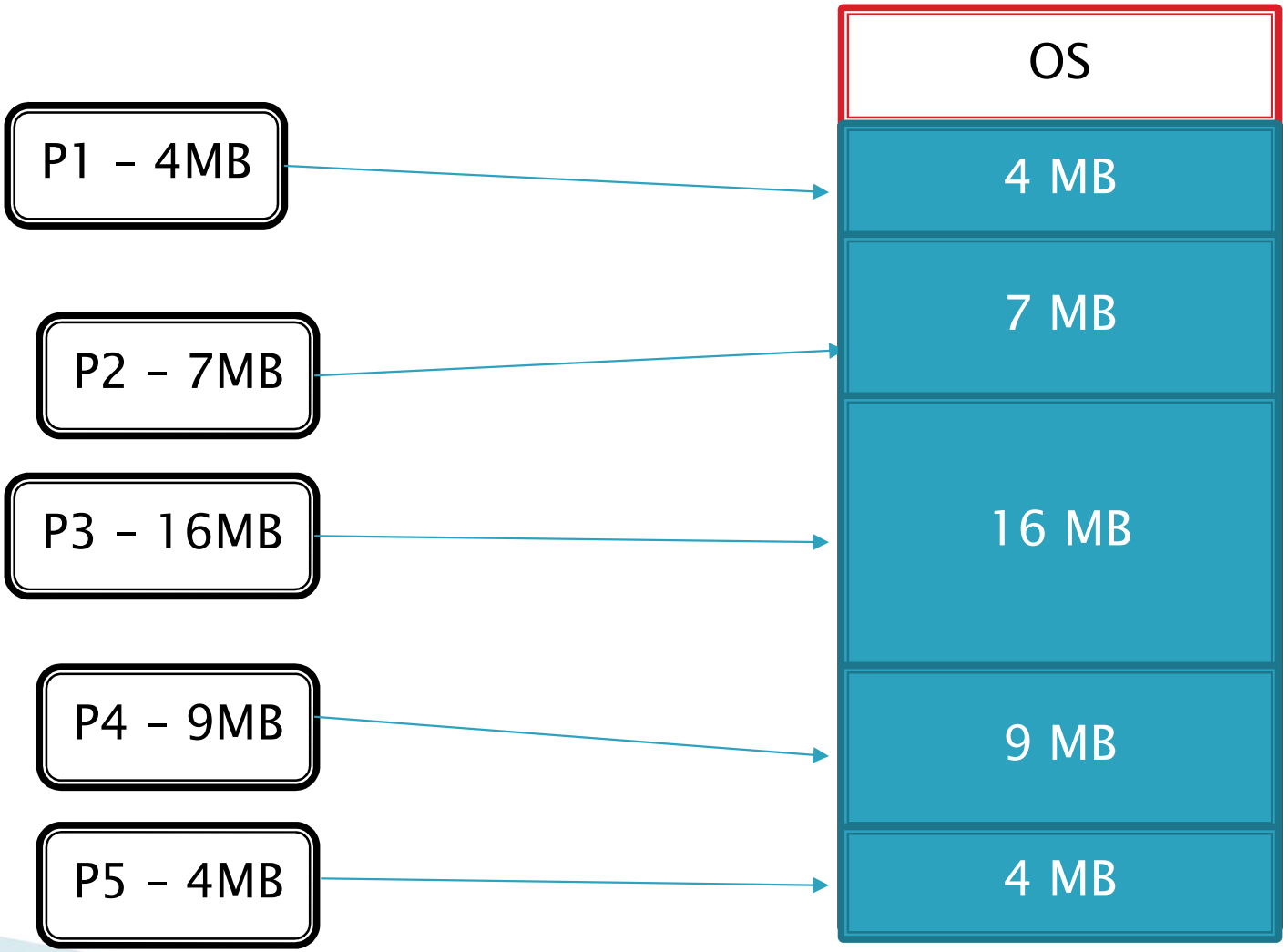
Disadvantages of Fixed Partitioning scheme

- Internal fragmentation - Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.
- Limit in process size as the memory blocks are of fixed sizes.
- Limitation on degree of multi-programming – number of partitions are fixed and hence there is hard limit to number processes that can be loaded into the memory.
- External Fragmentation - Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

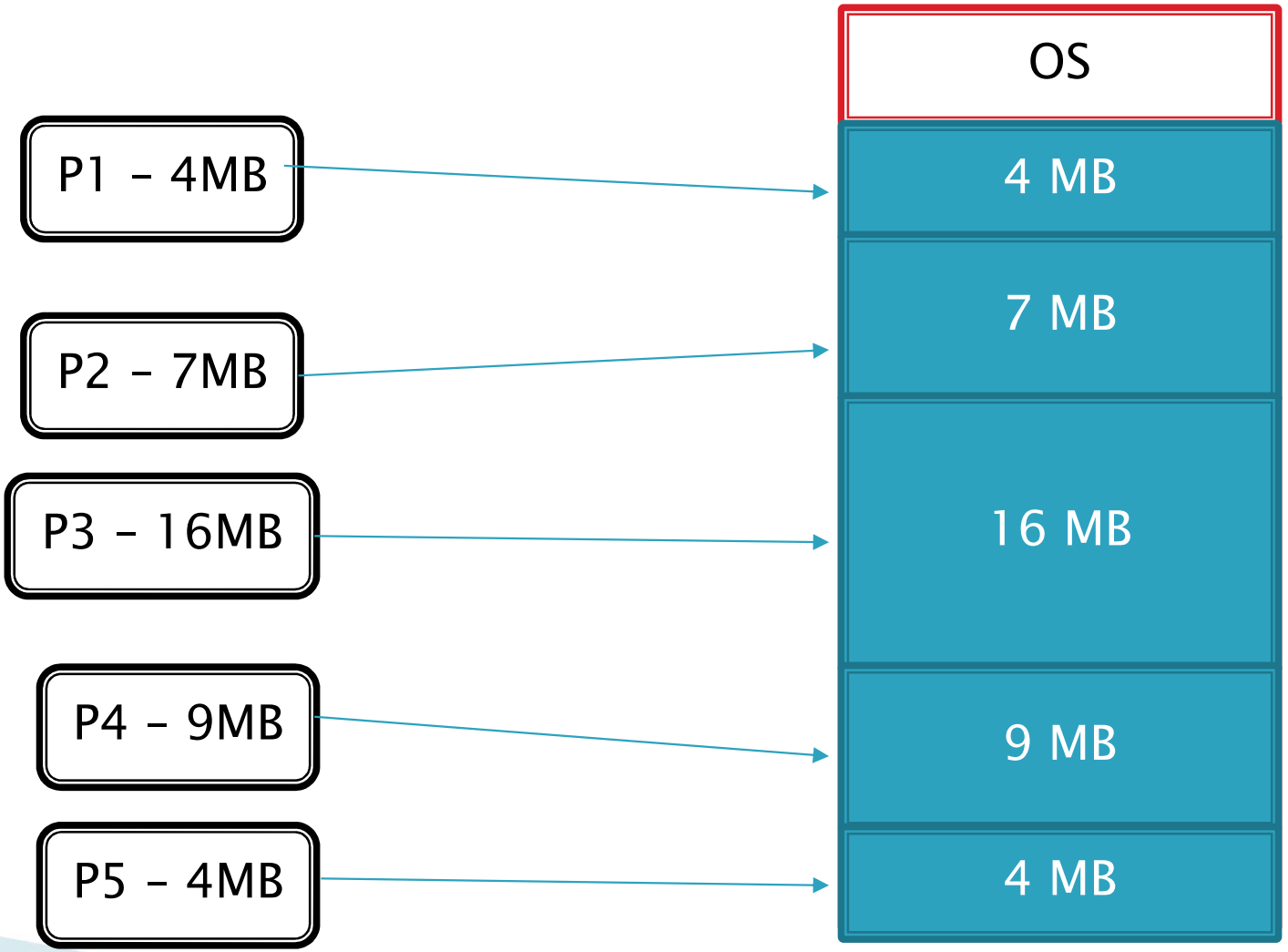
Variable (dynamic) Partitioning Scheme

- Partitions are not made before the execution or during system configure.
- The size of partition will be equal to incoming process.
- The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilization of RAM.
- Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.

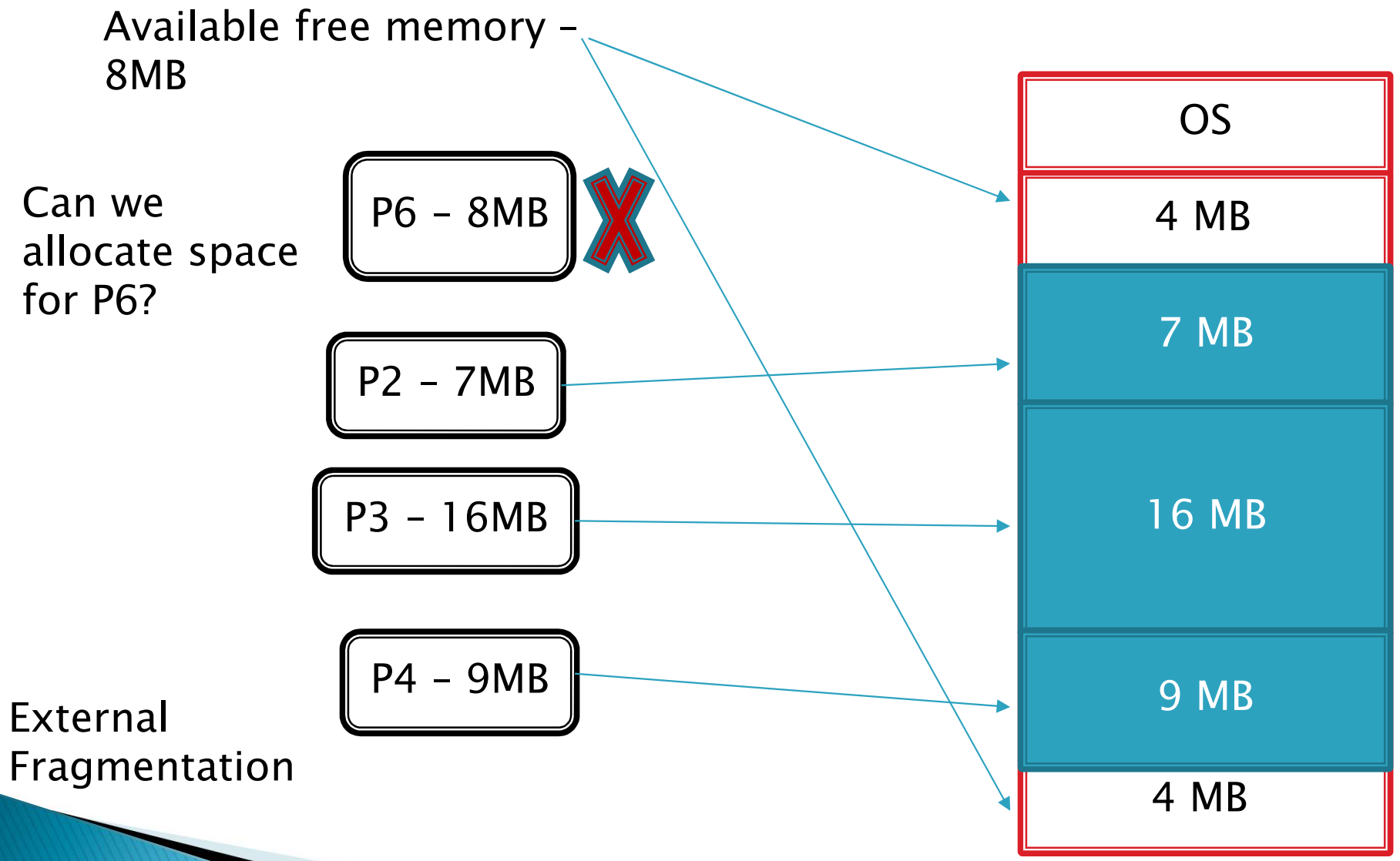
Variable Partitioning Scheme



Variable Partitioning Scheme



Variable Partitioning Scheme

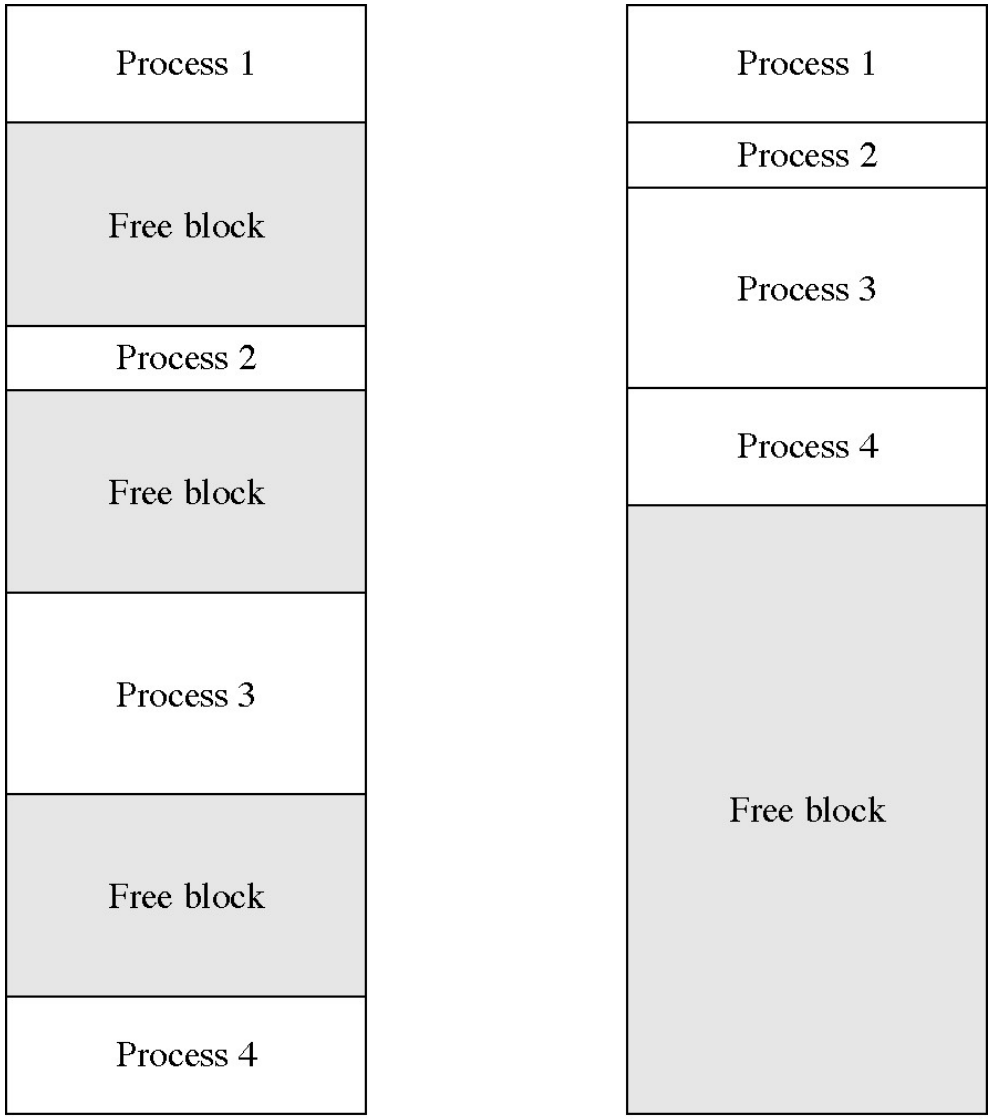


Variable Partitioning Scheme

- Advantages
  - No Internal Fragmentation
  - No restriction on Degree of Multiprogramming
  - No Limitation on the size of the process

- Disadvantages
  - Difficult Implementation: it involves allocation of memory during run-time rather than during system configure.
  - External Fragmentation: There will be external fragmentation inspite of absence of internal fragmentation.

Compacting memory



Before compaction

After compaction

Memory allocation strategies

How to satisfy a request of size *n* from a list of free holes

- First fit
- Best fit
- Worst fit

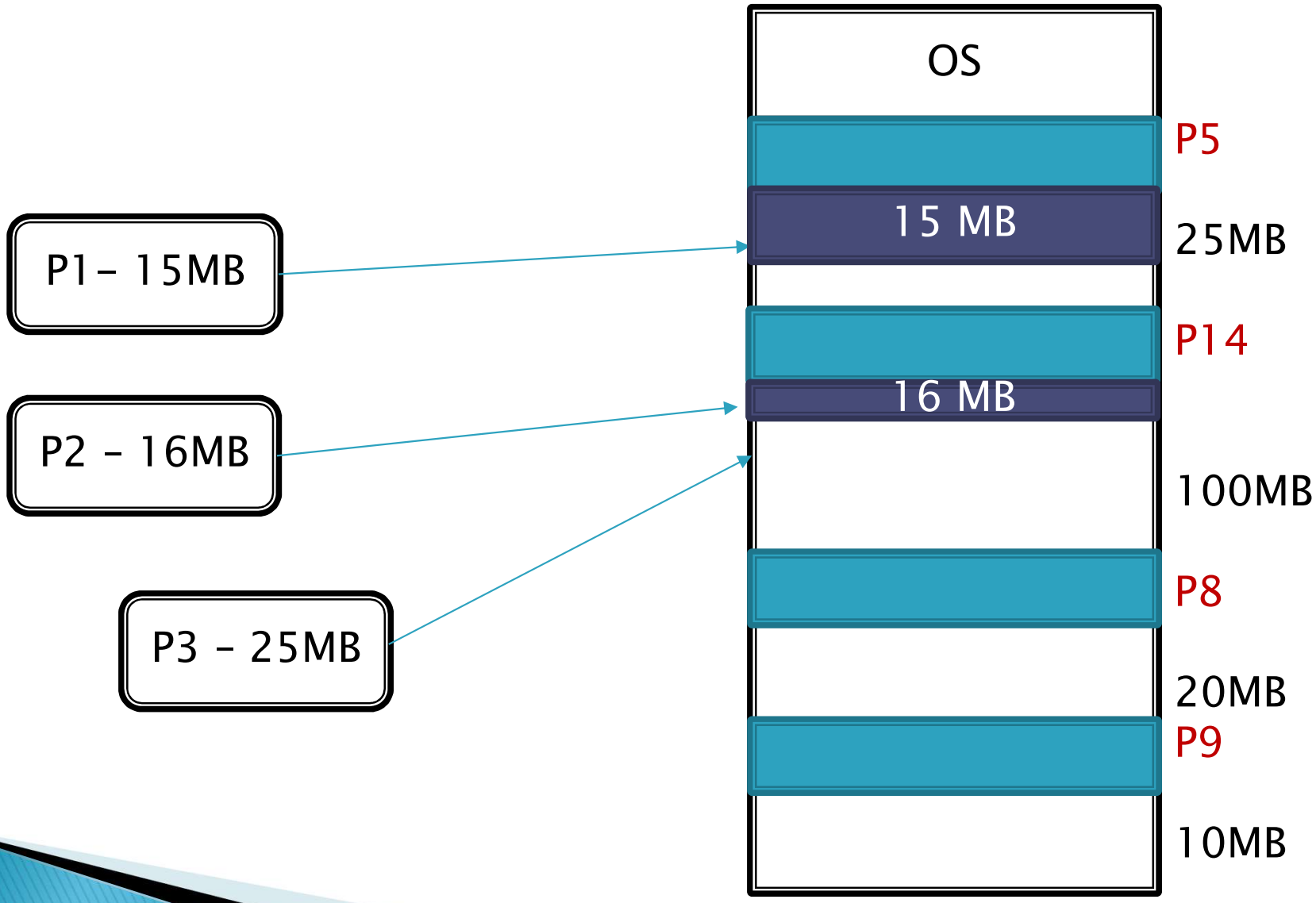
First Fit

First Fit Memory Allocation

- In this method, first job claims the first available memory with space more than or equal to it's size.
- The operating system doesn't search for appropriate partition but just allocate the job to the first memory partition available with sufficient size.

- Advantages
  - It is fast in processing.
- Disadvantages
  - Wastage of memory - smaller processes may be allocated larger spaces resulting in in-efficiency

First Fit


OS
P5
15 MB
25MB
P1- 15MB
P14
16 MB
P2 - 16MB
100MB
P8
P3 - 25MB
20MB
P9
10MB

Best Fit

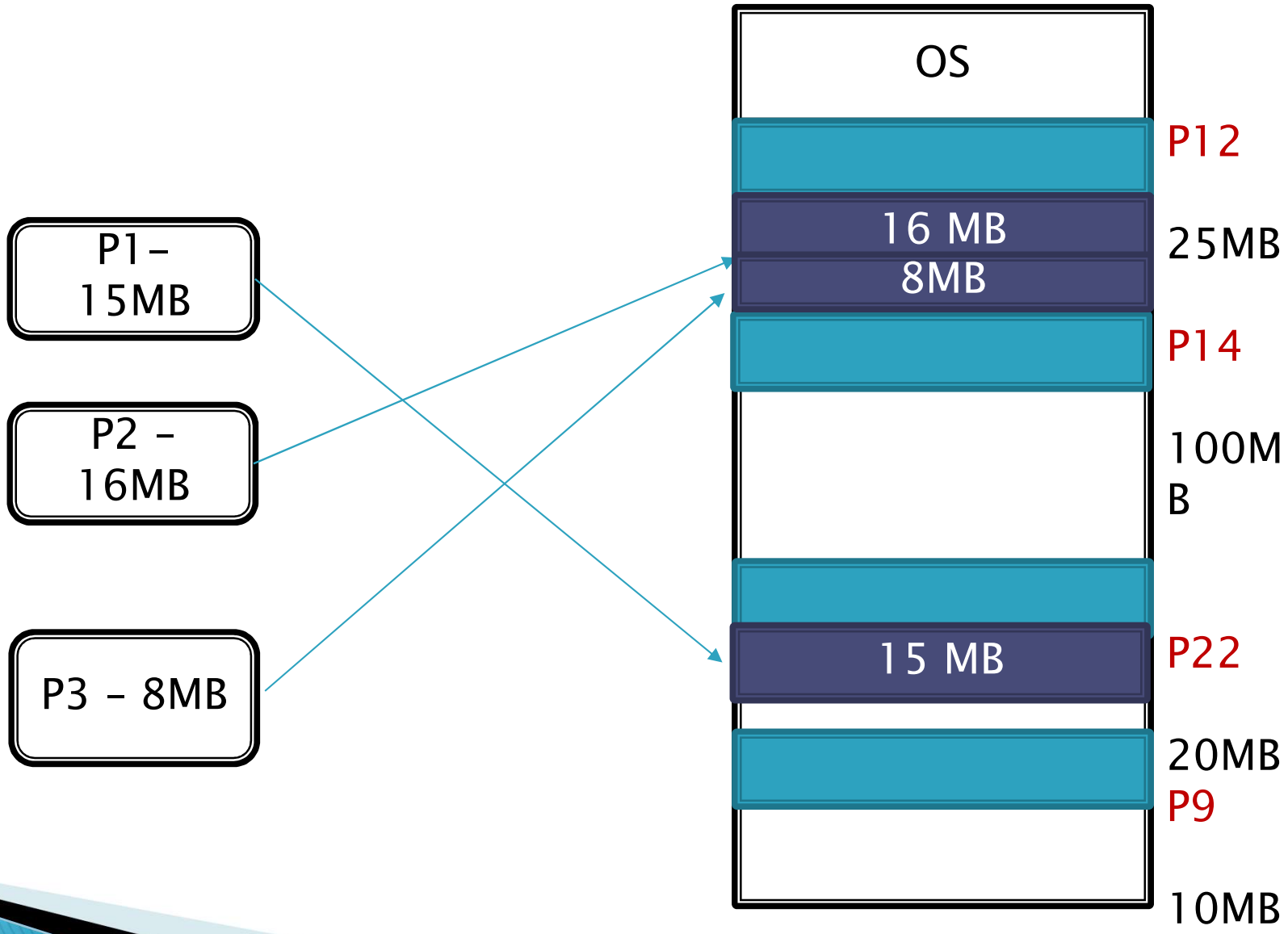
Best Fit Memory Allocation

- The operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory.

- Advantages
  - Memory efficient - allocates the process to the minimum possible space available in the memory
  - Minimal internal fragmentation

- Disadvantages
  - Slow Process - Checking the whole memory for each job makes the working of the operating system very slow.

Best Fit


P1 – 15MB
P2 – 16MB
P3 – 8MB
OS
16 MB
8MB
15 MB
P12
25MB
P14
100MB
P22
20MB
P9
10MB

Worst Fit

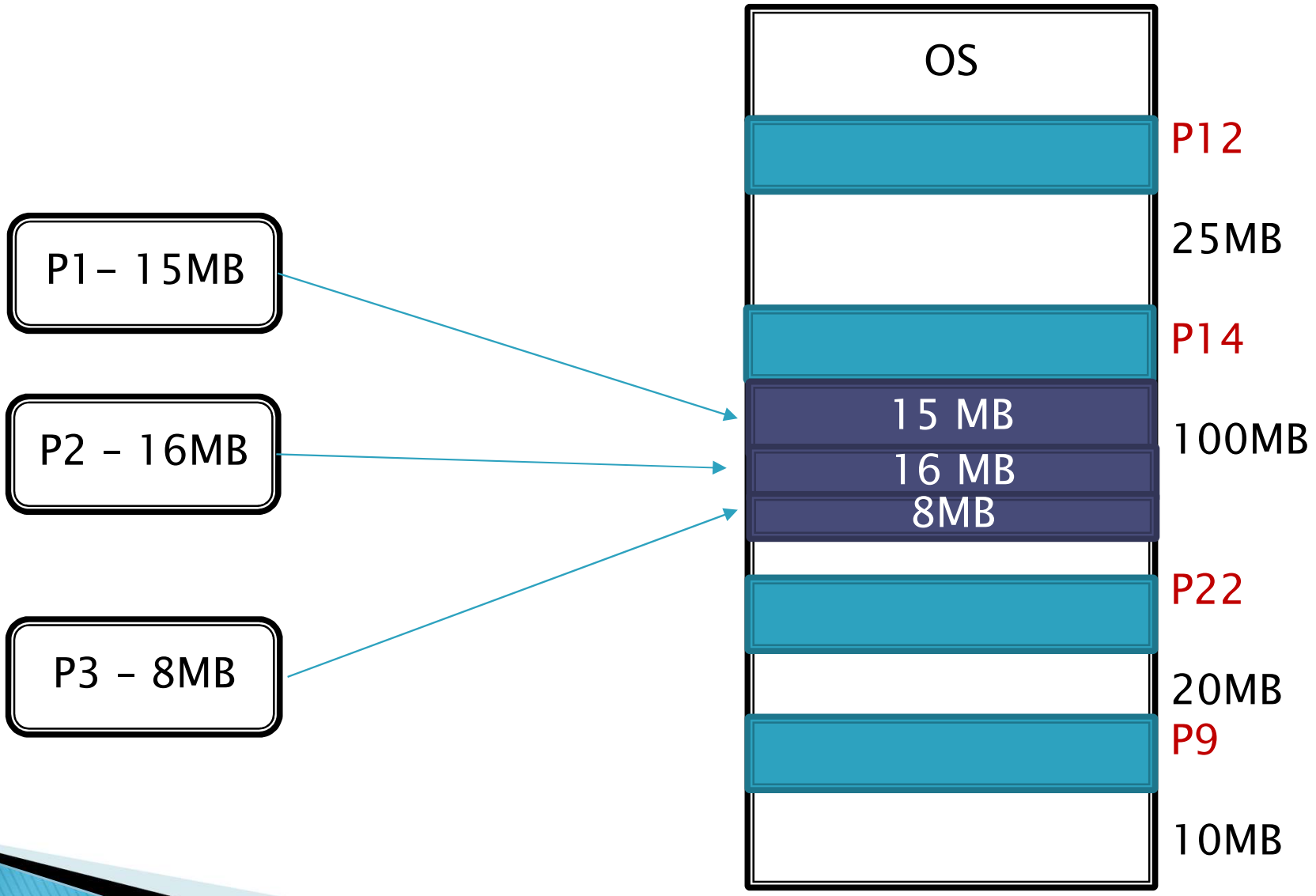
Worst Fit Memory Allocation

- The process traverses the whole memory and always search for the largest hole/partition, and then the process is placed in that hole/partition.

- Advantages
  - Since this process chooses the largest hole/partition, therefore there will be large internal fragmentation. This internal fragmentation is big enough for other small processes to be allocated memory.

- Disadvantages
  - Slow Process - Checking the whole memory for each job makes the working of the operating system very slow.

Worst Fit

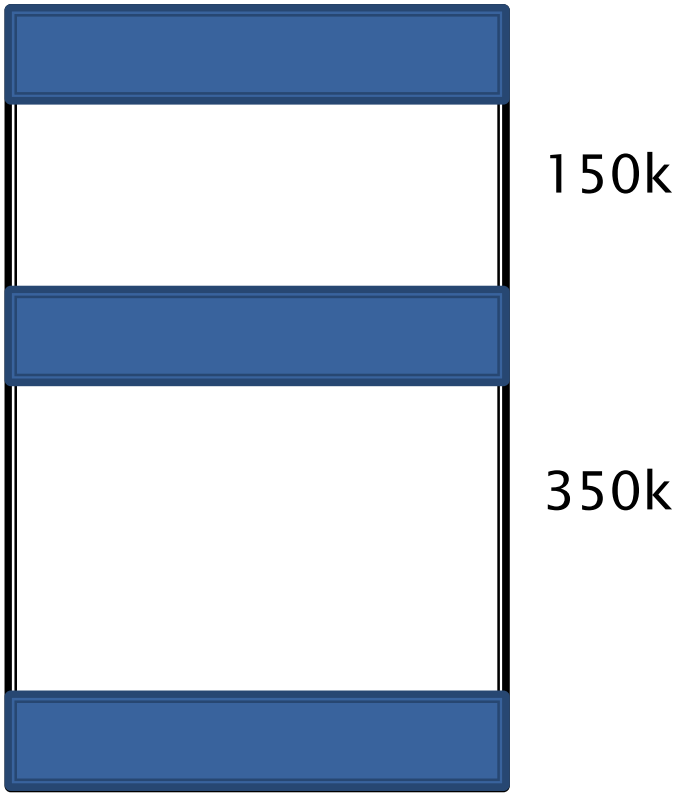

OS
P12
25MB
P14
15 MB
16 MB
8MB
100MB
P22
20MB
P9
10MB
P1- 15MB
P2 - 16MB
P3 - 8MB

Question?

Requests from processes are 300k, 25k, 125k, 50k respectively.

The requests can be satisfied with

A) Best fit but not first fit
B) First fit but not best fit
C) First fit and Worst fit
D) Both
E) None

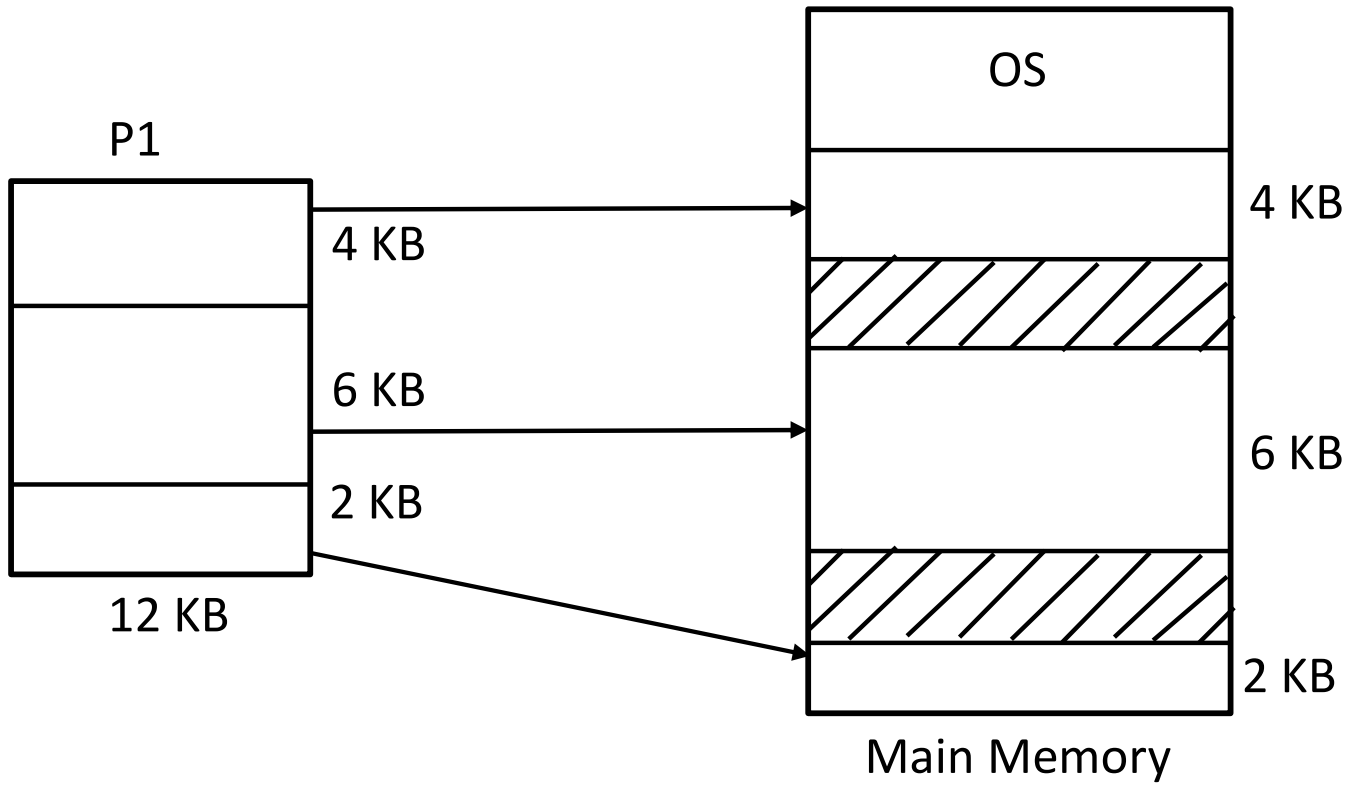


Topics

- Non contiguous Memory Allocation
  - Segmentation
    - About segmentation
    - HW required
    - Segment table
  - Paging
    - Paging and page table
    - TLB
    - Dirty bit
    - Shared pages and reentrant code
  - Shared pages and reentrant code

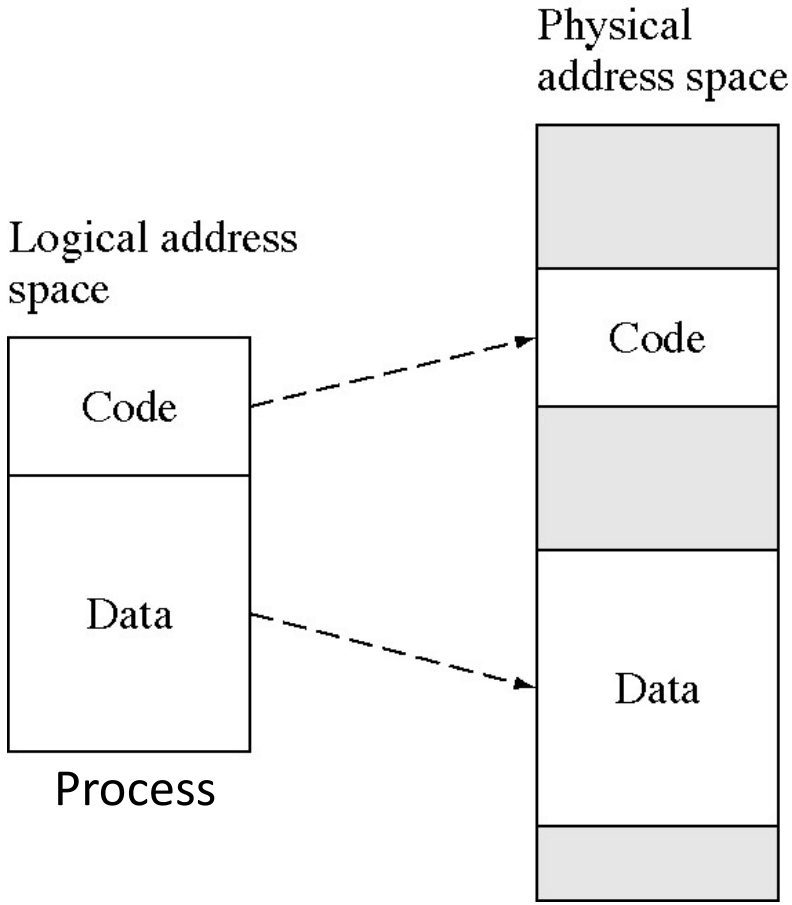
Non contiguous memory allocation

- Without memory mapping, programs require physical continuous memory
- Large blocks mean large fragments and wasted memory



Non-contiguous Memory allocation

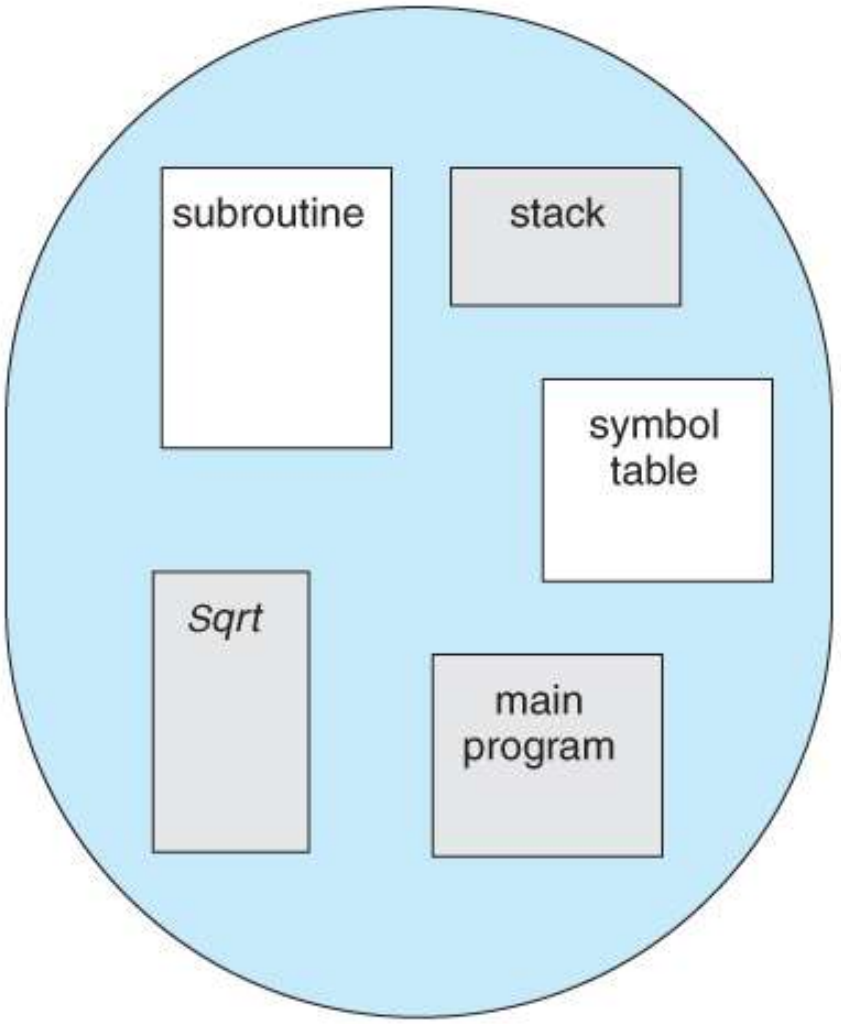
- We need hardware memory mapping to address this problem
  - Segments
  - Pages



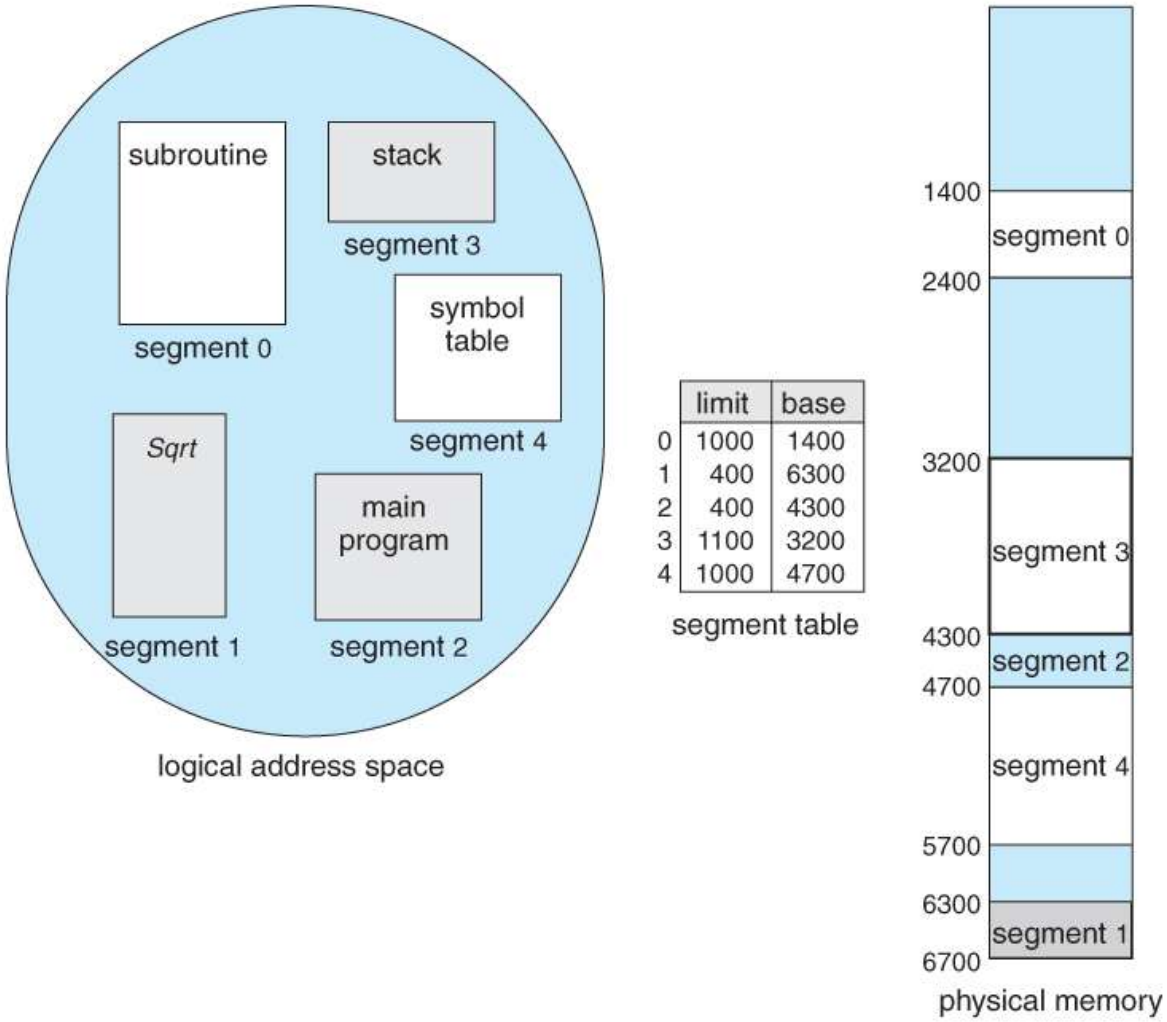
Separate code and data spaces

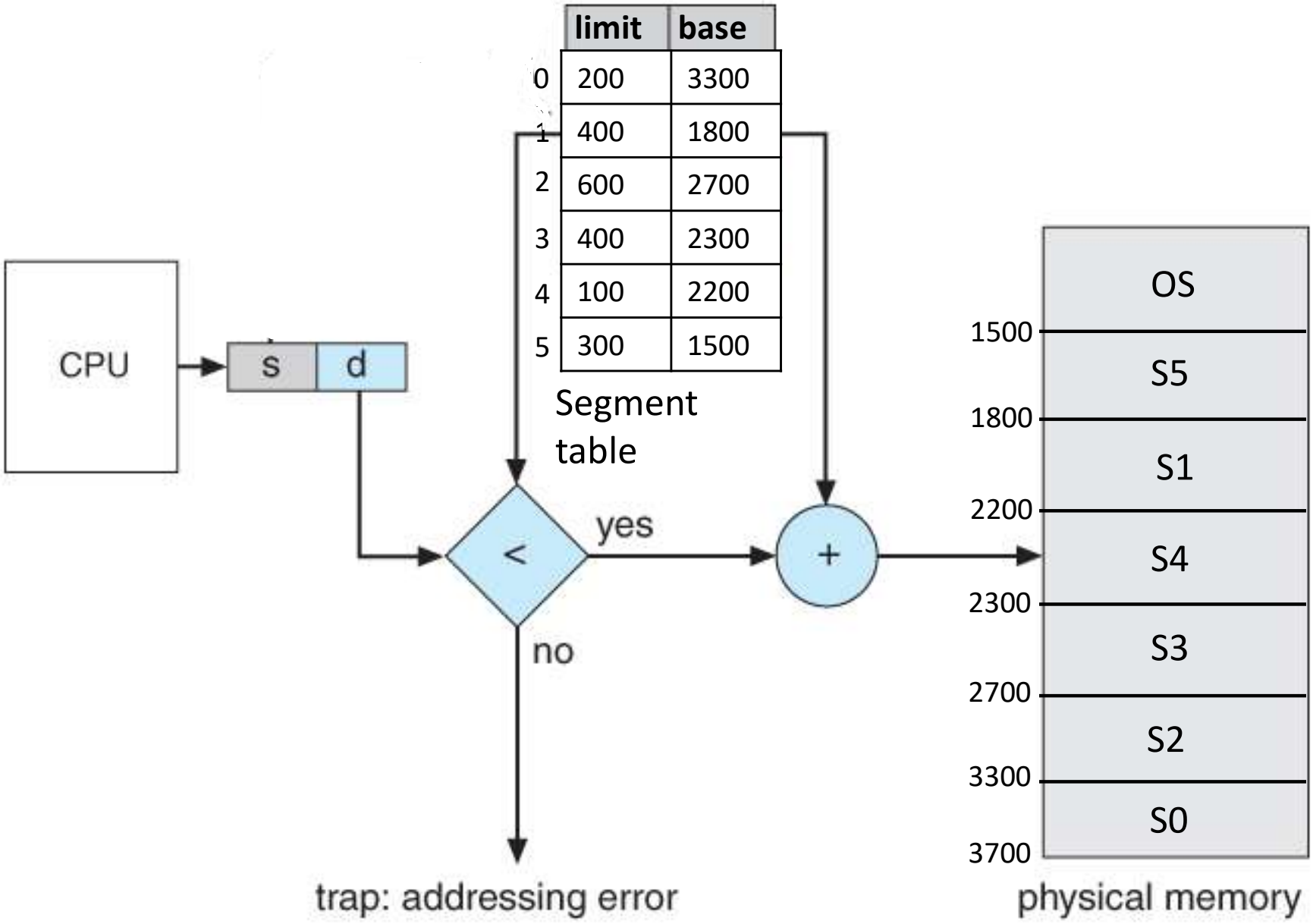
Segmentation

- Most **users** (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a **particular use**, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.



Segment table

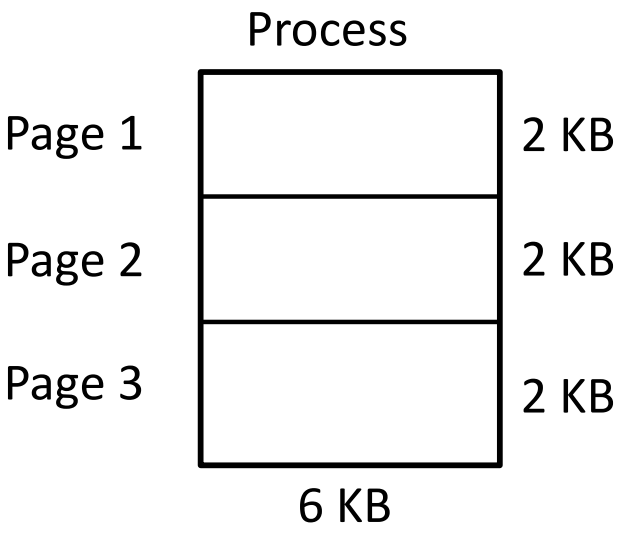


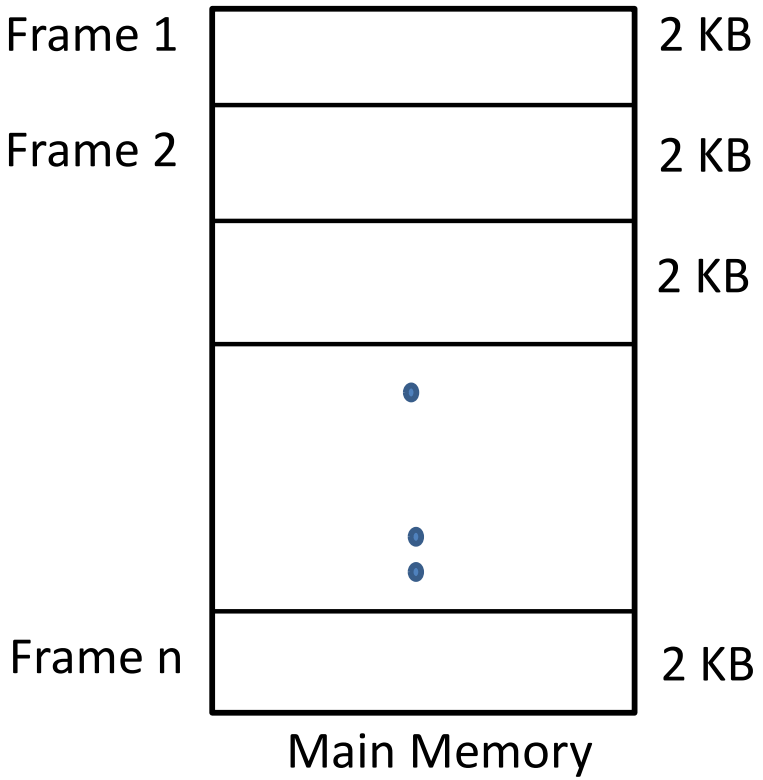

limit
base
0
200
3300
1
400
1800
2
600
2700
3
400
2300
4
100
2200
5
300
1500
Segment table
CPU
s
d
<
yes
no
+
trap: addressing error
OS
1500
S5
1800
S1
2200
S4
2300
S3
2700
S2
3300
S0
3700
physical memory

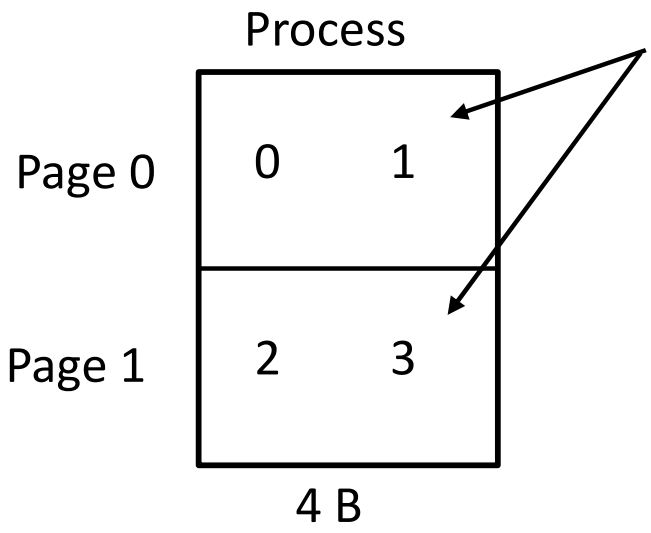
Segments to pages

- Large segments do not help the fragmentation problem
  - so we need small segments
- Small segments are usually full
  - so we don’t need a length/limit register
  - just make them all the same length
- Identical length segments are called *pages*
- We use page tables instead of segment tables
  - base register but no limit register

Paging


Process
Page 1
Page 2
Page 3
2 KB
2 KB
2 KB
6 KB


Frame 1
Frame 2
Frame n
2 KB
2 KB
2 KB
2 KB
Main Memory

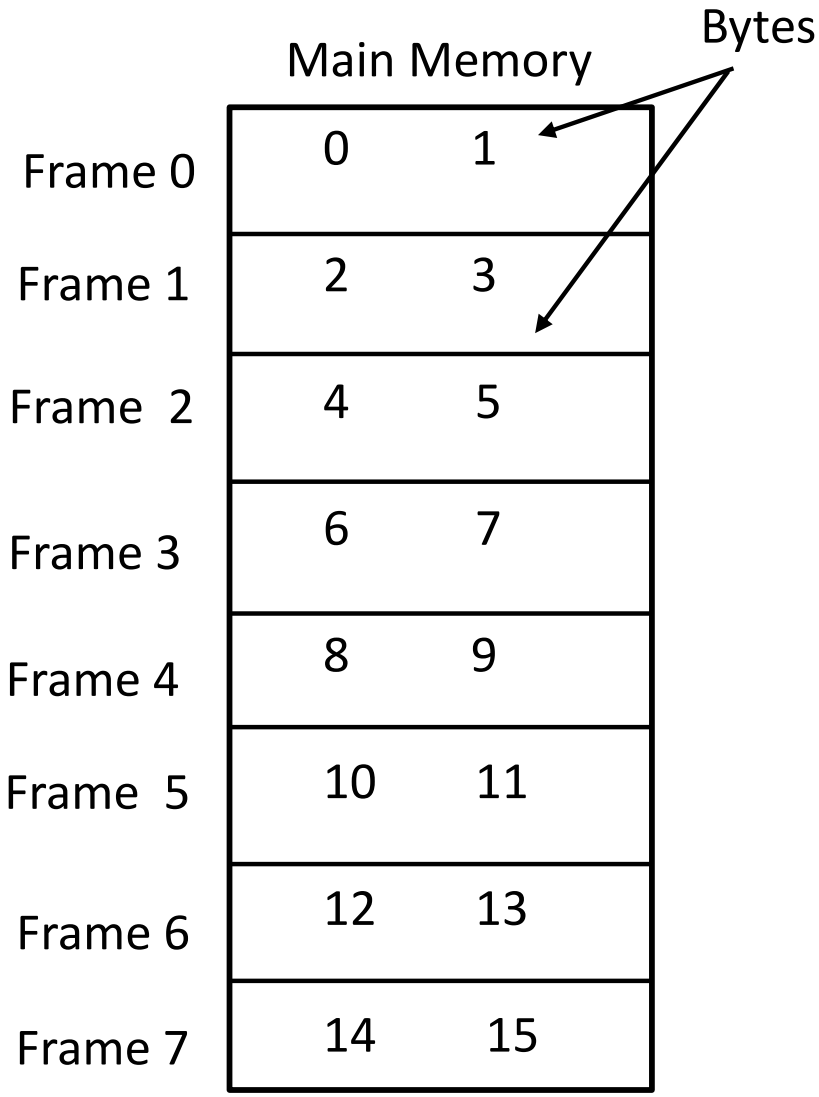


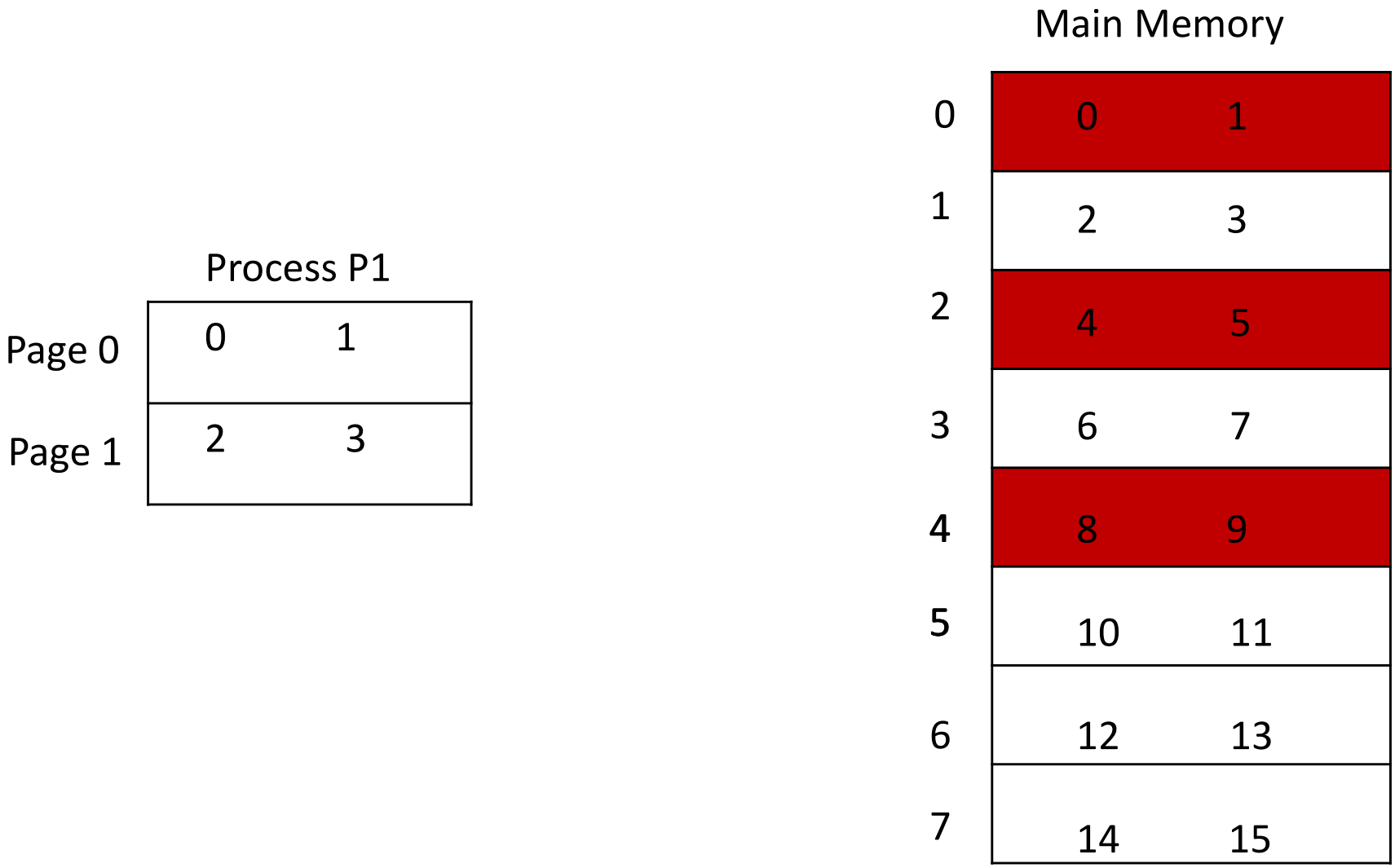
Process size = 4 B
Page size = 2 B

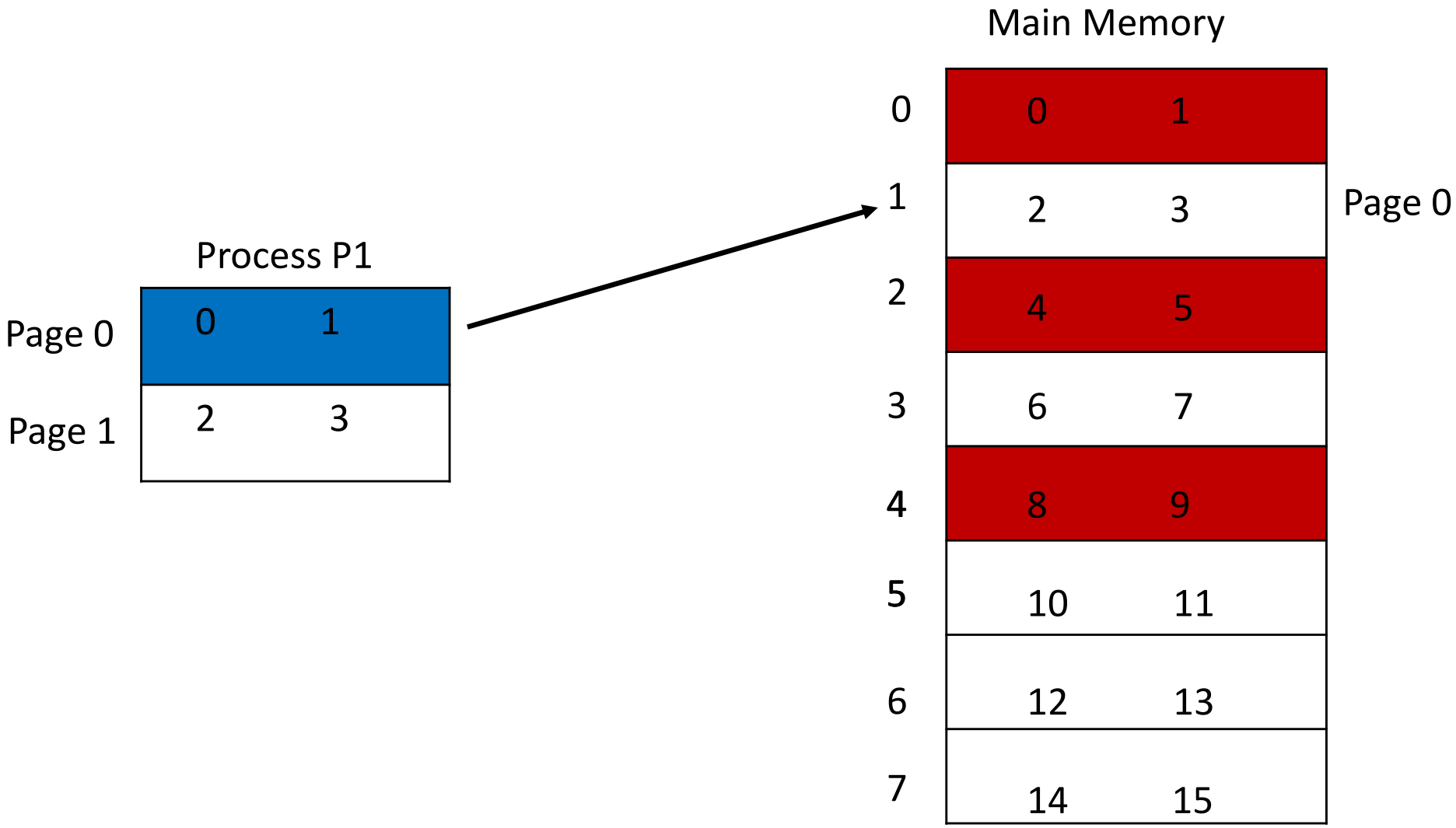
No. of pages = 2

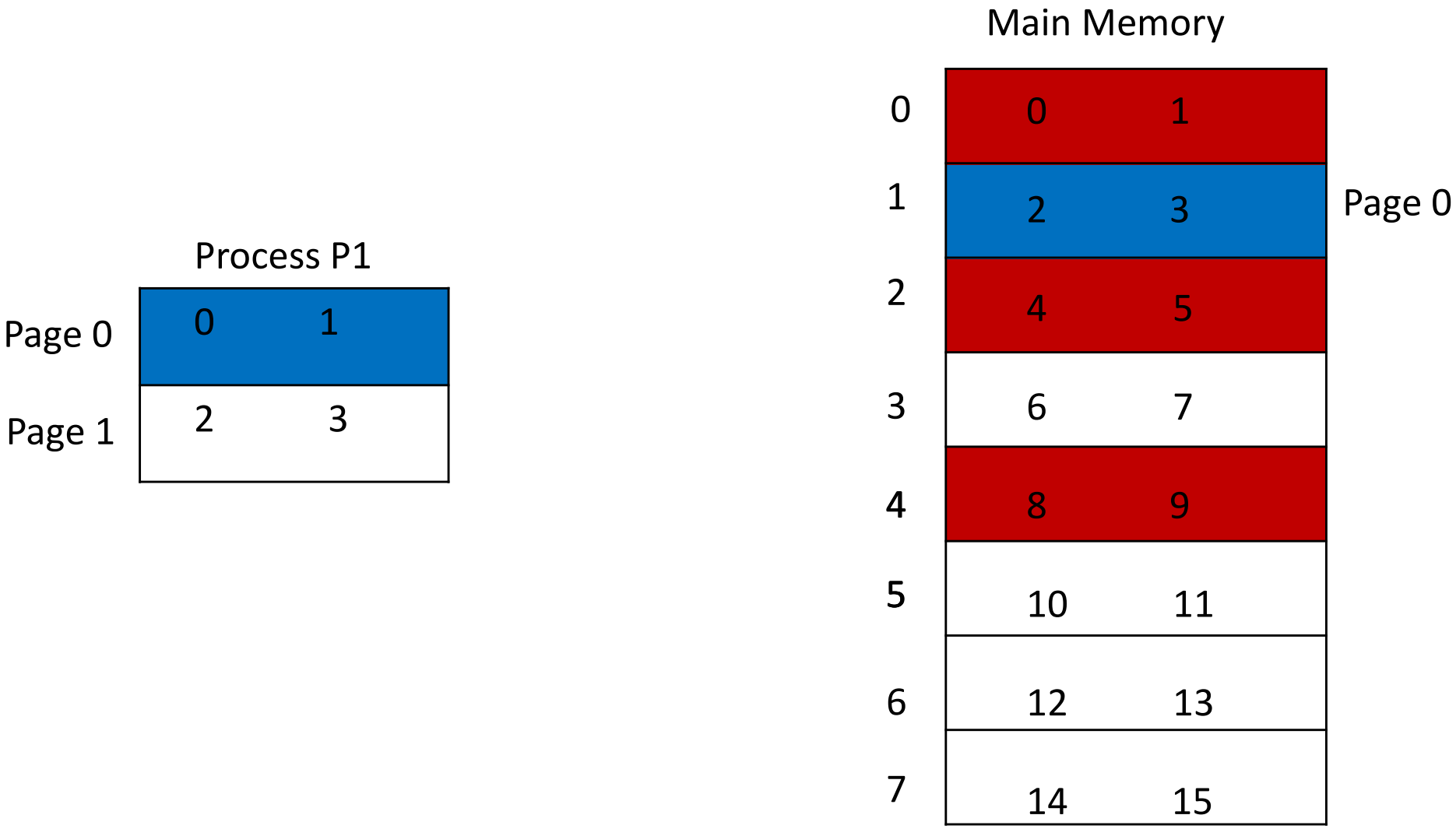
MM size = 16 B
Frame size = 2 B

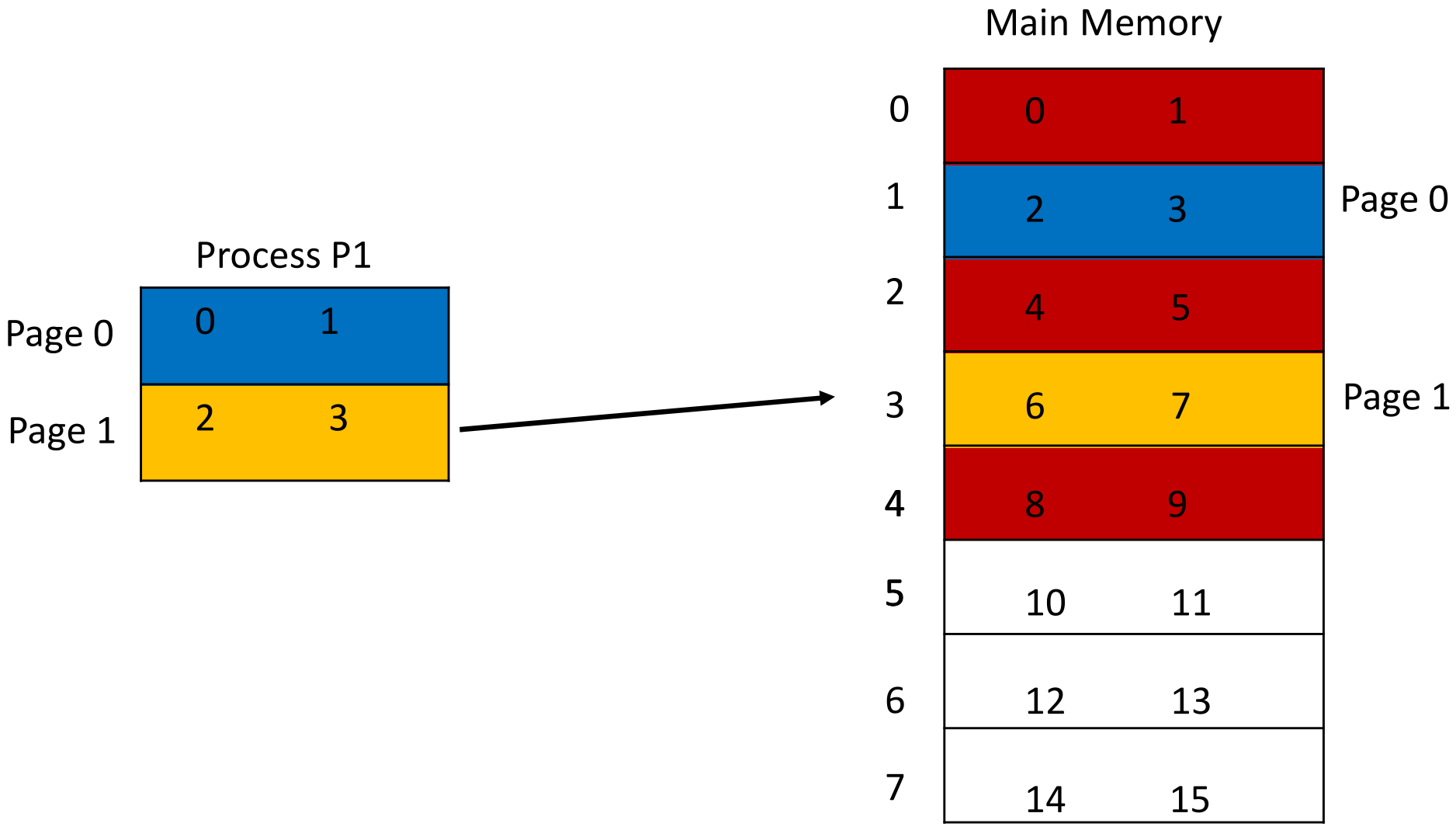
No. of frames = 8

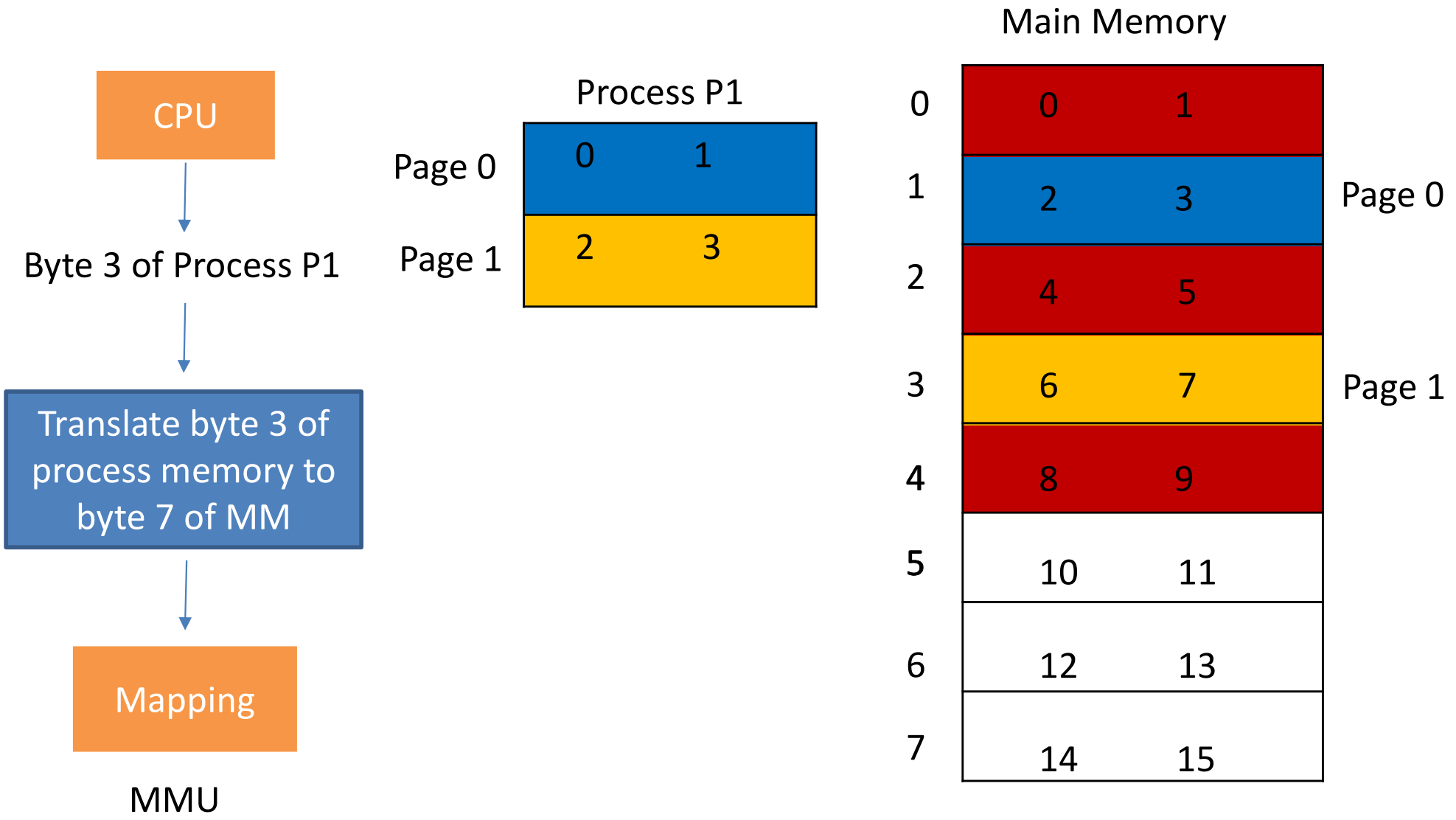


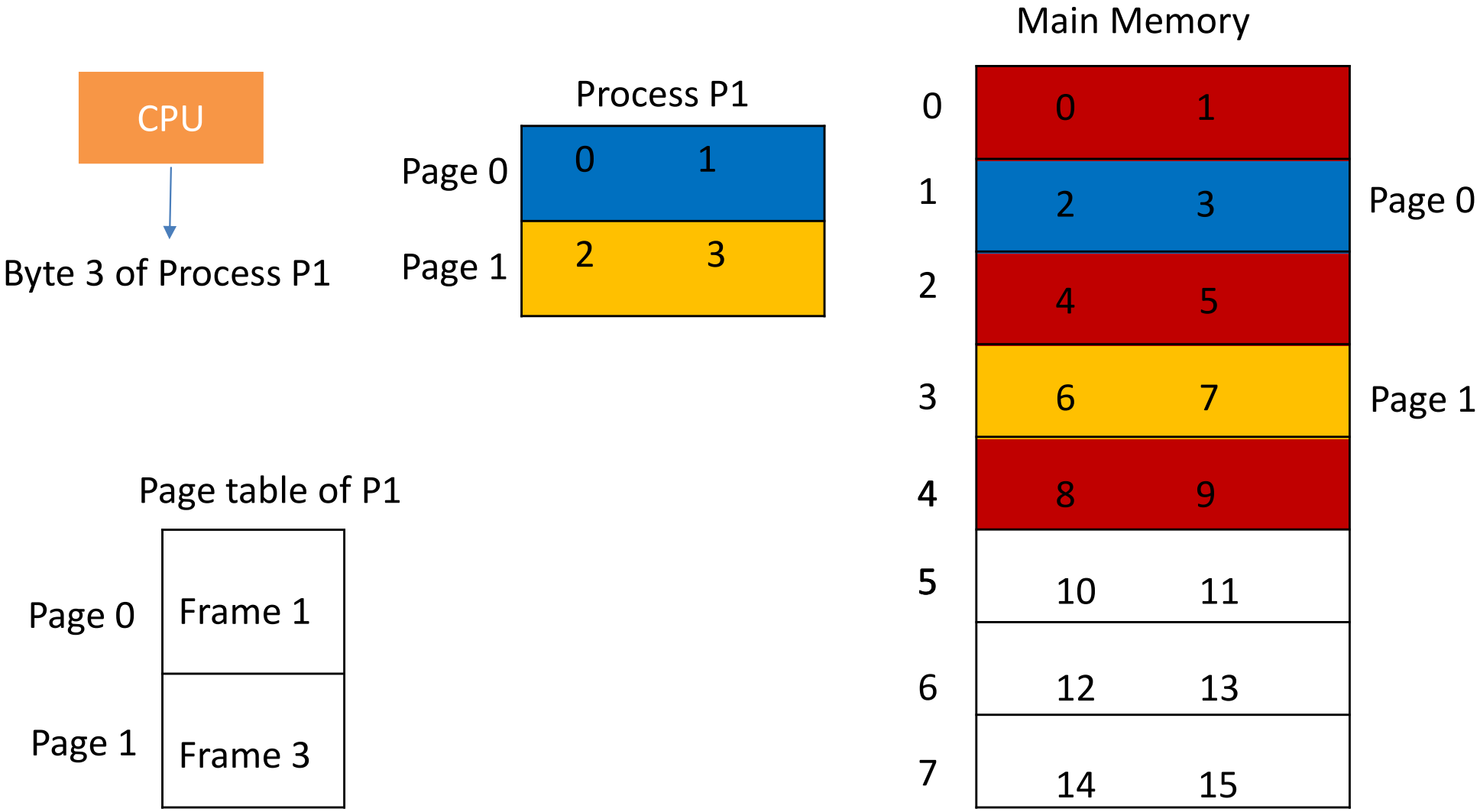

Process P1
Page 0
0
1
Page 1
2
3
Main Memory
0
0
1
1
2
3
2
4
5
3
6
7
4
8
9
5
10
11
6
12
13
7
14
15


Main Memory
Process P1
Page 0
Page 1
0 1
2 3
0 1
2 3
4 5
6 7
8 9
10 11
12 13
14 15
0
1
2
3
4
5
6
7
Page 0

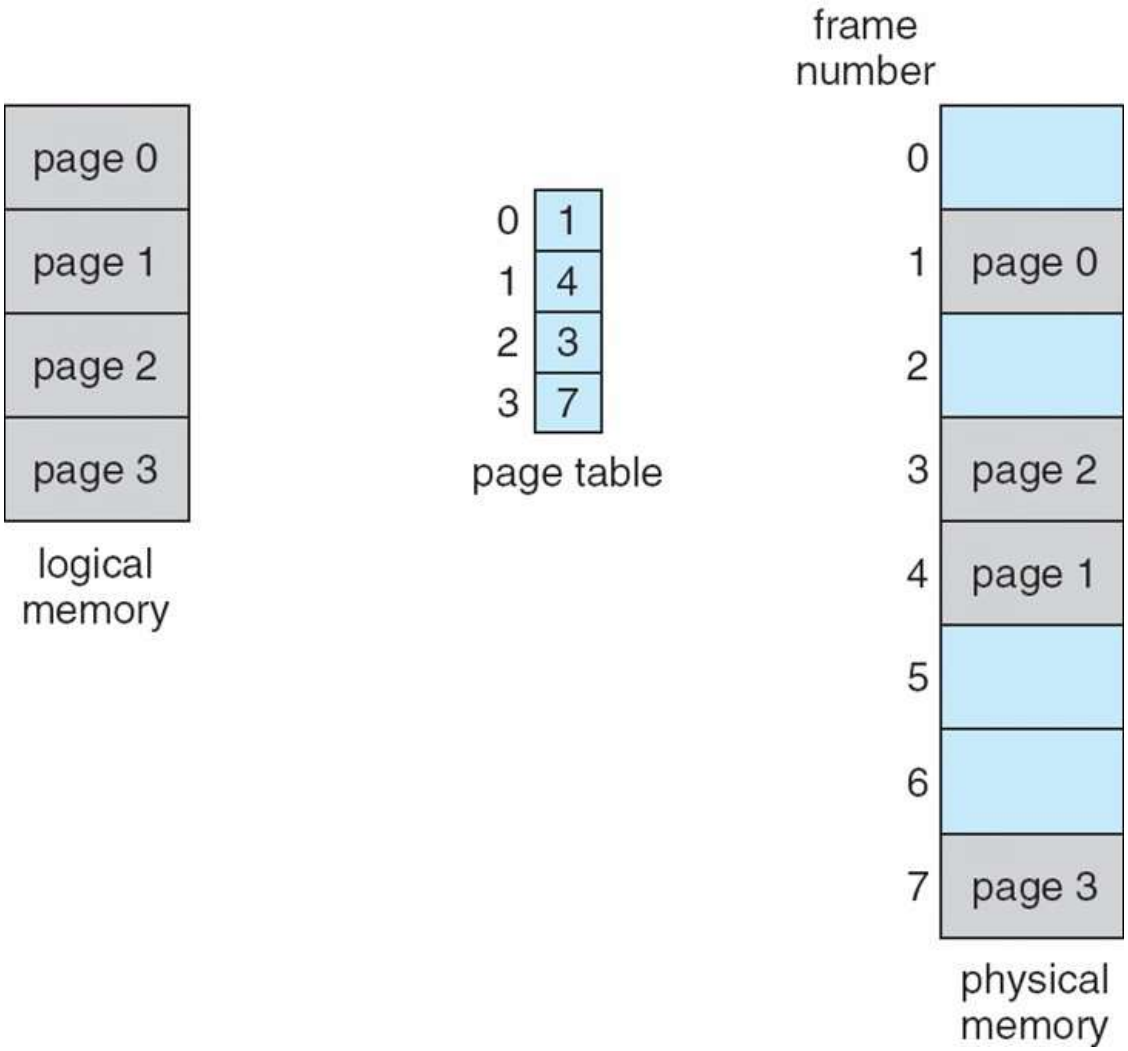

Process P1
Page 0
Page 1
0
1
2
3
Main Memory
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Page 0


Process P1
Page 0
Page 1
0
1
2
3
Main Memory
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Page 0
Page 1


CPU
Byte 3 of Process P1
Translate byte 3 of process memory to byte 7 of MM
Mapping
MMU
Process P1
Page 0
0
1
Page 1
2
3
Main Memory
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Page 0
Page 1

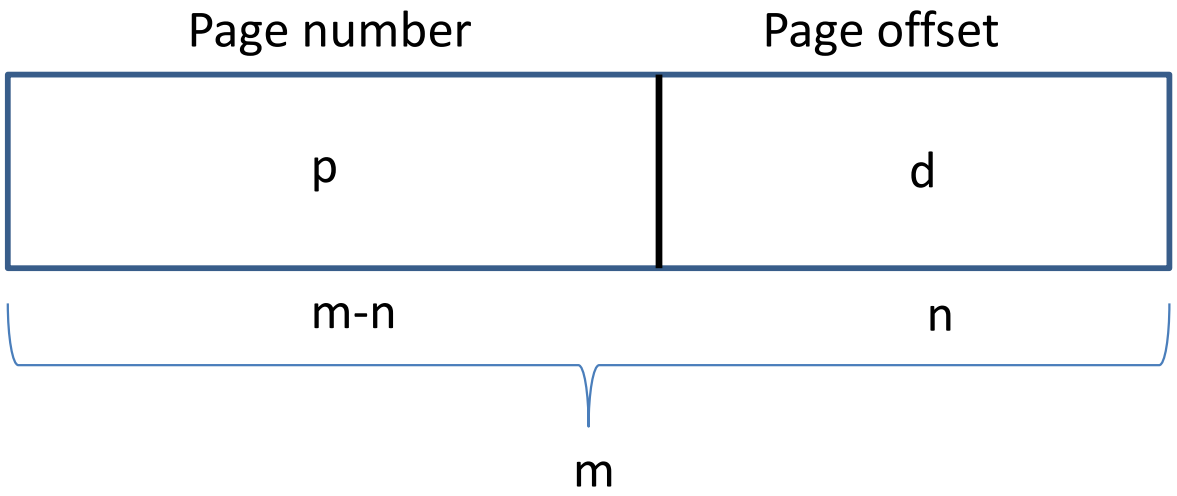

CPU
Byte 3 of Process P1
Process P1
Page 0
0
1
Page 1
2
3
Page table of P1
Page 0
Frame 1
Page 1
Frame 3
Main Memory
0
0
1
1
2
3
Page 0
2
4
5
3
6
7
Page 1
4
8
9
5
10
11
6
12
13
7
14
15

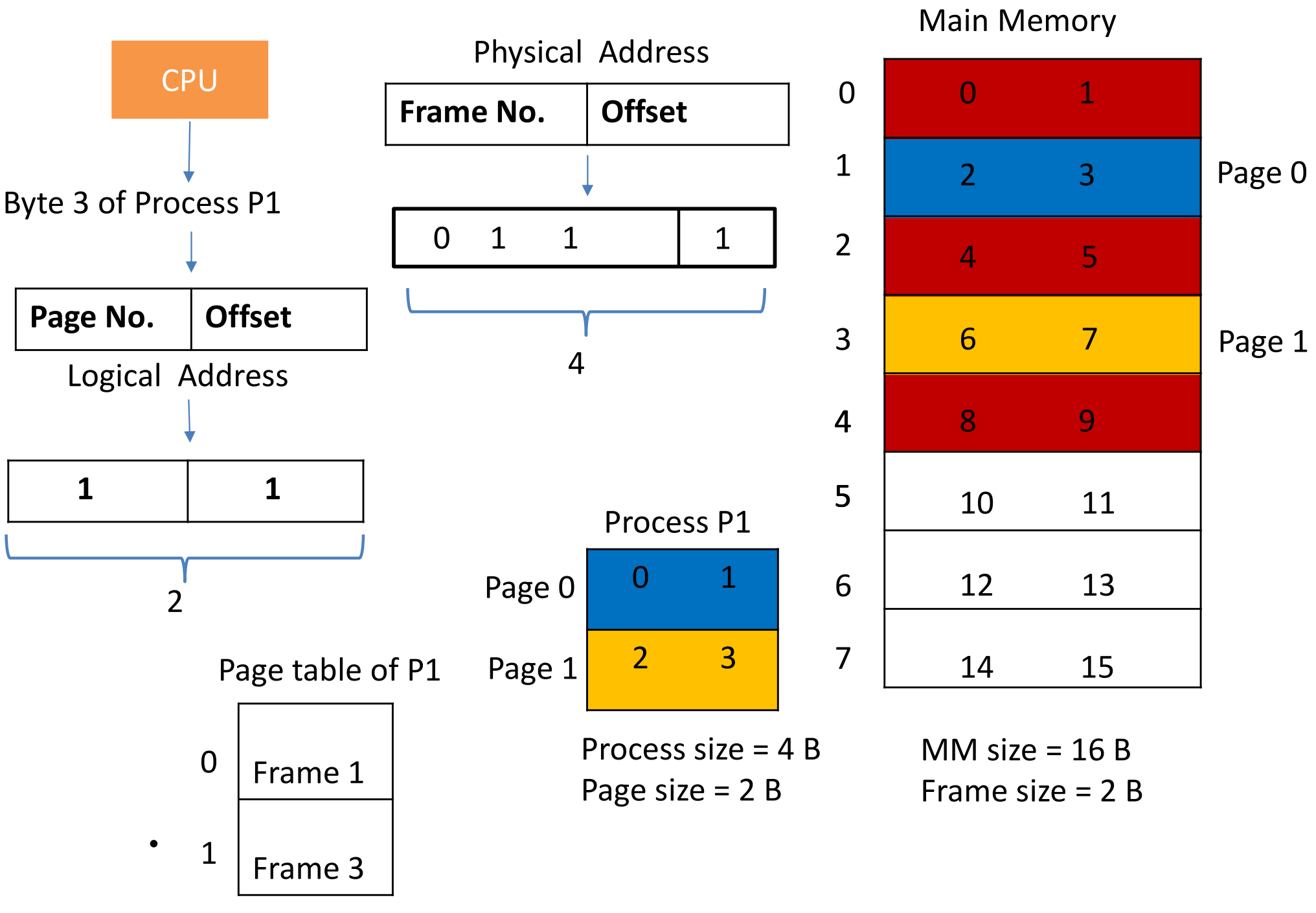
Paging model of Logical and Physical Memory



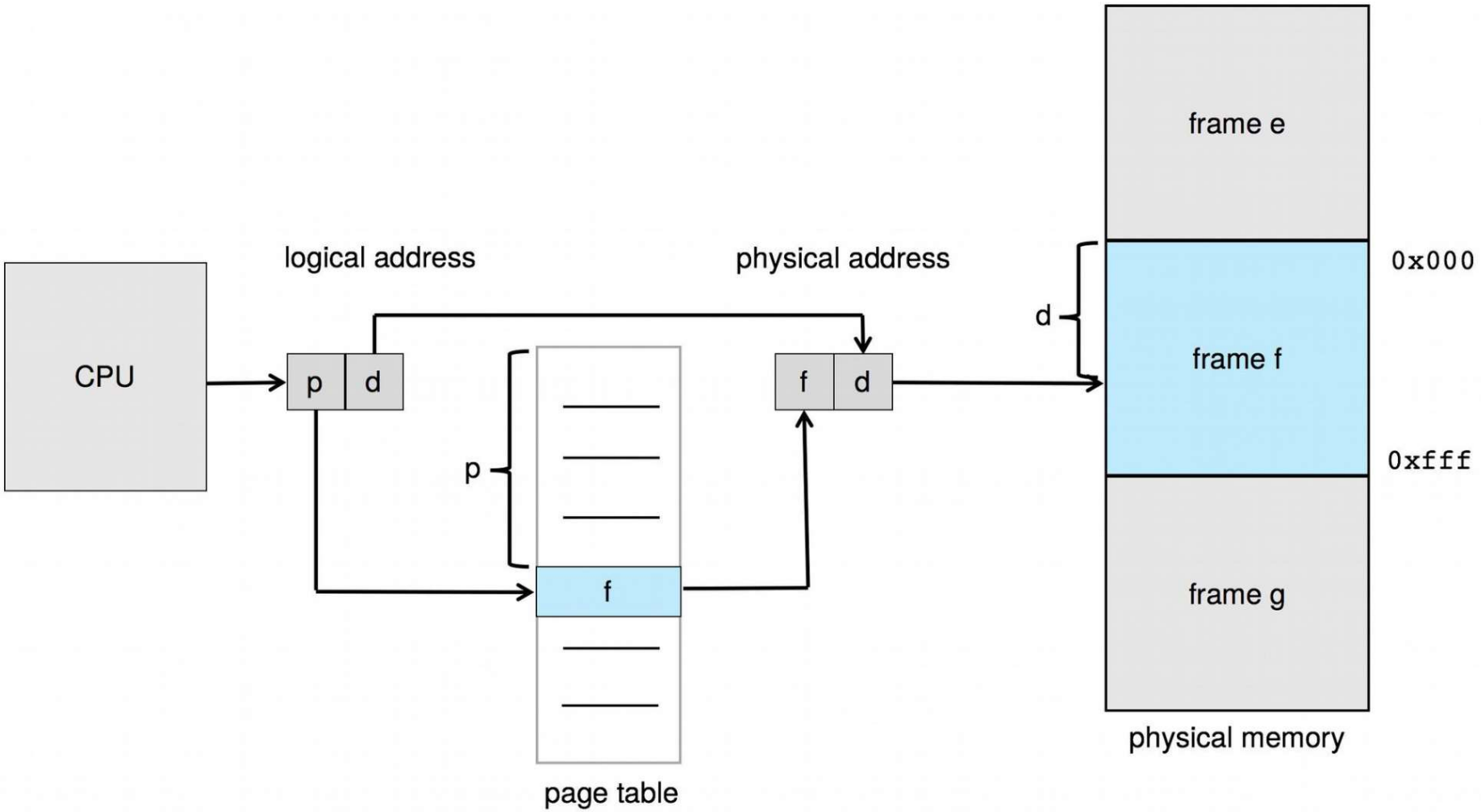
Logical address

- Address generated by CPU is divided into:
  - P**age number (*p) – used as an index into a page table which*** contains base address of each page in physical memory
  - P**age offset (*d) – combined with base address to define the*** physical memory address that is sent to the memory unit
  - For given logical address space 2^*m and page size 2^n*




CPU
Byte 3 of Process P1
Page No.
Offset
Logical Address
1
1
2
Physical Address
Frame No.
Offset
0 1 1
1
4
Page table of P1
0
Frame 1
1
Frame 3
Process P1
Page 0
0 1
Page 1
2 3
Process size = 4 B
Page size = 2 B
Main Memory
0
1
2
3
4
5
6
7
0 1
2 3
4 5
6 7
8 9
10 11
12 13
14 15
Page 0
Page 1
MM size = 16 B
Frame size = 2 B

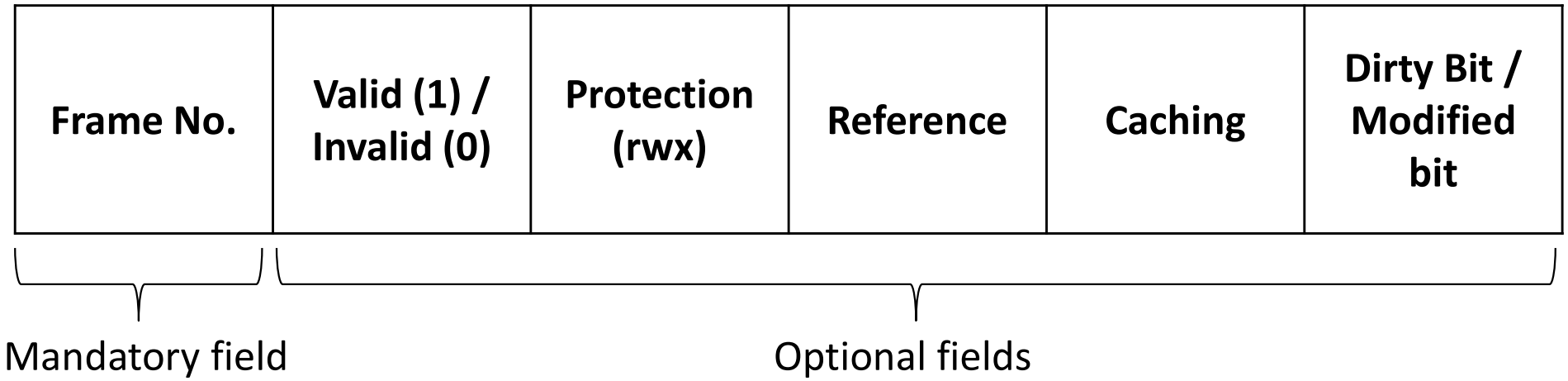
Paging Hardware



Paging summary

- Physical address space of a process can be noncontiguous
- Process is allocated physical memory whenever it is available
- Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in ***equal sized blocks*** known as ***pages***.
- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical memory into blocks of same size called **pages**
- To run a program of size ***N pages, need to find N free frames and*** load program
- Set up a **page table to translate logical to physical addresses**
- Any page (from any process) can be placed into any available frame.
- Can still have internal fragmentation

Page table entry



Page Table Implementation

- Page table is kept in main memory
  - **Page-table base register (PTBR) points to the page** table
  - **Page-table length register (PTLR) indicates size of the** page table

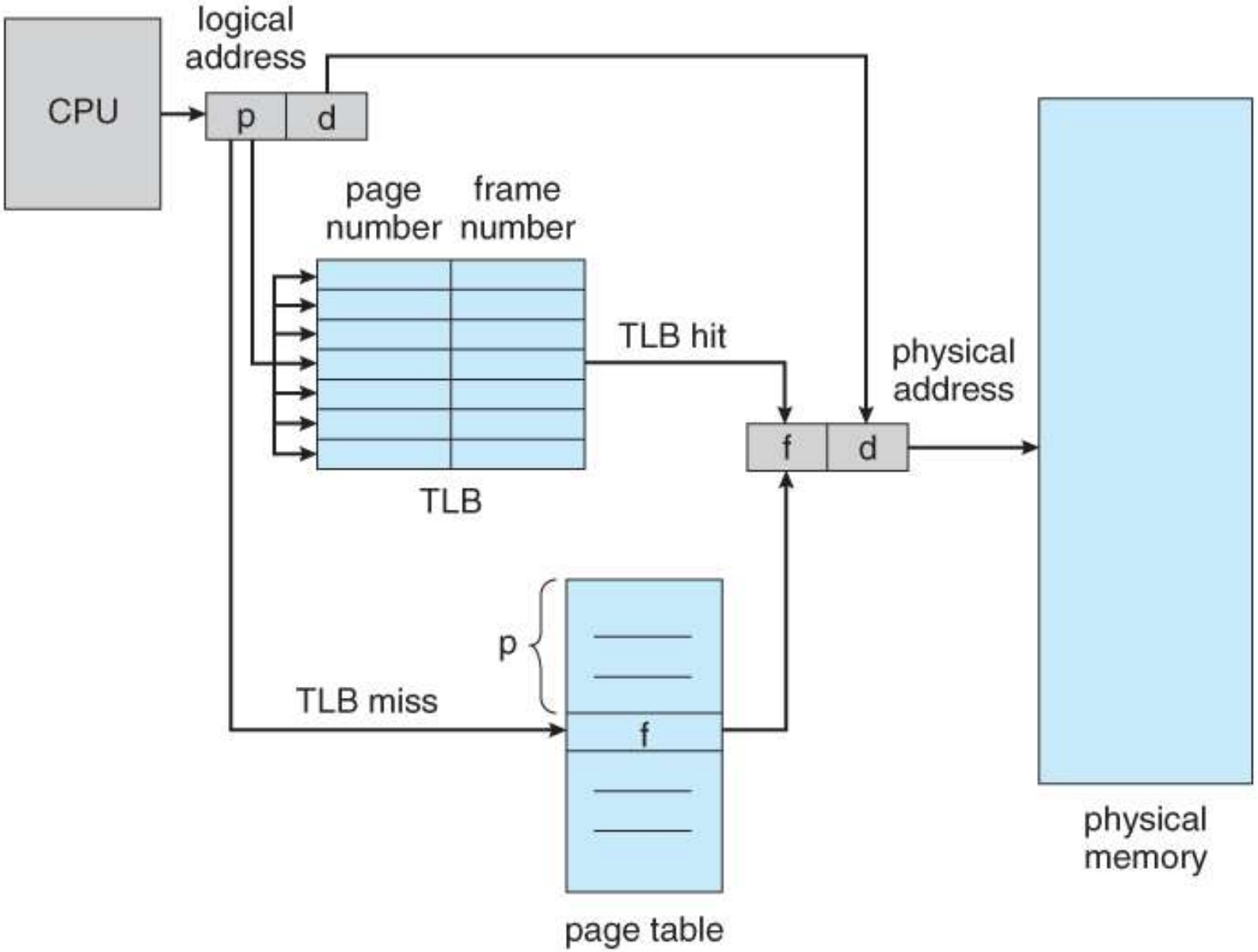
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers (TLBs) (also called associative memory).**

TLB

- We saw that with paging, every memory access requires ***two*** memory accesses - One to fetch the block number from page table and then another one to access the desired memory location.
- The solution - a very special high-speed memory device called the ***translation look-aside buffer, TLB.***
- The TLB is on the cache is very expensive and therefore very small. (Not large enough to hold the entire page table.)
- Addresses are first checked against the TLB, and if the info is not there (a ***TLB miss***), then the frame is looked up from main memory and the TLB is updated.
- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.

TLB


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

Effective access time

- Hit ratio – percentage of times that a page number is found in the TLB
  - An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
  - Effective Access Time = Hit (TLB access time + Main memory access time) + Miss (TLB + Page table access time + Main memory access time)

** This is assuming that there is no page fault, because in case there is a page fault then the page fault service time will be considered i.e. the page has to be brought to the main memory from the secondary memory.

Question

- TLB access time = 10 ns
- Main memory access time = 50ns
- What is the effective memory access time (in ns) if the TLB hit ratio is 90% and there is no page fault.

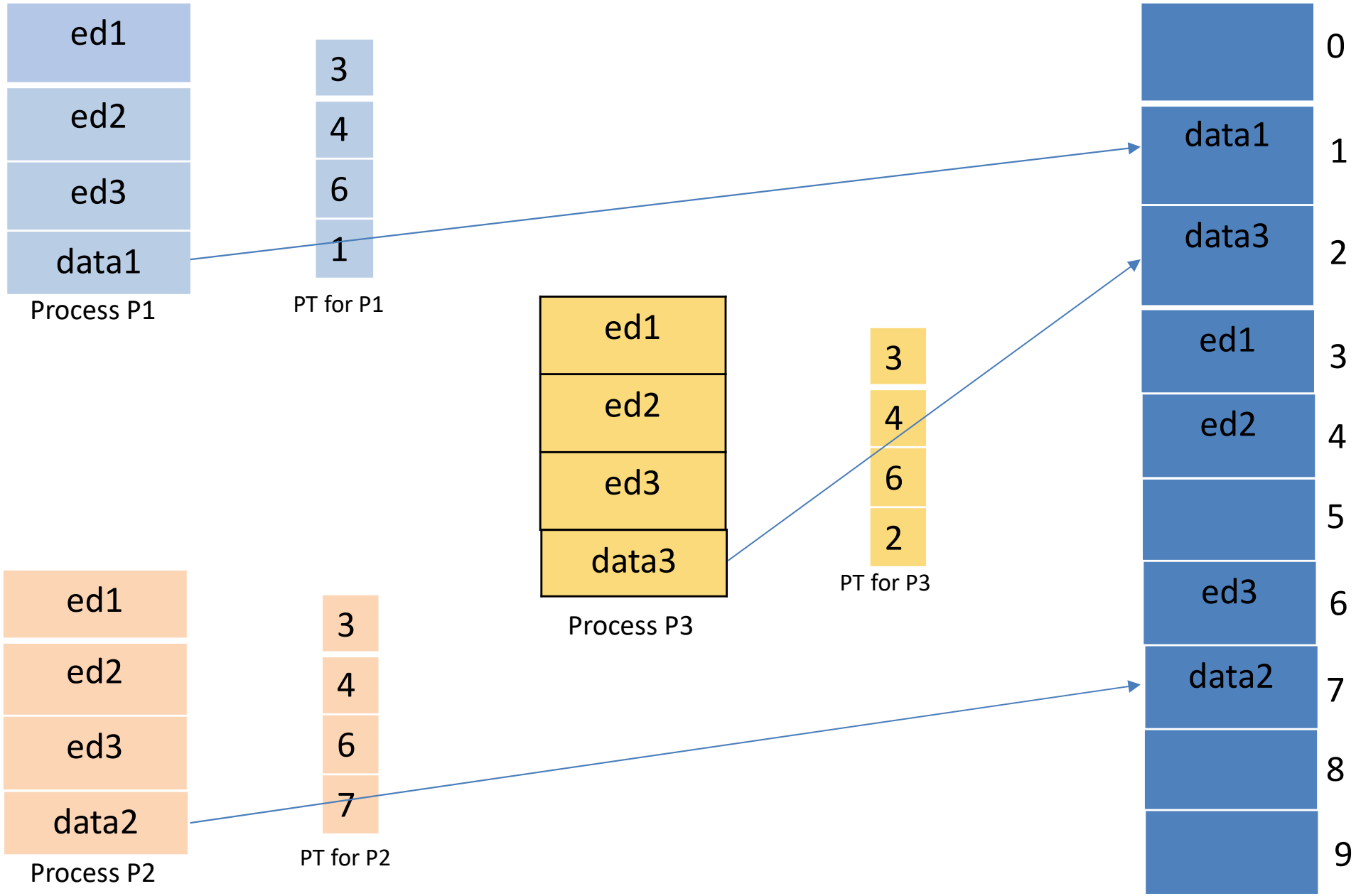
Paging- advantages and disadvantages

- **Advantages**:
  - Avoids external fragmentation
  - Provide more memory that can be used for more jobs
  - Higher degree of multiprogramming results in increased processor and memory utilization
  - Compaction overhead in relocatable partition schemes is eliminated

- **Disadvantages**:
  - Page address mapping hardware usually increases cost of computer and also slows down processor
  - Memory is used to store PMT; processor time (overhead) must be expended to maintain and update these tables
  - Though external fragmentation is eliminated, Internal Fragmentation / Page Breakage does occur.

Shared Pages & Reentrant code

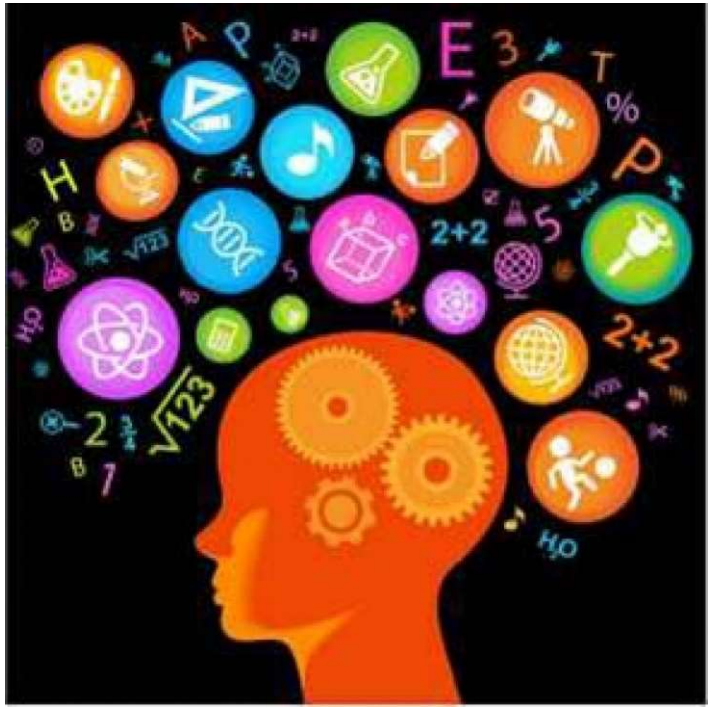
- One of the advantage of Paging
- Pages with common code shared between multiple processes
- Data is unique to each process in its own registers and data storage
- Two or more processes can execute same code at the same time
- Reentrant code is non-self modifying code
- Only reentrant code can be shared



Only one copy of editor(ed*) needs to be kept in memory

Questions ??

Memory Management
Part -3



Prachi Pandey

C-DAC Bangalore

prachip@cdac.in

Topics

- Virtual Memory
  - Demand Paging
  - Page Faults
  - Page Replacement algorithms
    - FIFO
    - LRU
    - Optimal

Memory Limitation

- Although code needs to be in memory to be executed, the entire program does not need to be
  - Only small sections execute in any small window of time, and
  - Error code, unusual routines, large data structures do not need to be in memory for the entire execution of the program
- What if we do not load the entire program into memory?
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running implies more programs run at the same time
  - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

Virtual Memory – contd.

- Separation of user logical memory from physical memory
  - Only part of the program and its data needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Also allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Virtual Memory – contd.

- Virtual address space – logical view of how process is stored in memory
  - Usually starts at address 0, contiguous addresses until end of space
  - 48-bit virtual addresses implies 2^48 bytes of virtual memory
  - Physical memory is still organized into page frames
  - MMU must map virtual to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paging

- **Demand paging** (as opposed to **anticipatory** paging) is a method of virtual memory management.
- In this method, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory (*i.e.*, if a **page fault** occurs).
- To implement Demand paging we must develop
  - Frame allocation algorithm
  - Page replacement algorithm
- It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory. This is an example of a **lazy loading** technique.

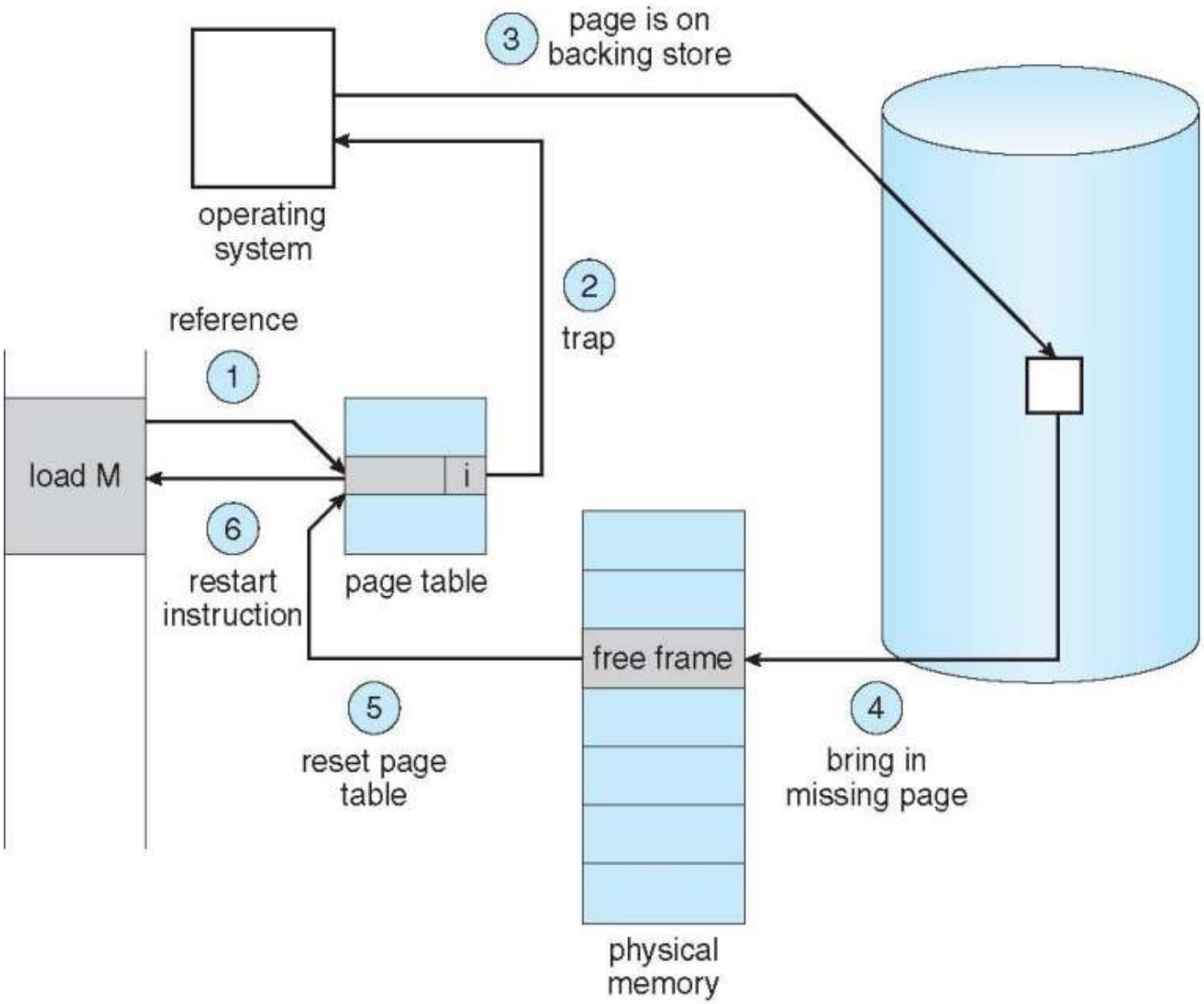
Demand Paging

- In demand paging pages are brought into memory only when needed:
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
- With each page table entry a valid–invalid bit is associated (v -> in-memory – memory resident, i -> not-in-memory)
  - Initially valid–invalid bit is set to i on all entries
- During MMU address translation, if valid–invalid bit in page table entry is i -> page fault

Handling page fault

- If there is a reference to a page, the first reference to that page will trap to operating system, i.,e. it is a
  - Page fault
- Operating system looks at another table to decide:
  - Invalid reference -> abort
  - Just not in memory
- Find free frame
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory Set validation bit = v
- Restart the instruction that caused the page fault

Steps in handling page fault

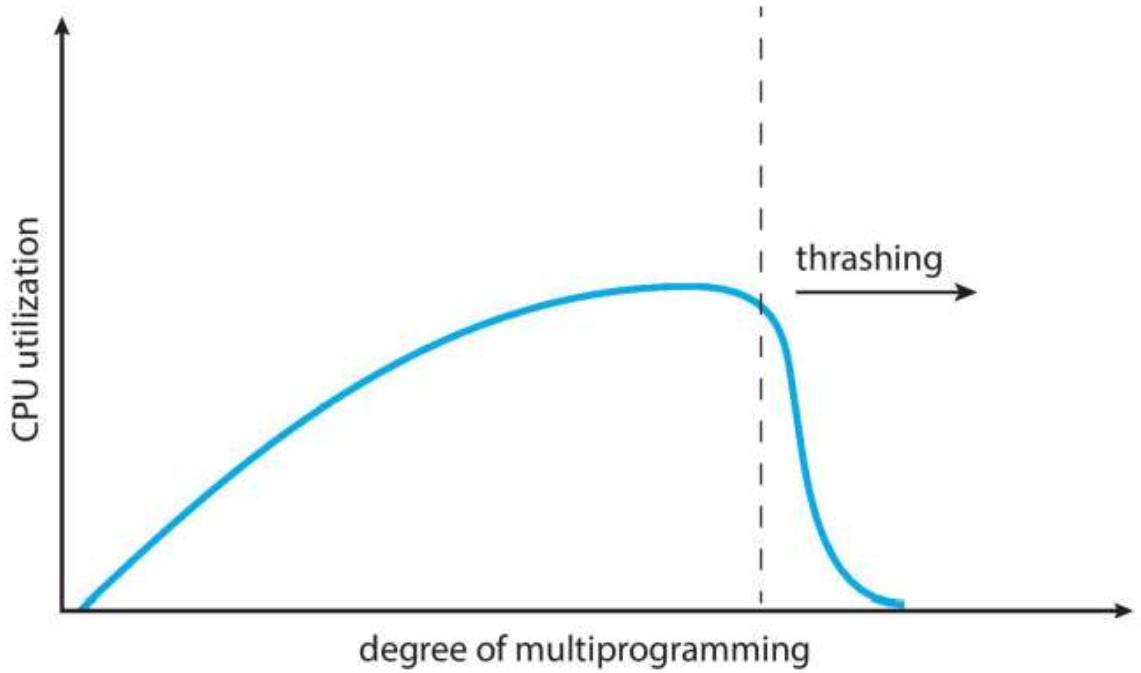


Thrashing

- If a process does not have “enough” pages in the main memory, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

Thrashing

- A process that is spending more time paging than executing is said to be ***thrashing.***



Page Replacement

Basic Scheme

1. Find the location of the desired page on the disk
2. Find a free frame
   - If there is a free frame, use it.
   - If there is **no free frame**, use a **page-replacement algorithm** to select a victim frame
   - Write the victim page to the disk; change the page and frame tables accordingly
3. Read the desired page into the (newly) free frame; change the page and frame tables
4. Restart the user process

- Main objective of a good replacement algorithm is to achieve a low *page fault rate*

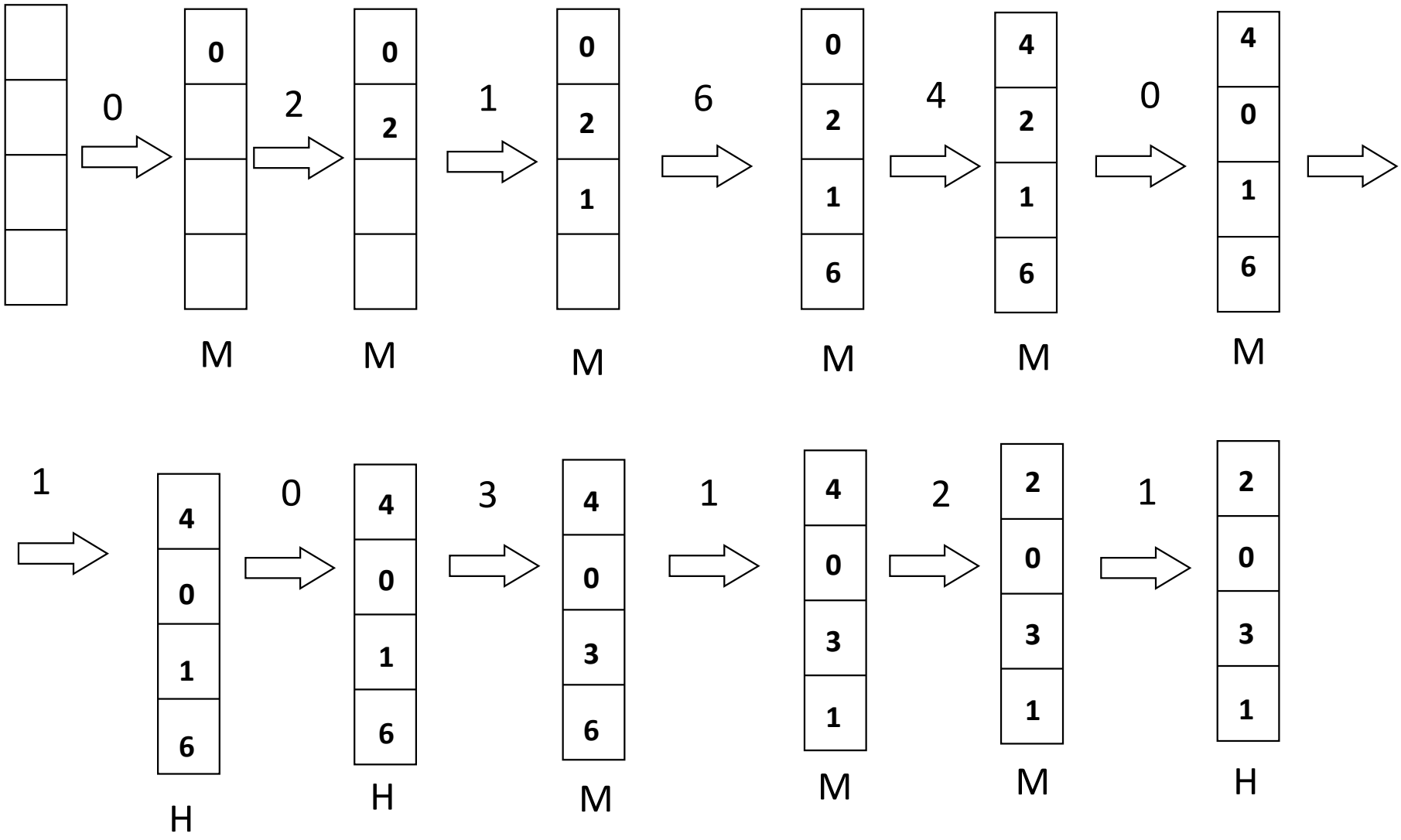
Terminologies

- The string of memory references is called ***Reference String***
- The page size is generally fixed (say 4K), so we need to consider only the page number (***p***)
- To determine the ***number of page faults***, for a given ***reference string,*** we need to know the number of ***page frames*** (memory blocks) available

First-In, First-Out (FIFO)

- The oldest page in physical memory is the one selected for replacement
- Very simple to implement
  - keep a list
    - victims are chosen from the tail
    - new pages in are placed at the head

Reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1



- Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 1 2 0

	7	0	1	2	0	3	0	4	2	3	0	3	1	2	0			
f3			1	1	1	1	0	0	0	3	3	3	3	2	2			
f2		0	0	0	0	3	3	3	2	2	2	2	1	1	1			
f1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0			
	M	M	M	M	H	M	M	M	M	M	M	H	M	M	H			

Hit ratio = 3/15 = .20

Miss ratio = 12/15 = .80

Solve

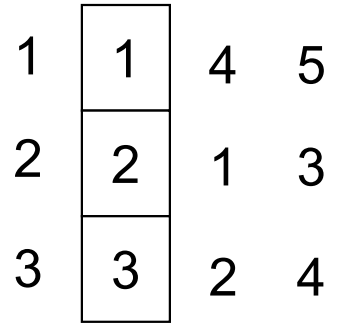
- Ref string 1 2 3 4 1 2 5 1 2 3 4 5
- With 3 frames
- With 4 frames
- Observe the result

Belady's Anomaly

- Normally Increasing Number of frames should reduce page faults
- But the number of page faults for four frames is 10 is greater than the number of faults for three frames (9)!
- ***Belady's anomaly***: is unexpected result - in which for some page replacement algorithms, the page fault rate may ***increase*** as the number of allocated frames increases!
- FIFO exhibits Belady's Anomaly

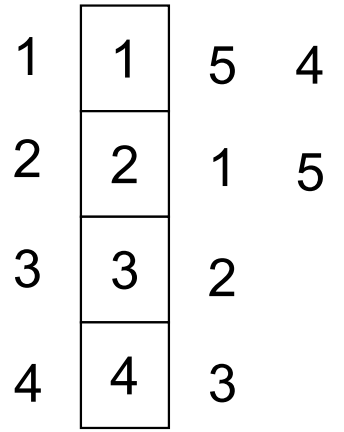
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process):



9 page faults

- 4 frames:



10 page faults

- FIFO Replacement manifests Belady's Anomaly:
  - more frames $\Rightarrow$ more page faults

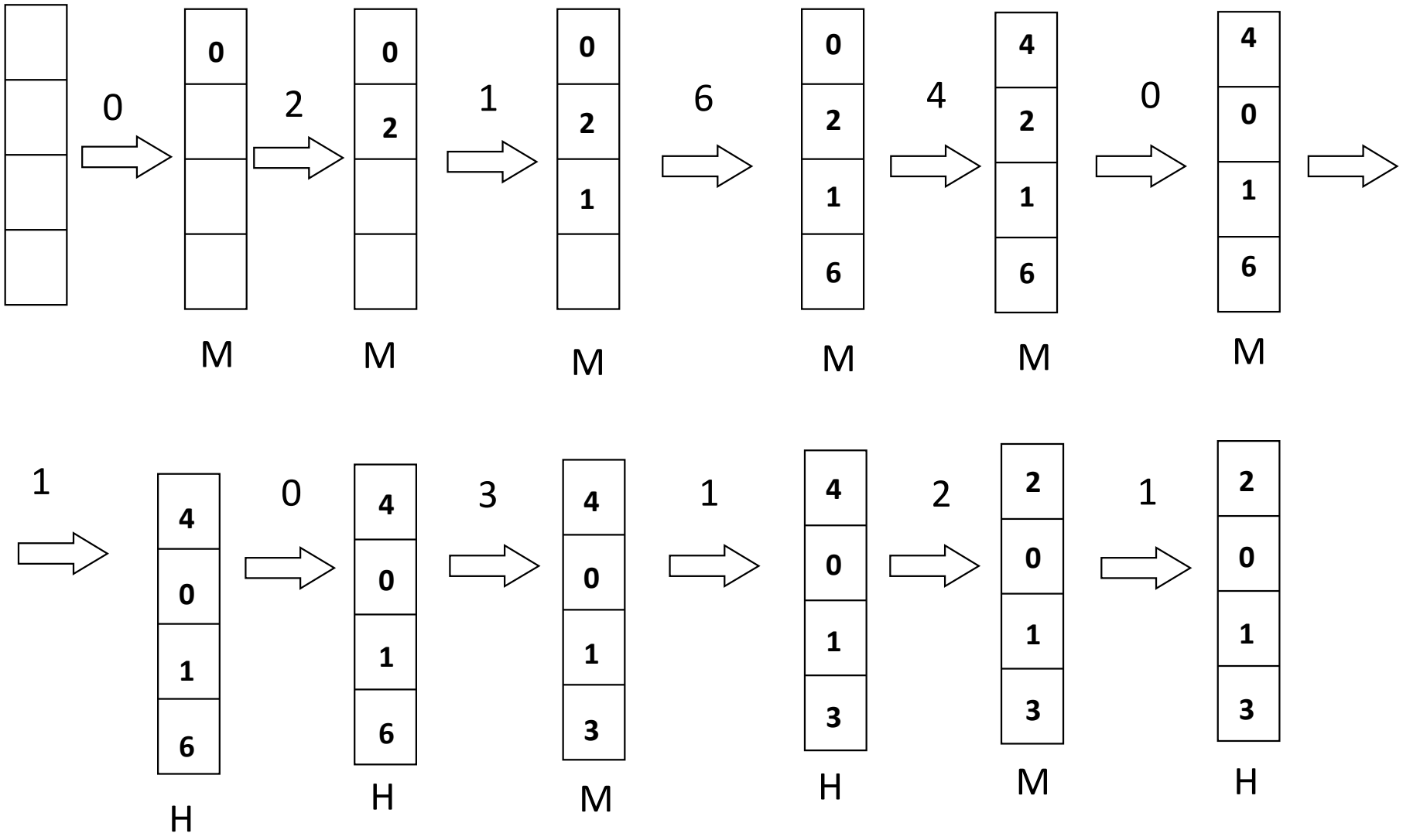
FIFO Issues

- Poor replacement policy
- Evicts the oldest page in the system
  - usually a heavily used variable should be around for a long time
  - FIFO replaces the oldest page - perhaps the one with the heavily used variable
- FIFO does not consider page usage

Least Recently Used (LRU)

- Basic idea
  - replace the page in memory that has not been accessed for the longest time
- Optimal policy looking back in time
  - as opposed to forward in time
  - fortunately, programs tend to follow similar behavior

Reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1



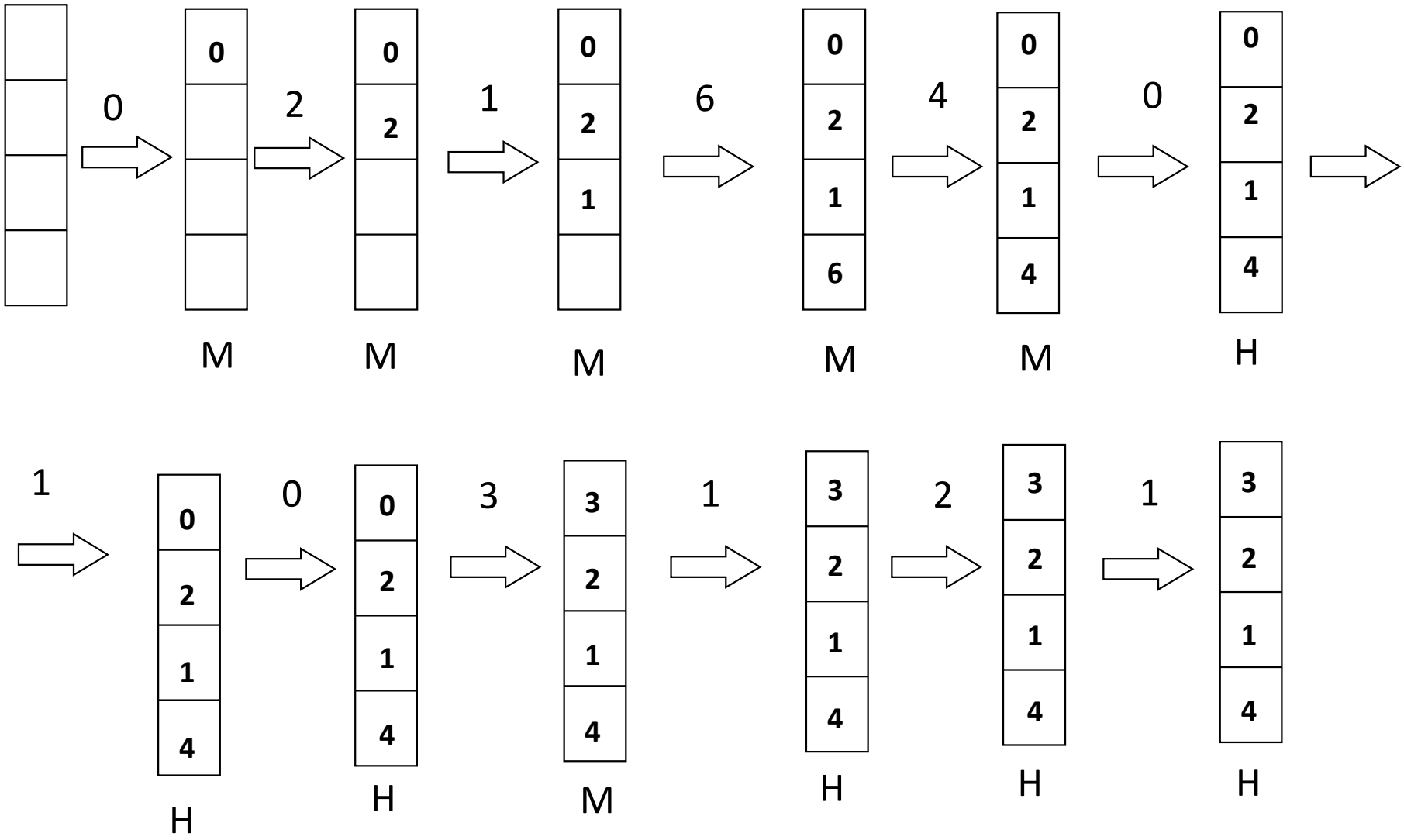
LRU Issues

- How to keep track of last page access?
  - requires special hardware support
- 2 major solutions
  - counters
    - hardware clock “ticks” on every memory reference
    - the page referenced is marked with this “time”
    - the page with the smallest “time” value is replaced
  - stack
    - keep a stack of references
    - on every reference to a page, move it to top of stack
    - page at bottom of stack is next one to be replaced

Optimal Page Replacement

- Basic idea
  - replace the page that will not be referenced for the longest time
- This gives the lowest possible fault rate
- Impossible to implement
- Does provide a good measure for other techniques

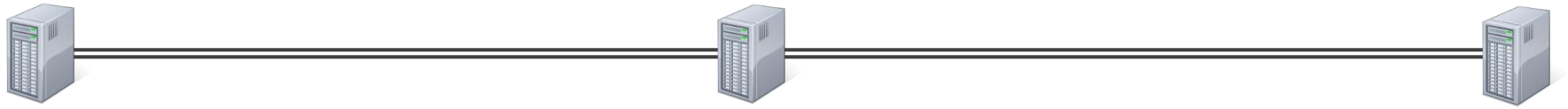
Reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1



Questions ??

Deadlocks

(Operating System)



Deepika H V

C-DAC Bengaluru

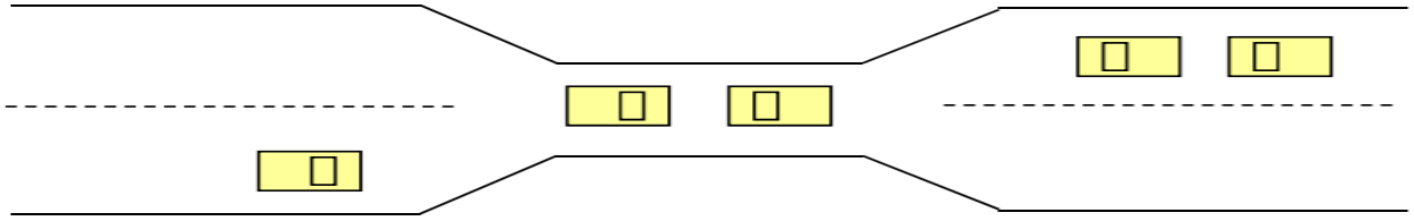
(deepikahv@cdac.in)

Agenda

- What is a Deadlock?
- Why does it happen?
- How can you detect deadlock happen?
- Consequences
- Dealing with deadlocks
- Semaphores
- Mutex
- Producer consumer problem
- Deadlock vs Starvation

What is a deadlock?

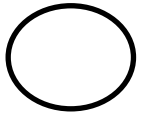
- **Definition**
  - When a waiting process is never again able to change state because the resource requested is held by other waiting process. This situation is **Deadlock**

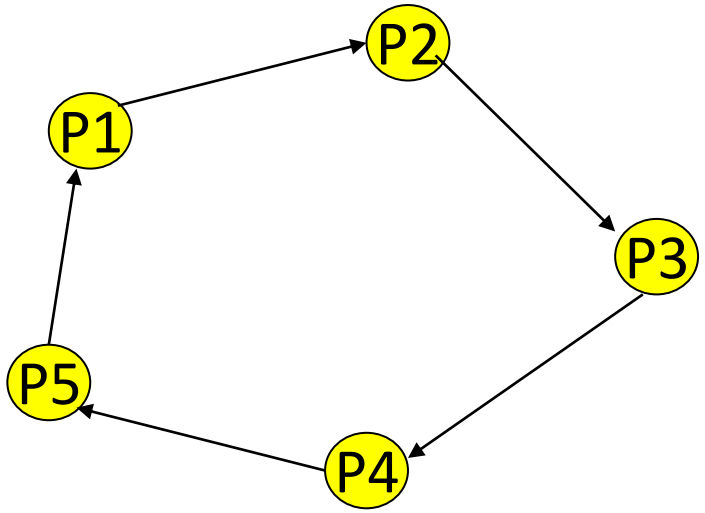


- **Resources in a system**
  - CPU cycles ; I/O devices (printers, tape drives) ; database (tables)
  - Two types – preemptable and non-premptable

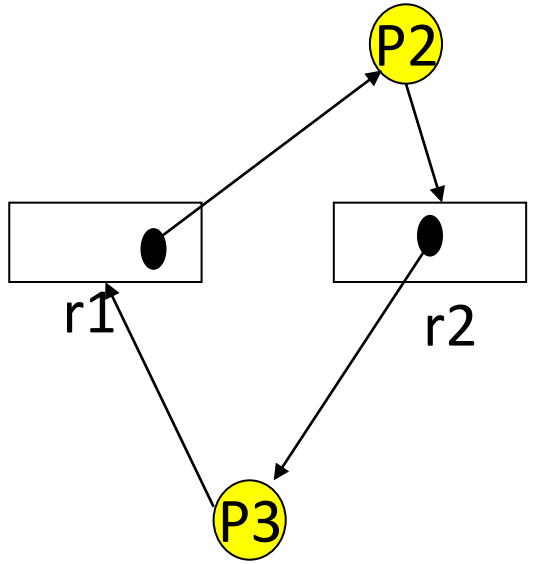
- **How to access a resource**
  - Sequence of events : (a) Request resource (b) Use resource (c) release resource
- **Must wait if resource is denied**
  - Requesting process is blocked
  - May fail with error code
- **Identify Deadlock (4 conditions)**
  - Mutual exclusion condition - each resource assigned to 1 process or is available
  - Hold and wait condition - process holding resources can request additional
  - No preemption condition - previously granted resources cannot forcibly taken away
  - Circular wait condition - must be a circular chain of 2 or more processes ; each process is waiting for resource held by next process of the chain

Graph Theoretic Models

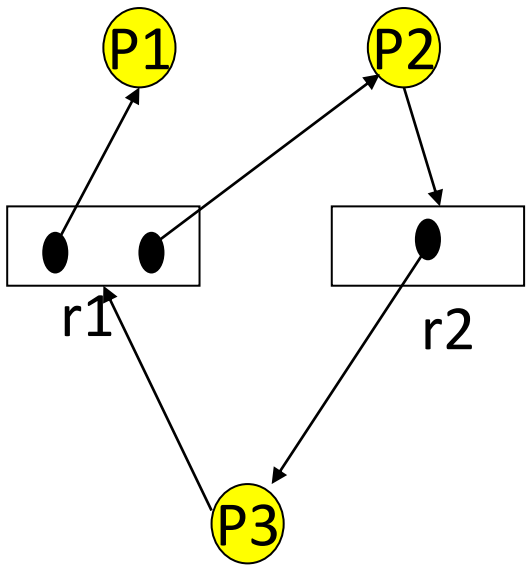
- **Resources :** 
- **Process:** 
- **Wait for Graph**



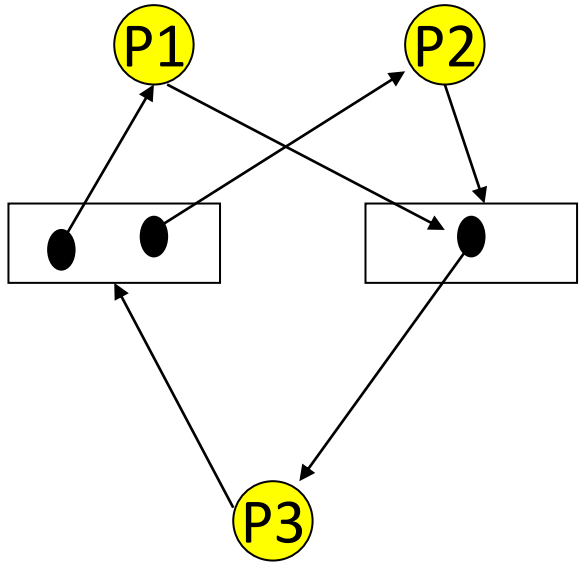
- **Resource Allocation Graph**



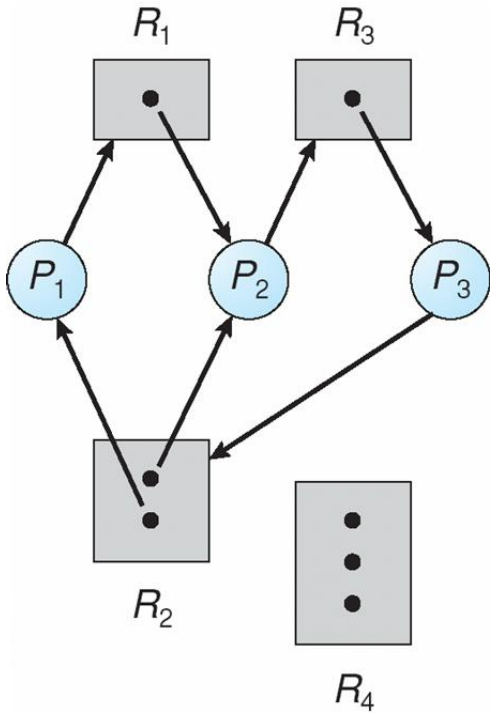
- **Resource Allocation Graph**



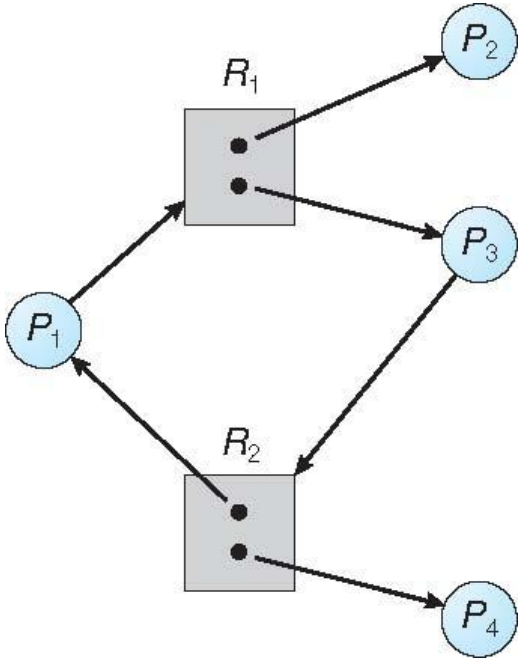
(A)



(B)



(C)



(D)

- **Which is deadlock and not?**

Strategies

- **Handling Deadlocks**
  - Ignorance
  - Prevention
  - Avoidance
  - Detection and recovery

Ignore

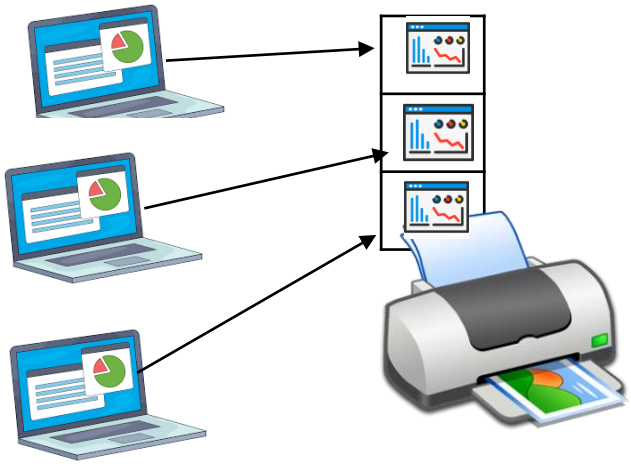
- **Ostrich Algorithm**
  - Pretend there's no problem
  - Reasonable if
    - Deadlocks occur very rarely
    - Cost of prevention is high
  - UNIX and Windows take this approach
    - Resources (memory, CPU, disk space) are plentiful
    - Deadlocks over such resources rarely occur
    - Deadlocks typically handled by rebooting
  - Trade off between convenience and correctness

Prevent

- **Ensure that at least one of the conditions for deadlock never occurs**
  - Mutual exclusion
  - No preemption
  - Hold & wait
  - Circular wait

1. Eliminate Mutual Exclusion

- **Cause for Mutual Exclusion**
  - Mutual Exclusion happens with non-sharable resources
- **How to handle mutex devices**
  - Spooling
    - Combination of buffering and queuing
    - Eg: Printer or tape drive
      - Only the printer daemon uses printer resource
      - This eliminates deadlock for printer
- **Not all devices can be spooled**



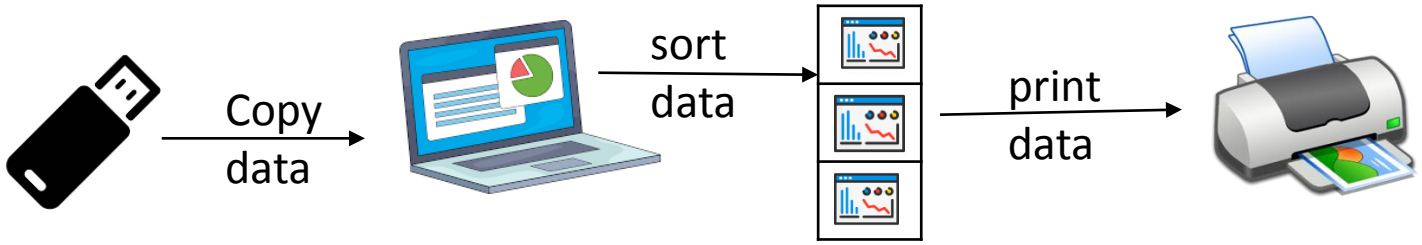
2. Preempt resources

- **Work for some resources**
  - Forcibly take away memory pages, suspending the process
  - Process may be able to resume with no ill effects
  - Eg : Round Robin scheduling of CPU cycles

- **This is not usually a viable option for all**
- **Consider a process given the printer**
  - Halfway through its job, take away the printer
  - Confusion ensues!

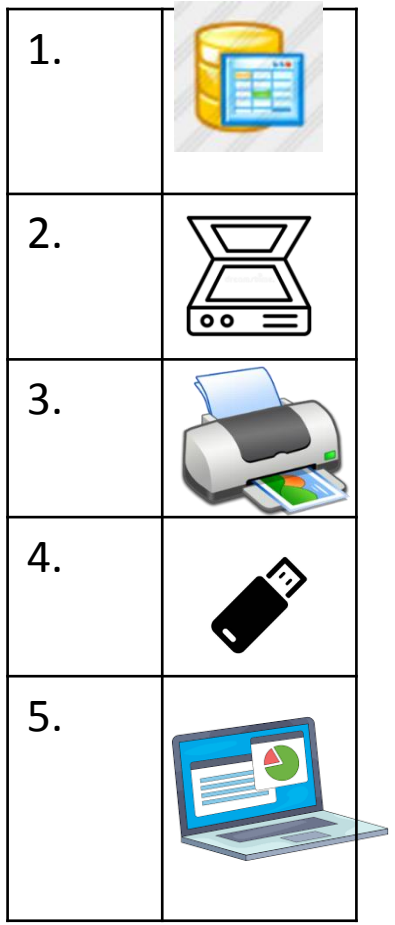
3. Stop Hold & Wait

- **Require processes to request resources before starting**
  - A process never has to wait for what it needs
- **Problem with the stratergy**
  - A process may not know required resources at start of run
    - Lower no. of resource request or over request resources
  - This also ties up resources other processes could be using
- **Variation:**
  - process must give up all held resources
  - Make a new request of resoureces required



4. No Circular Wait

- **Assign an order to resources**
- **Always acquire resources in numerical order**
  - Need not acquire them all at once!
- **Circular wait is prevented**



Prevention Summary

- **Mutual exclusion**
  - Spool everything
- **Hold and wait**
  - Request all resources initially
- **No preemption**
  - Take resources away
- **Circular wait**
  - Order resources numerically

Deadlock Avoidance

- **Do not grant a resource request if this allocation might lead to deadlock**
- **2 approaches**
  - do not start a process if its total demand might lead to deadlock: (“Process Initiation Denial”)
  - do not grant an incremental resource request if this allocation could lead to deadlock: (“Resource Allocation Denial”)
- **In both cases: maximum requirements of each resource must be stated in advance**
- **A Better Approach: Resource Allocation Denial**
  - Grant incremental resource requests if we can prove that this leaves the system in a state in which deadlock cannot occur.
  - Based on the concept of a “safe state”
  - Eg: Banker’s Algorithm

Banker's Algorithm

- **Steps involved in Banker's Algorithm**
  - Tentatively grant each resource request
  - Analyze resulting system state to see if it is "safe".
  - If safe, grant the request
  - if unsafe refuse the request (undo the tentative grant)
  - block the requesting process until it is safe to grant it.

The two states defined in Deadlock Avoidance

① Safe State

A system is in "safe state" if there is at least one assignment schedule to complete all the processes without deadlock and without any process releasing their resources

(deadlock will never happen even in the worst case)

➡ If a system is in safe state, deadlock can always be avoided

② Unsafe State

If a system is not in safe state, the system must be in "unsafe state"

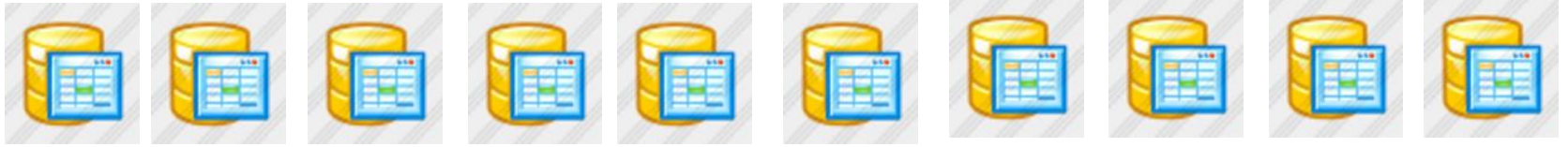
(deadlock can happen in the worst case)

➡ If a system is in unsafe state, deadlock may not be avoided

Courtesy :DR. Hiroshi Fujinoki, Southern Illinios Univerity Edwardsville

Sample Example

- **At Time T0**
  - Total Resources
    - A : 10
    - B : 5
    - C : 7

A 

B 

C 

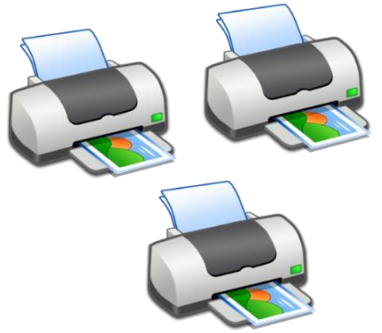
Sample Example

- **At Time T0**
  - Total Resources
    - A : 10
    - B : 5
    - C : 7
- **At Time T6**
  - Resource Avail
    - A : 3
    - B : 3
    - C : 2

AVAILABLE: 3 3 2		
Process	Allocation	Max
	A B C	A B C
P0	0 1 0	7 5 3
P1	2 0 0	3 2 2
P2	3 0 2	9 0 2
P3	2 1 1	2 2 2
P4	0 0 2	4 3 3

A

B

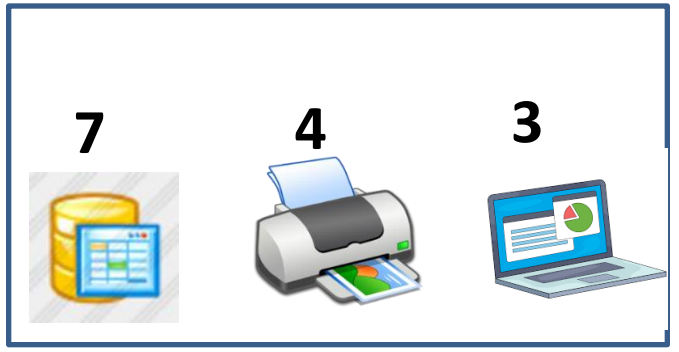


C

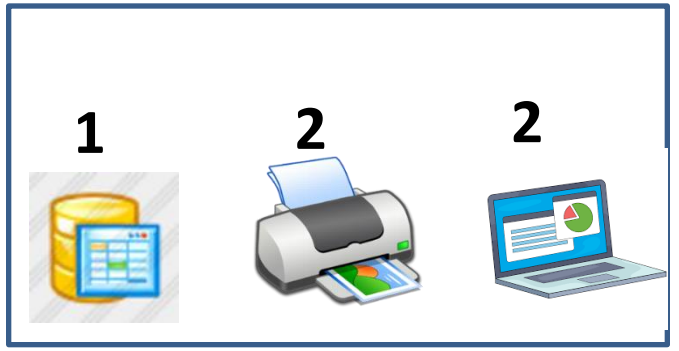
- **Need = Max – Allocation**
- **Pick P0**

AVAILABLE: 3 3 2			
Process	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

AVAILABLE:	3 3 2		
Process	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

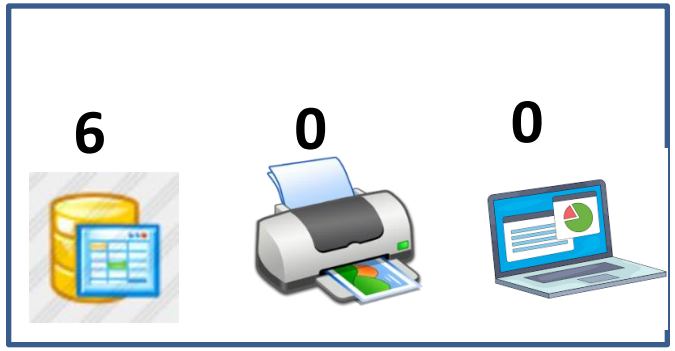


AVAILABLE: 3 3 2			
Process	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

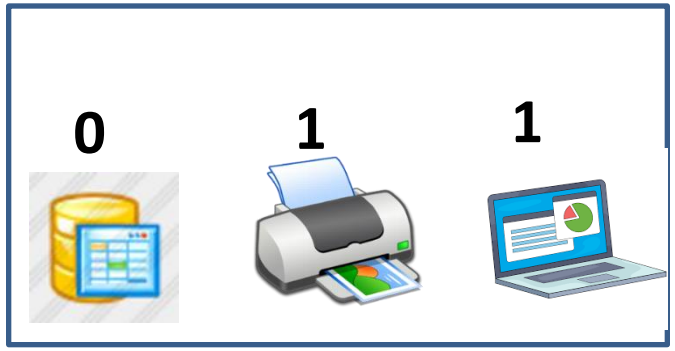


```
      3  3  2
   -  1  2  2
Rem   2  1  0
Complete P1
Rem   2  1  0
Ret + 3  2  2
Avail 5  3  2
```

AVAILABLE: 5 3 2			
Process	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3
P1	0 0 0	3 2 2	0 0 0
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

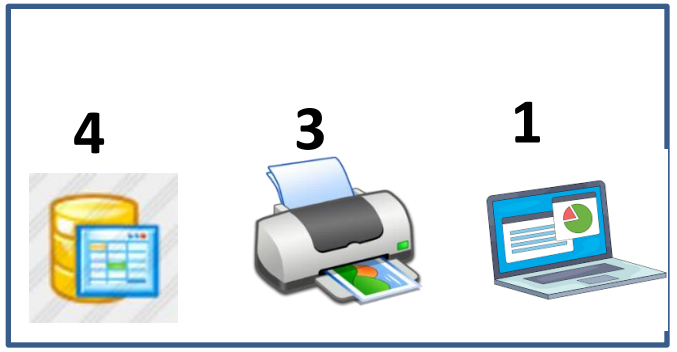


AVAILABLE: 5 3 2			
Process	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3
P1	0 0 0	3 2 2	0 0 0
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1



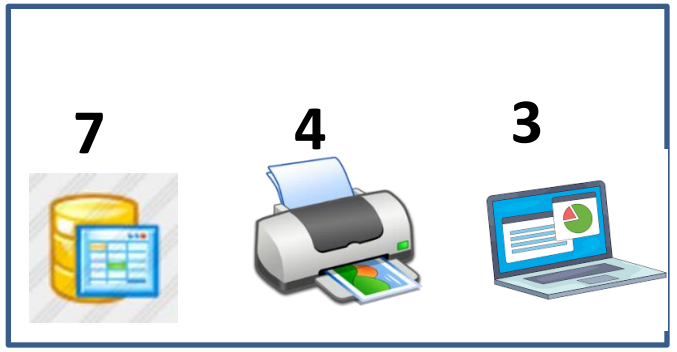
	5	3	2
-	0	1	1
Rem	5	2	1
Complete P3			
Rem	5	2	1
Ret +	2	2	2
Avail	7	4	3

AVAILABLE: 7 4 3			
Process	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3
P1	0 0 0	3 2 2	0 0 0
P2	3 0 2	9 0 2	6 0 0
P3	0 0 0	2 2 2	0 0 0
P4	0 0 2	4 3 3	4 3 1



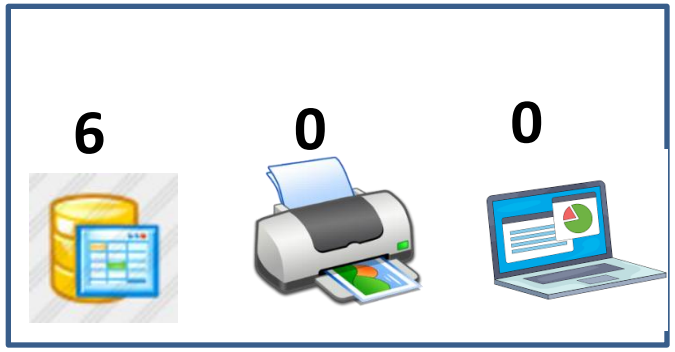
```
      7  4  3
  -   4  3  1
Rem   3  1  2
Complete P4
Rem   3  1  2
Ret + 4  3  3
Avail 7  4  5
```

AVAILABLE: 7 4 5			
Process	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3
P1	0 0 0	3 2 2	0 0 0
P2	3 0 2	9 0 2	6 0 0
P3	0 0 0	2 2 2	0 0 0
P4	0 0 0	4 3 3	0 0 0



```
      7  4  5
  -   7  4  3
Rem   0  0  2
Complete P0
Rem   0  0  2
Ret + 7  5  3
Avail 7  5  5
```

AVAILABLE: 7 5 5			
Process	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 0 0	7 5 3	0 0 0
P1	0 0 0	3 2 2	0 0 0
P2	3 0 2	9 0 2	6 0 0
P3	0 0 0	2 2 2	0 0 0
P4	0 0 0	4 3 3	0 0 0



7 5 5
- 6 0 0
Rem 1 5 5
Complete P2
Rem 1 5 5
Ret + 9 0 2
Avail 10 5 7

Four resources ABCD. A has 6 instances, B has 3 instances, C Has 4 instances and D has 2 instances.

Process	Allocation ABCD	Max ABCD
P1	3011	4111
P2	0100	0212
P3	1110	4210
P4	1101	1101
P5	0000	2110

Is the current state safe?

If P5 requests for (1,0,1,0), can this be granted?

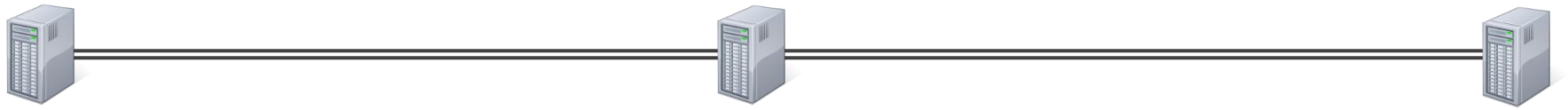
Deadlock Detection and Recovery

- **Allow the system to enter deadlock ; then detect and recover**
- **Detection**
  - For single instance of the resource using wait-for-graph
  - For multiple instance of resource use Banker's Algorithm
- **Recovery**
  - Abort
  - Preempt

Thank You

Process Synchronization

(Operating System)



Deepika H V

C-DAC Bengaluru

(deepikahv@cdac.in)

- **Process**
  - Independent Process : It cannot affect or be effected by the other processes executing in the system ;Does not share any data with any other processes
  - Cooperating Process: It can affect or be affected by the other processes executing in the system.
- **Cooperating Process**
  - Share variable, memory, buffer, code, resources
- **Synchronization**
  - Several processes access and manipulate the same data **concurrently**
  - Outcome of the execution depends on the particular **order** in which the access takes place
  - **data consistency** is must in co-operating processes

- **What is Synchronization?**
  - Process Synchronization means **sharing system resources by processes** in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data.
  - Maintaining **data consistency demands mechanisms** to ensure synchronized execution of cooperating processes.

Producer Consumer Problem

- We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0.
- It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer()

Consumer()

Producer

```
int count = 0;
Producer() {
while (TRUE)
        produce item();
        if (count == MAX_SIZE)   sleep();
        enter item();
        count ++;
        if (count == 1) wakeup(Consumer);
}
```

```
register1 = count
register1 = register1 + 1
count = register1
```

Consumer

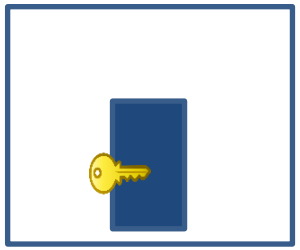
```
Consumer() {
while(TRUE)
        if (count == 0) sleep();
        remove item();
        count --;
        if (count == MAX_SIZE - 1)  wakeup(Producer);
        consume item();
}
```

- **Race Condition**
  - It is an undesirable situation which occurs when 2 computer program processes/threads attempt to access the same resource at the same time and cause problems in the system.
  - A race condition can be difficult to reproduce and debug because the end result is nondeterministic and depends on the relative timing between interfering threads.
- **Critical Section**
  - Critical Section is the part of a program which tries to access shared resources.
  - Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder** section

- **The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise**
- In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.
  - **Primary conditions :**
    - Mutual Exclusion : one process is executing inside critical section then the other process must not enter in the critical section
    - Progress : one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.
  - **Secondary conditions** :
    - Bounded Waiting : able to predict the waiting time for every process to get into the critical section.
    - Architectural Neutrality : our solution is working fine on one architecture then it should also run on the other ones as well.

CSP Methods - Mutex

- **Simplest is mutex lock**
- **Product critical regions with it by first `acquire()` a lock then `release()` it**
  - Boolean variable indicating if lock is available or not
- **Calls to `acquire()` and `release()` must be atomic**
  - Usually implemented via hardware atomic instructions
- **But this solution requires busy waiting**
  - This lock therefore called a **spinlock**



```
acquire() {
   while (!available); /* busy wait */
   available = false;;
}
```

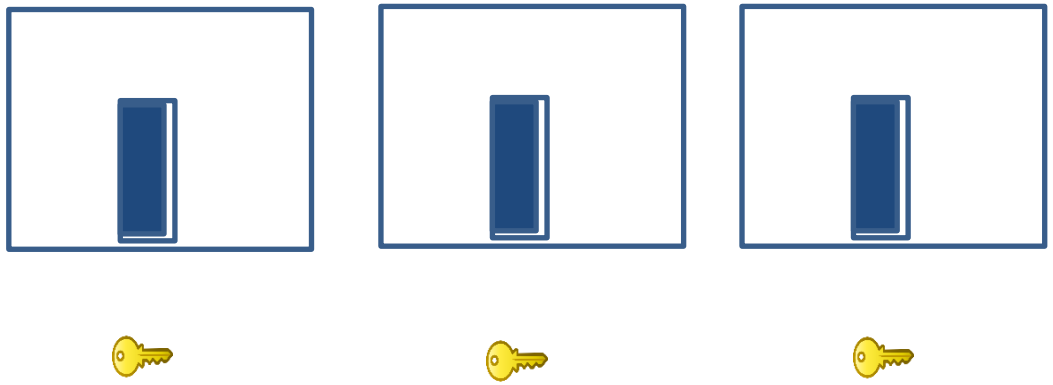
```
release() {
   available = true;
}
```

```
do{
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

CSP Methods - Semaphore

- Integer variable used in a mutual exclusive manner by various cooperating processes in order to achieve synchronization.
- tool that does not require busy waiting
- Semaphores are like integers, except NO negative values
- Accessed only through two standard operations **P()/acquire()/wait()/down() and V()/release()/ signal()/up()**.

```
Semaphore s; // initialized to no of instances of the CS
do {
        down(s);
        // Critical Section
        up(s);
        // remainder section
} while (TRUE);
```



- **Two Types**
  - Counting Semaphores: value can range over an unrestricted domain; varies from -∞ to +∞. Initialize to number of CS.
  - Binary Semaphores: integer value can range only between 0 and 1; also known as mutex locks. Initialize to 1

```
Semaphore s; // initialized to no of instances of the CS
do {
        down(s);
        // Critical Section
        up(s);
        // remainder section
} while (TRUE);
```

```
Down(Semaphore S){
        S.value= S.value -1;
        if(S.value<0){
                put process(PCB) in suspended list sleep();
        }
        else{
                return ;
        }
}
```

```
Up(Semaphore S){
        S.value = S.value +1;
        if(S.value <=0){
                select
process from sleep list
                wakeup();
        }
}
```

Binary Sempahore

```
Down(Semaphore S){
        if(S.value ==1){
                S.value=0;
        }
        else{
                Block this process and put in suspended list
                sleep() ;
        }
}
```

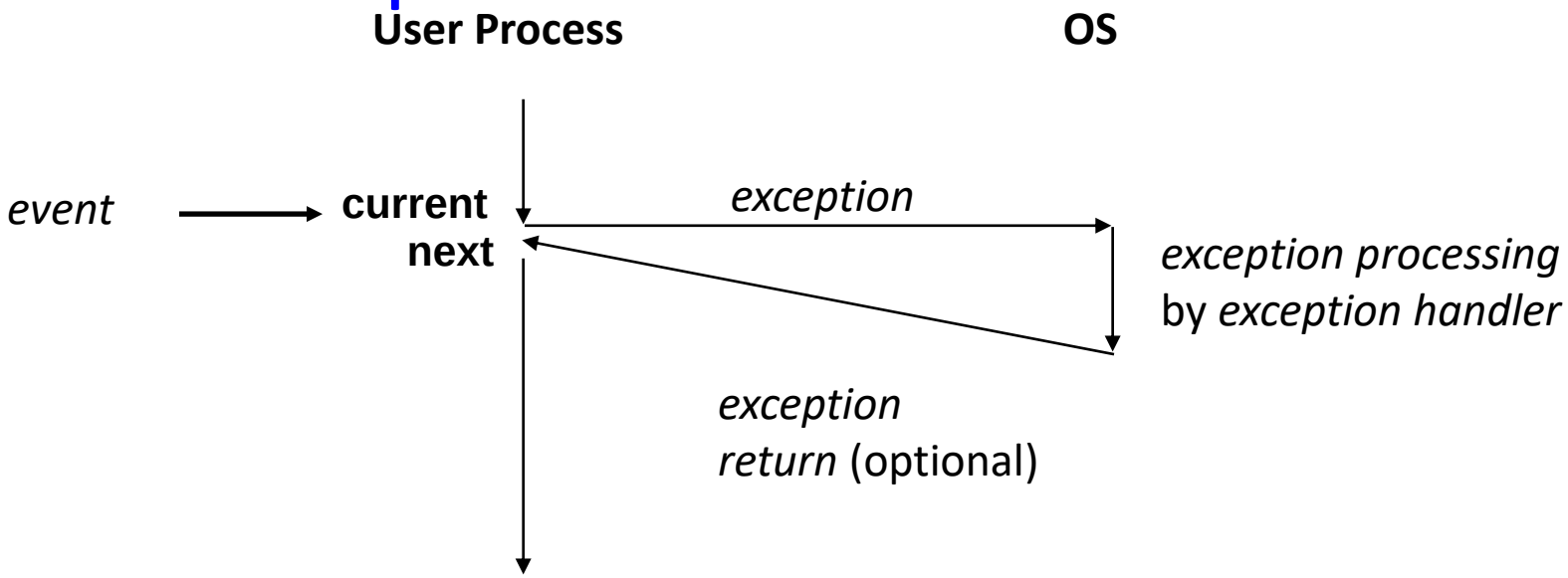
```
Up(Semaphore S){
    if( suspended list is Empty){
            S.value=1;
    }else{
            select process from sleep list
            wakeup();
    }
}
```

```
Semaphore s; // initialized to 1
do {
        down(s);
        // Critical Section
        up(s);
        // remainder section
} while (TRUE);
```

DEADLOCK	STARVATION
A situation that occurs with a set of processes in which every process is waiting for an event that can only be caused by another process in the set	A situation in which a process is perpetually denied necessary resources to process its work
Occurs when each process is holding a resource and waits for a resource held by another process	Occurs when a process waits for a resource for a long period of time
All the related processes cannot proceed	Some processes wait for resources but others can proceed

Interrupt

- An unexpected event that causes a processor to (temporarily) transfer control to another program, or function.
- When that function completes, control is (typically) returned to the interrupted process, which resumes from the point it was interrupted



Interrupt

Teacher is OS

Students is I/O device

Types

- **Hardware Interrupts**
  - asynchronous entities
  - typically employed to provide an effective means for a system to react to outside stimuli.
- **Software Interrupts (Exceptions)**
  - Has both synch and asynch
  - Exceptions generated by processes.
  - Caused by events that occur as result of executing an instruction

Hardware Interrupts

- **I/O interrupts**
  - hitting ctl-c at the keyboard
  - arrival of a packet from a network
  - arrival of a data sector from a disk
- **Hard reset interrupt**
  - hitting the reset button
- **Soft-reset interrupts**
  - hitting ctl-alt-delete on a PC

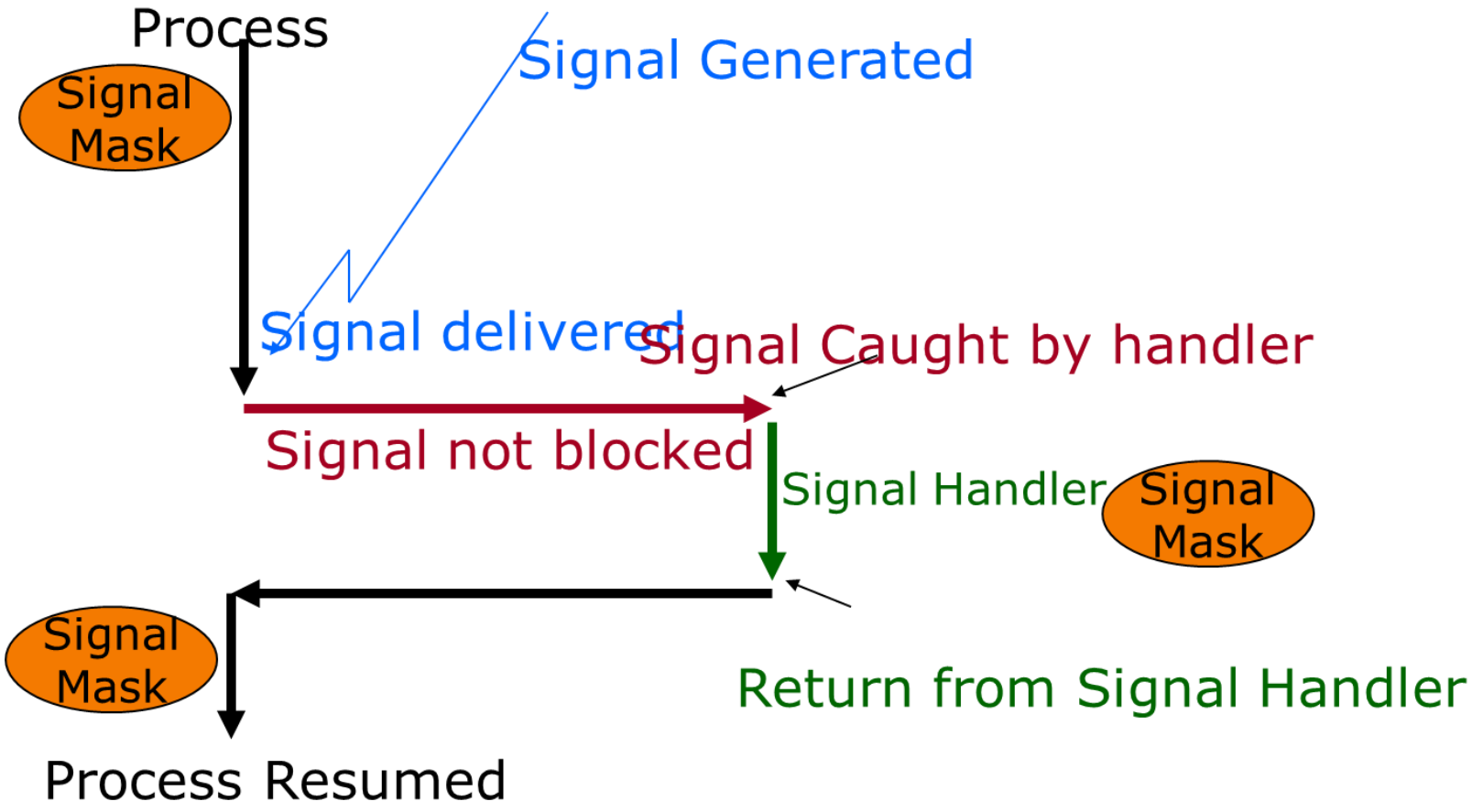
Software Interrupts

- **Traps**
  - Intentional - system calls, breakpoint traps, special instruction
  - Returns control to “next” instruction
- **Faults**
  - Unintentional but possibly recoverable - page faults.
  - Either re-executes faulting instruction.
- **Aborts**
  - unintentional and unrecoverable -parity error, machine check.
  - Aborts current program

Signals

- **A notification of an event**
  - Event gains attention of the OS
  - OS stops the current process, sending it a signal
  - Signal handler executes to completion
  - Application process resumes where it left off
- **Different signals are identified by small integer ID's**

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated


Process
Signal Mask
Signal Generated
Signal delivered
Signal Caught by handler
Signal not blocked
Signal Handler
Signal Mask
Signal Mask
Return from Signal Handler
Process Resumed

Signal Mask

- **Process can temporarily prevent signal from being delivered by *blocking* it.**
- ***Signal Mask* contains a set of signals currently blocked.**
- **Important! Blocking a signal is different from ignoring signal. Why?**
  - When a process blocks a signal, the OS does not deliver signal until the process unblocks the signal
  - A *blocked* signal is not delivered to a process until it is unblocked.
  - When a process ignores signal, signal is delivered and the process handles it by throwing it away.

Sending a Signal

- **Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process**
- **Kernel sends a signal for one of the following reasons:**
  - Kernel has detected a system event such as divide by zero (SIGFPE) or termination of a child process (SIGCHLD)
  - Another process has invoked the kill system call to explicitly request that the kernel send a signal to the destination process

Receiving

- **A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal**
- **Five possible ways to react:**
  - Ignore the signal (do nothing)
  - Terminate the process
  - Temporarily stop the process from running
  - Continue a stopped process (let it run again)
  - Catch the signal by executing a user-level function called a signal handler

Predefined Signals

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX
```

Signal via Keyboard

- **Ctrl-c -> 2/SIGINT signal**
  - Default handler exits process
- **Ctrl-z -> 20/SIGTSTP signal**
  - Default handler suspends process
- **Ctrl-\ -> 3/SIGQUIT signal**
  - Default handler exits process
- **Check using stty -a**

Signal via Commands

- **kill -signal pid**
  - Send a signal of type signal to the process with id pid
  - Can specify either signal type name (-SIGINT) or number (-2)
- **No signal type name or number specified => sends 15/SIGTERM signal**
  - Default 15/SIGTERM handler exits process
- **Examples**
  - kill –2 1234
  - kill -SIGINT 1234

Signal via function call – raise()

- **int raise(int iSig);**
  - Commands OS to send a signal of type iSig to current process
  - Returns 0 to indicate success, non-0 to indicate failure

- **Example**

```
int ret = raise(SIGINT); /* Process commits suicide. */
assert(ret != 0);        /* Shouldn't get here. */
```

```
#include<stdio.h>
#include<signal.h>

void main()
{
        printf(" the process id is %d\n",getpid());
        raise(SIGINT);
        printf(" the parent process id is %d\n",getppid());

}
```

Signal via function call – kill()

- **int kill(pid_t iPid, int iSig);**
  - Sends a iSig signal to the process whose id is iPid
  - Equivalent to raise(iSig) when iPid is the id of current process

- **Example**

```
pid_t iPid = getpid(); /* Process gets its id.*/
kill(iPid, SIGINT);
```

```
#include<stdio.h>
#include<signal.h>

void main()
{
	int ret;
	ret = fork();
	if(ret == 0)
	{
		printf(" the process id is %d\n",getpid());
		printf(" the parent process id is %d\n",getppid());
		for(;;)
			printf("Looping in child process\n");
	}
	else
	{
		kill(ret,SIGINT);
	}
}
```

Signal via function call - signal

- **sighandler_t signal(int iSig, sighandler_t pfHandler);**
  - Installs function pfHandler as the handler for signals of type iSig
  - pfHandler is a function pointer:

- **Returns the old handler on success, SIG_ERR on error**

- **pfHandler is invoked whenever process receives a signal of type iSig**

```
int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, SIG_IGN);
    …
}
```

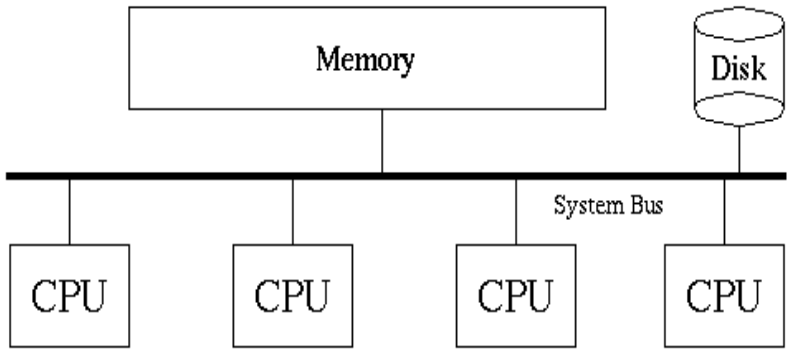
```
…
static FILE *psFile; /* Must be global. */
static void cleanup(int iSig) {
    fclose(psFile);
    remove("tmp.txt");
    exit(EXIT_FAILURE);
}
int main(void) {
    void (*pfRet)(int);
    psFile = fopen("temp.txt", "w");
    pfRet = signal(SIGINT, cleanup);
    …
    raise(SIGINT);
    return 0;  /* Never get here. */
}
```

Terminologies

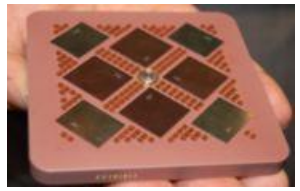
- Signal generated
- Signal Delivered
- Lifetime
- Pending
- Signal catched
- Signal ignored
- Signal blocked

Symmetric MultiProcessor

- **SMP – computer architecture where two or more identical processors can connect to a single shared memory.**



Shared Memory Machine



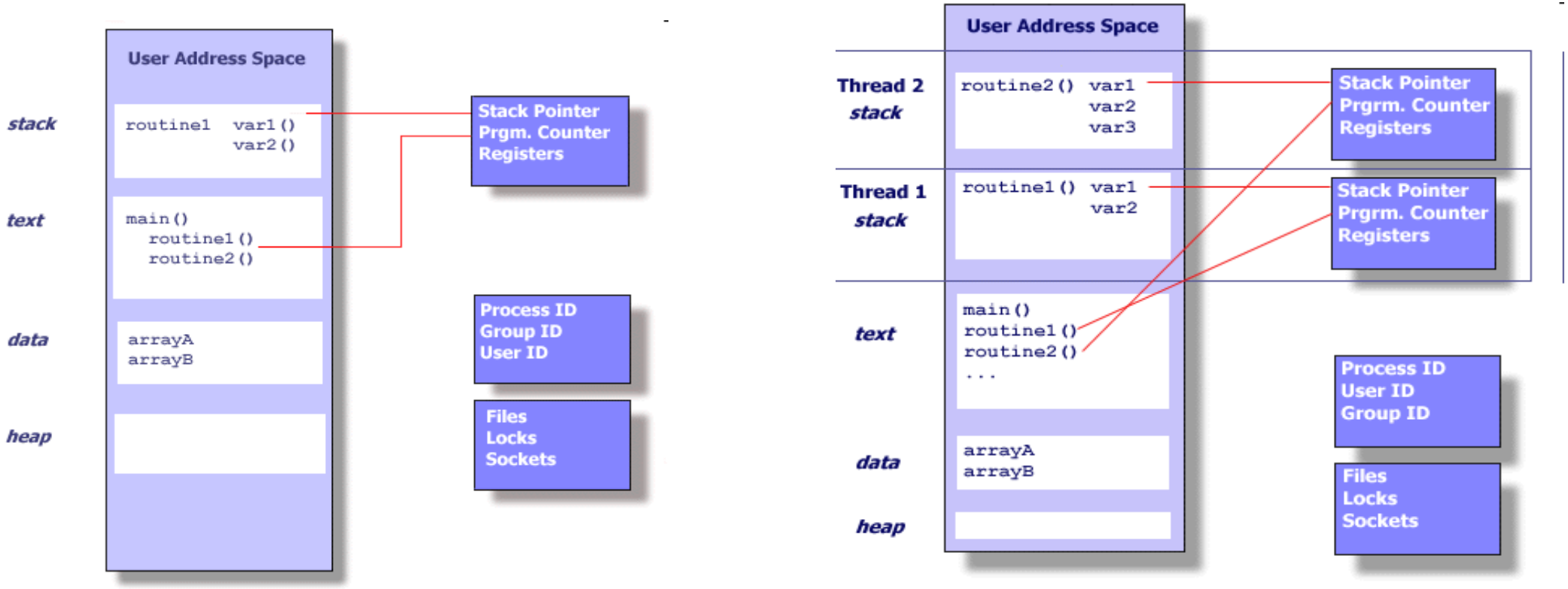
- **What is process?**
  - Program in execution

- **What is thread?**
  - Is an independent /different stream of control that can execute its instructions independently and can use the process resources

- **What is the connection b/w process and thread ?**

Thread

- **Imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to run simultaneously and/or independently. That would describe a "multi-threaded" program**



Thread Features

- **So, in summary, in the UNIX environment a thread:**
  - Exists within a process and uses the process resources
  - Has its own independent flow of control as long as its parent process exists and the OS supports it
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads that act equally independently (and dependently)
  - Dies if the parent process dies
  - Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

- **As threads within the same process share resources:**
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

	Process	Threads
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
4	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
5	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Value of Using Threads

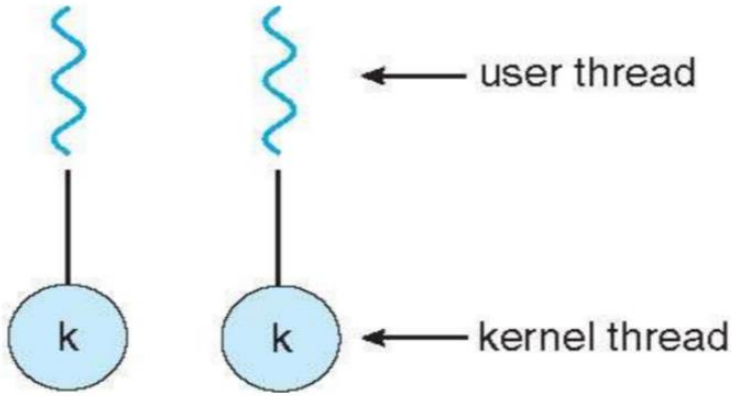
- Performance gains from multiprocessing hardware (parallelism)
- Increased application throughput
- Increased application responsiveness
- Replacing process-to-process communications
- Efficient use of system resources
- Simplified signal handling
- The ability to make use of the inherent concurrency of distributed objects

Types of Threads

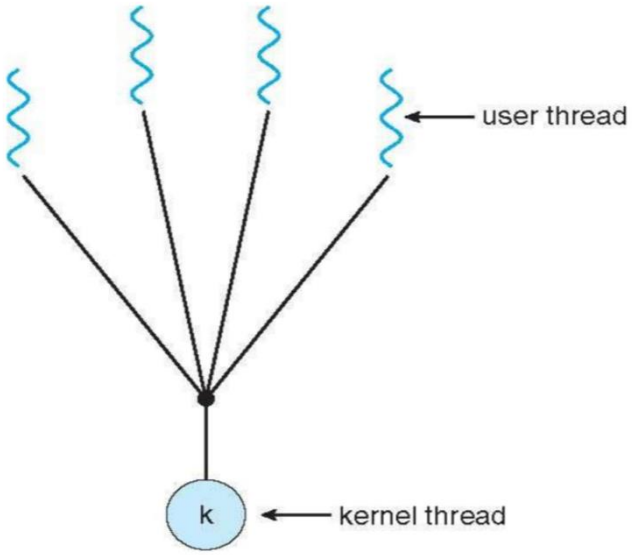
	User Level Thread	Kernel Level Threads
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.

Multithreading Models

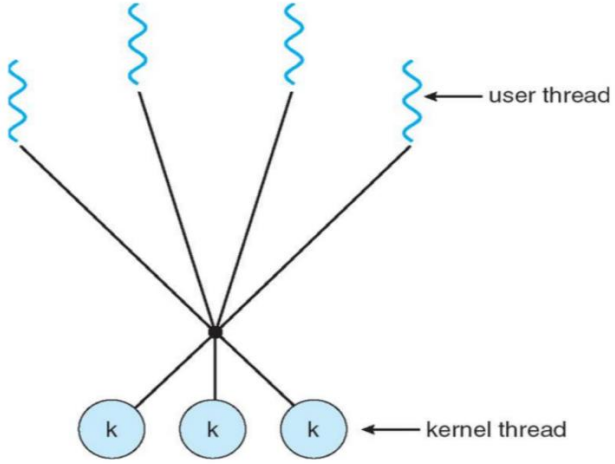
- **One to One**
  - Windows NT, Linux



- **Many to one**
  - Solaris Green Threads and GNU portable threads



- **Many to Many**
  - Solaris prior to ver 9
  - Windows 2000



Posix Threads (pthreads)

- specified by the IEEE POSIX 1003.1c standard (1995).
- set of C programming types & procedure calls, implemented with a pthread.h header file and a thread library.

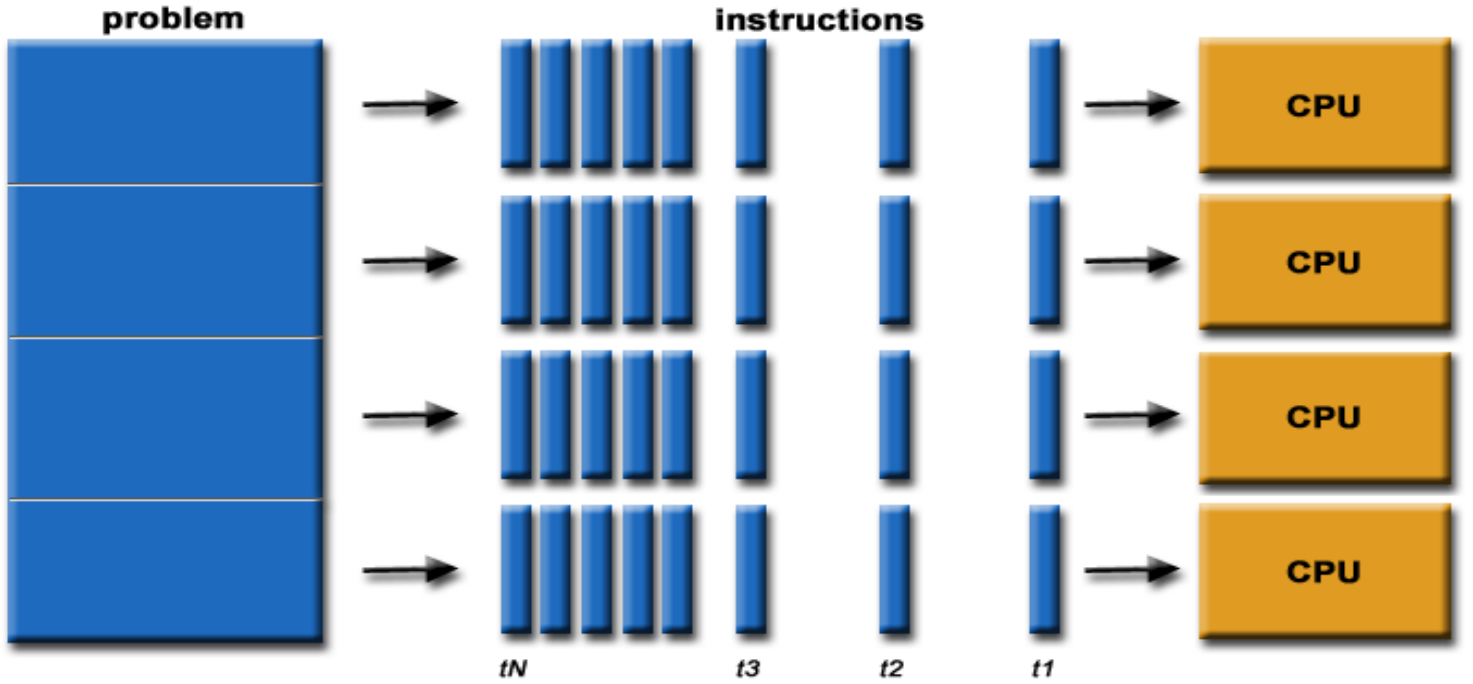
- **Why Pthreads.**
  - 5000

threads/process.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

When is threading Useful

- Independent Tasks
- Servers
- Repetitive tasks
- Asynchronous events



- **Considerations For Thread Programming**
  - Problem partitioning and complexity
  - Load balancing
  - Data dependencies
  - Synchronization and race conditions
  - Data communications
  - Memory, I/O issues

Naming convention

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

Compilation

Compiler / Platform	Compiler Command	Description
IBM AIX	`xlc_r / cc_r`	C (ANSI / non-ANSI)
	`xlC_r`	C++
	`xlf_r -qnosave` `xlf90_r -qnosave`	Fortran - using IBM's Pthreads API (non-portable)
INTEL Linux	`icc -pthread`	C
	`icpc -pthread`	C++
PathScale Linux	`pathcc -pthread`	C
	`pathCC -pthread`	C++
PGI Linux	`pgcc -lpthread`	C
	`pgCC -lpthread`	C++
GNU Linux, AIX	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++

Concept

- Concept of opaque objects pervades the design of API.
- Pthreads has over 100 subroutines
- For portability, pthread.h header file should be used for accessing pthread library.
- POSIX standard defined only for C language
- Once threads are created they are peers and may create other threads.
- Maximum number of threads created is implementation dependent.

1. Thread Management

- pthread_create (thread,attr,start_routine,arg)
- pthread_exit (status)
- pthread_attr_init (attr)
- pthread_attr_destroy (attr)
- pthread_join (threadid,status)
- pthread_detach (threadid,status)

Pthread_create

- **pthread_create (thread, attr, start_routine, arg)**
  - creates a new thread and makes it executable.

- **thread: An unique identifier for the new thread returned by the subroutine.**
- **attr: An attribute object that may be used to set thread attributes. NULL for the default values.**
- **start_routine: the C routine that the thread will execute once it is created.**
- **arg: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.**

Termination

- Thread returns from main routine.
- Thread calls pthread_exit (status) . This is used to explicitly exit a thread
- the pthread_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
- Thread is cancelled by other thread – pthread_cancel()
- Entire process is terminated.

Example Code - Pthread Creation and Termination

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Example 2 - Thread Argument Passing

This example shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

- pthread_attr_getstacksize (attr, stacksize)
- pthread_attr_setstacksize (attr, stacksize)
- pthread_attr_getstackaddr (attr, stackaddr)
- pthread_attr_setstackaddr (attr, stackaddr)

Mutex

- Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- Mutexes can be used to prevent "race" conditions.

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Sequence

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

Mutex Routines

- pthread_mutex_init (mutex,attr)
- pthread_mutex_destroy (mutex)
- pthread_mutexattr_init (attr)
- pthread_mutexattr_destroy (attr)
- pthread_mutex_lock (mutex)
- pthread_mutex_trylock (mutex)
- pthread_mutex_unlock (mutex)

Condition Variable

- Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock.

Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

Main Thread

Join / Continue

- pthread_cond_init (condition,attr)
- pthread_cond_destroy (condition)
- pthread_condattr_init (attr)
- pthread_condattr_destroy (attr)
- pthread_cond_wait (condition,mutex)
- pthread_cond_signal (condition)
- pthread_cond_broadcast (condition)

- **Processes within a system may be independent or cooperating**
  - **Independent –** It cannot affect or be effected by the other processes executing in the system – Does not share any data with any other processes
  - **Cooperating -** It can affect or be effected by the other processes executing in the system.
- **Reasons for cooperating processes:**
  - Information sharing -- Computation speedup
  - Modularity --Convenience
- **Cooperating processes need inter process communication (IPC)**

Types

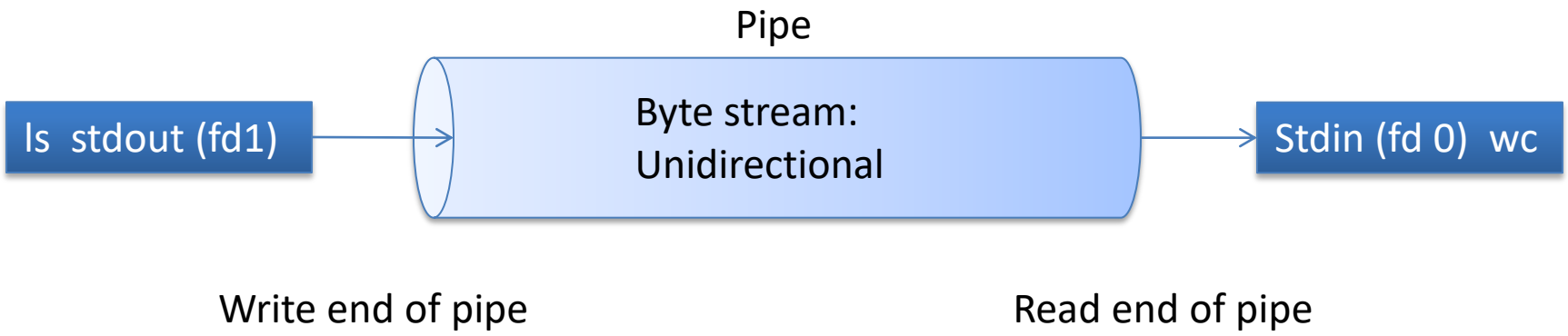
- **Pipe**
  - A pipe is a data channel that is unidirectional
- **File**
  - A file is a data record that may be stored on a disk or acquired on demand by a file server.
- **Signal**
  - signals are not used to transfer data but are used for remote commands between processes
- **Shared Memory**
  - memory that can be simultaneously accessed by multiple processes
- **Message Queue**
  - Multiple processes can read and write data to the message queue without being connected to each other
- **Socket**
  - the endpoint for sending or receiving data in a network

Pipes

- Method that contains two end points. Data is entered from one end of the pipe by a process and consumed from the other end by the other process
- **Pipe - byte stream buffer in kernel**
  - Sequential (can't *lseek()*)
  - Multiple readers/writers difficult
- **Unidirectional**
  - Write end + read end
- **2 Types**
  - Ordinary Pipes
  - Named Pipes

Pipes

ls | wc -l



Piping is a process where the *output* of one process is made the *input* of another.

Functioning of a Pipe

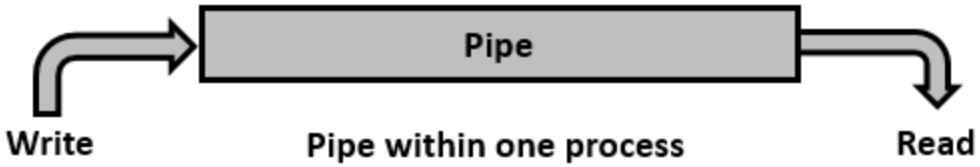
Step 1 – Create a pipe.

Step 2 – Send a message to the pipe.

Step 3 – Retrieve the message from the pipe and write it to the standard output.

1. Create Pipe

```
#include<unistd.h>

int pipe(int pipedes[2]);
```



- This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function

2. Send & receive message

Send message

```
#include<unistd.h>

ssize_t write(int fd, void *buf, size_t count)
```

Receive message

```
#include<unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

Program to write and read 2 messages using pipe

Algorithm

Step 1 – Create a pipe.

Step 2 – Send a message to the pipe.

Step 3 – Retrieve the message from the pipe and write it to the standard output.

Step 4 – Send another message to the pipe.

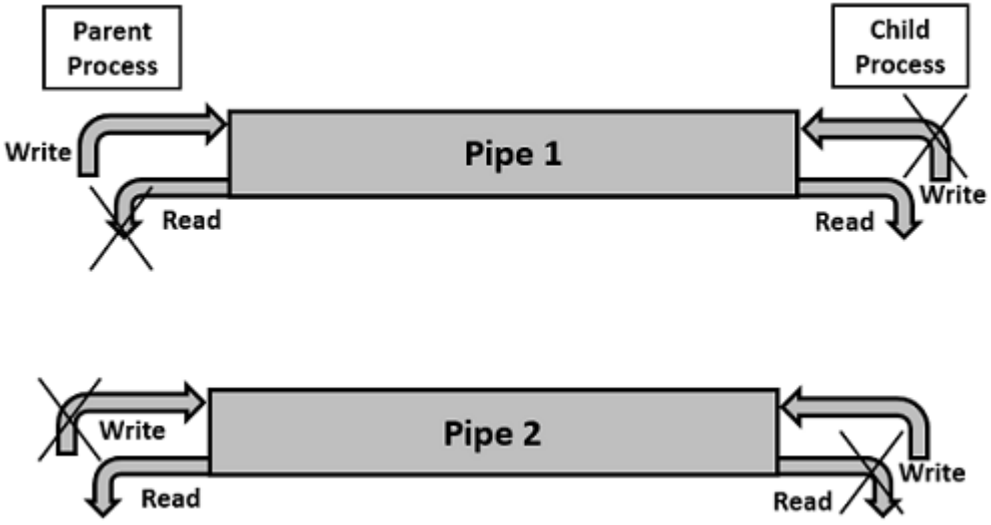
Step 5 – Retrieve the message from the pipe and write it to the standard output.

Note – Retrieving messages can also be done after sending all messages.

```
main() {
   int fds[2];
   int status;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   status = pipe(fds);

    write(fds[1], writemessages[0], sizeof(writemessages[0]));
    read(fds[0], readmessage, sizeof(readmessage));
    printf("Reading from pipe . Message 1 is %s\n", readmessage);

    write(fds[1], writemessages[1], sizeof(writemessages[1]));
    read(fds[0], readmessage, sizeof(readmessage));
    printf("Reading from pipe . Message 2 is %s\n", readmessage);
}
```

Program: Two-way Communication Using Pipes

- Pipe communication is only **one-way** communication i.e., either the parent process writes and the child process reads or vice-versa but not both.
- what if both the parent and the child needs to write and read from the pipes simultaneously.
- **Two pipes** are required to establish two-way communication.



Step 1 – Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

Step 2 – Create a child process.

Step 3 – Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step 4 – Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step 5 – Perform the communication as required.

FIFO – Named Pipes

- Pipes were meant for communication between **related processes**.
- Can we use pipes for unrelated process communication, say, we want to execute client program from one terminal and the server program from another terminal?
  - Can be achieved with Named Pipes also called FIFO
- We used one pipe for one-way communication and two pipes for bi-directional communication. Does the same condition apply for Named Pipes?
  - Named Pipe supports bi-directional communication

```
#include <sys/types.h>
#include <sys/stat.h>
 int mkfifo(const char *pathname, mode_t mode)
```

Mode:

S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH |S_IFIFO

Reverse String

Step 1 – Create two processes, one is fifo2way_server and another one is fifo2way_client.

Step 2 – Server process performs the following –

- Creates a named pipe (using library function **mkfifo()**) with name in /tmp directory
- Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.
- Opens the named pipe for read and write purposes.
- Waits infinitely for a message from the client.
- If the message received from the client , prints the message and reverses the string. The reversed string is sent back to the client.

Step 3 – Client process performs the following –

- Opens the named pipe for read and write purposes.
- Accepts string from the user.
- Sends message to the server.
- It waits for the message (reversed string) from the server and prints the reversed string on receiving.

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#define FIFO_FILE
"/tmp/fifo_twoway"
void reverse_string(char *str) {
  int last, limit, first;
  char temp;
  last = strlen(str) - 1;
  limit = last/2;
  first = 0;
  while (first < last) {
    temp = str[first];
    str[first] = str[last];
    str[last] = temp;
    first++;
    last--;
  } return; }
```

```
int main() {
  int fd, read_bytes;
  char readbuf[80];
/* Create the FIFO if it does not exist */
  mkfifo(FIFO_FILE, S_IFIFO|0640);

  fd = open(FIFO_FILE, O_RDWR);
read_bytes = read(fd, readbuf,
sizeof(readbuf));
    readbuf[read_bytes] = '\0';

 printf("FIFOSERVER: Received string: \"%s\"
and length is %d\n", readbuf,
(int)strlen(readbuf));

reverse_string(readbuf);
    printf("FIFOSERVER: Sending Reversed
String: \"%s\" and length is %d\n", readbuf,
(int) strlen(readbuf));
    write(fd, readbuf, strlen(readbuf));
sleep(2);
return 0;
}
```

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "/tmp/fifo_twoway"
int main() {
  int fd;
  int stringlen;
  int read_bytes;
  char readbuf[80];

  printf("FIFO_CLIENT: Send messages,
infinitely, to end enter \"end\"\n");

  fd = open(FIFO_FILE, O_CREAT|O_RDWR)

printf("Enter string: ");
    fgets(readbuf, sizeof(readbuf), stdin);
    stringlen = strlen(readbuf);
    readbuf[stringlen - 1] = '\0';
write(fd, readbuf, strlen(readbuf));
      printf("FIFOCLIENT: Sent string: \"%s\"
and string length is %d\n", readbuf,
(int)strlen(readbuf));
      read_bytes = read(fd, readbuf,
sizeof(readbuf));
      readbuf[read_bytes] = '\0';
      printf("FIFOCLIENT: Received string:
\"%s\" and length is %d\n", readbuf,
(int)strlen(readbuf));

close(fd);
return 0;
}
```

PIPEs Vs Named PIPEs

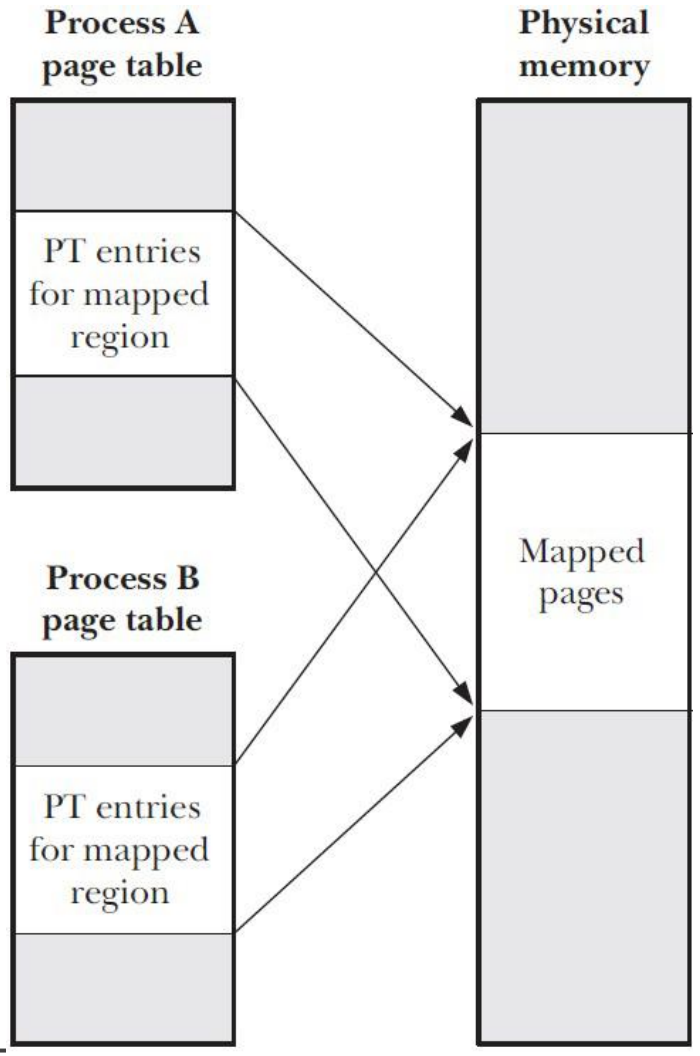
- **PIPE**
  - Between the processes which **share the same file descriptor table** (normally the parent and child processes or threads created by them)
- **Named PIPE**
  - Don't have to start the reading/writing processes at the same time
  - Can control ownership and permissions
  - Across different systems – If a common file system available

Shared Memory

- **The shared memory is a mechanism that allows processes to exchange data.**
  - a portion of the memory that is shared between processes.
- **How it works**
  - One process creates the memory area
  - Other process, which has appropriate permissions accesses the memory.
- **Shared among how many process**
  - Shared memory allows several processes to attach a segment of physical memory to their virtual addresses.

Shared memory

- **Processes share physical pages of memory**



Shared memory

- **Processes share same physical pages of memory**
- **Communication means copy data to memory**
- **Efficient compared to Data transfer**
  - Data transfer: user space -> kernel -> user space
- **Shared memory: single copy in user space**
  - Access need to be sync!

Steps to be followed

- Create the shared memory segment or use an already created shared memory segment (**shmget()**)
- Attach the process to the already created shared memory segment (**shmat()**)
- Detach the process from the already attached shared memory segment (**shmdt()**)
- Control operations on the shared memory segment (**shmctl()**)

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg)
```

- Key – Key for the shared memory location for sharing
- Size – size of the memory location requested
- Shmflg – Permissions & creation flag (IPC_CREAT,IPC_EXCL)

```
#include <sys/types.h>
#include <sys/shm.h>
void * shmat(int shmid, const void *shmaddr, int shmflg)
```

attaches the System V shared memory segment identified by shmid to the address space of the calling process

Shmid – shared memory id

Shmaddr- The attaching address , always NULL

Shmflg- Setting permissions

- **Detach the shared memory**

```
#include <sys/types.h>
#include <sys/shm.h>
 int shmdt(const void *shmaddr)
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
 int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

- **Control & deallocation**
- **Cmd option**

✓ **IPC_SET** – Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.

✓ **IPC_RMID** – Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.

✓ **IPC_INFO** – Returns the information about the shared memory limits and parameters in the structure pointed by buf.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
        int i;
        void *shared_memory;
        char buff[100];
        int shmid;

        shmid=shmget((key_t)2345, 1024, 0666);
        printf("Key of shared memory is %d\n",shmid);

        //process attached to shared memory segment
        shared_memory=shmat(shmid,NULL,0);
        printf("Process attached at %p\n",shared_memory);
        printf("Data read from shared memory is : %s\n",(char *)shared_memory);

}
```

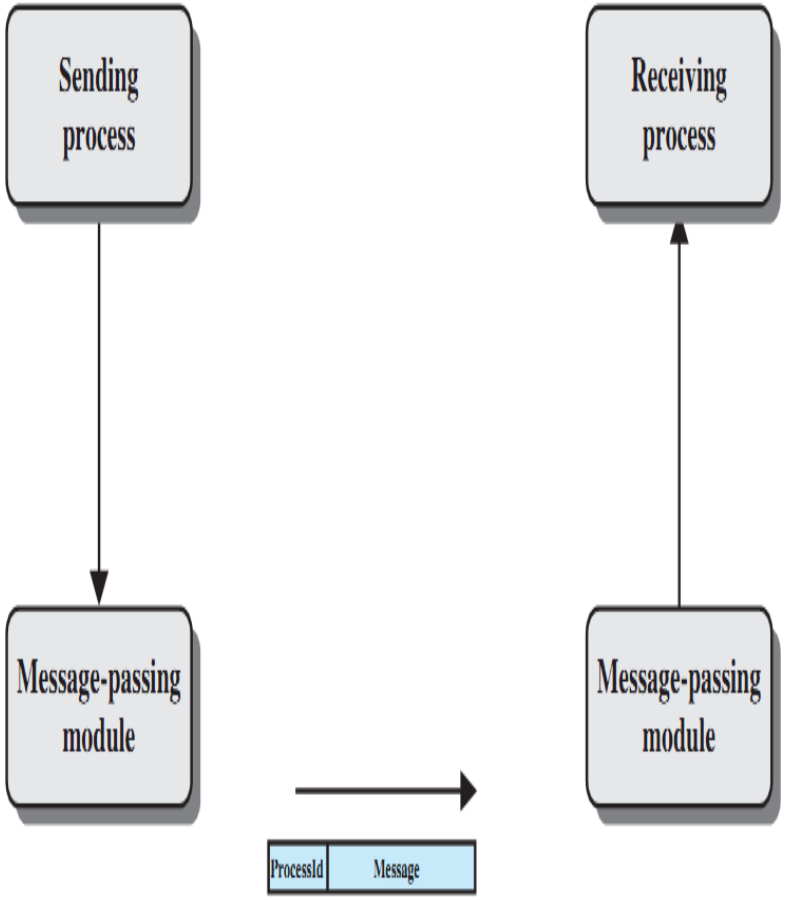
```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()  {
int i , shmid ;
void *shared_memory;
char buff[100];

shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
printf("Key of shared memory is %d\n",shmid);

shared_memory=shmat(shmid,NULL,0); //process attached to  segment
printf("Process attached at %p\n",shared_memory);

printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

Message Queue

- **Type of Message passing mechanism**
  - processes communicate with each other without resorting to shared variables.
- **It is useful in distributed environments where the processes may reside on different computers connected by network**
- **We have at least two primitives:**
  - **send**(*destination, message*) or **send**(*message*)
  - **receive**(*source, message*) or **receive**(*message*)
- **Message sent can be of fixed or variable size**

- **For 2 processes to communicate a communication link must exist**
- **Link implemented in one of three ways:**
  - Zero capacity – 0 messages. Sender must wait for receiver
  - Bounded capacity – finite length of *n* messages. Sender must wait if link full.
  - Unbounded capacity – infinite length. Sender never waits.

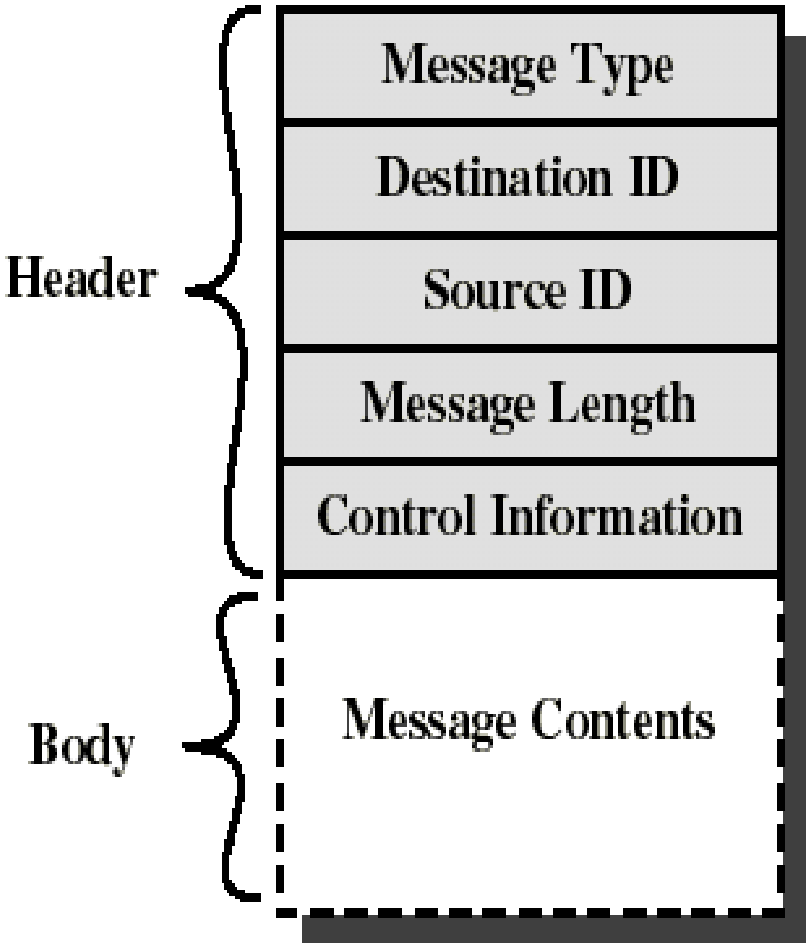
Message Queue



- In a message queue, the sending process calls a system routine for placing the message in a queue that can be read by another process.
- Each process is provided identification or a type, which allows other processes to identify and select it.
- A common key is shared between processes accessing the queue.

Message Format

- **Consists of header and body of message.**
- **Control info:**
  - what to do if run out of buffer space.
  - sequence numbers.
  - priority.
- **Queuing discipline: usually FIFO but can also include priorities.**



- To perform communication using message queues, following are the steps –

Step 1 – Create a message queue or connect to an already existing message queue (**msgget()**)

Step 2 – Write into message queue (**msgsnd()**)

Step 3 – Read from the message queue (**msgrcv()**)

Step 4 – Perform control operations on the message queue (**msgctl()**)

System calls for message queues:

- **msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
  - **int msgget(key_t key, int msgflg);**
- **ftok()**: is use to generate a unique key.
- **msgsnd()**: Data is placed on to a message queue by calling msgsnd().
  - **int msgsnd(int msgqid, const void *msgptr, size_t msgsize, int msgflag);**
- **msgrcv()**: messages are retrieved from a queue.
  - **int msgrcv(int msgqid, void *msgptr, size_t msgsize, long msgtype,int msgflag);**
- **msgctl()**: It performs various operations on a queue. Generally it is use to destroy message queue.
  - **int msgctl(int msqid, int cmd, struct msqid_ds *buf)**

Sending Message

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXSIZE     128
struct msgbuf
{
    long    mtype;
    char    mtext[MAXSIZE];
};
main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    struct msgbuf sbuf;
    size_t buflen;
    key = 1234;
```

```
if ((msqid = msgget(key, msgflg )) < 0){   //Get the message
queue ID for the given key
      perror("msgget");
        exit(1);
        }
    //Message Type
    sbuf.mtype = 1;
    printf("Enter a message to add to message queue : ");
    scanf("%[^\n]",sbuf.mtext);
    getchar();
    buflen = strlen(sbuf.mtext) + 1 ;
    if (msgsnd(msqid, &sbuf, buflen, IPC_NOWAIT) < 0)
    {
        printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype,
sbuf.mtext, buflen);
        perror("msgsnd");
        exit(1);
    }
    else
        printf("Message Sent\n");
    exit(0);
}
```

Receiving Message

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE     128
struct msgbuf
{
    long    mtype;
    char    mtext[MAXSIZE];
};
main()
{
    int msqid;
    key_t key;
    struct msgbuf rcvbuffer;
    key = 1234;
```

```
 if ((msqid = msgget(key, 0666)) < 0)
        {
      perror("msgget()");
        exit(1);
        }
     //Receive an answer of message type 1.
    if (msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0) < 0)
        {
      perror("msgrcv");
        exit(1);
        }
    printf("%s\n", rcvbuffer.mtext);
    exit(0);
}
```

Message Queue	Shared Memory
message is received by a process it would be no longer available for any other process	Message is available to access multiple times until the memory is updated or destroyed
communicate with small message formats	Can create based on the memory required
Messages are sent to destination process on receiving the message gets removed	Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.
Best when small messages needs to be exchanged	Can be used when frequency of writing and reading is high