Linux Internals & Networking
System programming using Kernel interfaces

Team Emertxe

Contents

Linux Internals & Networking
Contents

- Introduction

- Transition to OS programmer

- System Calls

- Process

- IPC

- Signals

- Networking

- Threads

- Synchronization

- Process Management
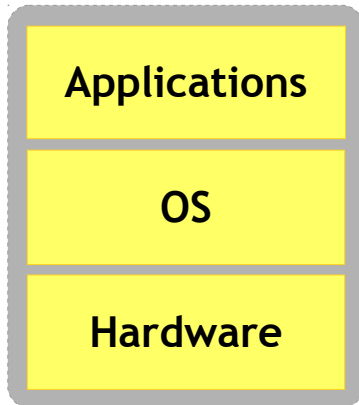
- Memory Management

ΣMERTXE

Introduction

Introduction

Let us ponder …

- What exactly is an Operating System (OS)?

- Why do we need OS?

- How would the OS would look like?

- Is it possible for a team of us (in the room) to create an OS of our own?

- Is it necessary to have an OS running in a Embedded System?

- Will the OS ever stop at all?

ΣMERTXE

What is an OS

Applications
OS
Hardware

OS is an interface between application and hardware
Which abstracts H/W layer from user
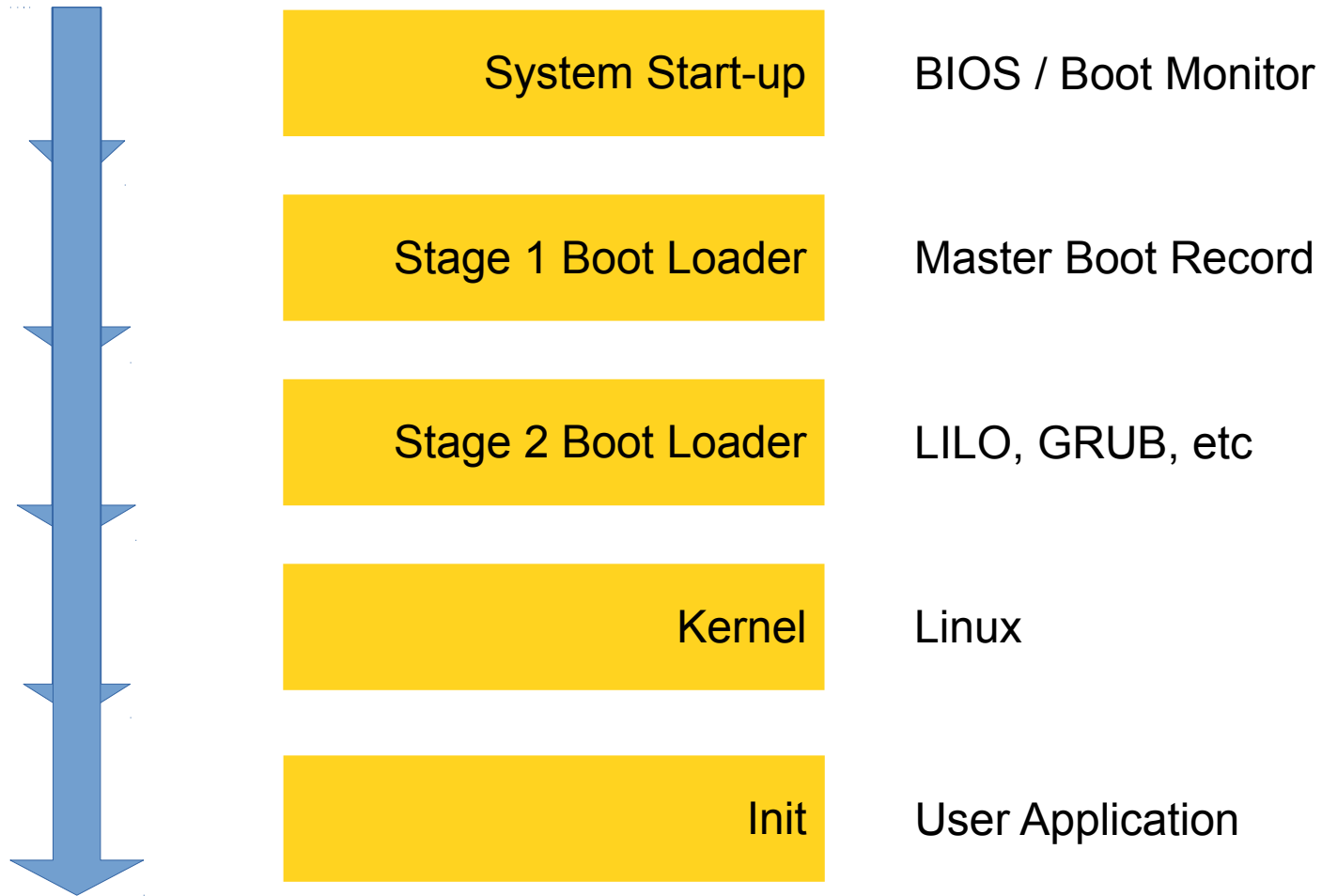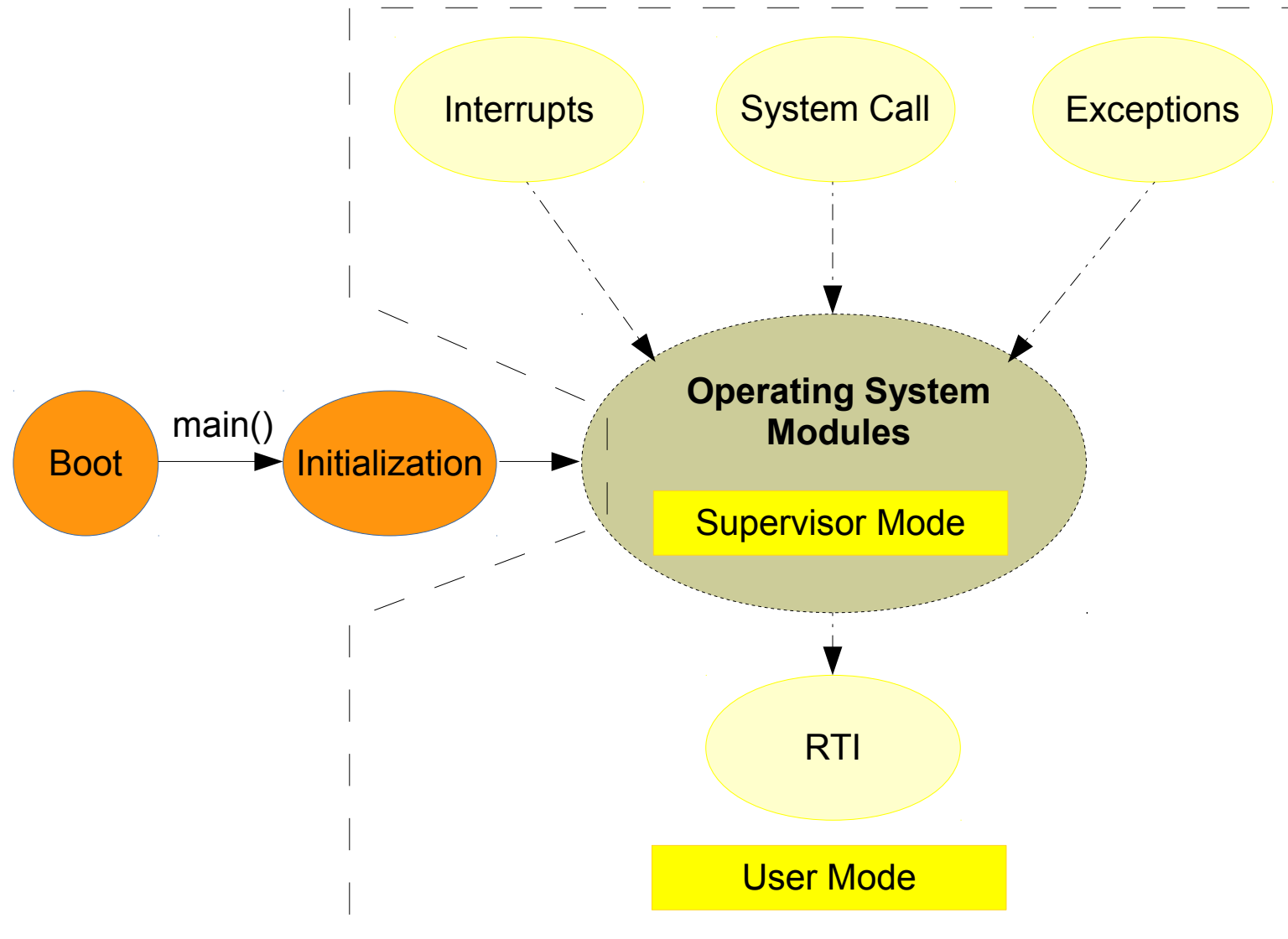
Is it possible to make an embedded without OS.?

Applications
Hardware
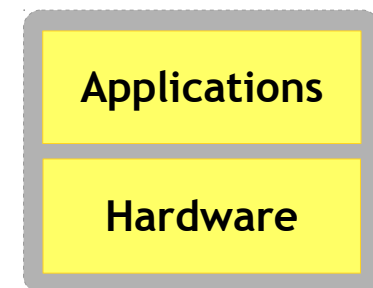
Why we need OS

- Multitasking
- Multi user
- Scheduling
- Memory management
  etc …

Linux Booting Sequence

System Start-up	BIOS / Boot Monitor
Stage 1 Boot Loader	Master Boot Record
Stage 2 Boot Loader	LILO, GRUB, etc
Kernel	Linux
Init	User Application

EMERTXE

Control flow in OS

History

Period	Description
1940 - 1955	ENIAC, Mechanical switches, Mainframe computers
1955 - 1970	Concept OS, FORTRAN, IBM OS/360, Multiprogramming, Minicomputers.

Vertical

Horizontal

Period	Description
1970 - 1980	UNIX, Microprocessors(intel), Personal computers age
1980 - 1990	First computer IBM 5150, DOS, Apple & Windows with GUI
1990 - NOW	Linux, ios, Android.....

EMERTXE

Vertical vs Horizontal

- Vertical
  - Hardware & Software made by same company
  - OS was integrated part of the Hardware
  - Applications were propitiatory

- Horizontal
  - Hardware & Software made by different company
  - OS is an independent software, that can run on diversified set of hardware
  - Applications are developed by everybody (propitiatory or open source)

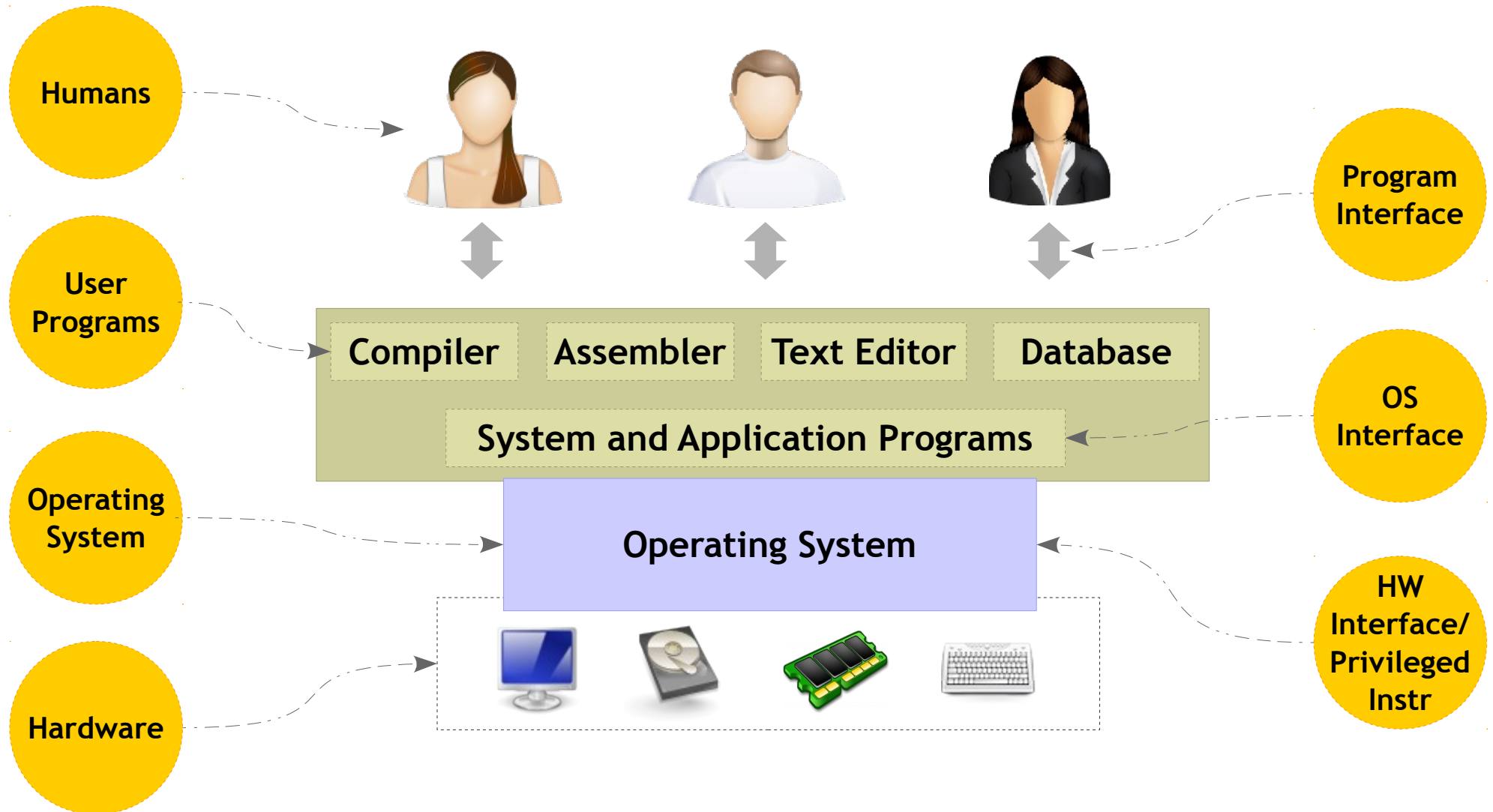ƩMERTXE

Quiz

- How would the OS look like ?

  a) H/W only

  b) S/W only

  c) S/W + H/W

- How big is OS ?

Introduction
Operating System

Humans

User Programs

Operating System

Hardware

Program Interface

OS Interface

HW Interface/ Privileged Instr

Compiler	Assembler	Text Editor	Database

System and Application Programs

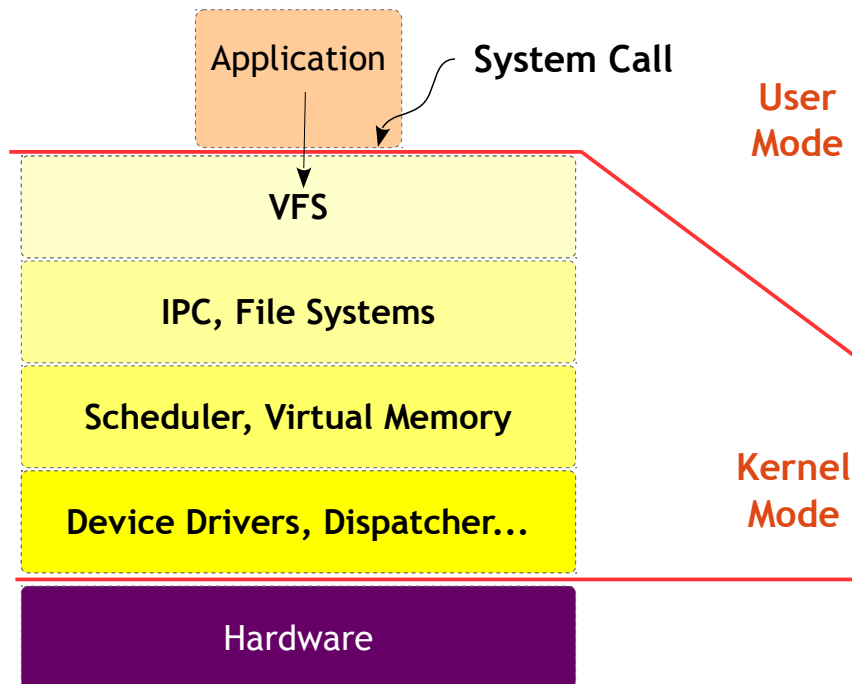**Operating System**

EMERTXE

- Most older operating systems are monolithic, that is, the whole operating system is a single executable file that runs in 'kernel mode'

- This binary contains the process management, memory management, file system and the rest (Ex: UNIX)

- The alternative is a microkernel-based system, in which most of the OS runs as separate processes, mostly outside the kernel

- They communicate by message passing. The kernel's job is to handle the message passing, interrupt handling, low-level process management, and possibly the I/O (Ex: Mach)
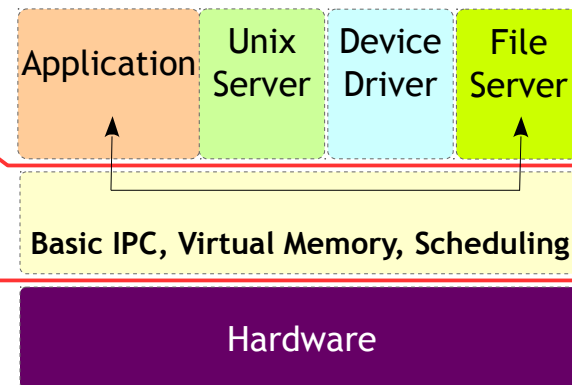
Introduction
Kernel Architecture

Mionolithic Kernel Based Operating System

Micro Kernel Based Operating System

Application

System Call

User Mode

VFS

IPC, File Systems

Scheduler, Virtual Memory

Device Drivers, Dispatcher...

Kernel Mode

Hardware

Application | Unix Server | Device Driver | File Server

Basic IPC, Virtual Memory, Scheduling

Hardware

ΣMERTXE

Introduction
Mono vs Micro

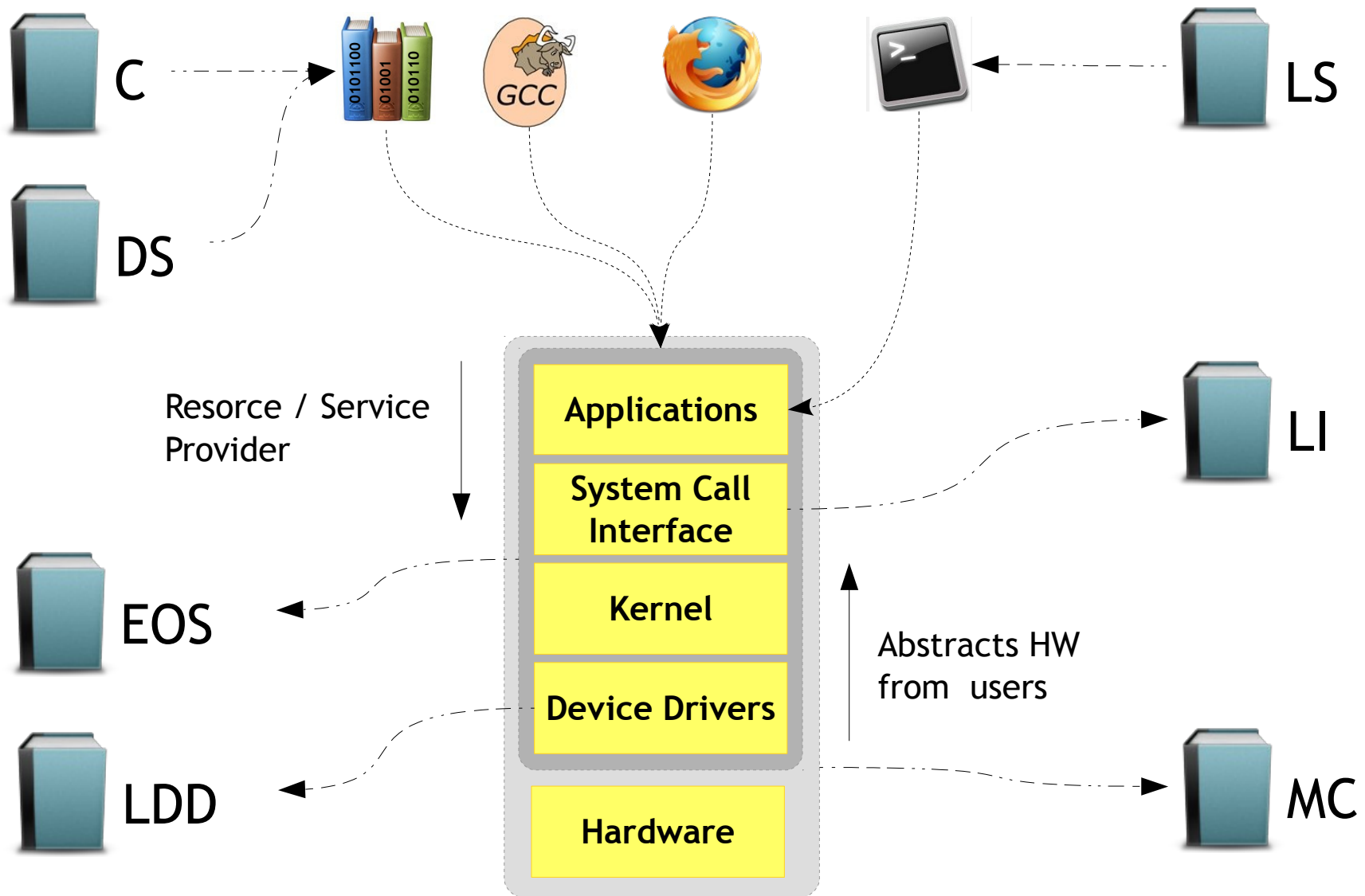| Monolithic kernel	Microlithic kernel
• Kernel size increases because kernel + kernel subsystems compiled as single binary • Difficult to extension or bug fixing, • Need to compile entire source code. • Bad maintainability • Faster, run as single binary • Communication between services is faster. • No crash recovery. • More secure • Eg: Windows, Linux etc	• Kernel size is small because kernel subsystems run as separate binaries. • Easily extensible and bug fixing. • Easily recover from crash • Slower due to complex message passing between services • Process management is complex • Communication is slow • Easily recoverable from crashing • Eg: MacOS, WinNT

RTOS

- Real time means fast..?

- RTOS is an operating system that guarantees a certain capabilities within a specified time constraint.

- RTOS must also must able to respond predictably to unpredictable events

- We use RTOS in Aircraft, Nuclear reactor control systems where time is crucial.

- Eg: LynxOS, OSE, RTLinux, VxWorks, Windows CE

Transition to OS programming

Course & module view

C

DS

Resorce / Service
Provider

EOS

LDD

LS

Applications

**System Call
Interface**

Kernel

Device Drivers

Hardware

LI

Abstracts HW
from users

MC

Application vs OS

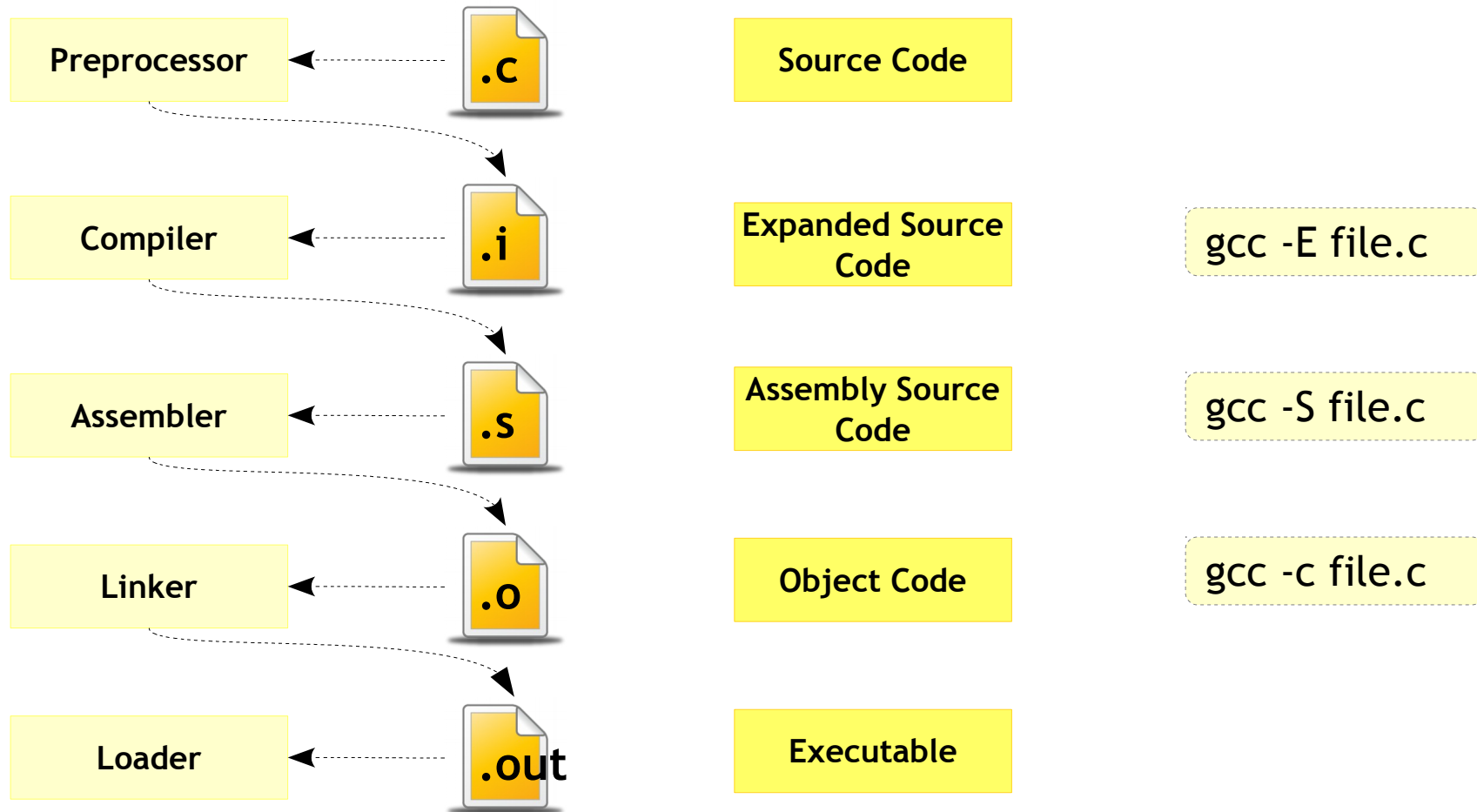| C	Algorithms, Syntax, Logic	• Preprocessor • Compiler • Assembler • Linker • Executable file (a.out)
OS	Memory segments, process, Threads Signals, IPC, Networking	• Executing a program • Loader

Application Programming
Compilation Stages

- **Preprocessor**
  - Expands header files
  - Substitute all macros
  - Remove all comments
  - Expands and all # directives
- **Compilation**
  - Convert to assembly level instructions
- **Assembly**
  - Convert to machine level instructions
  - Commonly called as object files
  - Create logical address
- **Linking**
  - Linking with libraries and other object files

ΣMERTXE

Application Programming
Compilation Stages

Stage	File	Description	Command
Preprocessor	.c	Source Code	
Compiler	.i	Expanded Source Code	gcc -E file.c
Assembler	.s	Assembly Source Code	gcc -S file.c
Linker	.o	Object Code	gcc -c file.c
Loader	.out	Executable	

gcc -save-temps file.c would generate all intermediate files

ΞMERTXE

Application Programming
Linking - Static

- Static linking is the process of copying all library modules used in the program into the final executable image.

- This is performed by the linker and it is done as the last step of the compilation process.

- Its both faster and more portable, since it does not require the presence of the library on the system where it is run.

- Compiling two .o files also a type of static linking.

ΣMERTXE

Application Programming
Linking – Static

- Create two files file1.c & file2.c with appropriate functions in them

- Compile them to create object files ("gcc -c fun1.c fun2.c"). It will create fun1.o and fun2.o files as output

- Create a static library by using ("ar rcs fun1.o fun2.o -o libstatic.a), with this you have completed creating your static library

- Create your main.c file by calling these two functions

- Perform a static linking (gcc main.c libstatic.a)

- This will generate your a.out, execute the program and observe the output

Note: ar - Archive command
       rcs option - r(replace), c(create), s(write an object file into archive)

ΣMERTXE

Application Programming
Linking - Dynamic

- It performs the linking process when programs are executed in the system.

- During dynamic linking the name of the shared library is placed in the final executable file.

- Actual linking takes place at run time when both executable file and library are placed in the memory.

- The main advantage to using dynamically linked libraries is that the size of executable programs is reduced

- To create a dynamic library (shared object file)

ΣMERTXE

Application Programming
Linking - Dynamic

- Create two files file1.c & file2.c with appropraite functions in them

- Create a dynamic library (shared object) ("gcc -fPIC -shared fun1.c fun2.c -o libdynamic.so)

- Set the environmental variable LD_LIBRARY_PATH to the directory where this new *.so is located   (ex: LD_LIBRARY_PATH="/home/LinuxInternals/DynamicLinking")

- Perform echo $LD_LIBRARY_PATH and ensure this variable is set properly

- Execute export LD_LIBRARY_PATH to ensure this gets into the shell's environmental variables ecosystem

- Create your main.c file by calling these two functions

- Perform a dynamic linking ("gcc main.c -L . -ldynamic)

- This will generate your a.out, execute the program and observe the output

Note: fPIC option - Generating Position Independent Code (as it is dynamically linked)

ƩMERTXE

Application Programming
Linking - Static vs Dynamic

Parameter \ Linking	Static	Dynamic
Executable Size	⬆	⬇
Loding Time	⬇	⬆
Memory Usage	⬆	⬇
No of Sys Calls	⬇	⬆

ΣMERTXE

Executing a process

RAM

P₁
P₂
P₃
.
.
.
Pₙ₋₁
Pn

Memory Layout

Command Line Arguments
Stack
Hole
Heap
Data Segment — **.BSS** (Uninitialized)
Initialized
Code Segment

Stack Frame

Local Variables
Return Address
Parameter List

EMERTXE
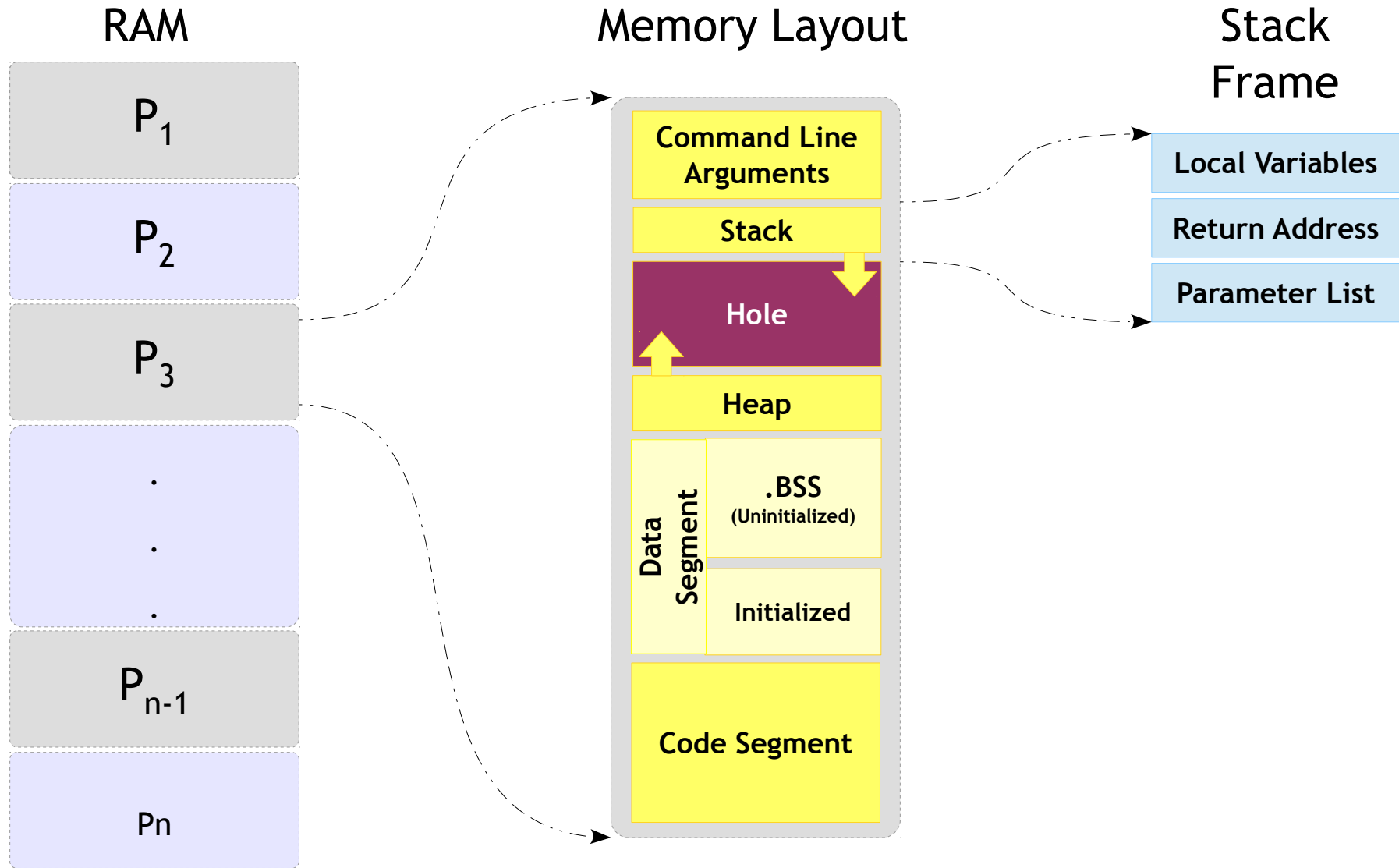
Quiz

- How a user defined function works?

- How a library function works?

Storage Classes

Storage Class	Scope	Lifetime	Memory Allocation
auto	Within the block / Function	Till the end of the block / function	Stack
register	Within the block / Function	Till the end of the block / function	Register
static local	Within the block / Function	Till the end of the program	Data Segment
static global	File	Till the end of the program	Data segment
extern	Program	Till the end of the program	Data segment

Hands-on

- Access a static variable from outside file.

- Access a global variable from outside file.

- Combination of both static and local.

Common errors

with various memory segments

Memory Layout
Command Line Arguments
Stack
Hole
Heap
Data Segment / .BSS (Uninitialized)
Data Segment / Initialized
Text Segment
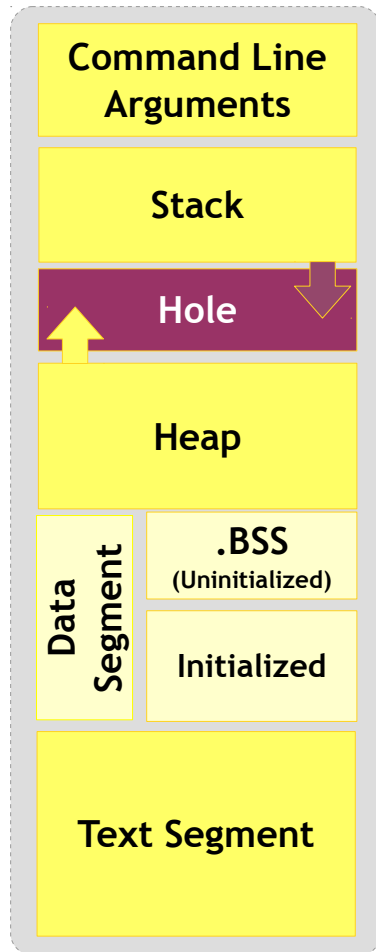
Stack Overflow / Stack Smashing

- When ever process stack limit is over
  Eg: Call a recursive function inifinite times.
- When you trying to access array beyond limits.
  Eg int arr[5]; arr[100];

Memory Leak

- When you never free memory after allocating.
  Eventually process heap memory will run-out

Segmentation Fault

When you try to change text segment, which is a read-only memory or try trying to access a memory beyond process memory limit (like NULL pointer)

Introduction
What is Linux?

- Linux is a free and open source operating system that is causing a revolution in the computer world

- Originally created by Linus Torvalds with the assistance of developers called community

- This operating system in only a few short years is beginning to dominate markets worldwide
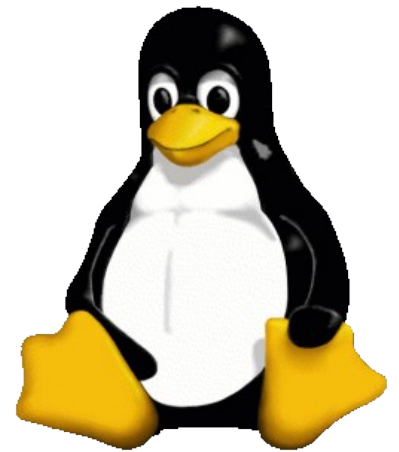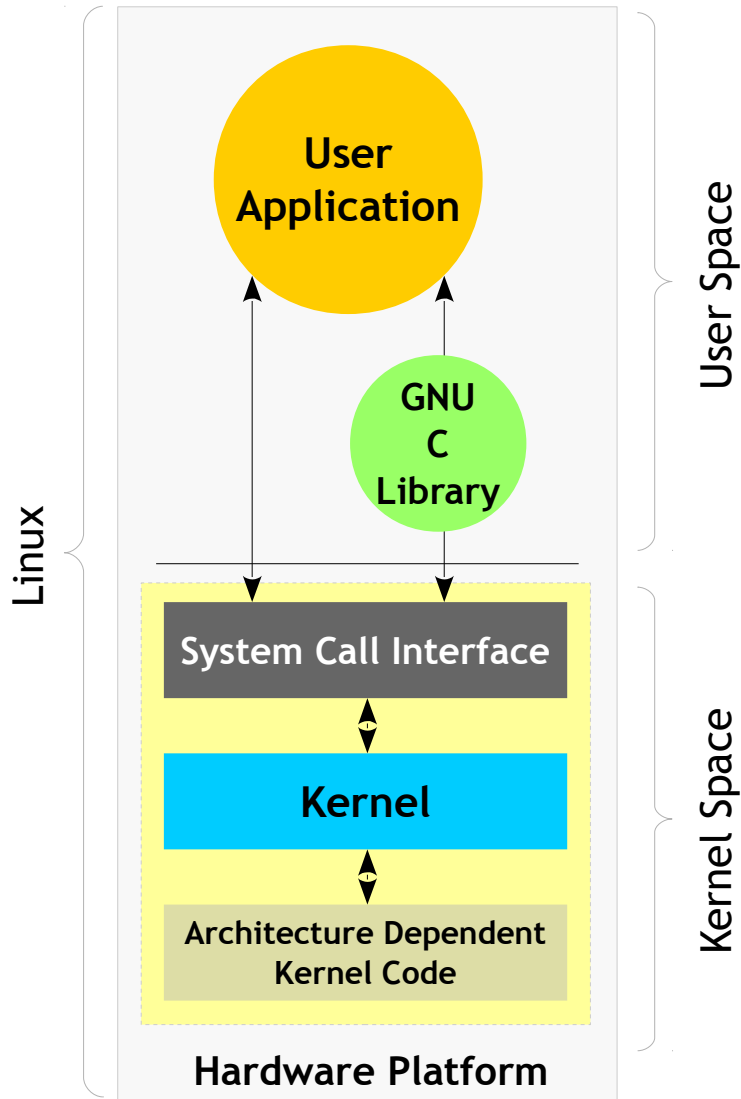
Introduction
Why use Linux?

- Free & Open Source –GPL license, no cost

- Reliability –Build systems with 99.999% upstream

- Secure –Monolithic kernel offering high security

- Scalability –From mobile phone to stock market servers
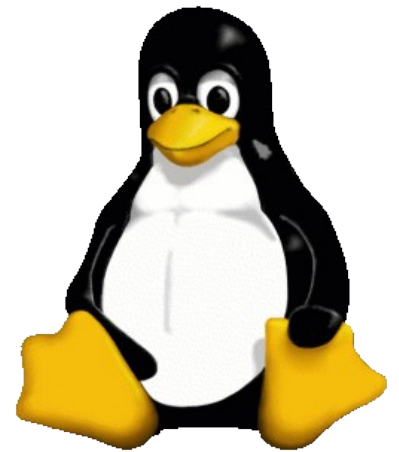
Introduction
Linux Components

- **Hardware Controllers:** This subsystem is comprised of all the possible physical devices in a Linux installation - CPU, memory hardware, hard disks

- **Linux Kernel:** The kernel abstracts and mediates access to the hardware resources, including the CPU. A kernel is the core of the operating system

- **O/S Services:** These are services that are typically considered part of the operating system (e.g. windowing system, command shell)

- **User Applications:** The set of applications in use on a particular Linux system (e.g. web browser)

Introduction
Linux Kernel Subsystem
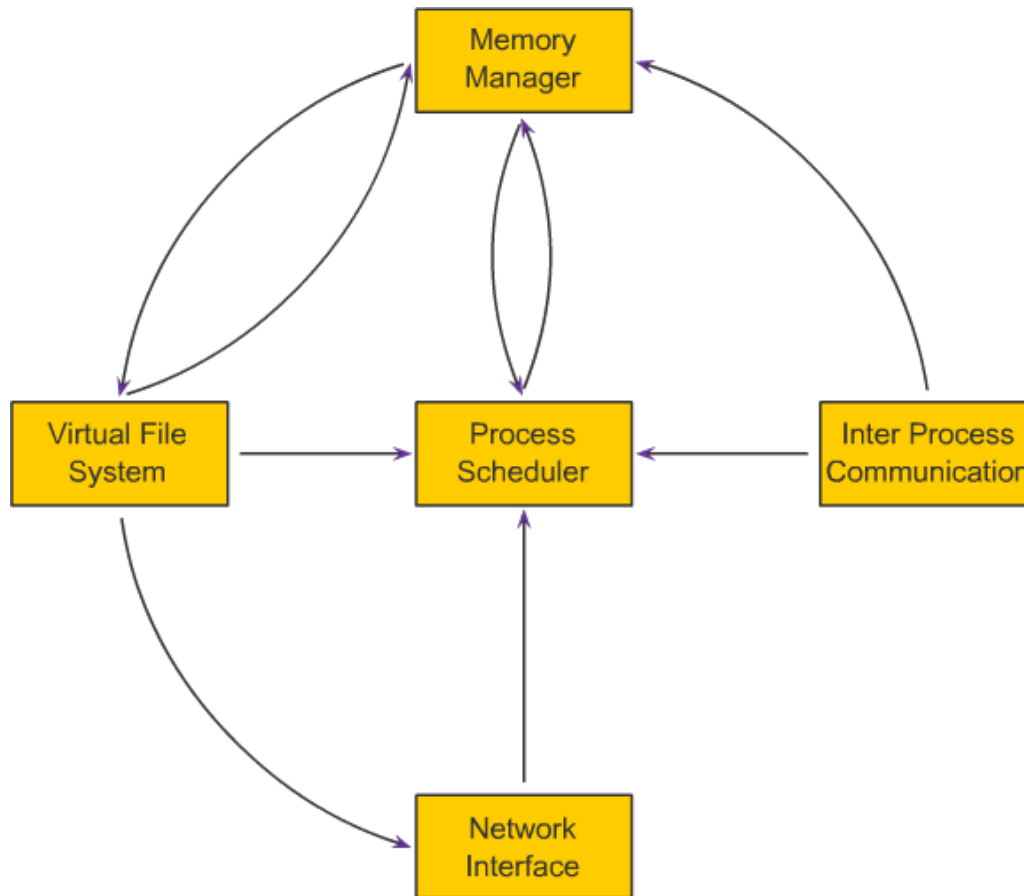
- Process Scheduler (SCHED):
  - To provide control, fair access of CPU to process, while interacting with HW on time

- Memory Manager (MM):
  - To access system memory securely and efficiently by multiple processes. Supports Virtual Memory in case of huge memory requirement

- Virtual File System (VFS):
  - Abstracts the details of the variety of hardware devices by presenting a common file interface to all devices
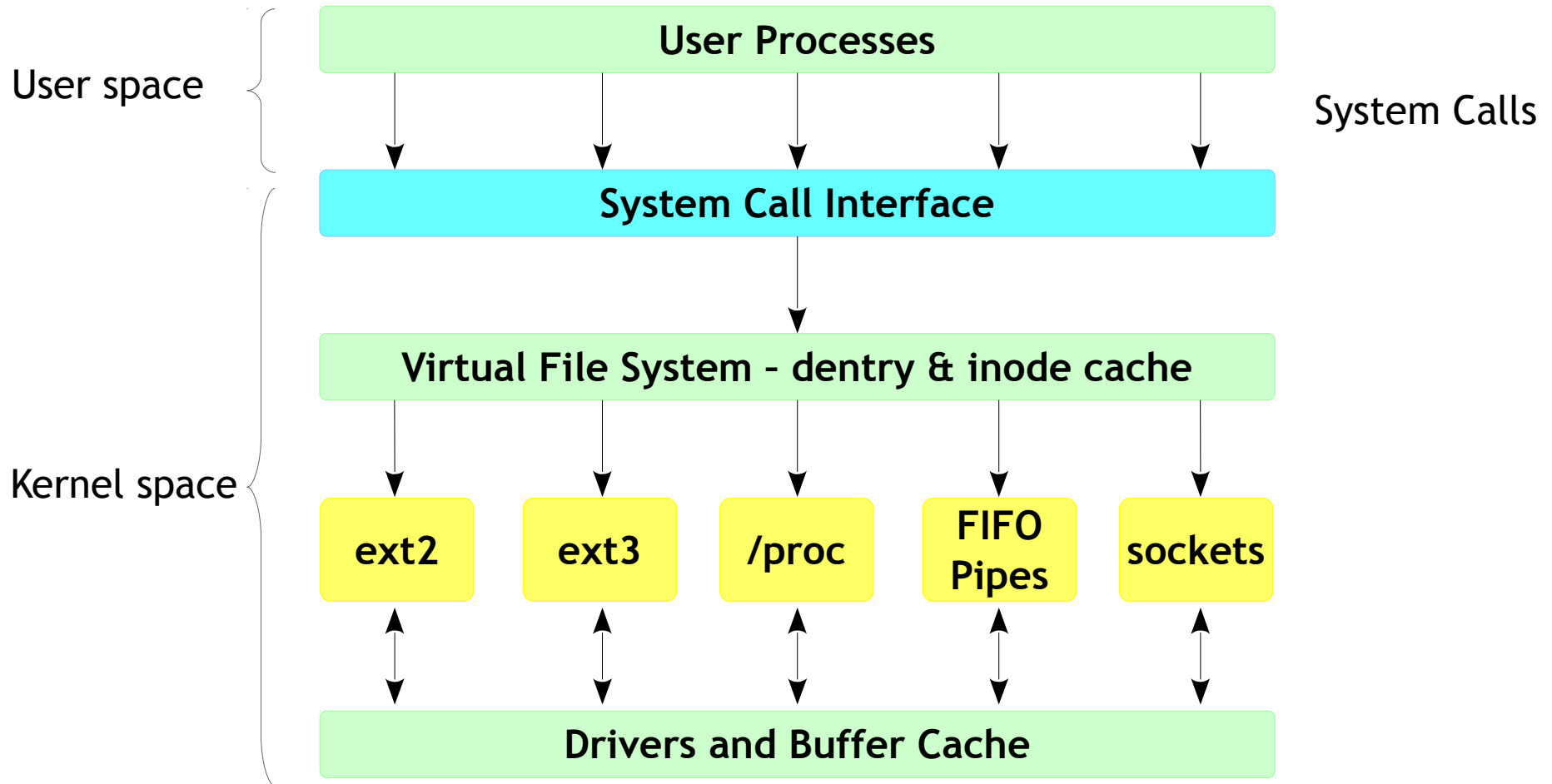
Introduction
Linux Kernel Subsystem

- Network Interface (NET):

  – provides access to several networking standards and a variety of network hardware

- Inter Process Communications (IPC):

  – supports several mechanisms for process-to-process communication on a single Linux system
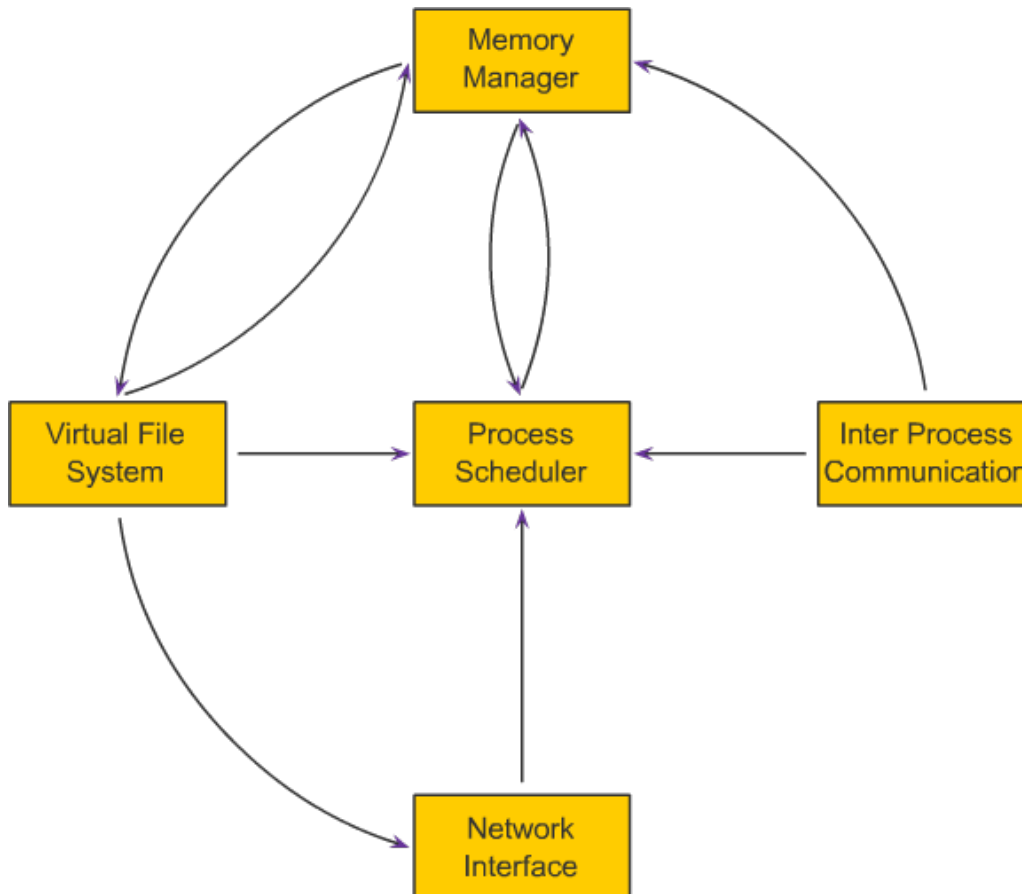
Introduction
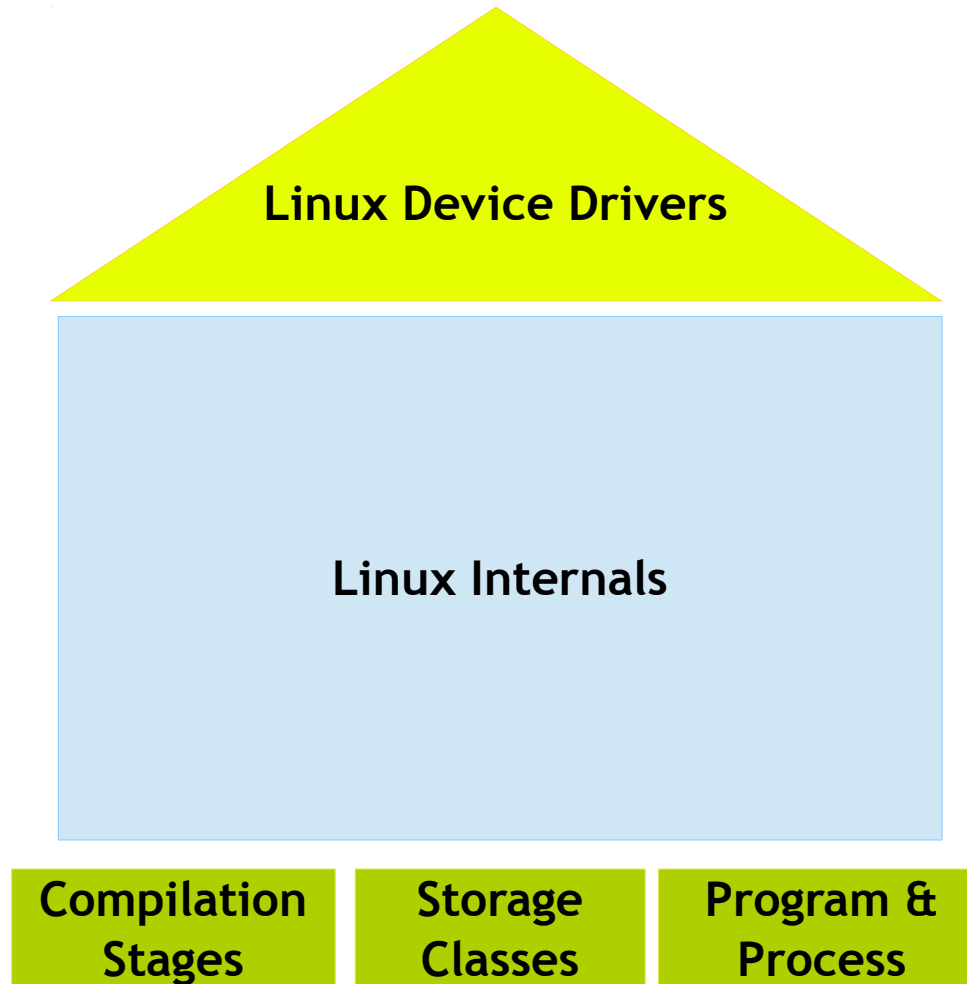## Virtual File System

Introduction
Virtual File System

- Presents the user with a unified interface, via the file-related system calls.

- The VFS interacts with file-systems which interact with the buffer cache, page-cache and block devices.

- Finally, the VFS supplies data structures such as the dcache, inodes cache and open files tables.

  - Allocate a free file descriptor.

  - Try to open the file.

  - On success, put the new 'struct file' in the fd table of the process.On error, free the allocated file descriptor.

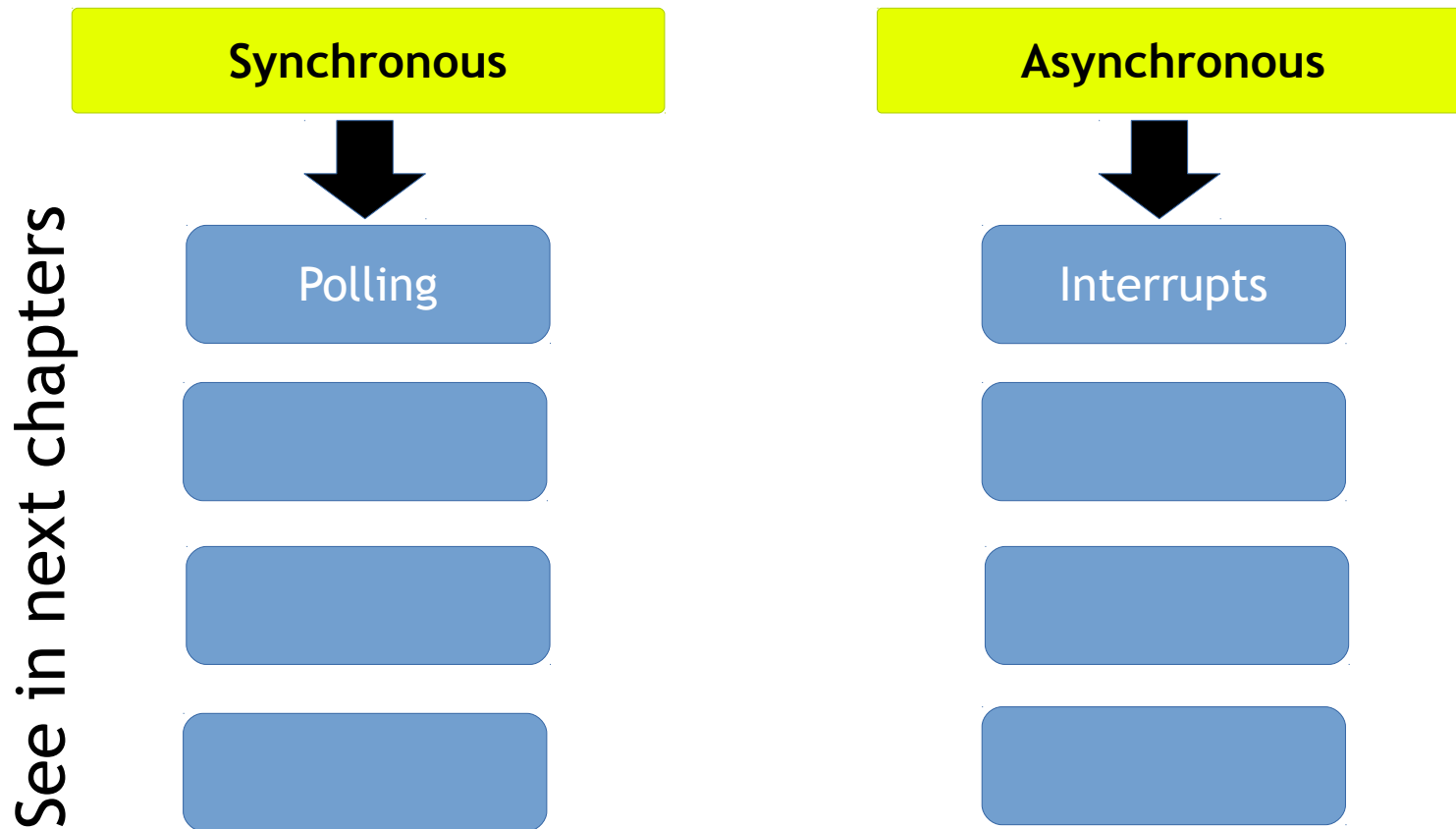**NOTE: VFS makes "Everythig is file" in Linux**

Summary

Linux Device Drivers

Linux Internals

Compilation Stages	Storage Classes	Program & Process

EMERTXE

System Calls

Synchronous & Asynchronous

- Communications are two types

See in next chapters

Synchronous	Asynchronous
Polling	Interrupts

EMERTXE

Interrupts

Interrupts

Hardware	Software
Generates whenever a Hardware changes happens	Generates by instruction from code (eg: INT 0x80)
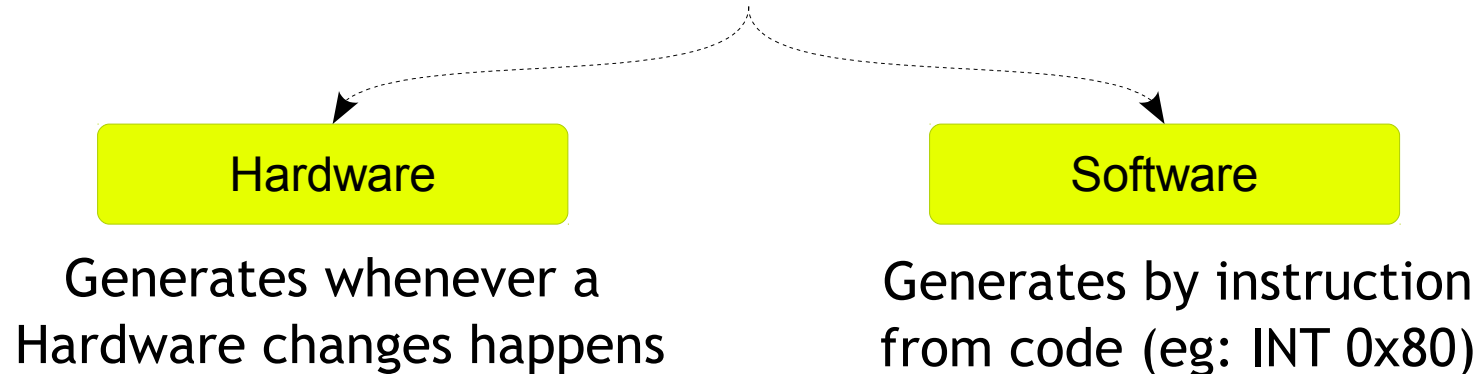
- Interrupt controller signals CPU that interrupt has occurred, passes interrupt number

- Basic program state saved

- Uses interrupt number to determine which handler to start

- CPU jumps to interrupt handler

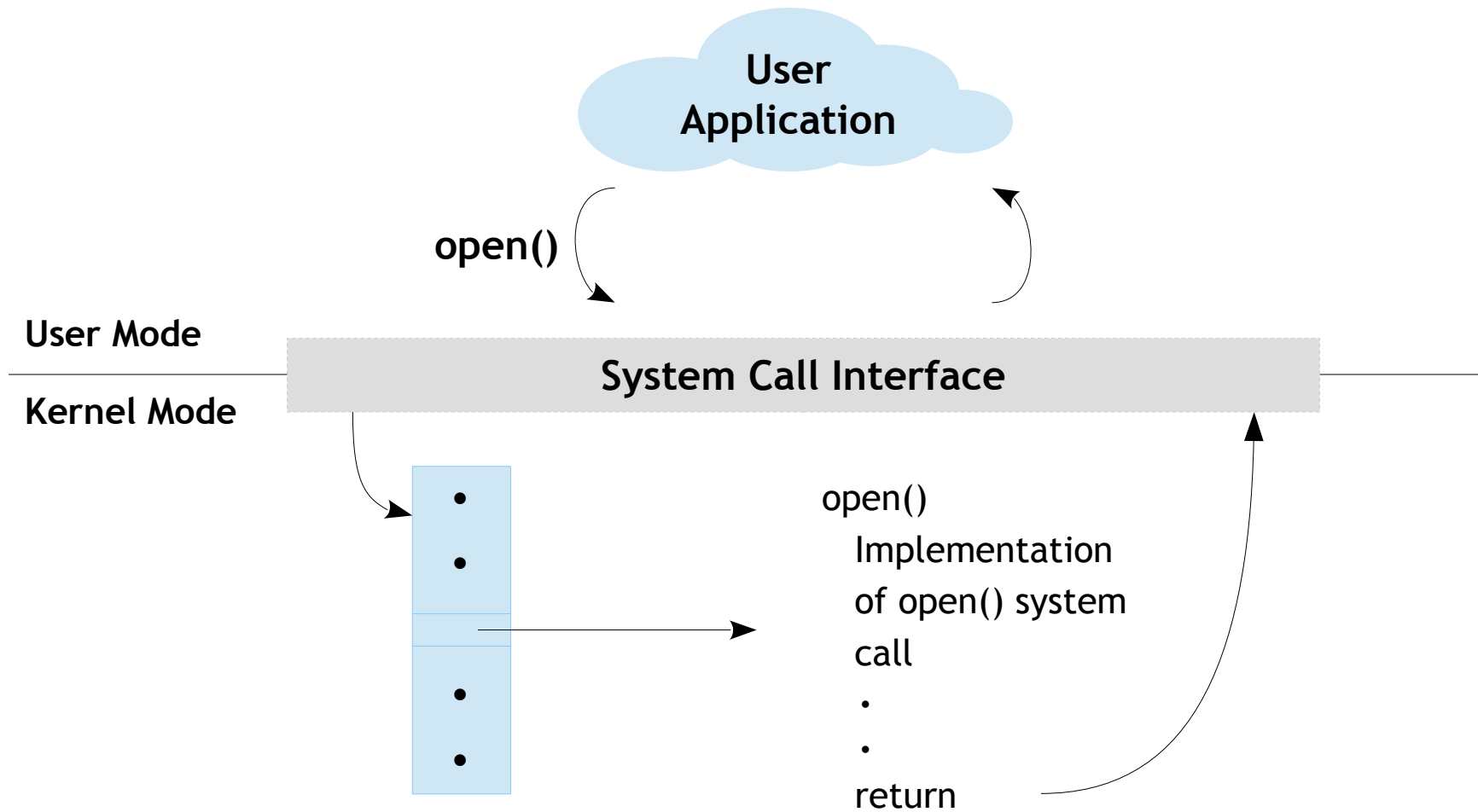- When interrupt done, program state reloaded and program resumes

ΣMERTXE

System calls

- A set of interfaces to interact with hardware devices such as the CPU, disks, and printers.

- Advantages:

  - Freeing users from studying low-level programming

  - It greatly increases system security

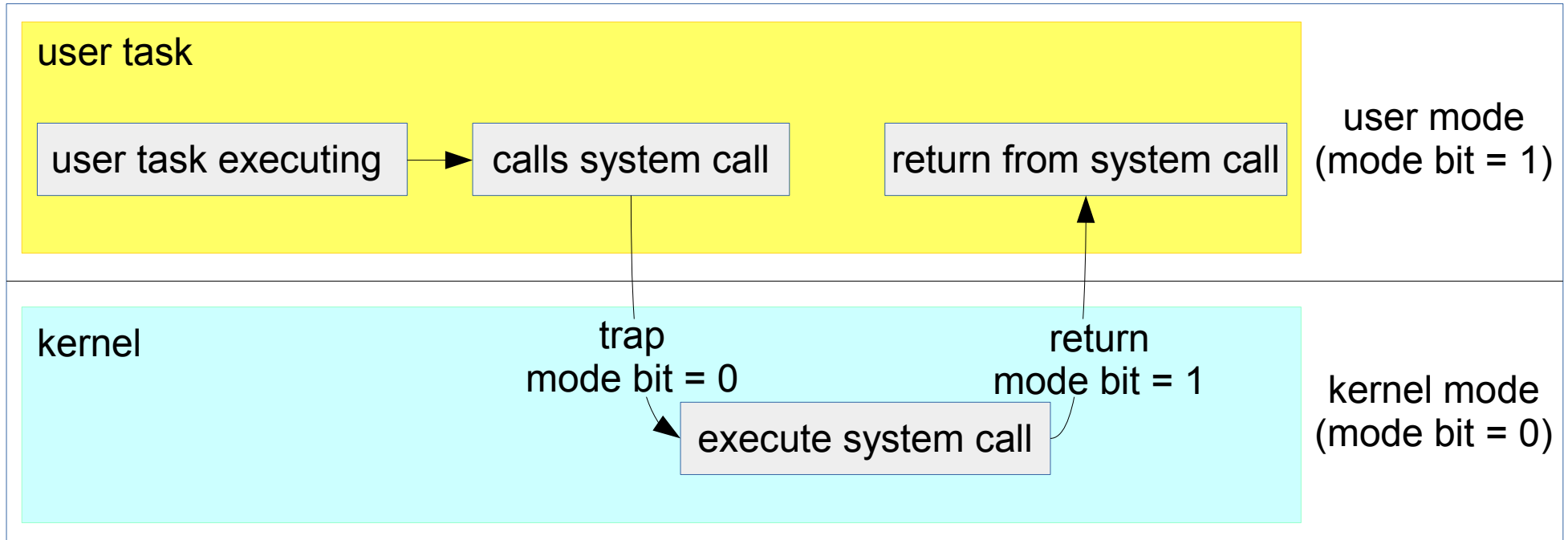 - These interfaces make programs more portable

For a OS programmer, calling a system call is no different from a normal function call. But the way system call is executed is way different.

EMERTXE

System calls

User Application

open()

User Mode

System Call Interface

Kernel Mode

open()
Implementation
of open() system
call
·
·
return

System Call
Calling Sequence

user task			user mode (mode bit = 1)
user task executing →	calls system call	return from system call	

kernel			kernel mode (mode bit = 0)
	trap mode bit = 0	return mode bit = 1	
	execute system call		

Logically the system call and regular interrupt follow the same flow of steps. The source (I/O device v/s user program) is very different for both of them. Since system call is generated by user program they are called as 'Soft interrupts' or 'Traps'

ΣMERTXE

System Call
vs Library Function

- A library function is an ordinary function that resides in a library external to your program. A call to a library function is just like any other function call

- A system call is implemented in the Linux kernel and a special procedure is required in to transfer the control to the kernel

- Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ

✓ Understand the differences between:
  - Functions
  - Library functions
  - System calls
✓ From the programming perspective they all are nothing but simple C functions

System Call
Implementation

....
xyz()
....

xyz() {
....
int 0x80
....
}

system_call:
  ...
  sys_xyz()
  ...
ret_from_sys_call:
  ...
  iret

sys_xyz() {

  ...

}

System Call
Invocation in
application
program

Wrapper
routine in libc
standard
library

System call
handler

System call
service
routine
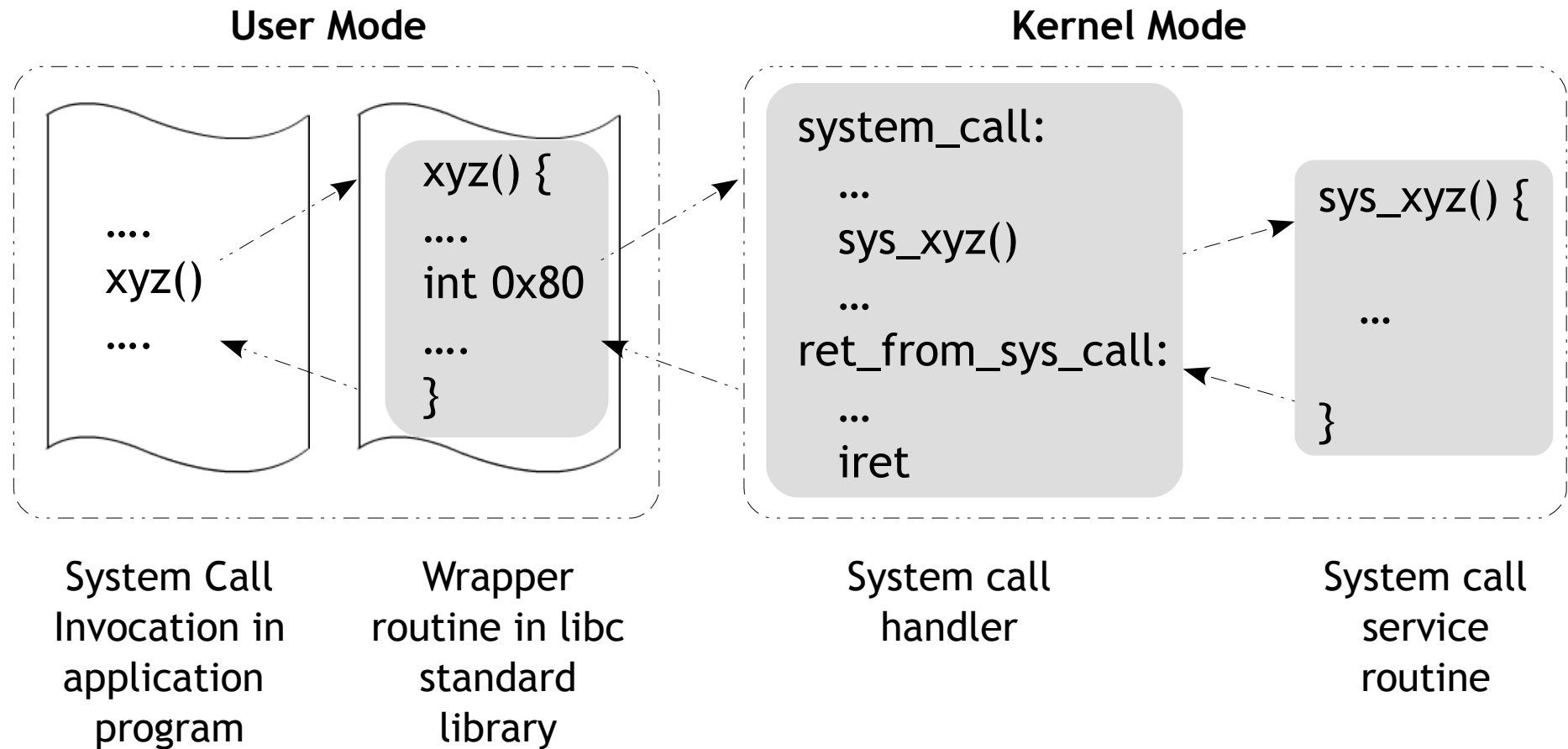
System Call
Example: gettimeofday()

- Gets the system's wall-clock time.

- It takes a pointer to a struct timeval variable. This structure represents a time, in seconds, split into two fields.

  - tv_sec field - integral number of seconds
  - tv_usec field - additional number of usecs

EMERTXE

System Call
Example: nanosleep()

- A high-precision version of the standard UNIX sleep call

- Instead of sleeping an integral number of seconds, **nanosleep** takes as its argument a pointer to a **struct timespec** object, which can express time to nanosecond precision.

  - tv_sec field - integral number of seconds
  - tv_nsec field - additional number of nsecs

System Call
Example: Others

- open

- read

- write

- exit

- close

- wait

- waitpid

- getpid

- sync

- nice

- kill etc..

ΣMERTXE

Process

Process

- Running instance of a program is called a PROCESS

- If you have two terminal windows showing on your screen, then you are probably running the same terminal program twice-you have two terminal processes

- Each terminal window is probably running a shell; each running shell is another process

- When you invoke a command from a shell, the corresponding program is executed in a new process

- The shell process resumes when that process complete

ΣMERTXE

Process
vs Program

- A program is a passive entity, such as file containing a list of instructions stored on a disk

- Process is a active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

- A program becomes a process when an executable file is loaded into main memory

Factor	Process	Program
Storage	Dynamic Memory	Secondary Memory
State	Active	Passive

Process
vs Program

Program

```
int global_1 = 0;
int global_2 = 0;

void do_somthing()
{
    int local_2 = 5;
    local_2 = local_2 + 1;
}

int main()
{
    char *local_1 = malloc(100);

    do_somthing();
    …..
}
```

Task
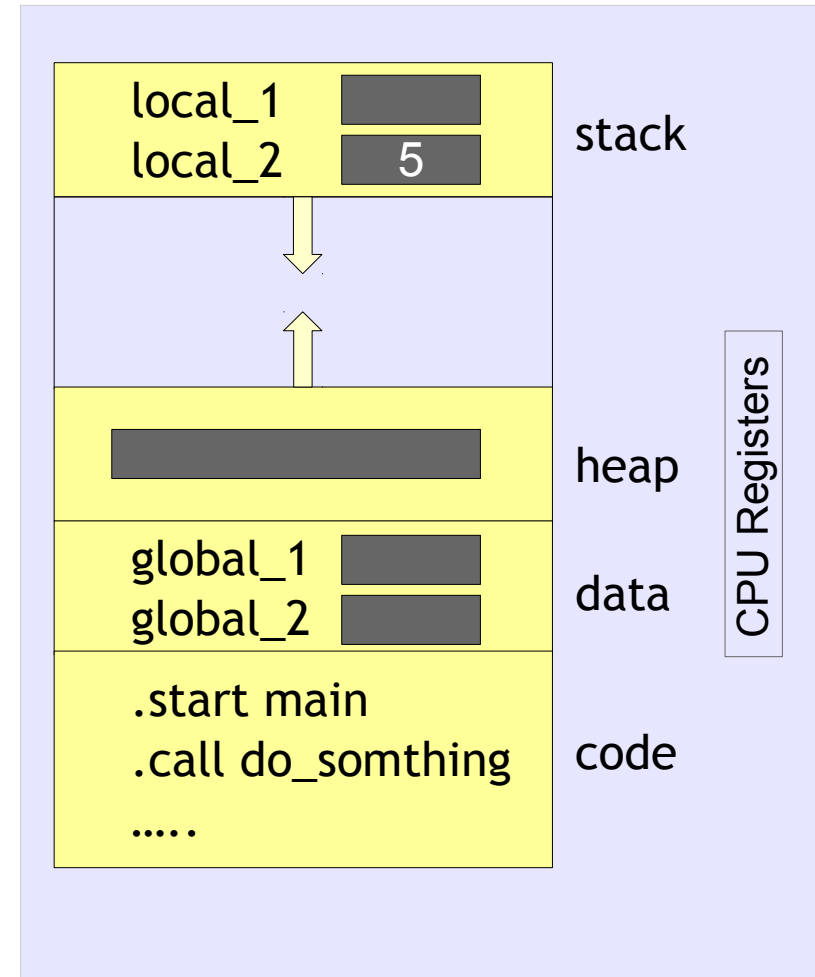
Process
More processes in memory!

P0
Free Space
P1
P2
Free Space
OS

Stack
⬇ ⬆
Heap
Data
Code
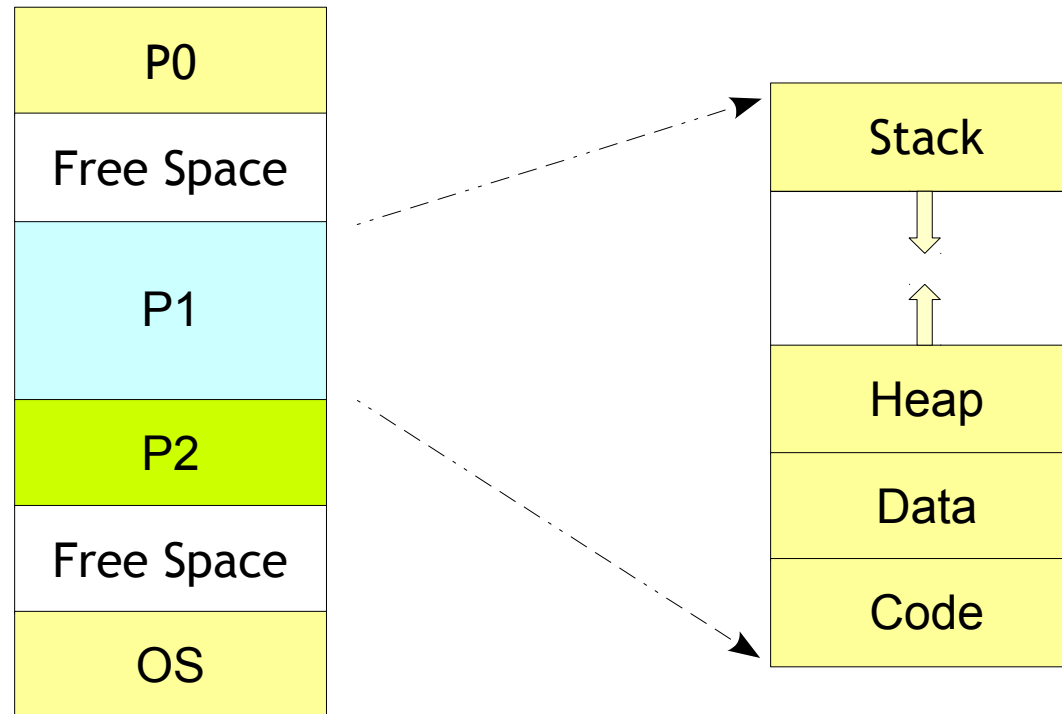
Each Process will have its own Code, Data, Heap and Stack
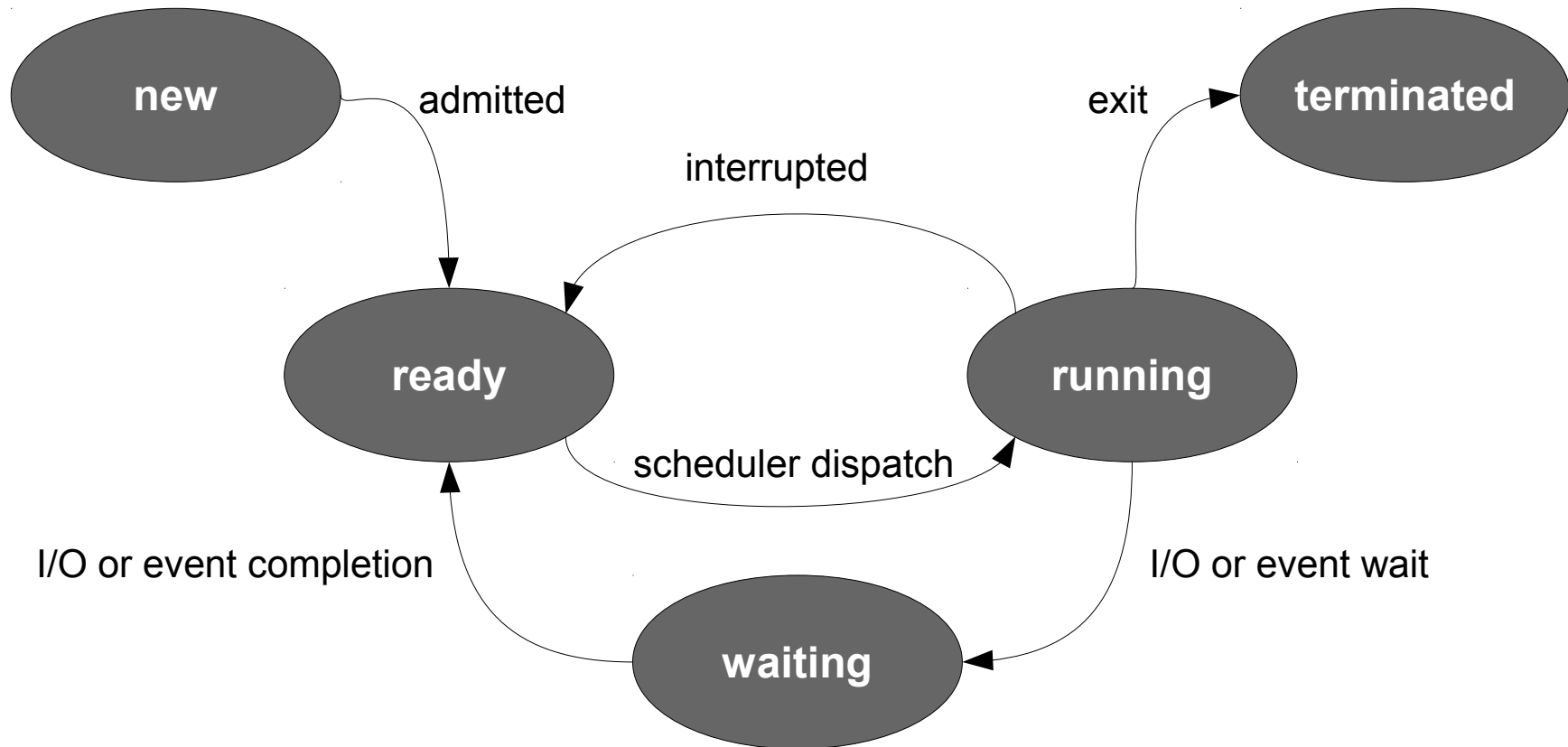
EMERTXE

Process
State Transition Diagram

Process
State Transition Diagram
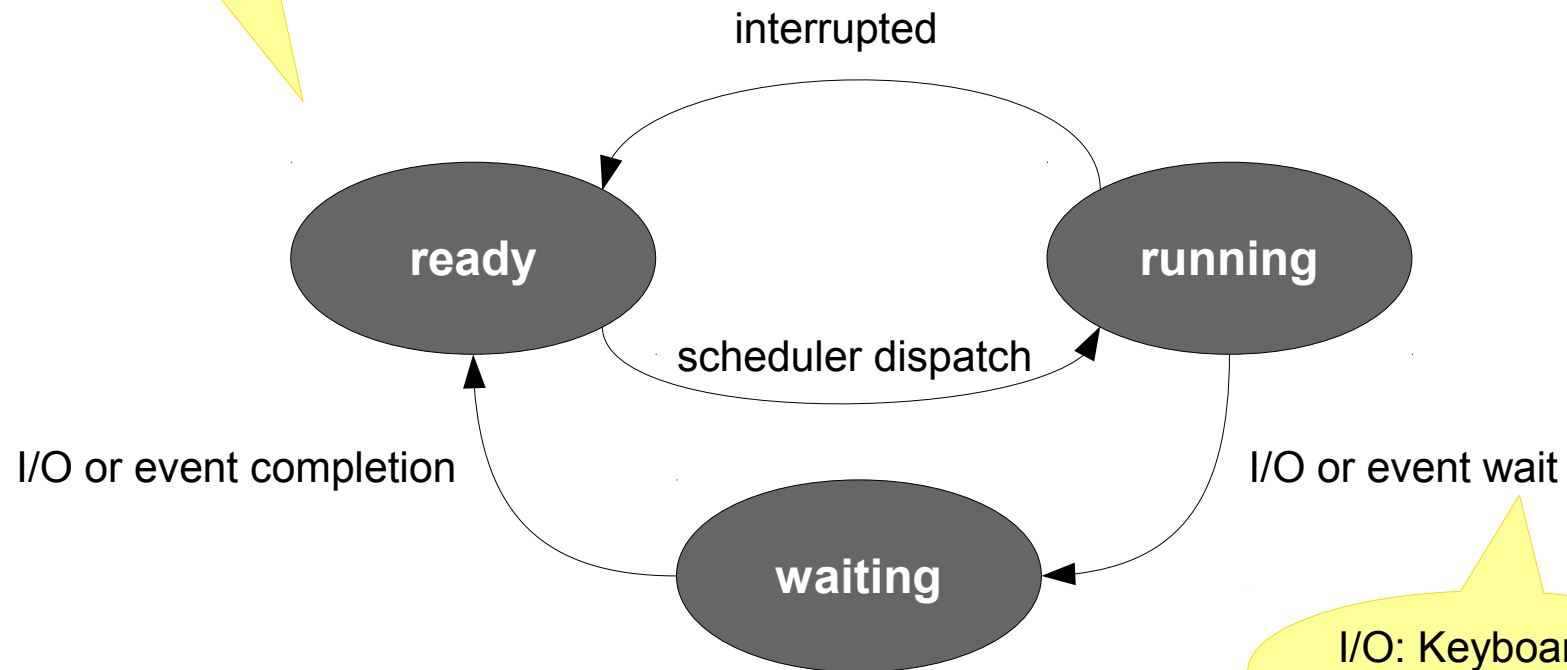
Process
States

- A process goes through multiple states ever since it is created by the OS

State	Description
New	The process is being created
Running	Instructions are being executed
Waiting	The process is waiting for some event to occur
Ready	The process is waiting to be assigned to processor
Terminated	The process has finished execution

Process
Descriptor

- To manage tasks:

  - OS kernel must have a clear picture of what each task is doing.

  - Task's priority

  - Whether it is running on the CPU or blocked on some event

  - What address space has been assigned to it

  - Which files it is allowed to address, and so on.

- Usually the OS maintains a structure whose fields contain all the information related to a single task

Process
Descriptor

Pointer	Process State
Process ID	
Program Counter	
Registers	
Memory Limits	
List of Open Files	
.	
.	
.	
.	
.	
.	

- Information associated with each process.

- Process state

- Program counter

- CPU registers

- CPU scheduling information

- Memory-management information

- I/O status information

Process
Descriptor – State Field

- State field of the process descriptor describes the state of process.

- The possible states are:

State	Description
TASK_RUNNING	Task running or runnable
TASK_INTERRUPTIBLE	process can be interrupted while sleeping
TASK_UNINTERRUPTIBLE	process can't be interrupted while sleeping
TASK_STOPPED	process execution stopped
TASK_ZOMBIE	parent is not issuing wait()

Process
Descriptor - ID

- Each process in a Linux system is identified by its unique process ID, sometimes referred to as PID

- Process IDs are numbers that are assigned sequentially by Linux as new processes are created

- Every process also has a parent process except the special init process

- Processes in a Linux system can be thought of as arranged in a tree, with the init process at its root

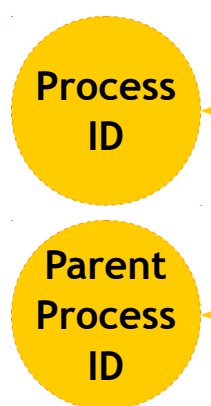- The parent process ID or PPID, is simply the process ID of the process's parent

Process
Schedule
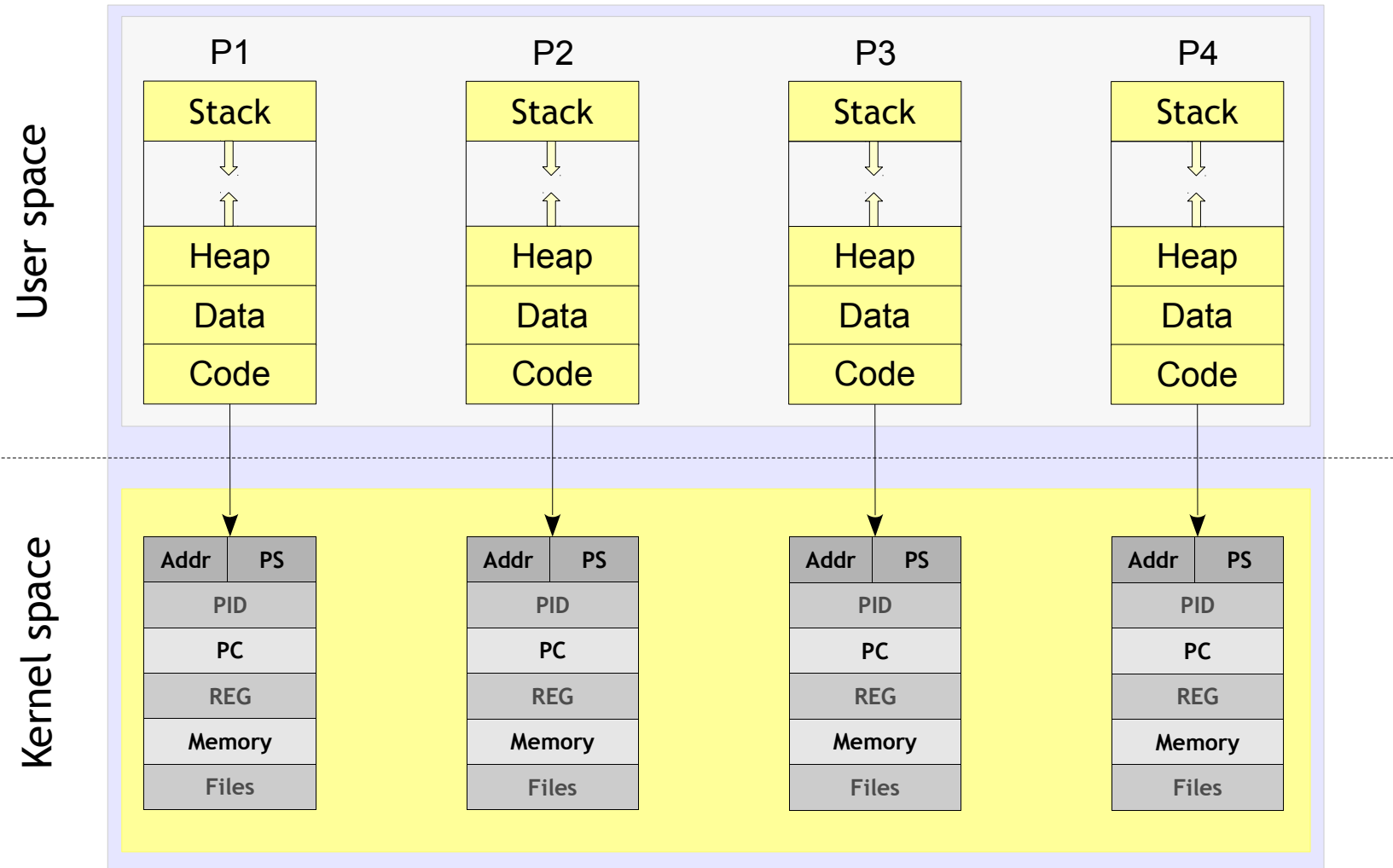
Process
Active Processes

- The *ps* command displays the processes that are running on your system

- By default, invoking ps displays the processes controlled by the terminal or terminal window in which ps is invoked

- For example (Executed as "ps –aef"):

```
user@user:~] ps -aef
UID        PID   PPID  C STIME TTY          TIME CMD
root         1      0  0 12:17 ?        00:00:01 /sbin/init
root         2      0  0 12:17 ?        00:00:00 [kthreadd]
root         3      2  0 12:17 ?        00:00:02 [ksoftirqd/0]
root         4      2  0 12:17 ?        00:00:00 [kworker/0:0]
root         5      2  0 12:17 ?        00:00:00 [kworker/0:0H]
root         7      2  0 12:17 ?        00:00:00 [rcu_sched]
```

Process ID

Parent Process ID

Process
Context Switching

- Switching the CPU to another task requires saving the state of the old task and loading the saved state for the new task

- The time wasted to switch from one task to another without any disturbance is called context switch or scheduling jitter

- After scheduling the new process gets hold of the processor for its execution

Context Switching
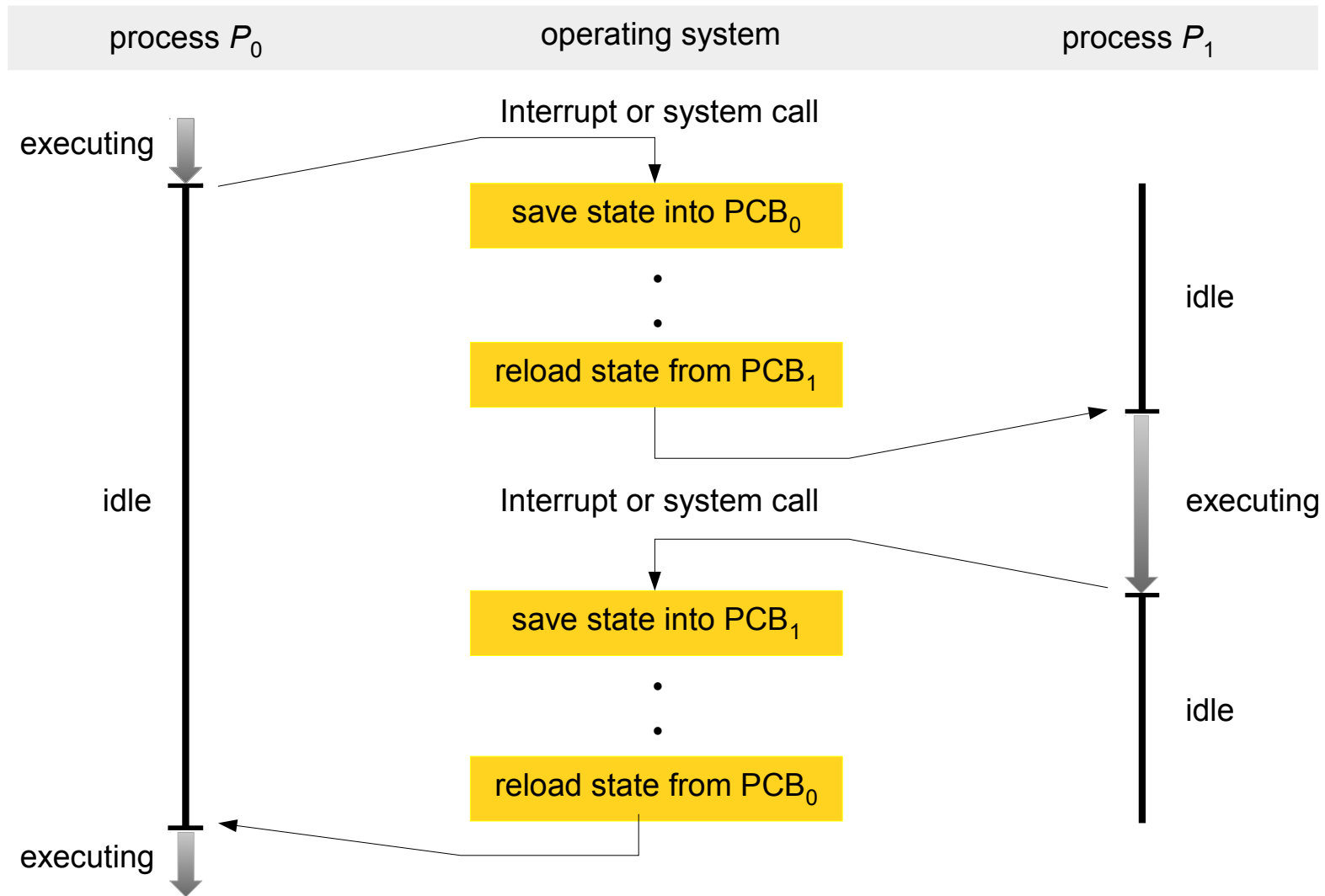
Process
Creation

- Two common methods are used for creating new process

- Using system(): Relatively simple but should be used sparingly because it is inefficient and has considerably security risks

- Using fork() and exec(): More complex but provides greater flexibility, speed, and security

Process
Creation - system()

- It creates a sub-process running the standard shell

- Hands the command to that shell for execution

- Because the system function uses a shell to invoke your command, it's subject to the features and limitations of the system shell

- The system function in the standard C library is used to execute a command from within a program

- Much as if the command has been typed into a shell

Process
Creation - fork()

- fork makes a child process that is an exact copy of its parent process

- When a program calls fork, a duplicate process, called the child process, is created

- The parent process continues executing the program from the point that fork was called

- The child process, too, executes the same program from the same place

- All the statements after the call to fork will be executed twice, once, by the parent process and once by the child process

Process
Creation - fork()

- The execution context for the child process is a copy of parent's context at the time of the call

```c
int child_pid;
int child_status;

int main()
{
    int ret;

    ret = fork();
    switch (ret)
    {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            <code for child process>
            exit(0);
        default:
            <code for parent process>
            wait(&child_status);
    }
}
```
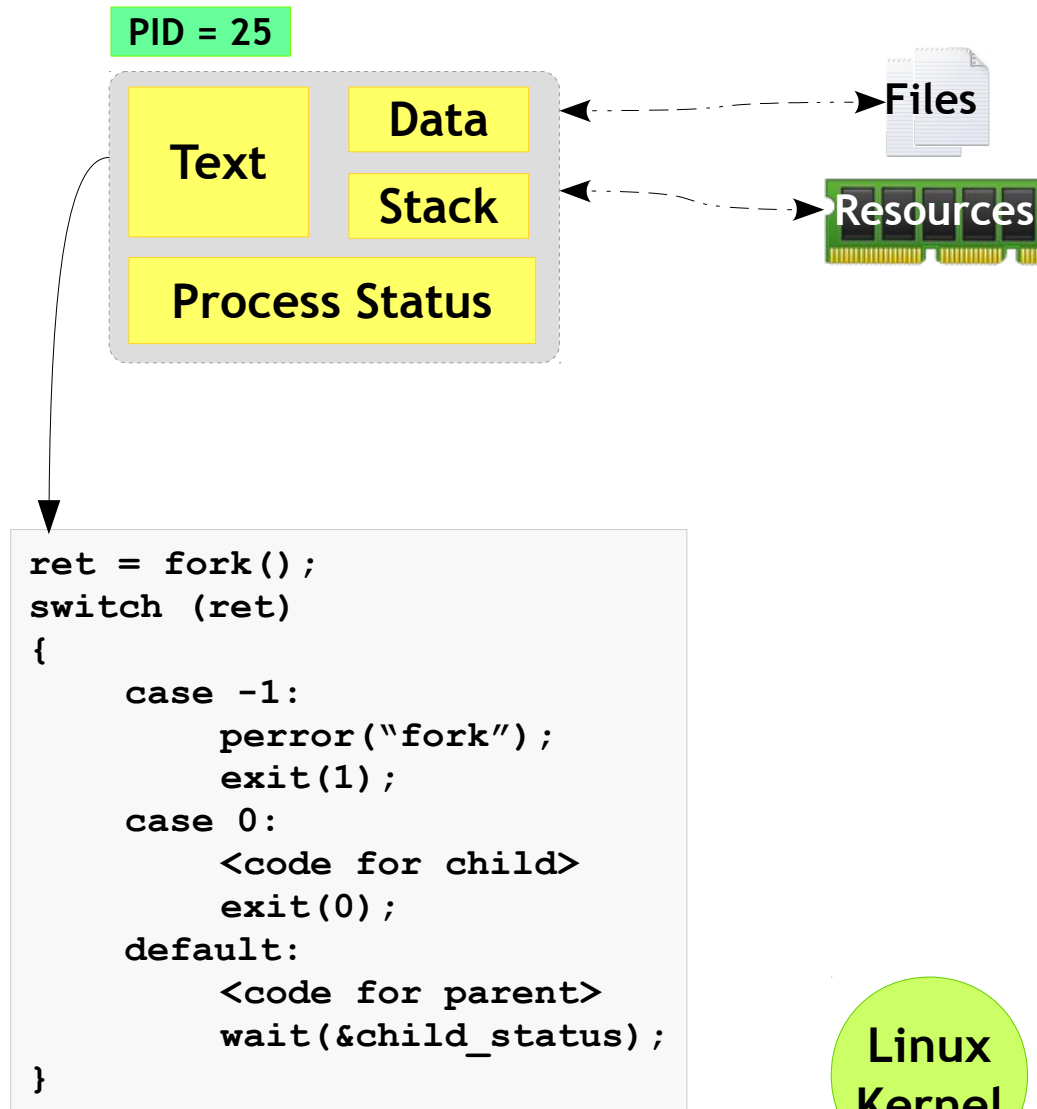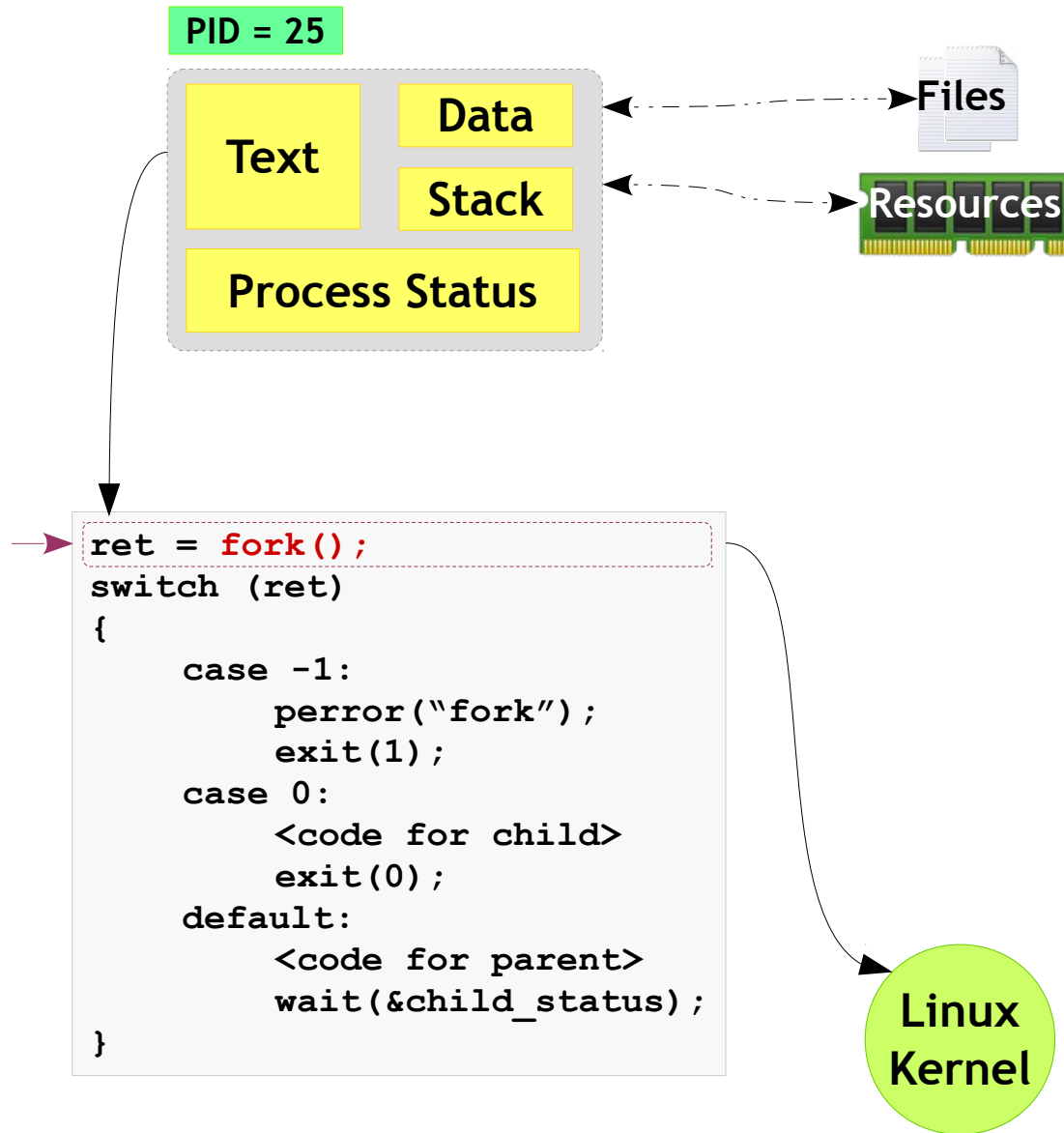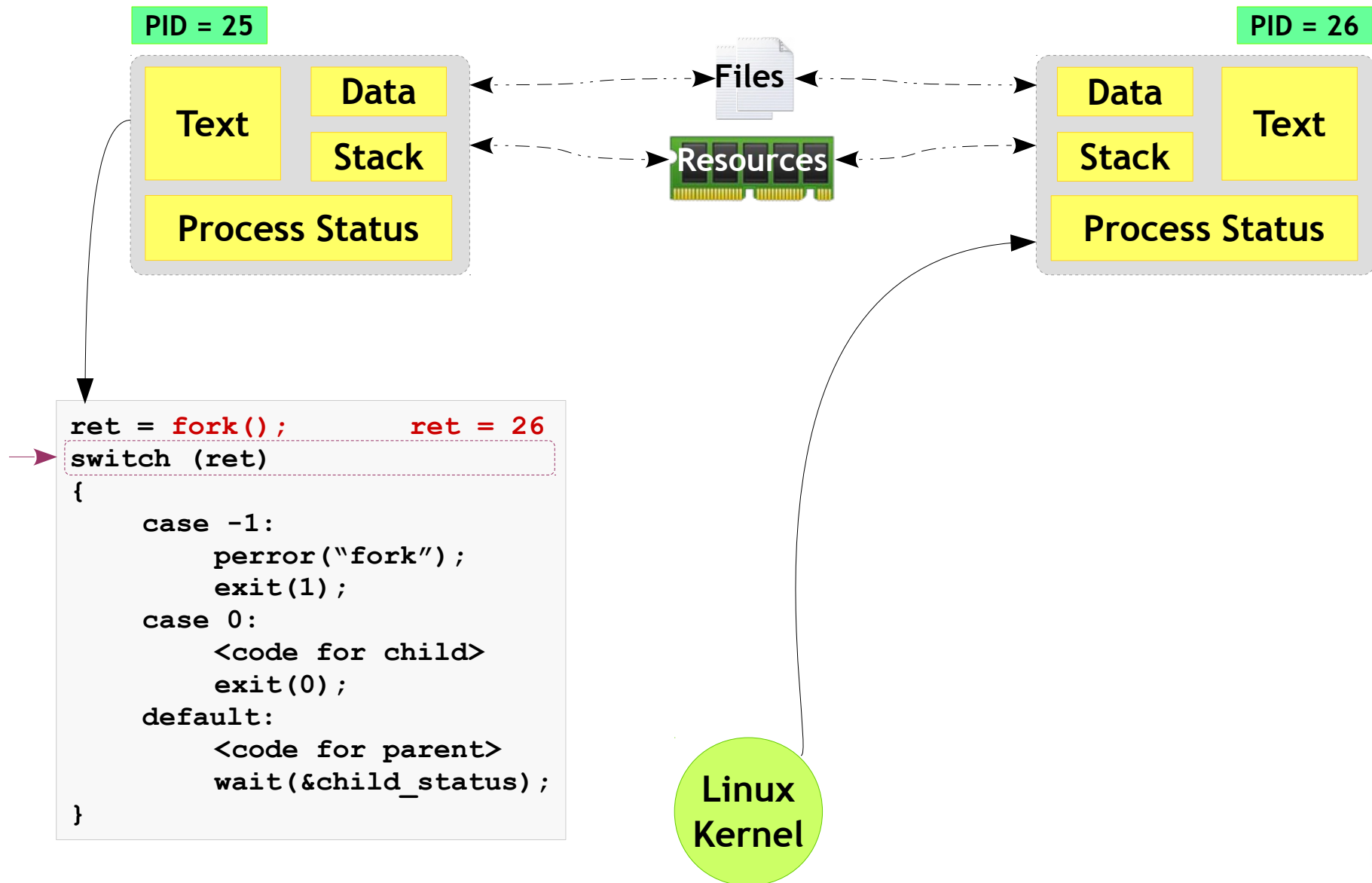
fork()

Stack
Heap
Data
Code

ret = 0

ret = xx

Stack
Heap
Data
Code

Process
fork() - The Flow

PID = 25

Text

Data

Stack

Process Status

Linux Kernel

Process
fork() - The Flow

PID = 25

Text	Data
	Stack
Process Status	

Files

Resources

```
ret = fork();
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux
Kernel

ΣMERTXE

Process
fork() - The Flow

PID = 25

Text | Data
Stack
Process Status

Files

Resources

```
ret = fork();
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux
Kernel

Process
fork() - The Flow

PID = 25

Text	Data
	Stack
Process Status	

Files

Resources

PID = 26

Data	Text
Stack	
Process Status	

```
ret = fork();         ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux
Kernel

ΣMERTXE

fork() - The Flow

Text
Data
Stack
Process Status

Files

Resources

Data
Text
Stack
Process Status

```
ret = fork();        ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

```
ret = fork();
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux
Kernel

Process
fork() - The Flow

Text

Data

Stack

Process Status

Files

Resources

Data

Text

Stack

Process Status

```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

```
ret = fork();          ret = 0
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux Kernel

EMERTXE

Process
fork() - The Flow

PID = 25

Text
Data
Stack
Process Status

Files

Resources

PID = 26

Data
Stack
Text
Process Status

```
ret = fork();        ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

```
ret = fork();        ret = 0
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux Kernel

EMERTXE

Process
fork() - The Flow

PID = 25

Text | Data | Stack

Process Status

Files

Resources

PID = 26

Data | Stack | Text

Process Status

```
ret = fork();        ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

```
ret = fork();        ret = 0
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux Kernel

EMERTXE

Process
fork() - The Flow

PID = 25


```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Text

Data

Stack

Process Status

Files

Resources

Linux
Kernel

Process
fork() - How to Distinguish?

- First, the child process is a new process and therefore has a new process ID, distinct from its parent's process ID

- One way for a program to distinguish whether it's in the parent process or the child process is to call getpid

- The fork function provides different return values to the parent and child processes

- One process "goes in" to the fork call, and two processes "come out," with different return values

- The return value in the parent process is the process ID of the child

- The return value in the child process is zero

EMERTXE

- What would be output of the following program?

```
int main()
{
    fork();
    fork();
    fork();

    printf("Hello World\n");

    return 0;
}
```

P

```
int main()
{
    fork();
    fork();
    fork();

    printf("Hello World\n");

    return 0;
}
```

```
int main()
{
    fork();
    fork();
    fork();

    printf("Hello World\n");

    return 0;
}
```

P

C1

Process
fork() - Example

```c
int main()
{
    fork();
    fork();
    fork();

    printf("Hello World\n");

    return 0;
}
```


Note: The actual order of execution based on scheduling

Process
fork() - Example

```
int main()
{
    fork();
    fork();
→   fork();

    printf("Hello World\n");

    return 0;
}
```


Note: The actual order of execution based on scheduling

ΣMERTXE
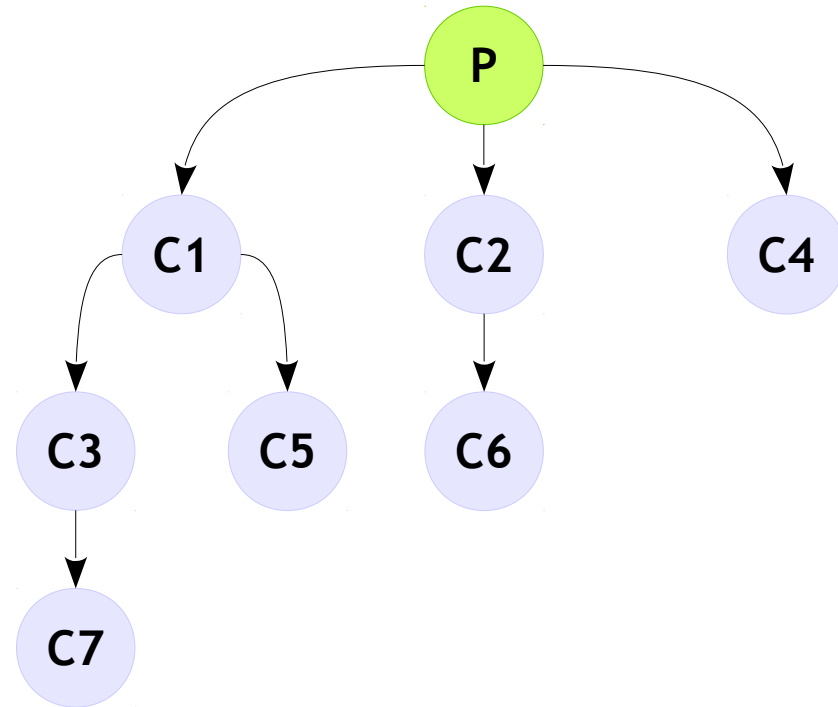
Process
Zombie

- Zombie process is a process that has terminated but has not been cleaned up yet

- It is the responsibility of the parent process to clean up its zombie children

- If the parent does not clean up its children, they stay around in the system, as zombie

- When a program exits, its children are inherited by a special process, the init program, which always runs with process ID of 1 (it's the first process started when Linux boots)

- The init process automatically cleans up any zombie child processes that it inherits.

Process
Orphan

- An orphan process is a computer process whose parent process has finished or terminated, though it remains running itself.

- Orphaned children are immediately "adopted" by init .

- An orphan is just a process. It will use whatever resources it uses. It is reasonable to say that it is not an "orphan" at all since it has a parent but "adopted".

- Init automatically reaps its children (adopted or otherwise).

- So if you exit without cleaning up your children, then they will not become zombies.

ΣMERTXE

Process
Overlay - exec()

- The exec functions replace the program running in a process with another program

- When a program calls an exec function, that process immediately ceases executing and begins executing a new program from the beginning

- Because exec replaces the calling program with another one, it never returns unless an error occurs

- This new process has the same PID as the original process, not only the PID but also the parent process ID, current directory, and file descriptor tables (if any are open) also remain the same

- Unlike fork, exec results in still having a single process

Process
Overlay - exec()

- Let us consider an example of execlp (variant of exec() function) shown below

```
/* Program: my_ls.c */

int main()
{
    print("Executing my ls :)\n");
    execlp("/bin/ls", "ls", NULL);
}
```

Process
Overlay - exec()

- After executing the exec function, you will note the following changes

```
/* Program: my_ls.c */

int main()
{
    print("Executing my ls :)\n");
    execlp("/bin/ls", "ls", NULL);
}
```

PID	→ Preserved
Program Counter	
Registers	→ Reset
Stack	→ Overwritten by New Code
Heap	→ Overwritten by New Code
Data	→ Overwritten by New Code
Code	→ Overwritten with New Code
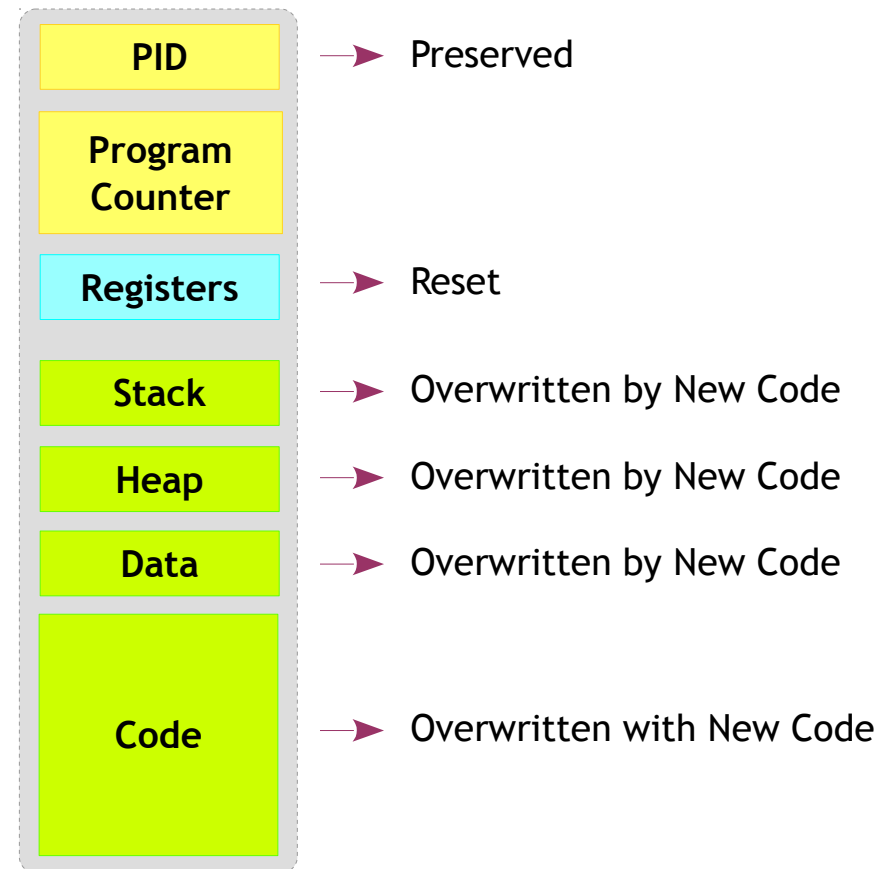
Process
exec() - Variants

- The exec has a family of system calls with variations among them

- They are differentiated by small changes in their names

- The exec family looks as follows:

System call	Meaning
execl(const char *path, const char *arg, ...);	Full path of executable, variable number of arguments
execlp(const char *file, const char *arg, ...);	Relative path of executable, variable number of arguments
execv(const char *path, char *const argv[]);	Full path of executable, arguments as pointer of strings
execvp(const char *file, char *const argv[]);	Relative path of executable, arguments as pointer of strings

Process
Blending fork() and exec()

- Practically calling program never returns after exec()

- If we want a calling program to continue execution after exec, then we should first fork() a program and then exec the subprogram in the child process

- This allows the calling program to continue execution as a parent, while child program uses exec() and proceeds to completion

- This way both fork() and exec() can be used together

Process
COW – Copy on Write

- Copy-on-write (called COW) is an optimization strategy

- When multiple separate process use same copy of the same information it is not necessary to re-create it

- Instead they can all be given pointers to the same resource, thereby effectively using the resources

- However, when a local copy has been modified (i.e. write) , the COW has to replicate the copy, has no other option

- For example if exec() is called immediately after fork() they never need to be copied the parent memory can be shared with the child, only when a write is performed it can be re-created

Process
Termination

- When a parent forks a child, the two process can take any turn to finish themselves and in some cases the parent may die before the child

- In some situations, though, it is desirable for the parent process to wait until one or more child processes have completed

- This can be done with the wait() family of system calls.

- These functions allow you to wait for a process to finish executing, enable parent process to retrieve information about its child's termination

Synchronous & Asynchronous

- Wait for child to finish

Synchronous	Asynchronous
Polling	Interrupts
sleep	wait

EMERTXE
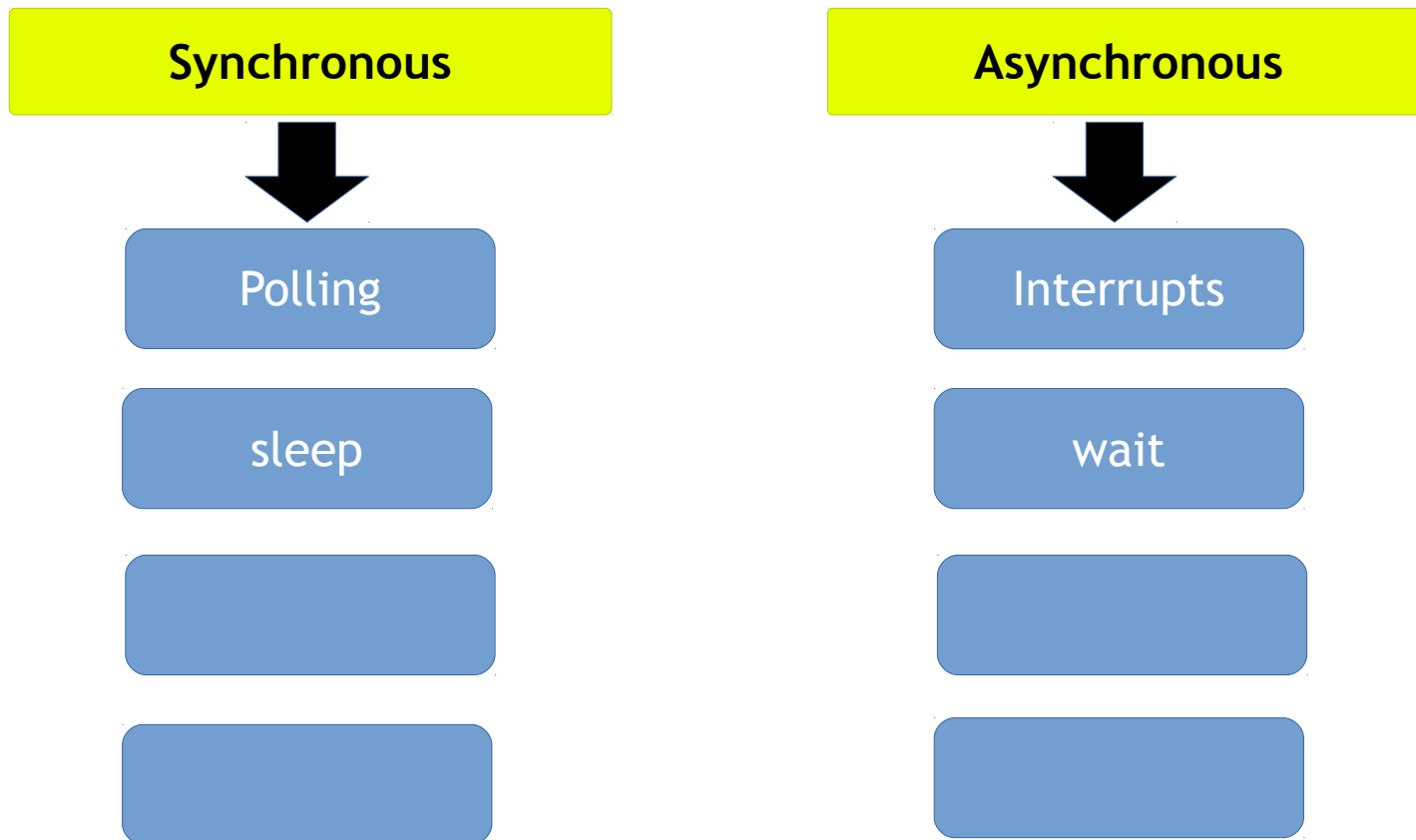
Process
Wait

- fork() in combination with wait() can be used for child monitoring

- Appropriate clean-up (if any) can be done by the parent for ensuring better resource utilization

- Otherwise it will result in a ZOMBIE process

- There are four different system calls in the wait family

System call	Meaning
wait(int *status)	Blocks & waits the calling process until one of its child processes exits. Return status via simple integer argument
waitpid (pid_t pid, int* status, int options)	Similar to wait, but only blocks on a child with specific PID
wait3(int *status, int options, struct rusage *rusage)	Returns resource usage information about the exiting child process.
wait4 (pid_t pid, int *status, int options, struct rusage *rusage)	Similar to wait3, but on a specific child

```
struct rusage {

    struct timeval ru_utime; /* user CPU time used */

    struct timeval ru_stime; /* system CPU time used */

    long   ru_maxrss;          /* maximum resident set size */

    long   ru_ixrss;           /* integral shared memory size */

    long   ru_idrss;           /* integral unshared data size */

    long   ru_isrss;           /* integral unshared stack size */

    long   ru_minflt;          /* page reclaims (soft page faults) */

    long   ru_majflt;          /* page faults (hard page faults) */

    long   ru_nswap;           /* swaps */

    long   ru_inblock;         /* block input operations */

    long   ru_oublock;         /* block output operations */

    long   ru_msgsnd;          /* IPC messages sent */

    long   ru_msgrcv;          /* IPC messages received */

    long   ru_nsignals;        /* signals received */

    long   ru_nvcsw;           /* voluntary context switches */

    long   ru_nivcsw;          /* involuntary context switches */

};
```

Inter Process Communications (IPC)

Communication
in real world

- Face to face

- Fixed phone

- Mobile phone

- Skype

- SMS

ΣMERTXE

Inter Process Communications
Introduction

- *Inter process communication (IPC)* is the mechanism whereby one process can communicate, that is exchange data with another processes

- There are two flavors of IPC exist: System V and POSIX

- Former is derivative of UNIX family, later is when standardization across various OS (Linux, BSD etc..) came into picture

- Some are due to "UNIX war" reasons also

- In the implementation levels there are some differences between the two, larger extent remains the same

- Helps in portability as well

Inter Process Communications
Introduction

- IPC can be categorized broadly into two areas:

    Data exchange   Resource usage/access/control

    | Communication | | Synchronization |

    - Pipes   - Semaphores
    - FIFO
    - Shared memory
    - Signals
    - Sockets

- Even in case of Synchronization also two processes are talking.

> Each IPC mechanism offers some advantages & disadvantages. Depending on the program design, appropriate mechanism needs to be chosen.

Application and Tasks

A → T

Example: Read from a file
$ cat file.txt

A → T1, T2, T3, T4
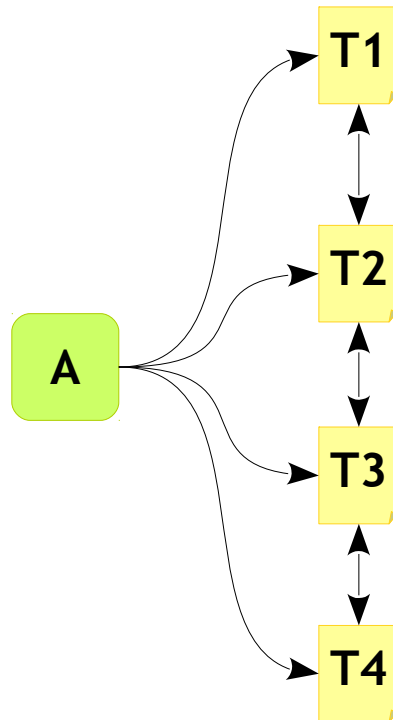
Example: Paper jam handling
in printer

ΣMERTXE

Inter Process Communications
User vs Kernel Space

- Protection domains - (virtual address space)
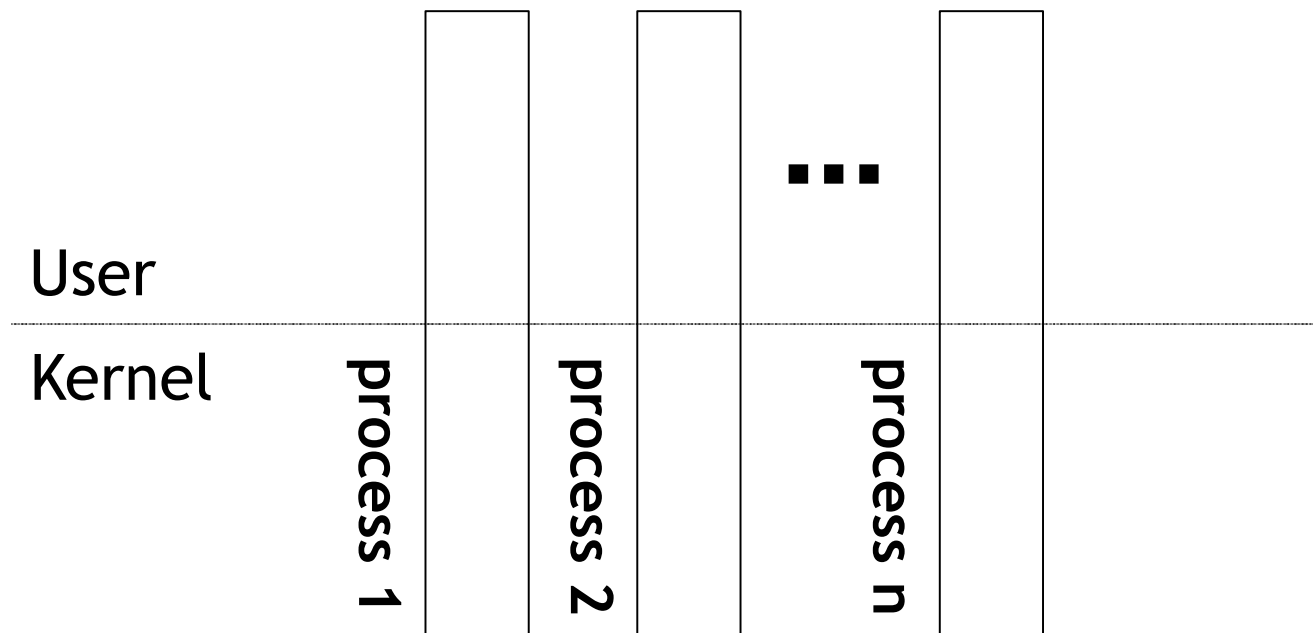
How can processes communicate with each other and the kernel? The answer is nothing but IPC mechanisms

Inter Process Communications
Pipes

- A pipe is a communication device that permits unidirectional communication

- Data written to the "write end" of the pipe is read back from the "read end"

- Pipes are serial devices; the data is always read from the pipe in the same order it was written

Water In End

Water Out End

Data In End

Data in Out

EMERTXE
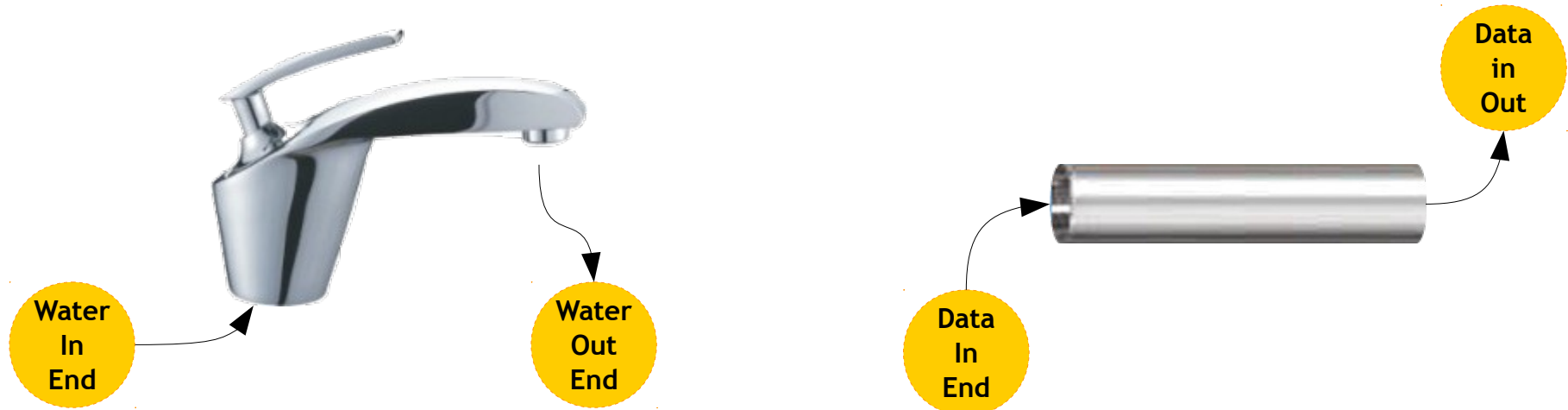
Inter Process Communications
Pipes - Creation

- To create a pipe, invoke the pipe system call

- Supply an integer array of size 2

- The call to pipe stores the reading file descriptor in array position 0

- Writing file descriptor in position 1

Function	Meaning
int pipe(int pipe_fd[2])	✓ Pipe gets created ✓ READ and WRITE pipe descriptors are populated ✓ RETURN: Success (0)/Failure (Non-zero)

Pipe read and write can be done simultaneously between two processes by creating a child process using fork() system call.

Inter Process Communications
Pipes – Direction of communication

- Let's say a Parent wants to communicate with a Child

- Generally the communication is possible both the way!
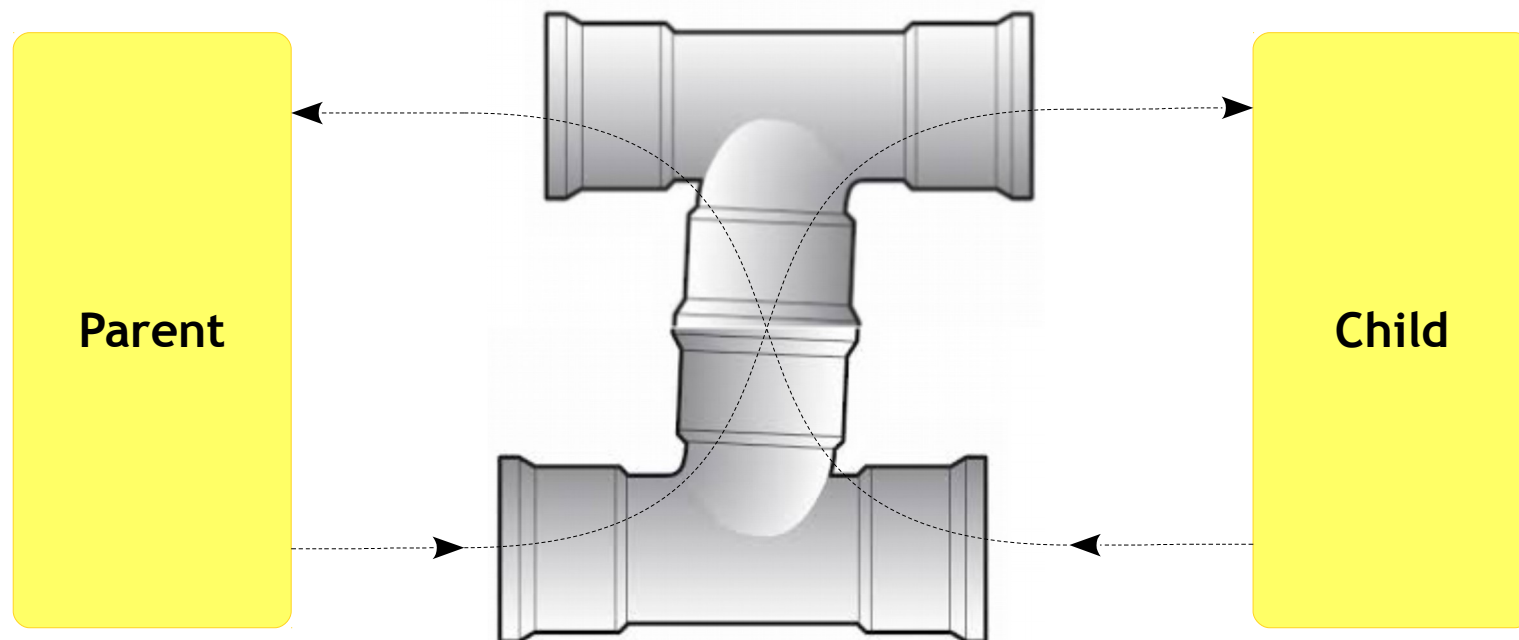
Inter Process Communications
Pipes – Direction of communication

- So it necessary to close one of the end form both sides
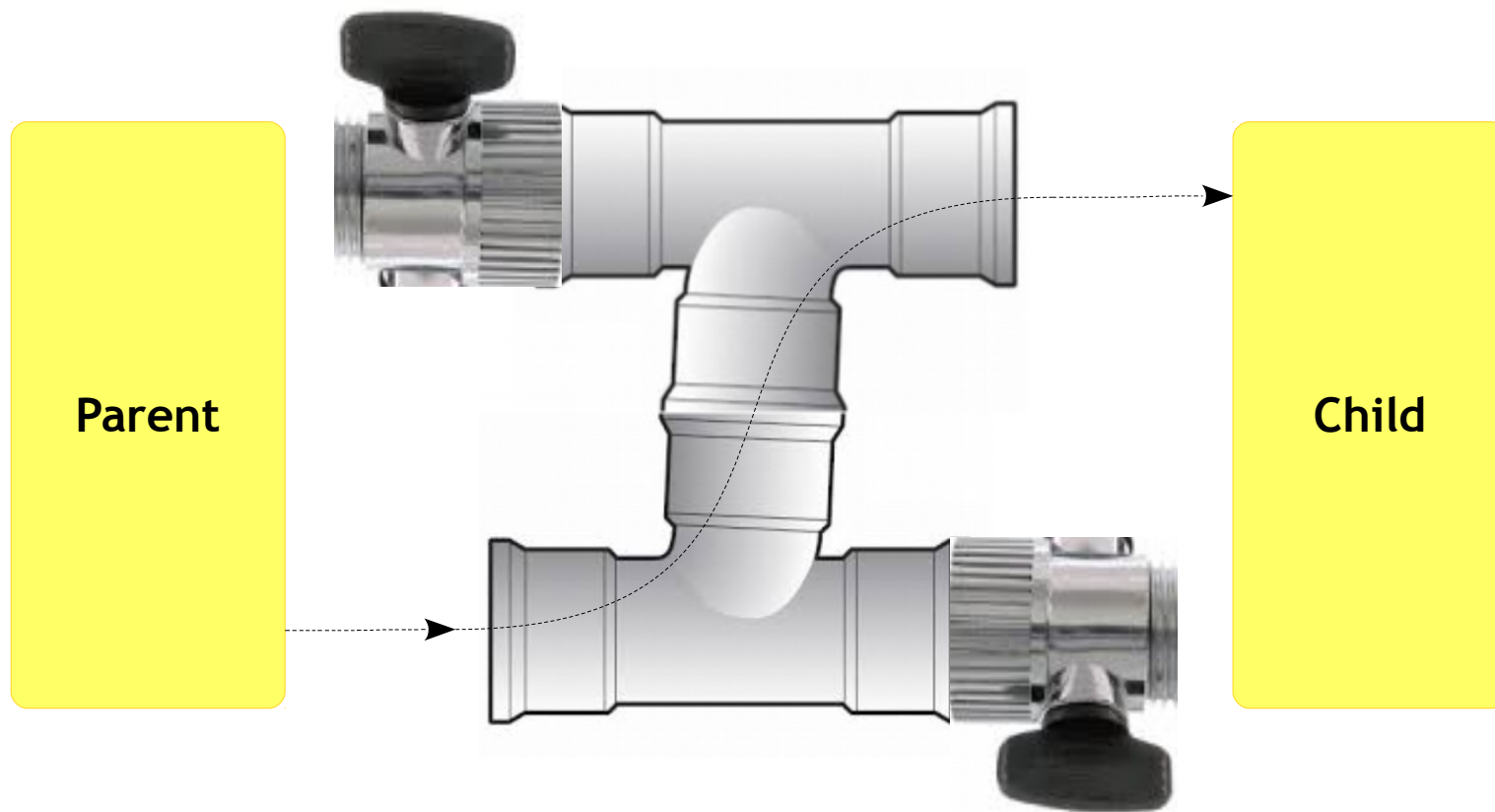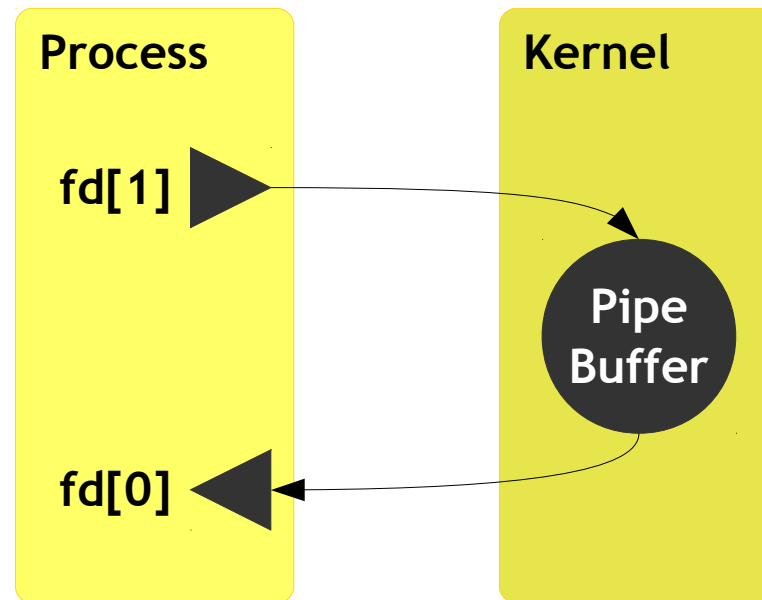
Inter Process Communications
Pipes - Pros & Cons

PROS	CONS
• Naturally synchronized	• Less memory size (4K)
• Simple to use and create	• Only related process can communicate.
• No extra system calls required to communicate (read/write)	• Only two process can communicate
	• One directional communication
	• Kernel is involved

ΣMERTXE

Inter Process Communications
Summary

- We have covered

    Data exchange

    Communication

    - Pipes
    - FIFO
    - Shared memory
    - Signals
    - Sockets

    Resource usage/access/control

    Synchronization

    - Semaphores

Inter Process Communications
FIFO - Properties

- A *first-in, first-out (FIFO)* file is a pipe that has a name in the file-system

- FIFO file is a pipe that has a name in the file-system

- FIFOs are also called Named Pipes

- FIFOs is designed to let them get around one of the shortcomings of normal pipes
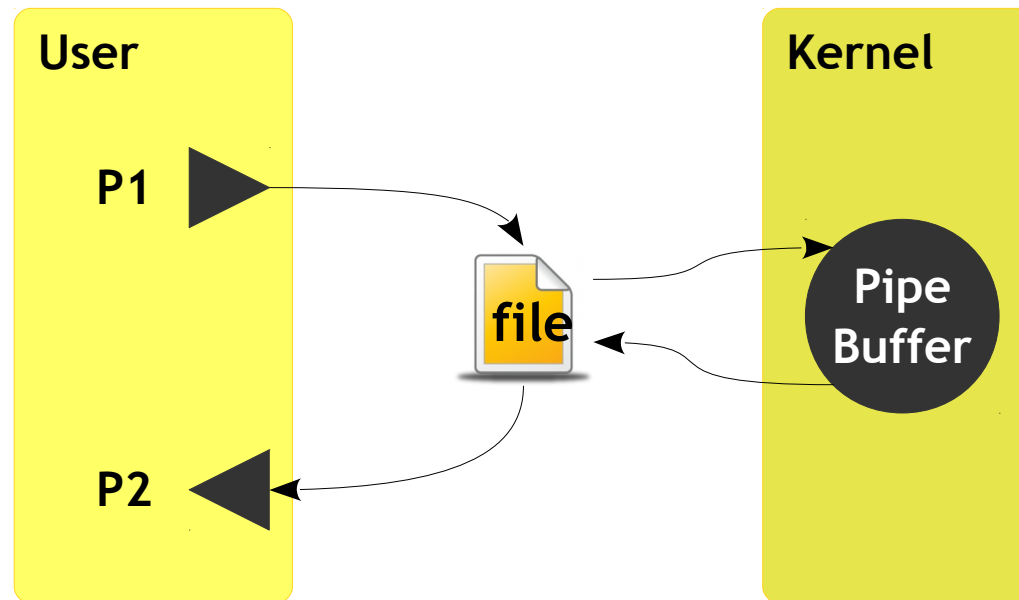
Inter Process Communications
FIFO - Creation

- FIFO can also be created similar to directory/file creation with special parameters & permissions

- After creating FIFO, read & write can be performed into it just like any other normal file

- Finally, a device number is passed. This is ignored when creating a FIFO, so you can put anything you want in there

- Subsequently FIFO can be closed like a file

Function	Meaning
int mknod(const char *path, mode_t mode, dev_t dev)	✓ path: Where the FIFO needs to be created (Ex: "/tmp/Emertxe") ✓ mode: Permission, similar to files (Ex: 0666) ✓ dev: can be zero for FIFO

- Access a FIFO just like an ordinary file

- To communicate through a FIFO, one program must open it for writing, and another program must open it for reading

- Either low-level I/O functions (open, write, read, close and so on) or C library I/O functions (fopen, fprintf, fscanf, fclose, and so on) may be used.

```
user@user:~] ls -l my_fifo
prw-rw-r-- 1 biju biju      0 Mar 8  17:36 my_fifo
```

prw-

Inter Process Communications
FIFO vs Pipes

- Unlike pipes, FIFOs are not temporary objects, they are entities in the file-system

- Any process can open or close the FIFO

- The processes on either end of the pipe need not be related to each other

- When all I/O is done by sharing processes, the named pipe remains in the file system for later use

Inter Process Communications
FIFO - Example

- Unrelated process can communicate with FIFO

Shell 1

```
user@user:~] cat > /tmp/my_fifo
Hai hello
```

Shell 2

```
user@user:~] cat /tmp/my_fifo
Hai hello
```

Inter Process Communications
FIFO - Pros & Cons

PROS

- Naturally synchronized

- Simple to use and create

- Unrelated process can communicate.

- No extra system calls required to communicate (read/write)

- Work like normal file

CONS

- Less memory size (4K)

- Only two process can communicate

- One directional communication

- Kernel is involved

ΣMERTXE

Inter Process Communications
Summary

- We have covered

    Data exchange

    Communication

    - Pipes
    - FIFO
    - Shared memory
    - Signals
    - Sockets

    Resource usage/access/control

    Synchronization

    - Semaphores

Inter Process Communications
Shared Memories - Properties

- Shared memory allows two or more processes to access the same memory

- When one process changes the memory, all the other processes see the modification

- Shared memory is the fastest form of Inter process communication because all processes share the same piece of memory

- It also avoids copying data unnecessarily

Note:
- Each shared memory segment should be explicitly de-allocated
- System has limited number of shared memory segments
- Cleaning up of IPC is system program's responsibility ☺

Inter Process Communications
Shared vs Local Memory
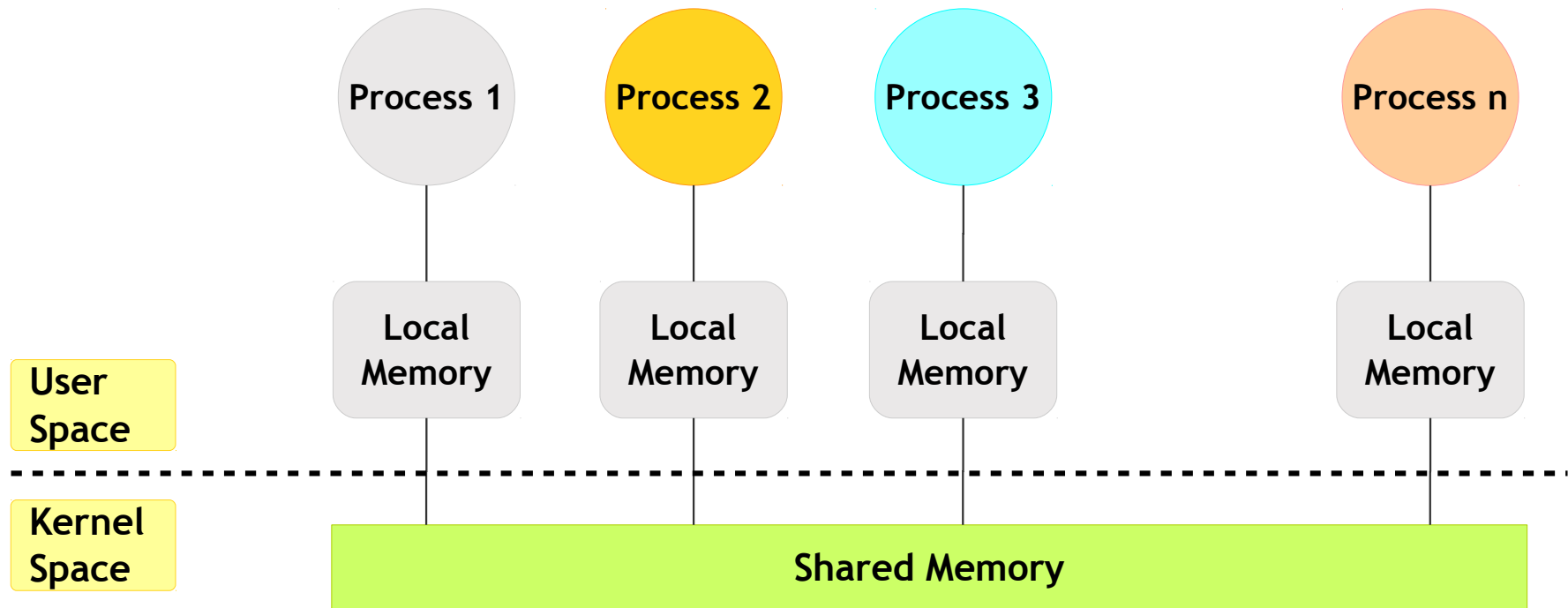
Inter Process Communications
Shared Memories - Procedure

- Create

- Attach

- Read/Write   } **95%**

- Detach

- Remove

EMERTXE

Inter Process Communications
Shared Memories - Procedure

- To start with one process must allocate the segment

- Each process desiring to access the segment must attach to it

- Reading or Writing with shared memory can be done only after attaching into it

- After use each process detaches the segment

- At some point, one process must de-allocate the segment

While shared memory is fastest IPC, it will create synchronization issues as more processes are accessing same piece of memory. Hence it has to be handled separately.

Inter Process Communications
Shared Memories – Function calls

Function	Meaning
int shmget(key_t key, size_t size, int shmflag)	✓ Create a shared memory segment ✓ key: Seed input ✓ size: Size of the shared memory ✓ shmflag: Permission (similar to file) ✓ RETURN: Shared memory ID / Failure
void *shmat(int shmid, void *shmaddr, int shmflag)	✓ Attach to a particular shared memory location ✓ shmid: Shared memory ID to get attached ✓ shmaddr: Exact address (if you know or leave it 0) ✓ shmflag: Leave it as 0 ✓ RETURN: Shared memory address / Failure
int shmdt(void *shmaddr)	✓ Detach from a shared memory location ✓ shmaddr: Location from where it needs to get detached ✓ RETURN: SUCCESS / FAILURE (-1)
shmctl(shmid, IPC_RMID, NULL)	✓ shmid: Shared memory ID ✓ Remove and NULL

Inter Process Communications
Synchronization - Debugging

- The *ipcs* command provides information on inter-process communication facilities, including shared segments.

- Use the -m flag to obtain information about shared memory.

- For example, this image illustrates that one shared memory segment, numbered 392316, is in use:

Semaphores In the system

```
user@user:~] ipcs -s

------ Semaphore Arrays --------
key        semid     owner     perms     nsems
```

Shared Memory in the system

```
user@user:~] ipcs -m | more

------ Shared Memory Segments --------
key          shmid      owner     perms     bytes       nattch     status
0x00000000 393216      user       600        524288      2           dest
0x00000000 557057      user       700        2116         2           dest
0x00000000 589826      user       700        5152         2           dest
```

Inter Process Communications
Summary

- We have covered

Data exchange

Communication

- Pipes
- FIFO
- Shared memory
- Signals
- Sockets

Resource usage/access/control

Synchronization

- Semaphores

Signals

Signals

- Signals are used to notify a process of a particular event

- Signals make the process aware that something has happened in the system

- Target process should perform some pre-defined actions to handle signals

- This is called 'signal handling'

- Actions may range from 'self termination' to 'clean-up'

Get Basics Right
Function pointers

- What is function pointer?

  - Datatype *ptr ;   normal pointer

  - Datatype (*ptr)(datatype,..); Function pointer

- How it differs from normal data pointer?

Function Pointer	Data Pointer
• Holds address of function	• Holds address of an object
• Pointing to a address from code segment.	• Pointing to a address from stack/heap/data
• Dereference to execute the function	• Dereference to get value from address
• Pointer arithmetic not valid	• Pointer arithmetic is valid

ΣMERTXE

Get Basics Right
Call back functions
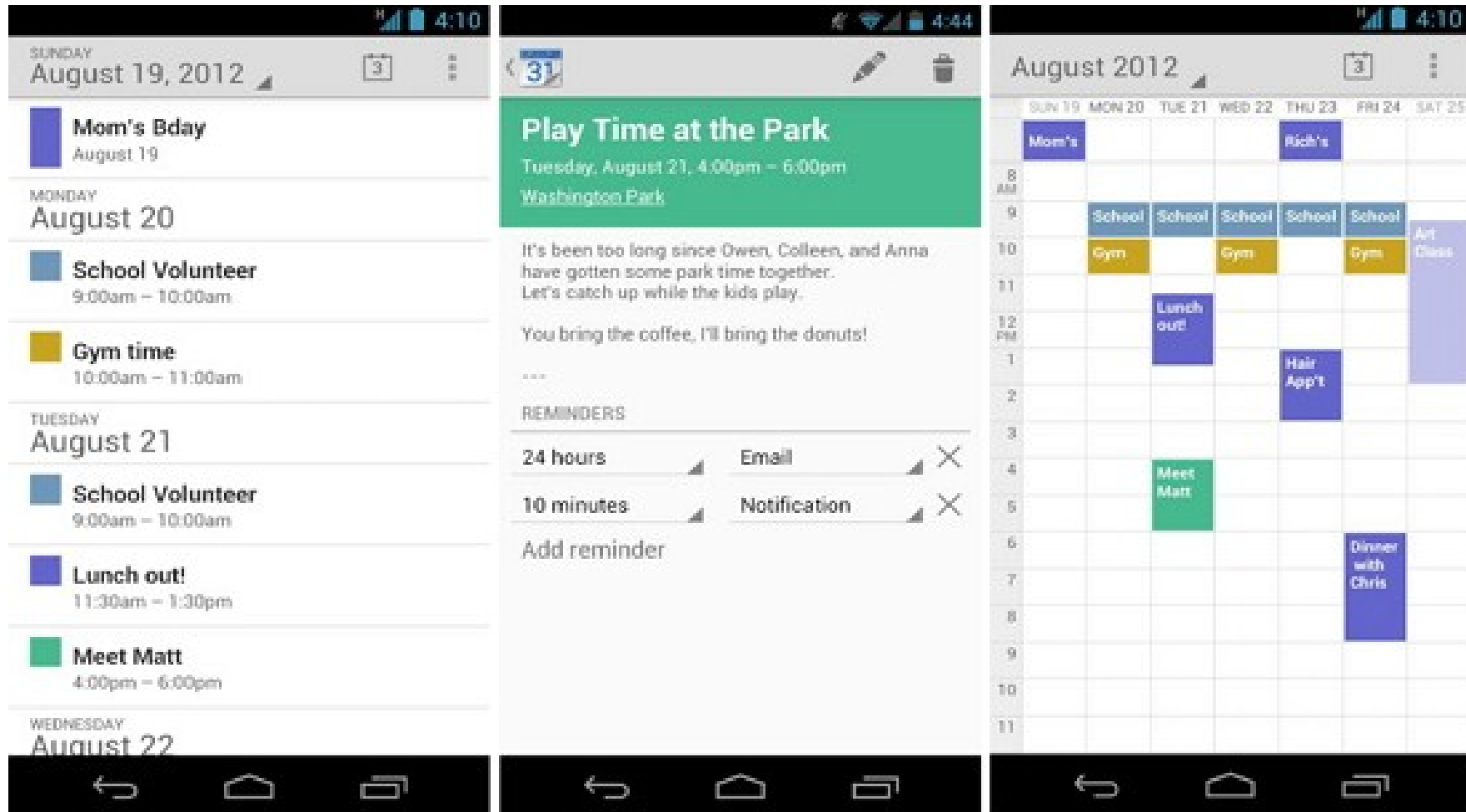
Registering an event for later use

Get Basics Right
Call back functions

- In computer programming, a callback is a reference to executable code, or a piece of executable code, that is passed as an argument to other code. This allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer.

ƩMERTXE

Signals
Names

- Signals are standard, which are pre-defined

- Each one of them have a name and number

- Examples are follows:

Signal name	Number	Description
SIGINT	2	Interrupt character typed
SIGQUIT	3	Quit character typed (^\)
SIGKILL	9	Kill -9 was executed
SIGSEGV	11	Invalid memory reference
SIGUSR1	10	User defined signal
SIGUSR2	12	User defined signal

To get complete signals list, open /usr/include/bits/signum.h in your system.

Signals
Origins

- The kernel

- A Process may also send a Signal to another Process

- A Process may also send a Signal to itself

- User can generate signals from command prompt:

  'kill' command:

  ```
  $ kill <signal_number> <target_pid>
  $ kill -KILL 4481
  ```

  Sends kill signal to PID 4481

  ```
  $ kill -USR1 4481
  ```

  Sends user signal to PID 4481

Signals
Handling

- When a process receives a signal, it processes

- Immediate handling

- For all possible signals, the system defines a default disposition or action to take when a signal occurs

- There are four possible default dispositions:

  - Exit: Forces process to exit

  - Core: Forces process to exit and create a core file

  - Stop: Stops the process

  - Ignore: Ignores the signal

- Handling can be done, called 'signal handling'

Signals
Handling

- The signal() function can be called by the user for capturing signals and handling them accordingly

- First the program should register for interested signal(s)

- Upon catching signals corresponding handling can be done

Function	Meaning
signal (int signal_number, void *(fptr) (int))	signal_number : Interested signal fptr: Function to call when signal handles

Signals
Handler
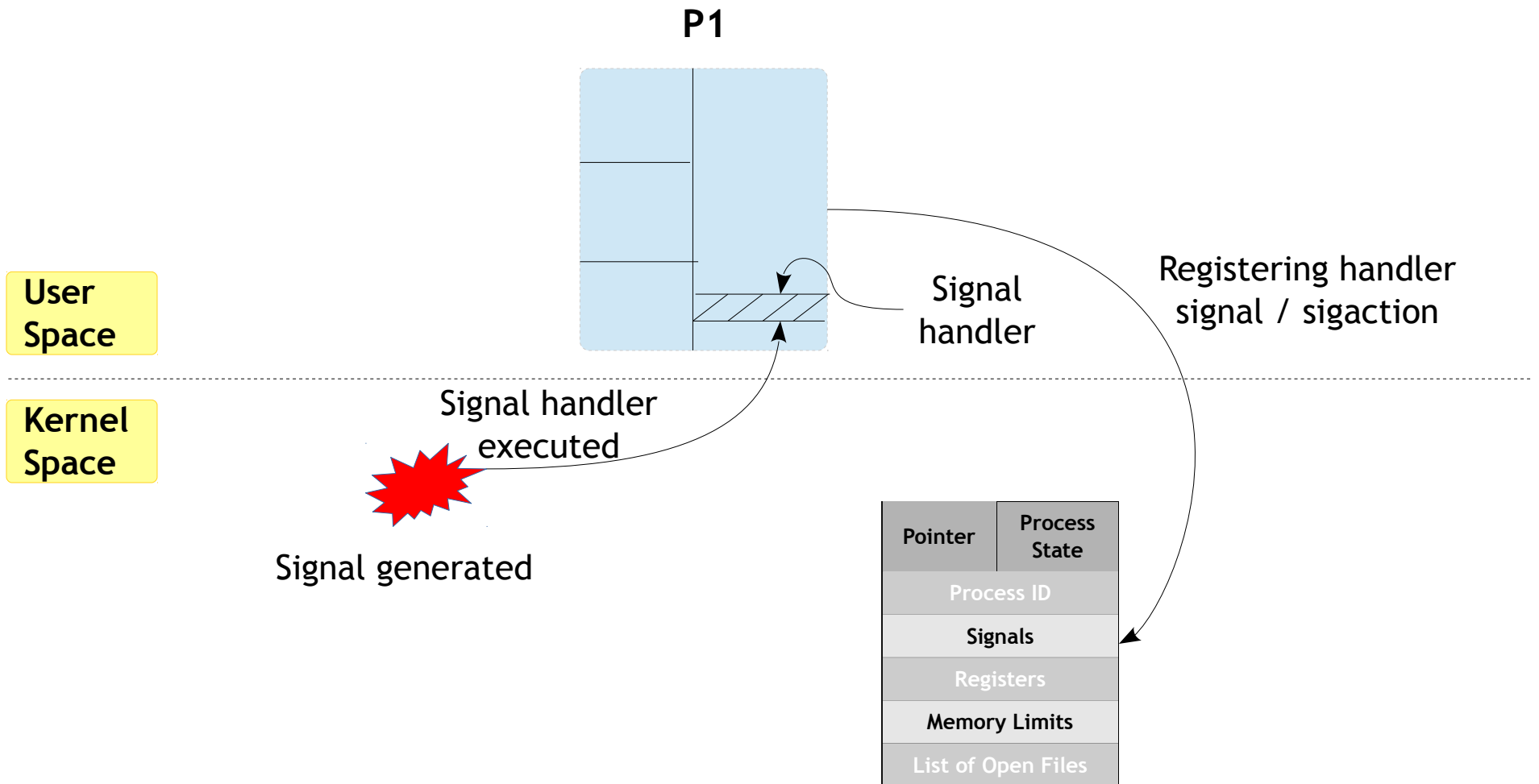
- A signal handler should perform the minimum work necessary to respond to the signal

- The control will return to the main program (or terminate the program)

- In most cases, this consists simply of recording the fact that a signal occurred or some minimal handling

- The main program then checks periodically whether a signal has occurred and reacts accordingly

- Its called as **asynchronous handling**

Signals
vs Interrupt

- Signals can be described as soft-interrupts

- The concept of 'signals' and 'signals handling' is analogous to that of the 'interrupt' handling done by a microprocessor

- When a signal is sent to a process or thread, a signal handler may be entered

- This is similar to the system entering an interrupt handler

- System calls are also soft-interrupts. They are initiated by applications.
- Signals are also soft-interrupts. Primarily initiated by the Kernel itself.

Signals
Advanced Handling

- The signal() function can be called by the user for capturing signals and handling them accordingly

- It mainly handles user generated signals (ex: SIGUSR1), will not alter default behavior of other signals (ex: SIGINT)

- In order to alter/change actions, sigaction() function to be used

- Any signal except SIGKILL and SIGSTOP can be handled using this

Function	Meaning
sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)	signum : Signal number that needs to be handled act: Action on signal oldact: Older action on signal

Signals
Advanced Handling – sigaction structure

```
struct sigaction
{
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

- sa_handler: SIG_DFL (default handling) or SIG_IGN (Ignore) or Signal handler function for handling
- Masking and flags are slightly advanced fields
- Try out sa_sigaction during assignments/hands-on session along with Masking & Flags

Signals
vs system calls

P2   Sending Signal   P1

User Space

Kernel Space

System call
S/w interrupt from
U/S to K/S

Signals
S/w interrupt from
K/S to U/S

Synchronous & Asynchronous

- Wait for child to finish

Synchronous	Asynchronous
Polling	Interrupts
sleep	wait
sleep	Pause

EMERTXE
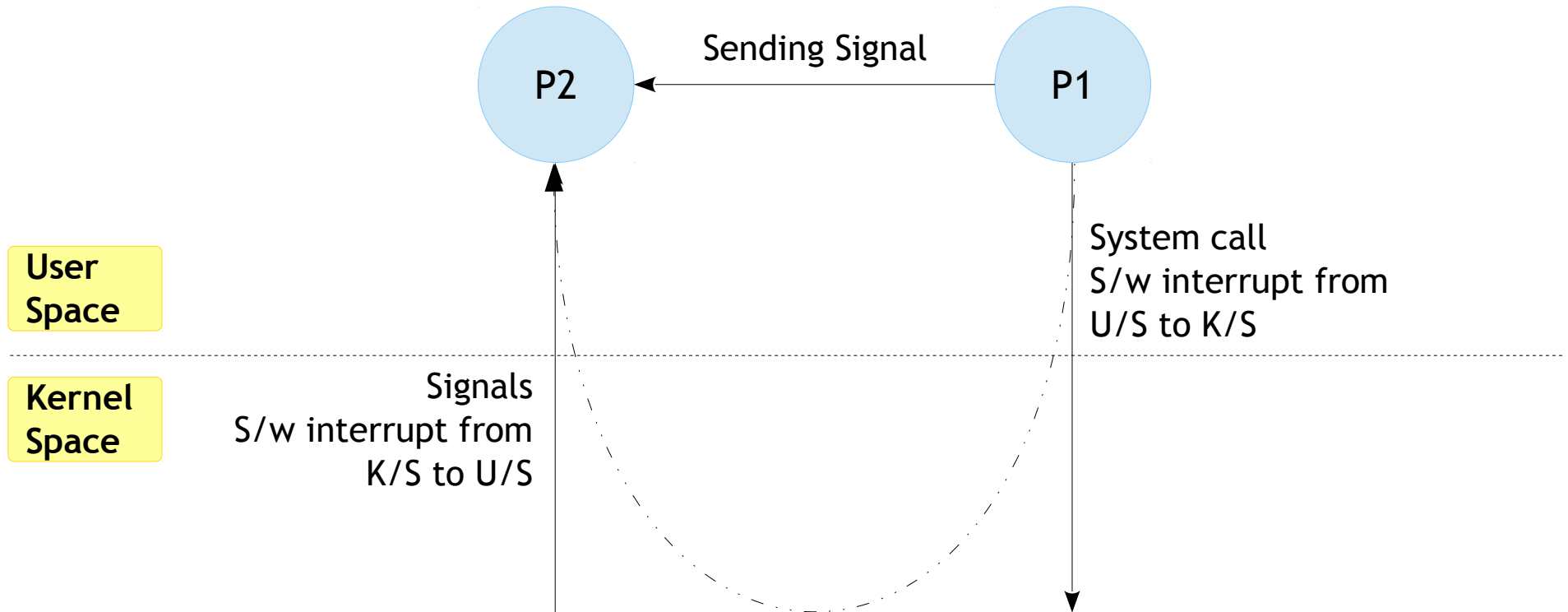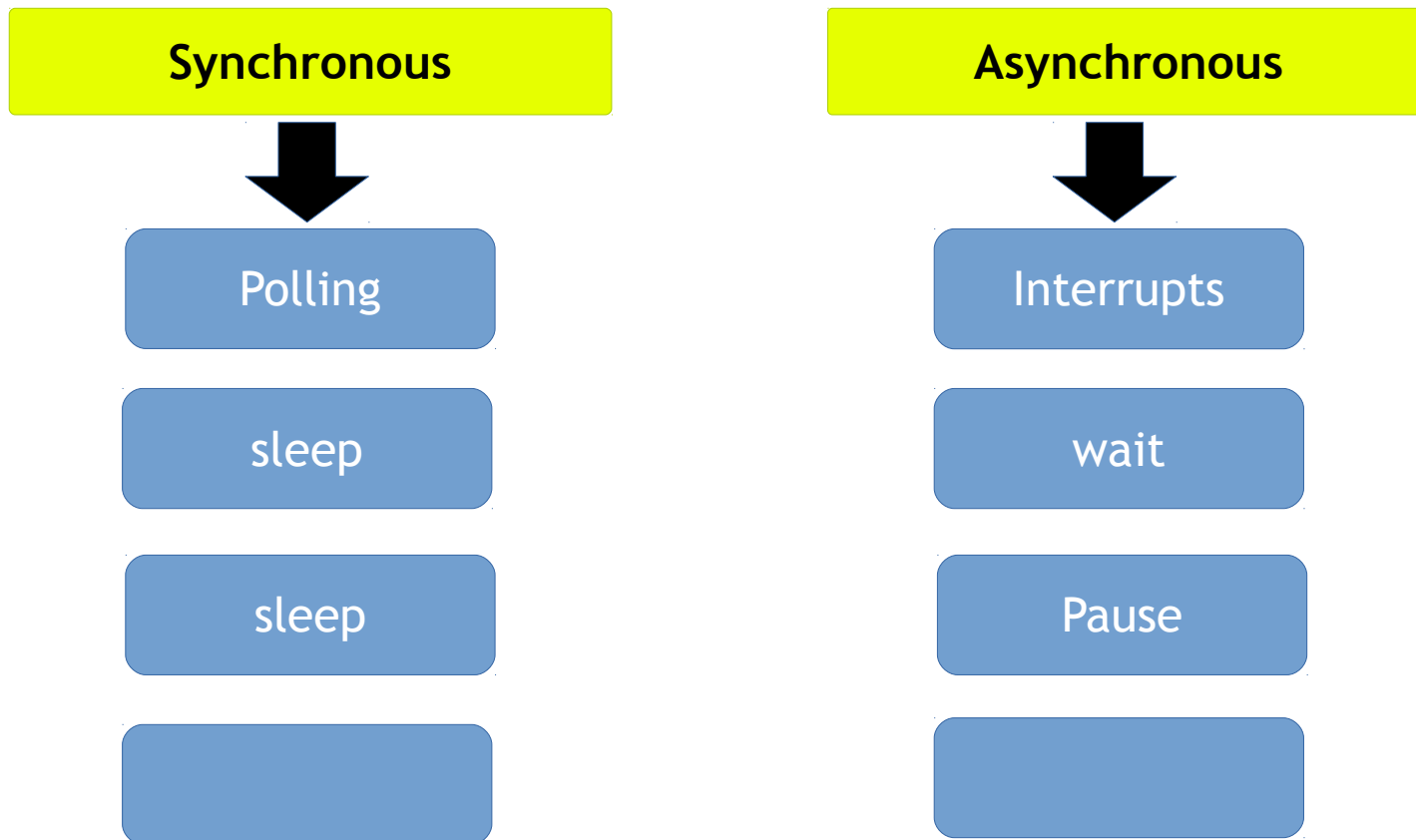
Signals
Self Signaling

- A process can send or detect signals to itself

- This is another method of sending signals

- There are three functions available for this purpose

- This is another method, apart from 'kill'

Function	Meaning
raise (int sig)	Raise a signal to currently executing process. Takes signal number as input
alarm (int sec)	Sends an alarm signal (SIGALRM) to currently executing process after specified number of seconds
pause()	Suspends the current process until expected signal is received. This is much better way to handle signals than sleep, which is a crude approach

Inter Process Communications
Summary

- We have covered

Data exchange

Communication

- Pipes
- FIFO
- Shared memory
- Signals
- Sockets

Resource usage/access/control

Synchronization

- Semaphores

EMERTXE

Networking Fundamentals

Networking Fundamentals
Introduction

- Networking technology is key behind today's success of Internet

- Different type of devices, networks, services work together

- Transmit data, voice, video to provide best in class communication

- Client-server approach in a scaled manner towards in Internet

- Started with military remote communication

- Evolved as standards and protocols

> **Organizations like IEEE, IETF, ITU etc…work together in creating global standards for interoperability and compliance**

ƐMERTXE

Networking Fundamentals
TCP / IP Model
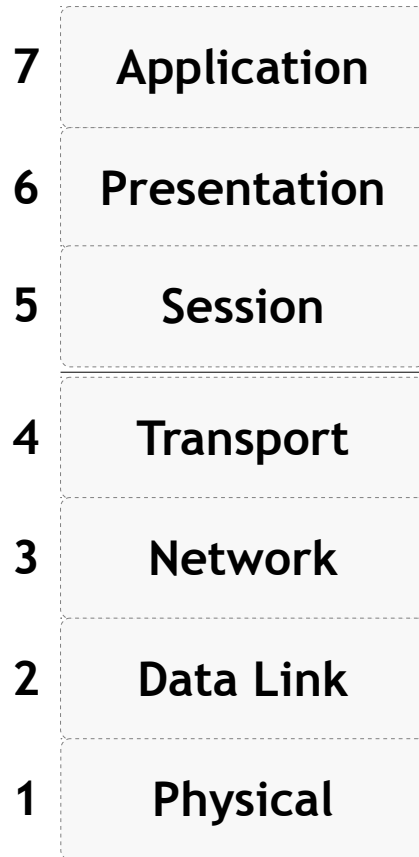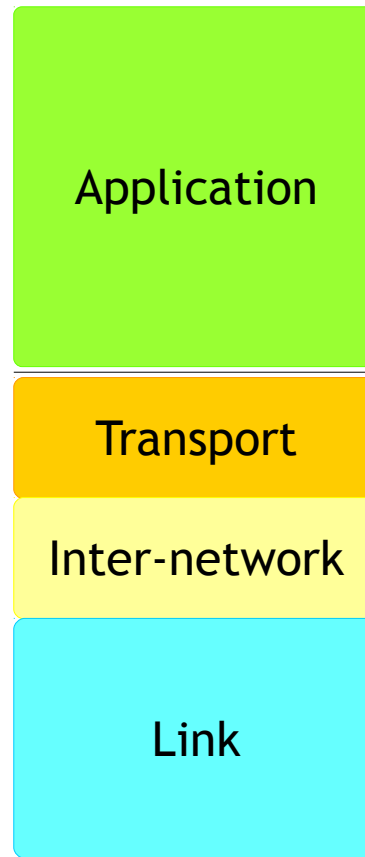
	OSI Model		TCP / IP Protocol Layers		Internet Protocol Suite	

OSI Model

7 Application
6 Presentation
5 Session
4 Transport
3 Network
2 Data Link
1 Physical

TCP / IP Protocol Layers

Application
Transport
Inter-network
Link

Internet Protocol Suite

Telnet	DNS
FTP	RIP
SMTP	SNMP

| TCP | UDP |

| IP | ARP | ICMP | IGMP |

| Ethernet | Token Ring |
| ATM | Wireless |

ΣMERTXE
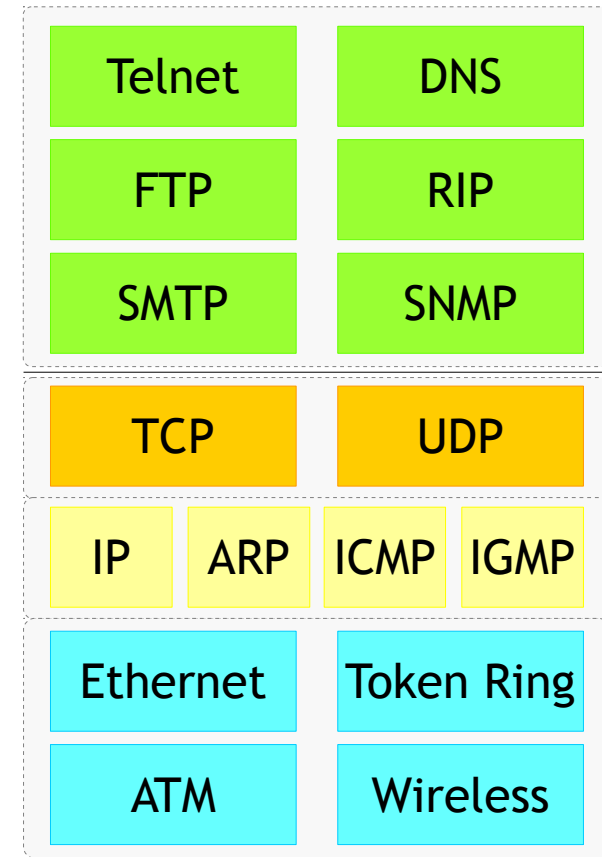
Networking Fundamentals
TCP / IP Model – Implementation in Linux

OSI Model

7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data Link
1	Physical

Internet Protocol Suite

Application

TCP | UDP

IPv4, IPv6

Device Drivers And Hardware

socket XTI

Application details

User process

kernel

Communications details
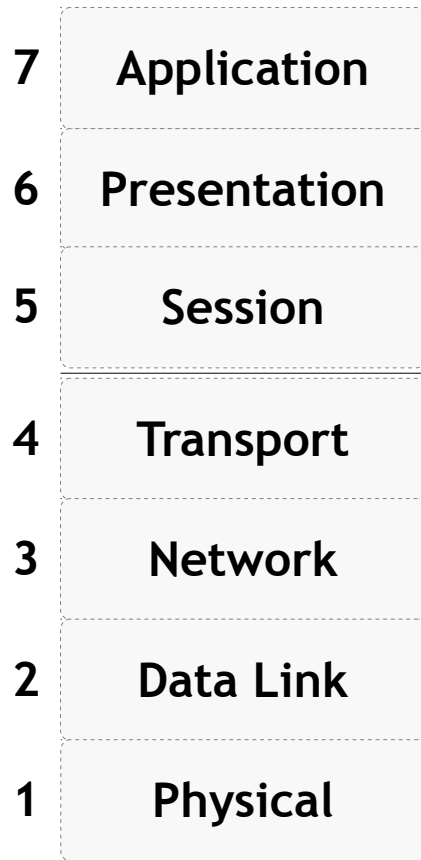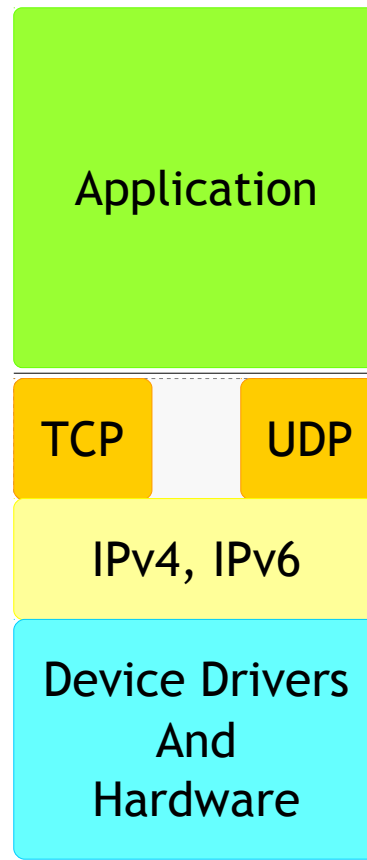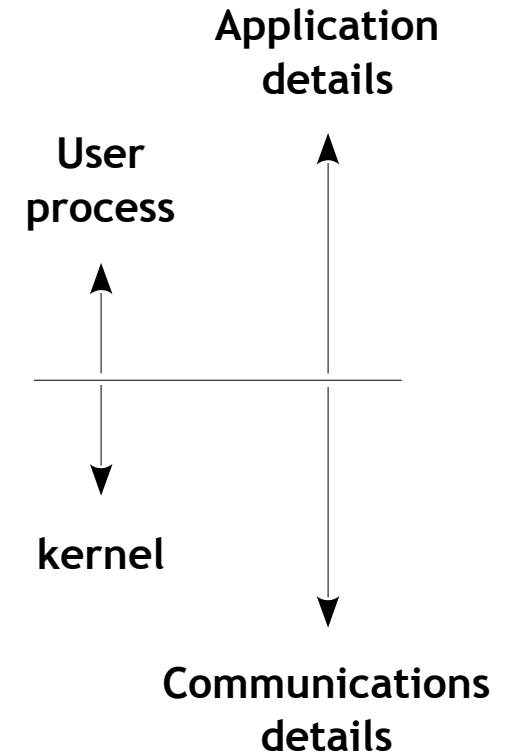
Networking Fundamentals
Protocols

User
process

Web
client

application protocol

Web
server

Application
Layer

Protocol
stack
within
kernel

TCP

TCP protocol

TCP

Transport
Layer

IP

IP protocol

IP

Network
Layer

Ethernet
driver

Ethernet protocol

Ethernet
driver

Data Link
Layer

actual flow between client and server

Ethernet
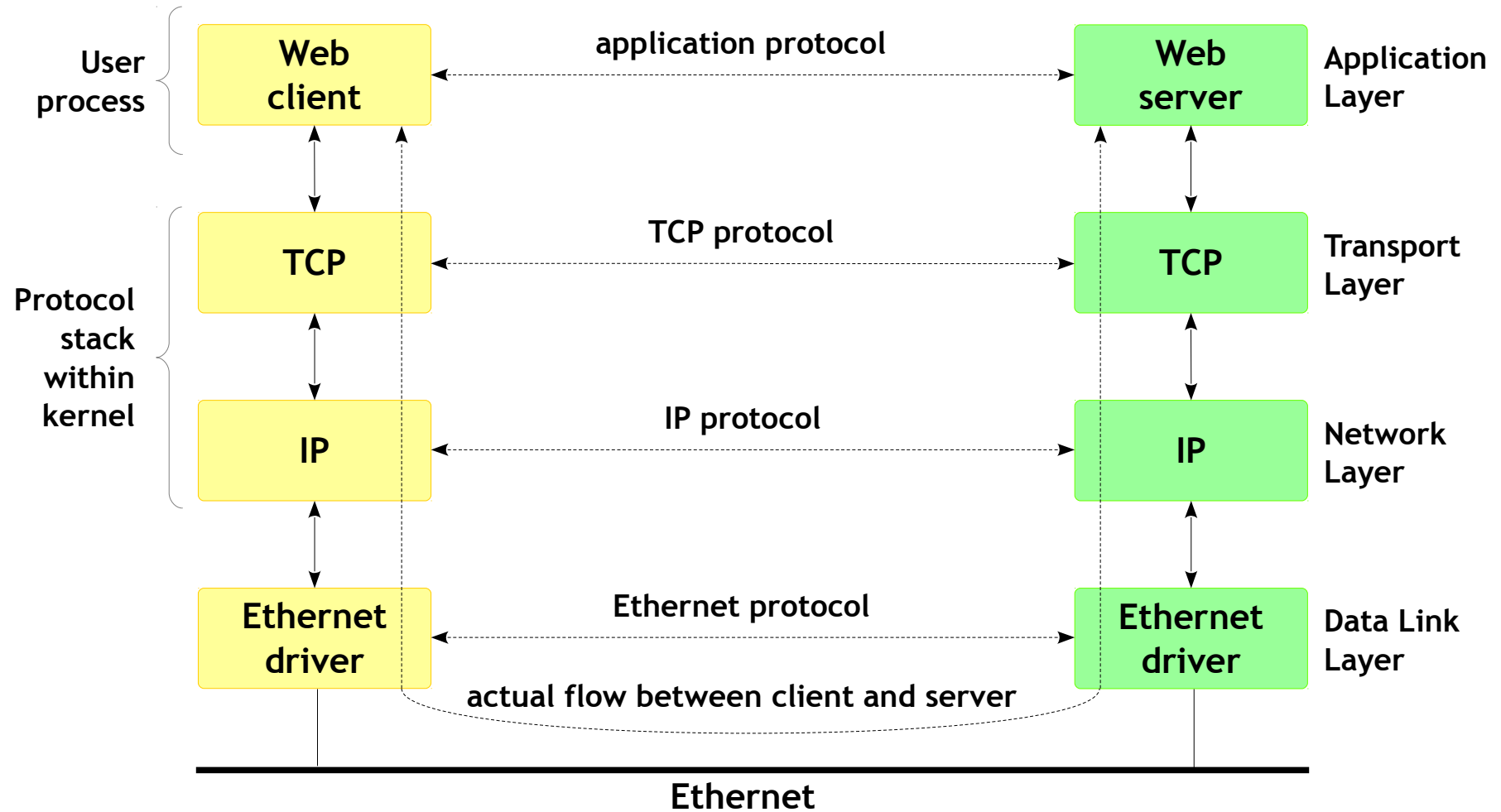
Networking Fundamentals
ATM Protocols stack
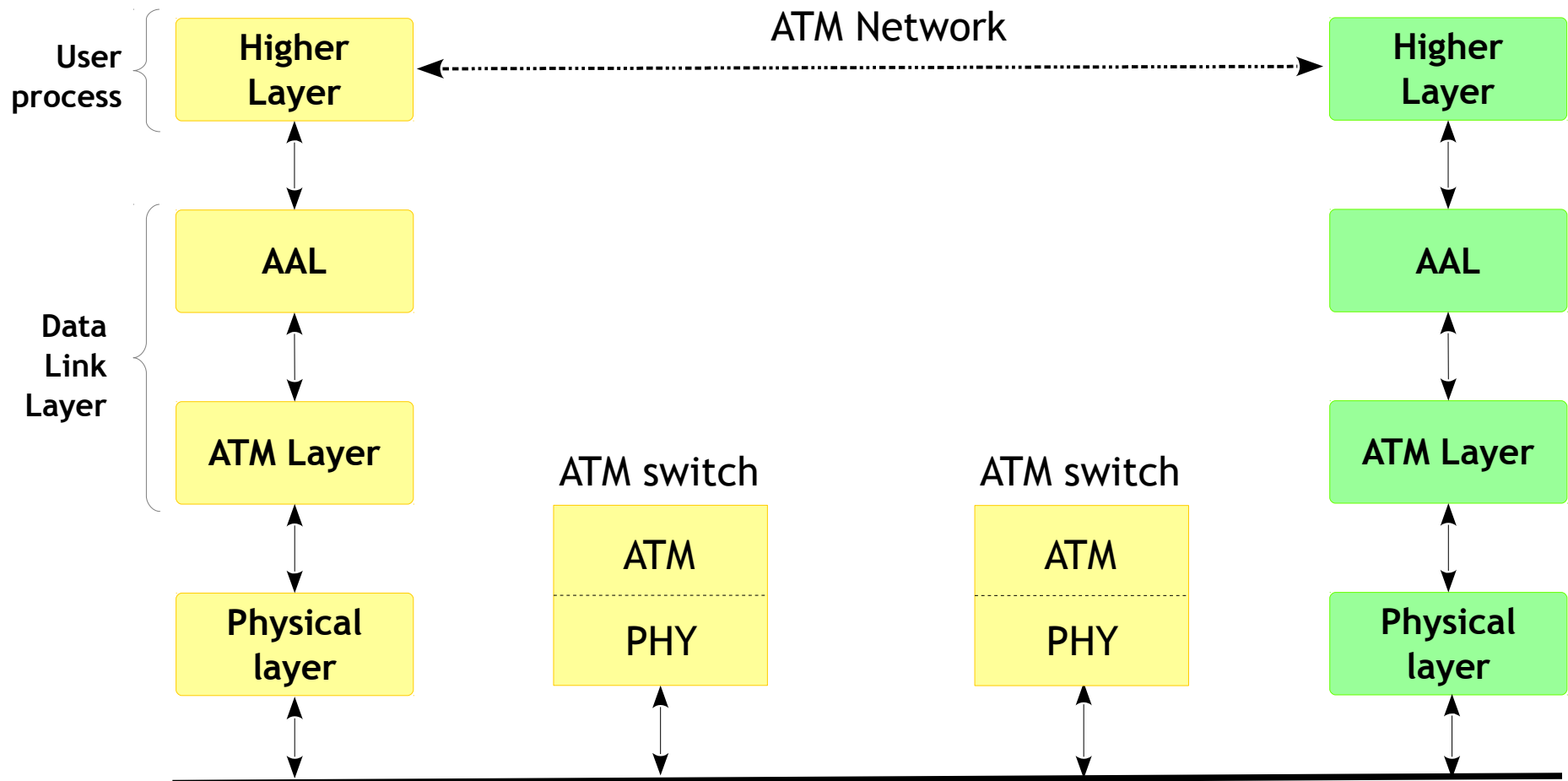
Networking Fundamentals
x.25 Protocols stack

X.25 Network

User process	**Higher Layer**

X.25 Protocol suite

Network Layer	**PLP**

Data Link Layer	**LAPB**

Physical Layer	**EIA/TIA-232, EIA/TIA-449 EIA-530**

Higher Layer

PLP

LAPB

EIA/TIA-232, EIA/TIA-449 EIA-530

EMERTXE
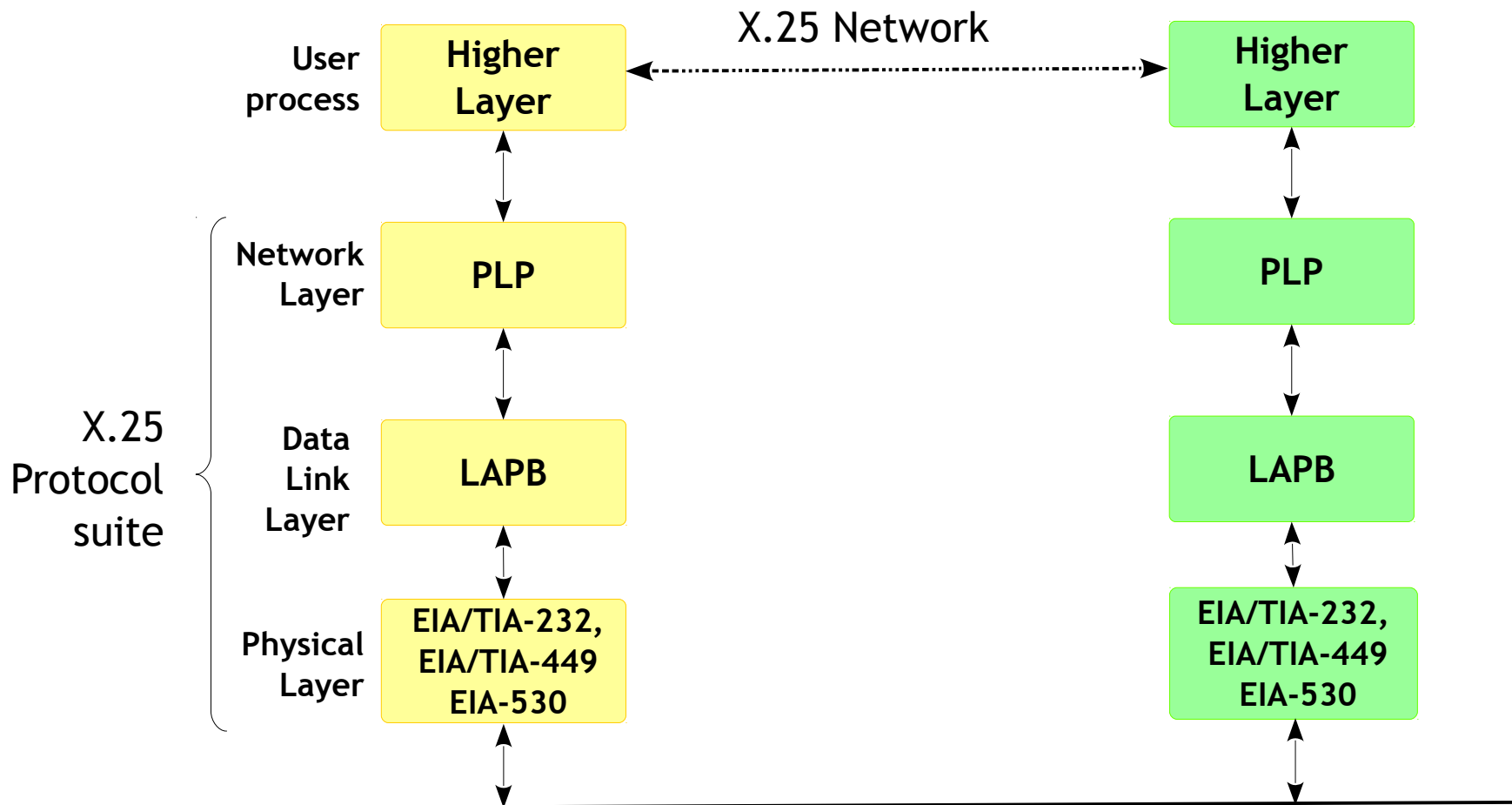
Networking Fundamentals
Addressing

- IP layer: IP address

  - Dotted decimal notation ("192.168.1.10")

  - 32 bit integer is used for actual storage

  - IP address must be unique in a network

  - Two modes IPv4 (32 bits) and IPv6 (128 bits)

  - Total bits divided into two parts

    - Network

    - Host

  - Host part obtained using subnet mask

ΣMERTXE

Networking Fundamentals
IPv4

An IPv4 address (dotted-decimal notation)

172 . 16 . 254 . 1

⬇   ⬇   ⬇   ⬇

10101100.00010000.11111110.00000001
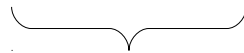
One byte = Eight bits

Thirty-two bits (4 x 8), or 4 bytes

ΣMERTXE

Networking Fundamentals
IPv4 classes

←— 8 Bits —→

A | 0 |

←— Network —→←——————————— Host ———————————→

1.0.0.0 to 127.255.255.255

←——————— 16 Bits ———————→

B | 1 0 |

←————————— Network —————————→←———————— Host ————————→

128.0.0.0 to 191.255.255.255

←————————————— 24 Bits —————————————→

C | 1 1 0 |

←———————————————— Network ————————————————→←—— Host ——→

192.0.0.0 to 223.255.255.255

D | 1 1 1 0 |

←———————————— Multi-cast Address ————————————→

224.0.0.0 to 239.255.255.255

E | 1 1 1 1 |

←———————————— Reserved for future use ————————————→

240.0.0.0 to 255.255.255.255

ƩMERTXE
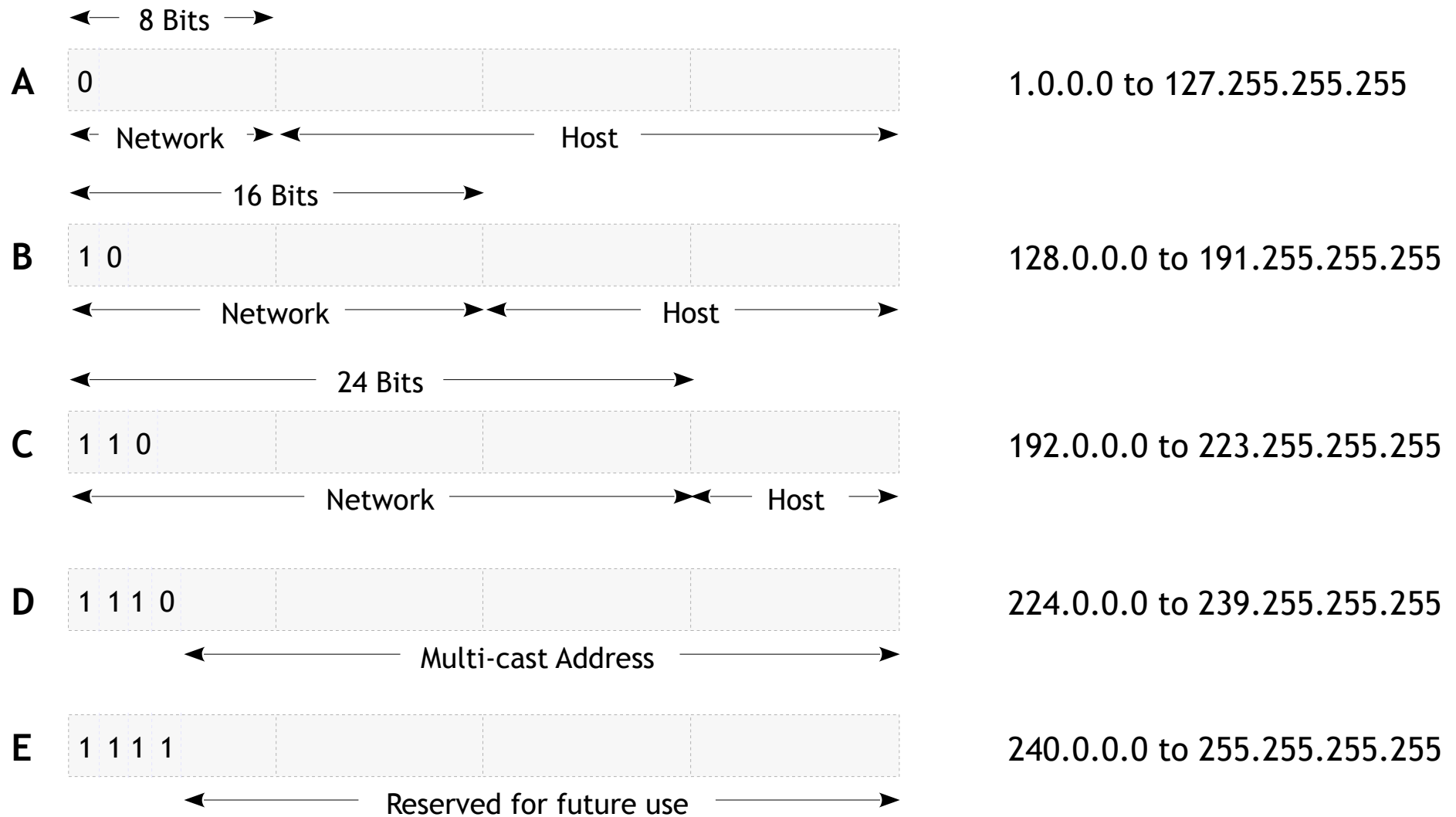
Networking Fundamentals
Ipv6

An IPv6 Address (in Hexadecimal)

2001:0DB8:AC10:FE01:0000:0000:0000:0000

⬇ ⬇ ⬇ ⬇

2001:0DB8:AC10:FE01:: Zero can be omitted

0010000000000001:0000110110111000:0000010000010000:1111111000000001:
0000000000000000:0000000000000000:0000000000000000:0000000000000000

ΣMERTXE

Networking Fundamentals
ip address and domain name

- Commands related to networking

  - Ifconfig (/sbin/ifconfig) command to find the ip-address of system

  - Ping – To check the connectivity using ICMP protocol

  - Host – To convert domain name to ip-address

    Eg: host emertxe.com

Networking Fundamentals
Ports

ENT

Pedia

FTP

TELNET

General

Ortho

TFTP

SSH

Hospital Reception

Transport Layer

Visitor

Packet

EMERTXE
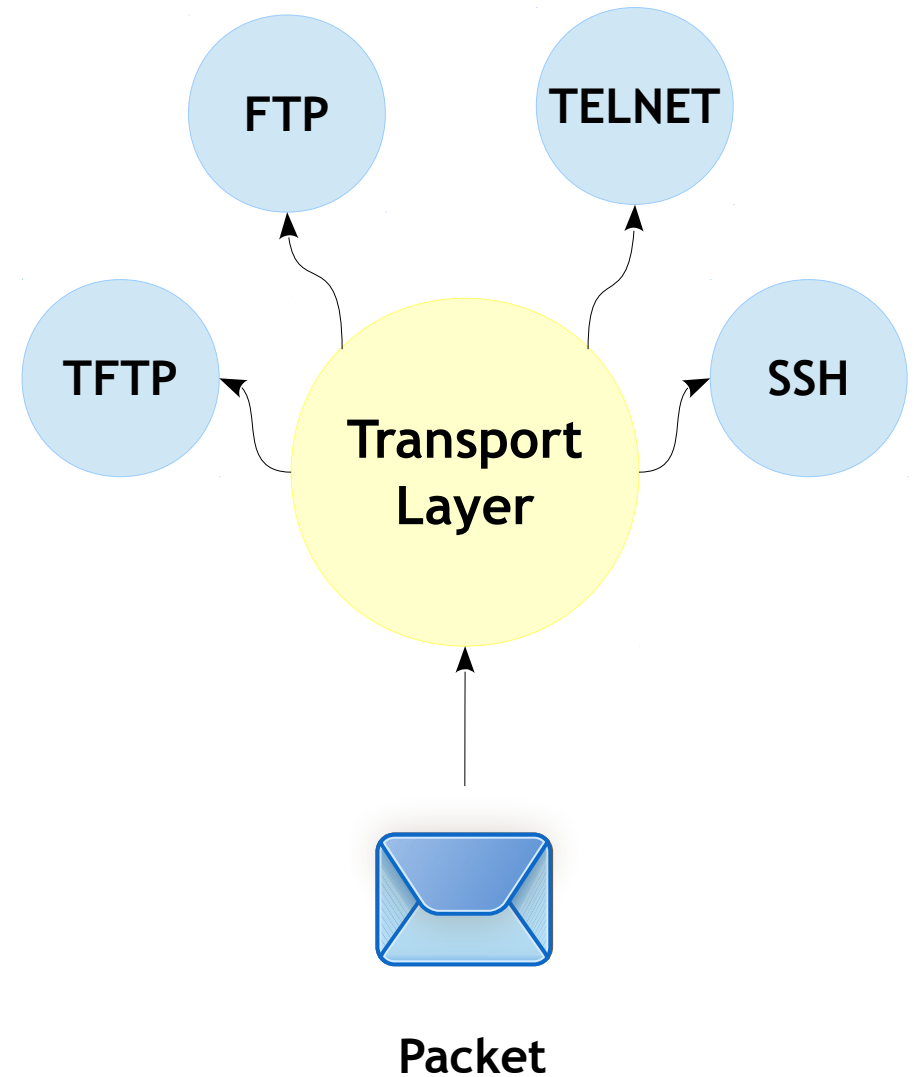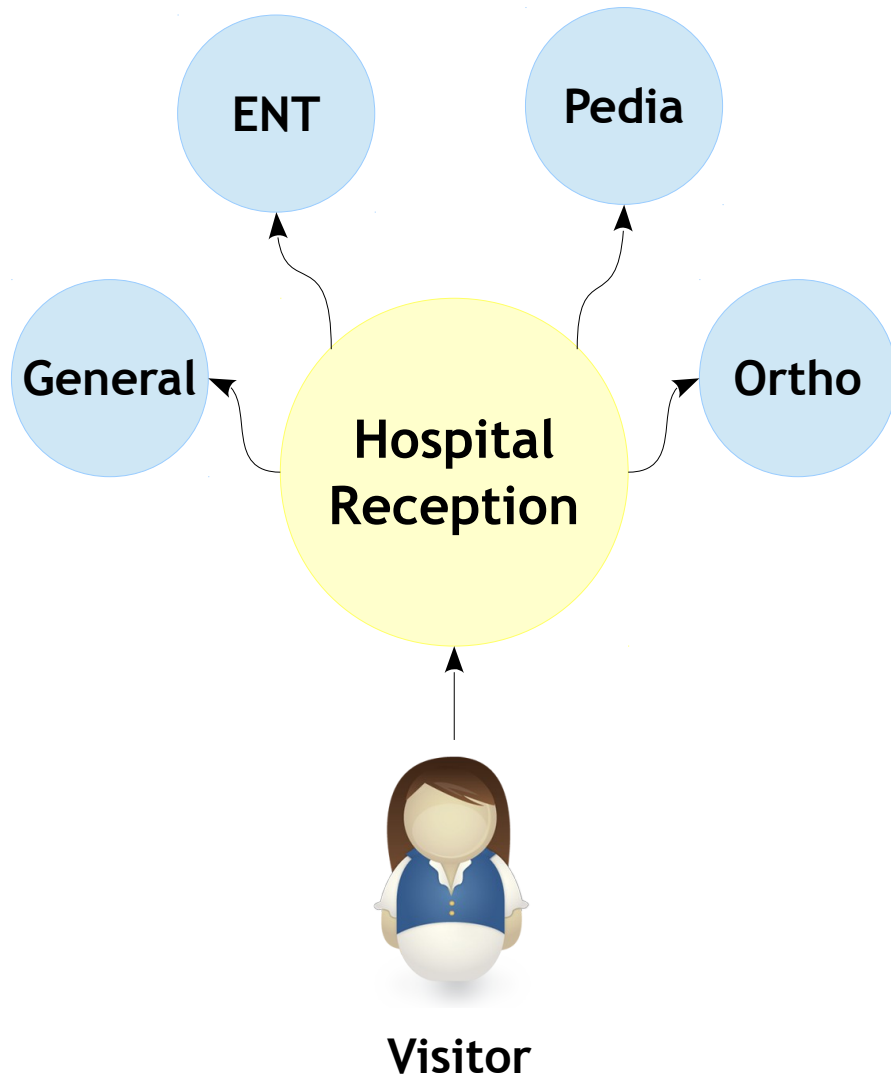
Networking Fundamentals
Ports

- TCP/UDP layer: Port numbers

  - Well known ports [ex: HTTP (80), Telnet (23)]

  - System Ports (0-1023)

  - User Ports (1024-49151)

  - Dynamic and/or Private Ports (49152-65535)

- Port number helps in multiplexing and de-multiplexing the messages

- To see all port numbers used in system by opening a file /etc/services

ΣMERTXE

Networking Fundamentals
Socket as a IPC

Clients

Server

Process ⟷ Socket ⟶ Process

EMERTXE
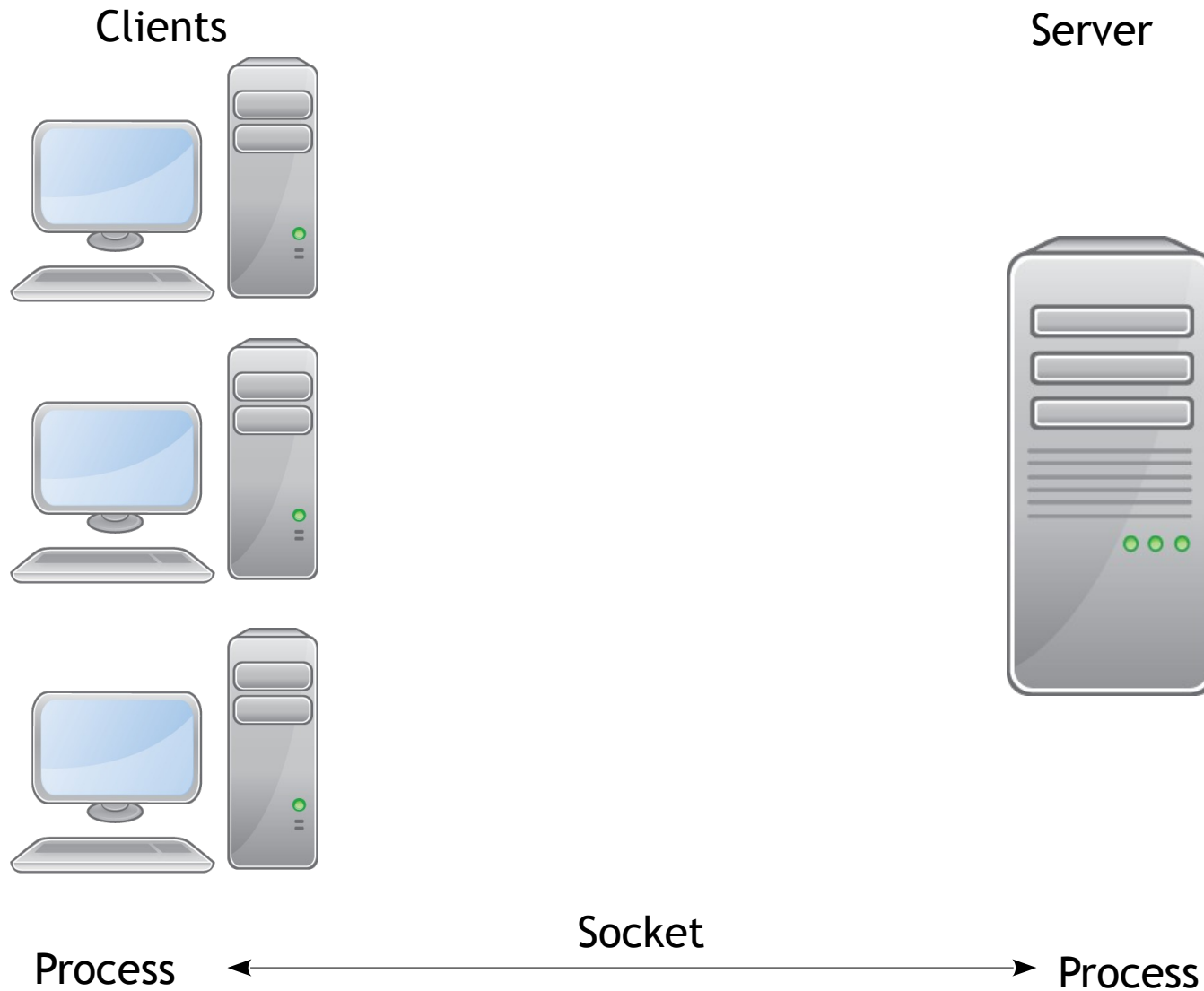
Networking Fundamentals
TCP/IP three way handshake connection

Server

Client

Syn req

Syn req + ACK

ACK

Connection established

ΣMERTXE
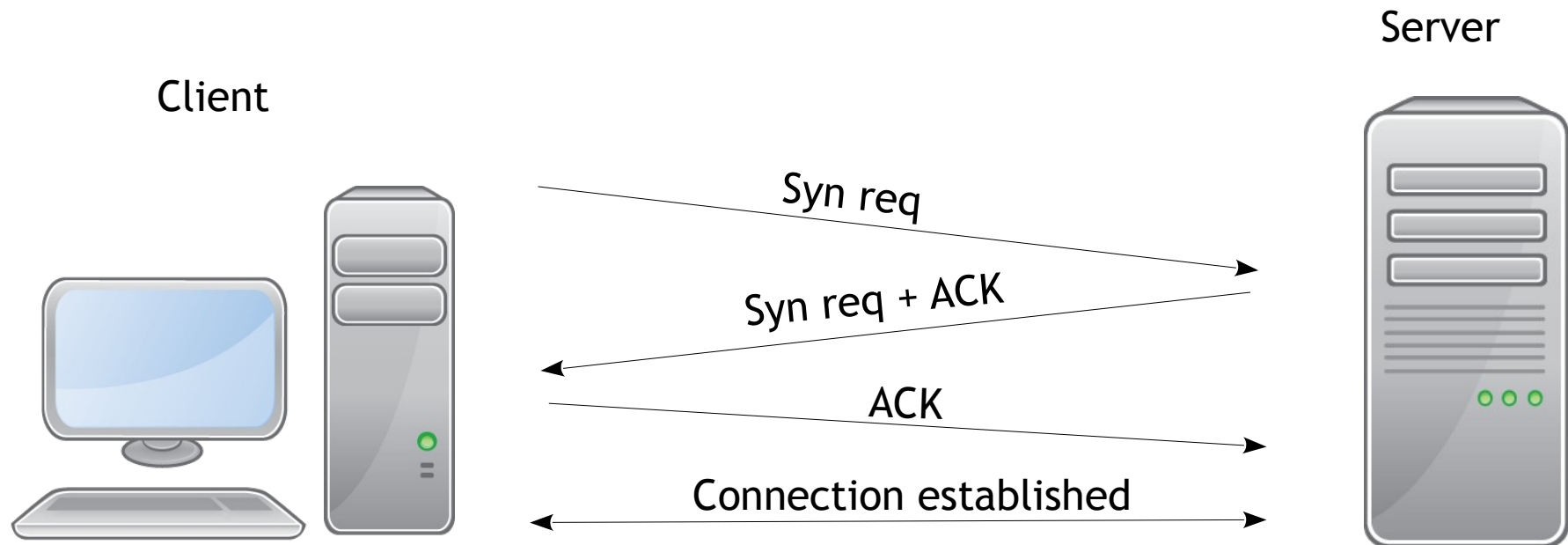
Socket

Sockets

- Sockets is another IPC mechanism, different from other mechanisms as they are used in networking

- Apart from creating sockets, one need to attach them with network parameter (IP address & port) to enable it communicate it over network

- Both client and server side socket needs to be created & connected before communication

- Once the communication is established, sockets provide 'read' and 'write' options similar to other IPC mechanisms

Get Basics Right
Between big endian & little endian

Example

```c
#include <stdio.h>

int main()
{
    int num = 0x12345678;

    return 0;
}
```

- Let us consider the following example and how it would be stored in both machine types

Big Endian

1000			
12	34	56	78

num

$\rightarrow$

1000	1001	1002	1003
12	34	56	78

Little Endian

1000			
78	56	34	12

num

$\rightarrow$

1000	1001	1002	1003
78	56	34	12

EMERTXE
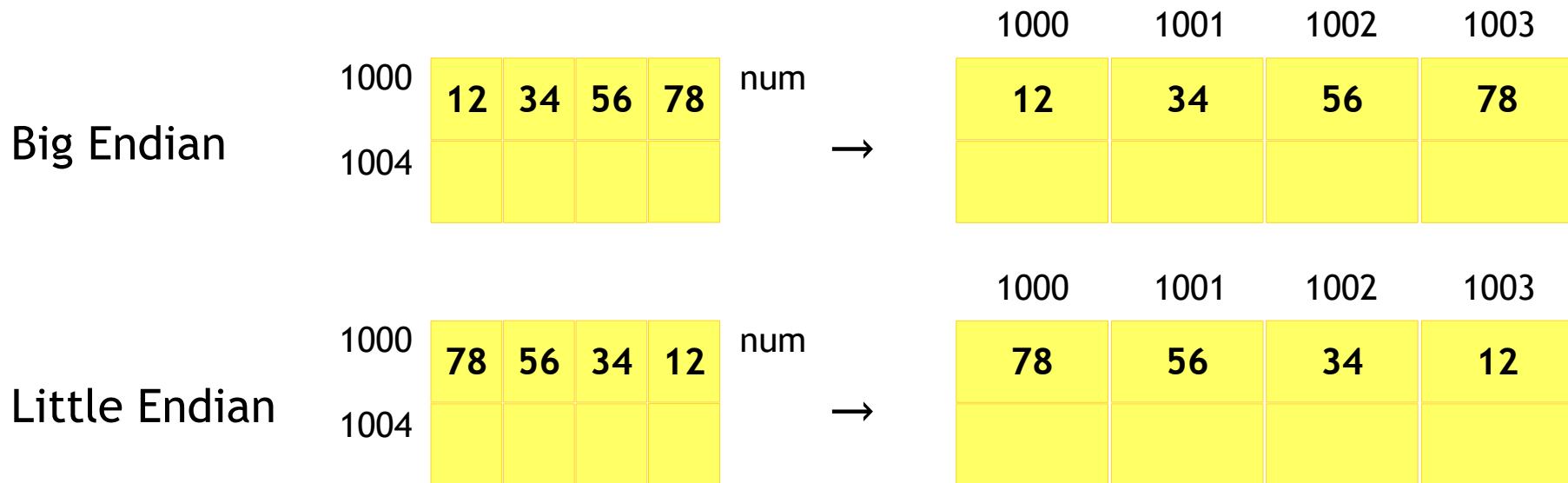
Sockets
Help Functions
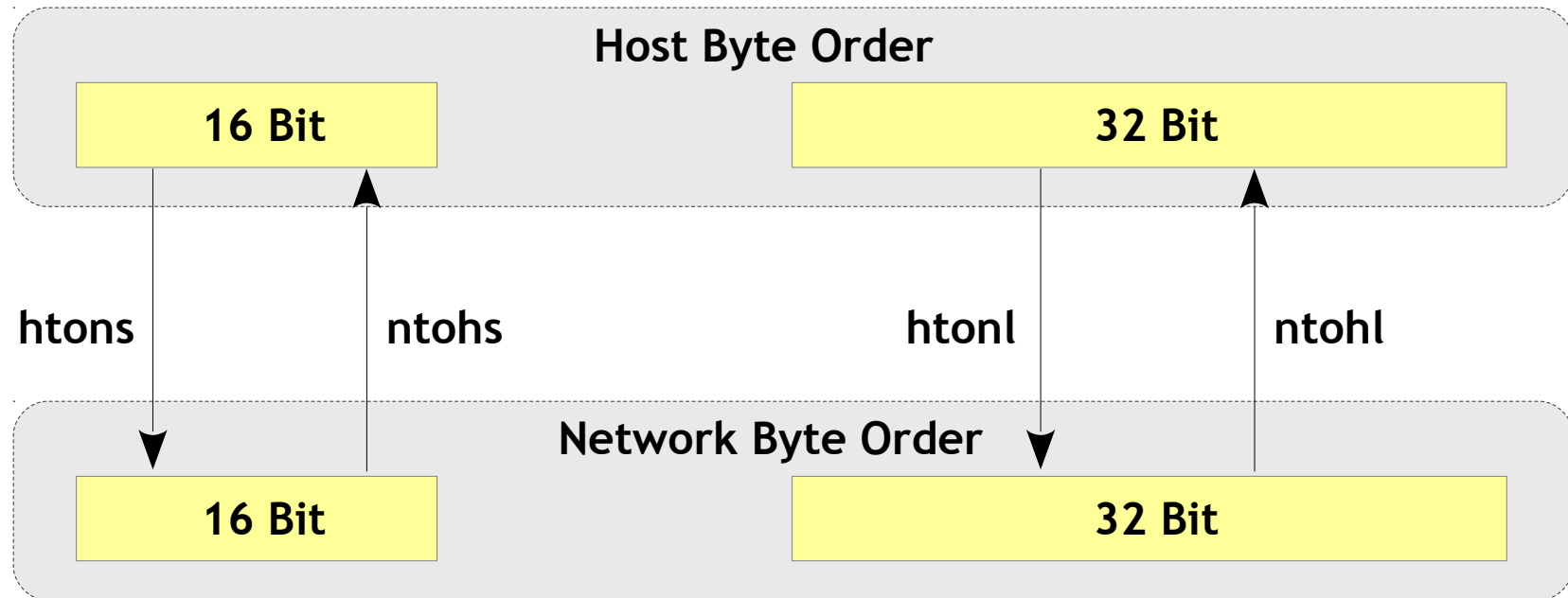

```
uint16_t htons(uint16_t host_short);
uint16_t ntohs(uint16_t network_short);
uint32_t htonl(uint32_t host_long);
uint32_t ntohl(uint32_t network_long);
```

Sockets
Help Functions

- Since machines will have different type of byte orders (little endian v/s big endian), it will create undesired issues in the network

- In order to ensure consistency network (big endian) byte order to be used as a standard

- Any time, any integers are used (IP address, Port number etc..) network byte order to be ensured

- There are multiple help functions (for conversion) available which can be used for this purpose

- Along with that there are some utility functions (ex: converting dotted decimal to hex format) are also available

Sockets
Address

- In order to attach (called as "bind") a socket to network address (IP address & Port number), a structure is provided

- This (nested) structure needs to be appropriately populated

- Incorrect addressing will result in connection failure

```
struct sockaddr_in
{
    short int sin_family;          /* Address family */
    unsigned short int sin_port;   /* Port number */
    struct in_addr sin_addr;       /* IP address structure */
    unsigned char sin_zero[8];     /* Zero value, historical purpose */
};
```

```
/* IP address structure for historical reasons */
struct in_addr
{
    unsigned long s_addr;          /* 32 bit IP address */
};
```

Sockets
Calls - socket

Function	Meaning
int socket(int domain, int type, int protocol)	✓ Create a socket ✓ domain: Address family (AF_INET, AF_UNIX etc..) ✓ type: TCP (SOCK_STREAM) or UDP (SOCK_DGRAM) ✓ protocol: Leave it as 0 ✓ RETURN: Socket ID or Error (-1)

```
Example usage:
sockfd = socket(AF_INET, SOCK_STREAM, 0);           /* Create a TCP socket */
```

Sockets
Calls - bind

Function	Meaning
int bind(int sockfd, struct sockaddr *my_addr, int addrlen)	✓ Bind a socket to network address ✓ sockfd: Socket descriptor ✓ my_addr: Network address (IP address & port number) ✓ addrlen: Length of socket structure ✓ RETURN: Success or Failure (-1)

```
Example usage:
int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(AF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = 3500;
my_addr.sin_addr.s_addr = 0xC0A8010A;    /* 192.168.1.10 */
memset(&(my_addr.sin_zero), '\0', 8);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
```

Sockets
Calls - connect

Function	Meaning
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)	✓ Create to a particular server ✓ sockfd: Client socket descriptor ✓ serv_addr: Server network address ✓ addrlen: Length of socket structure ✓ RETURN: Socket ID or Error (-1)

```
Example usage:
struct sockaddr_in my_addr, serv_addr;

/* Create a TCP socket  & Bind */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = 4500;                      /* Server port */
serv_addr.sin_addr.s_addr = 0xC0A8010B;       /* Server IP = 192.168.1.11 */
```

Sockets
Calls - listen

Function	Meaning
int listen(int sockfd, int backlog)	✓ Prepares socket to accept connection ✓ MUST be used only in the server side ✓ sockfd: Socket descriptor ✓ Backlog: Length of the queue

```
Example usage:
listen (sockfd, 5);
```

Sockets
Calls - accept

Function	Meaning
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)	✓ Accepting a new connection from client ✓ sockfd: Server socket ID ✓ addr: Incoming (client) address ✓ addrlen: Length of socket structure ✓ RETURN: New socket ID or Error (-1)

```
Example usage:
new_sockfd = accept(sockfd,&client_address, &client_address_length);
```

- The accept() returns a new socket ID, mainly to separate control and data sockets
- By having this servers become concurrent
- Further concurrency is achieved by fork() system call

Sockets
Calls – recv

Function	Meaning
int recv (int sockfd, void *buf, int len, int flags)	✓ Receive data through a socket ✓ sockfd: Socket ID ✓ msg: Message buffer pointer ✓ len: Length of the buffer ✓ flags: Mark it as 0 ✓ RETURN: Number of bytes actually sent or Error(-1)

Sockets
Calls – send

Function	Meaning
int send(int sockfd, const void *msg, int len, int flags)	✓ Send data through a socket ✓ sockfd: Socket ID ✓ msg: Message buffer pointer ✓ len: Length of the buffer ✓ flags: Mark it as 0 ✓ RETURN: Number of bytes actually sent or Error(-1)

Sockets
Calls – close

Function	Meaning
close (int sockfd)	✓ Close socket data connection ✓ sockfd: Socket ID

Sockets
TCP - Summary

Server

| socket |
| bind |
| listen |
| accept |
| recv |
| send |
| close |

Client

| socket |
| connect |
| send |
| recv |
| close |

Connection establishment

Data (request)

Data (reply)
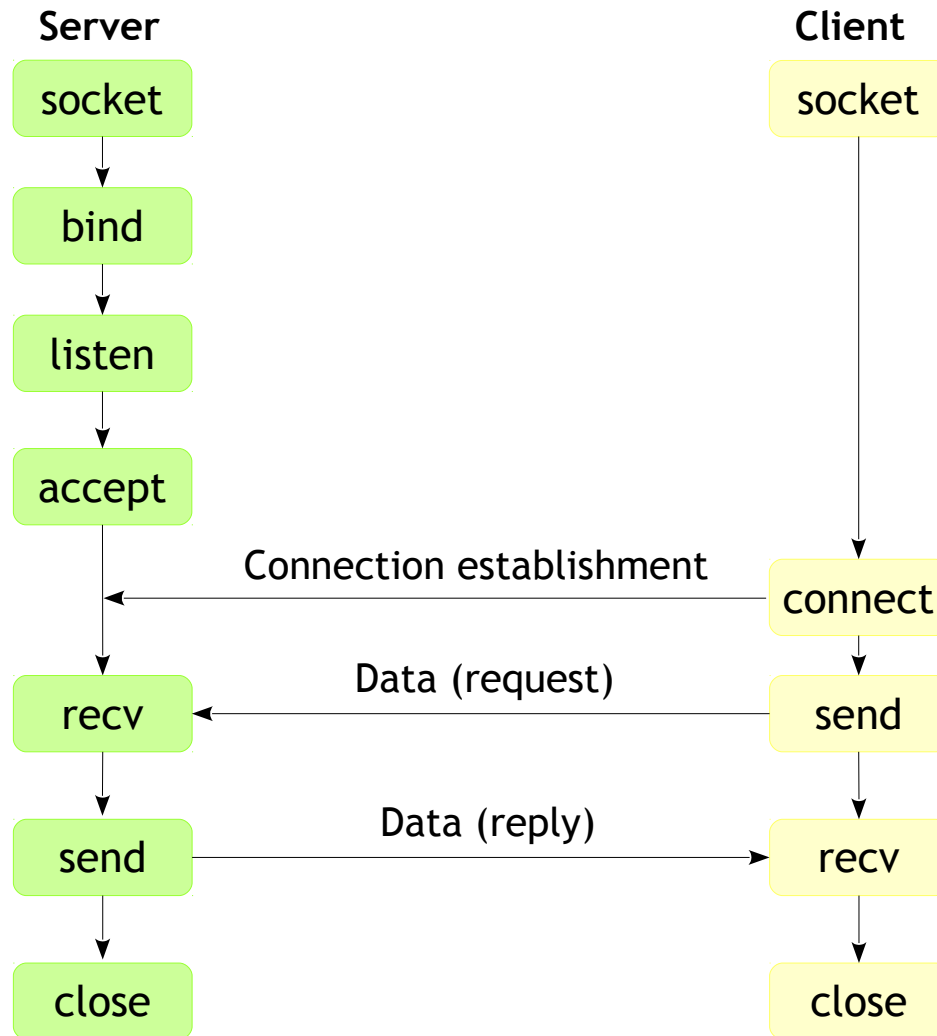
NOTE: Bind() – call is optional from client side

ΣMERTXE

Sockets
TCP vs UDP

TCP socket (SOCK_STREAM)	UDP socket (SOCK_DGRAM)
• Connection oriented TCP • Reliable delivery • In-order guaranteed • Three way handshake • More network BW	• Connectionless UDP • Unreliable delivery • No-order guarantees • No notion of "connection" • Less network BW

Sockets
UDP

Server

socket

↓

bind

↓

recvfrom

↓

sendto

Client

socket

↓

sendto

↓

recvfrom

↓

close
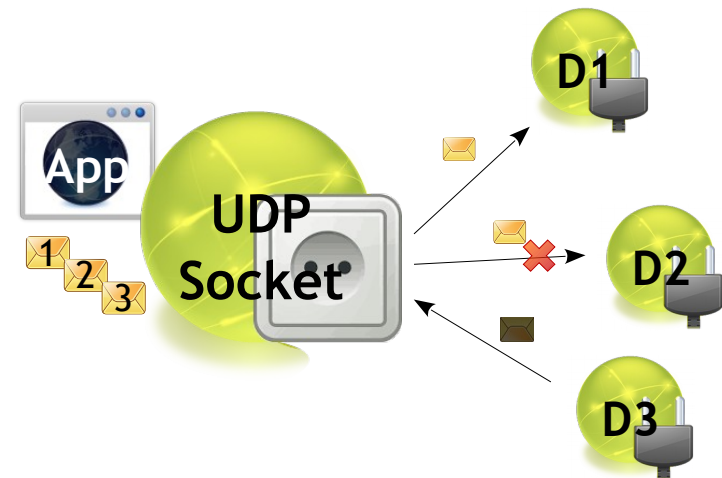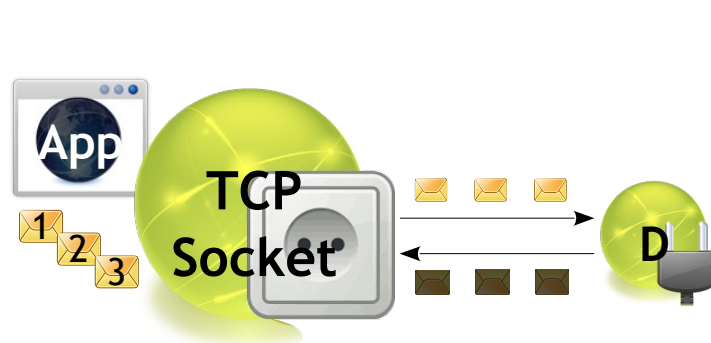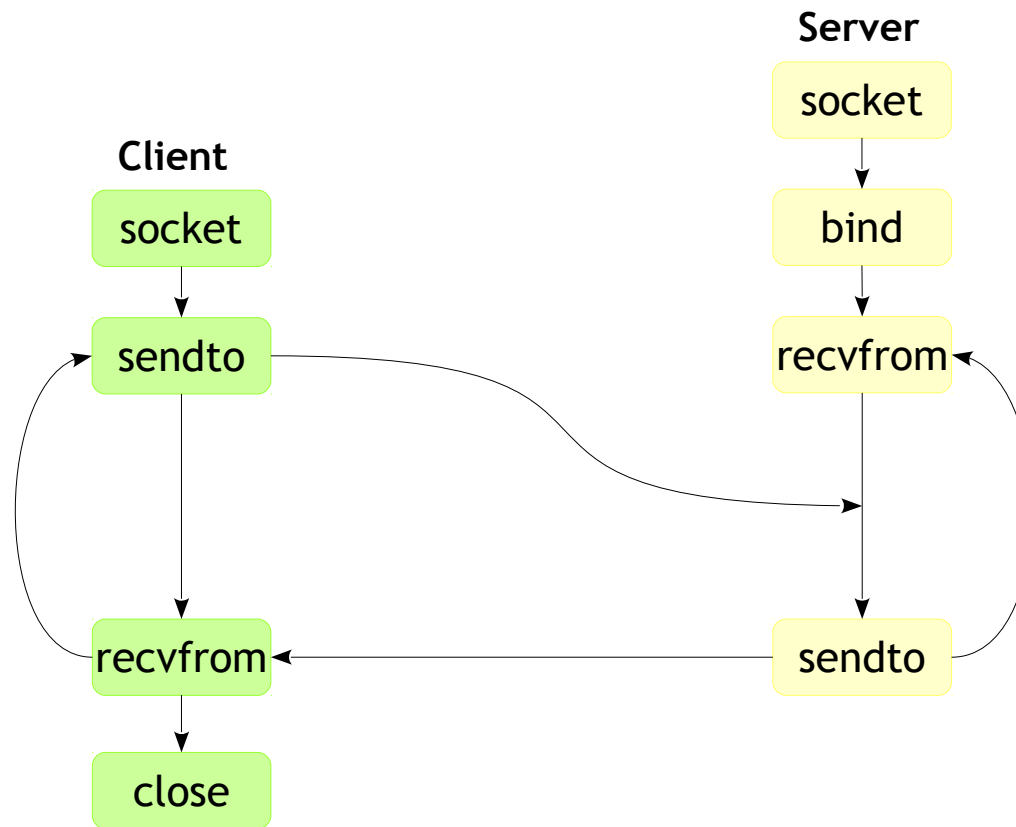
Each UDP data packet need to be addressed separately. sendto()
and recvfrom() calls are used

Sockets
UDP – Functions calls

Function	Meaning
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, socklen_t length);	✓ Send data through a UDP socket ✓ sockfd: Socket ID ✓ msg: Message buffer pointer ✓ len: Length of the buffer ✓ flags: Mark it as 0 ✓ to: Target address populated ✓ length: Length of the socket structure ✓ RETURN: Number of bytes actually sent or Error(-1)
int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *length);	✓ Receive data through a UDP socket ✓ sockfd: Socket ID ✓ buf: Message buffer pointer ✓ len: Length of the buffer ✓ flags: Mark it as 0 ✓ to: Receiver address populated ✓ length: Length of the socket structure ✓ RETURN: Number of bytes actually received or Error(-1)

Client – Server Models

Client – Server Models

- Iterative Model
  - The Listener and Server portion coexist in the same task
  - So no other client can access the service until the current running client finishes its task.

- Concurrent Model
  - The Listener and Server portion run under control of different tasks
  - The Listener task is to accept the connection and invoke the server task
  - Allows higher degree of concurrency

ΣMERTXE

Client – Server Models
Iterative Model – The Flow

- Create a socket

- Bind it to a local address

- Listen (make TCP/IP aware that the socket is available)

- Accept the connection request

- Do data transaction
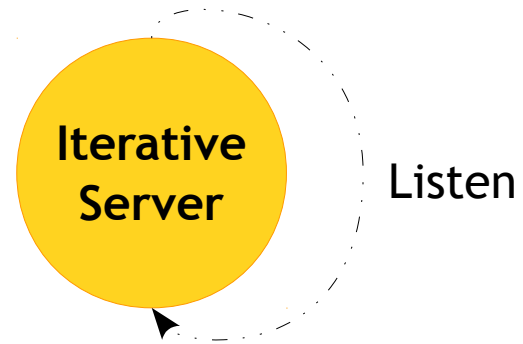
- Close

Iterative Server

Listen

Client – Server Models
Iterative Model – The Flow

Client – Server Models
Iterative Model – The Flow

Client – Server Models
Iterative Model – The Flow

Client – Server Models
Iterative Model – The Flow

Client – Server Models
Iterative Model – The Flow

Client – Server Models
Iterative Model – The Flow

Iterative Server

Listen
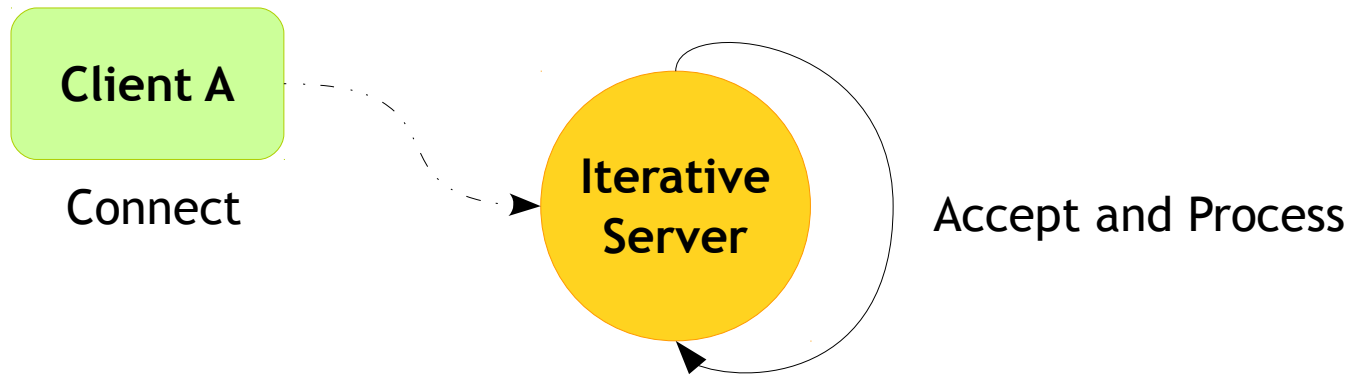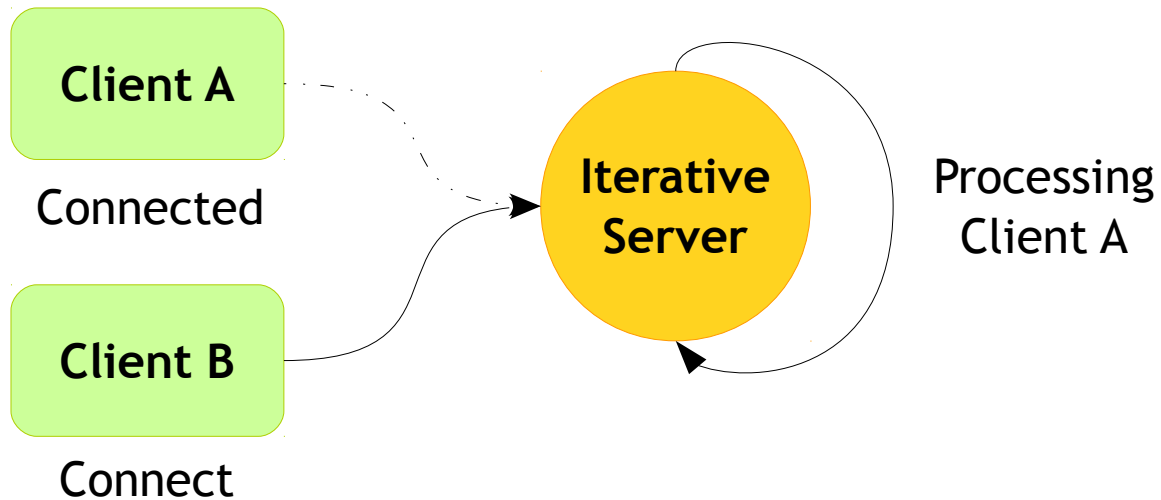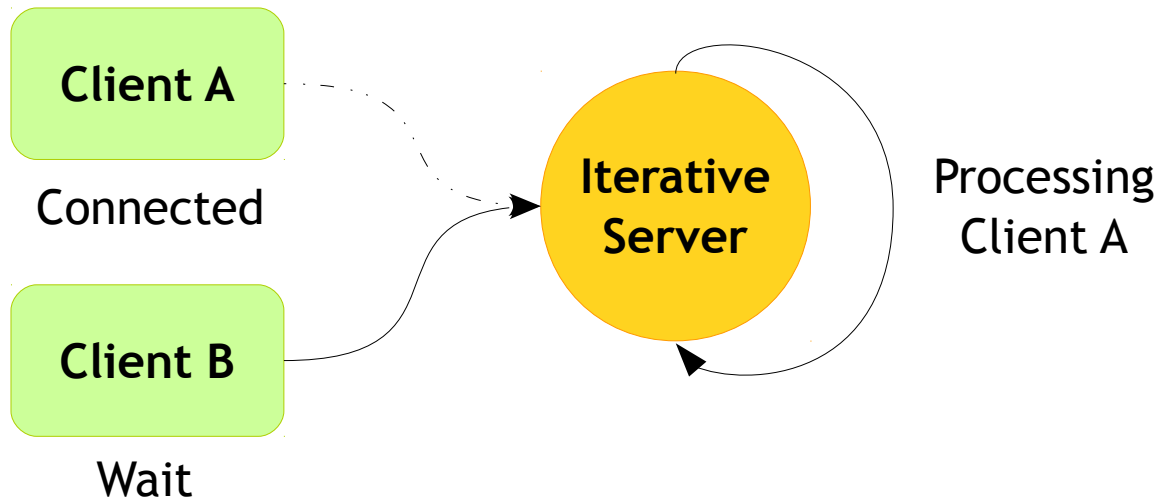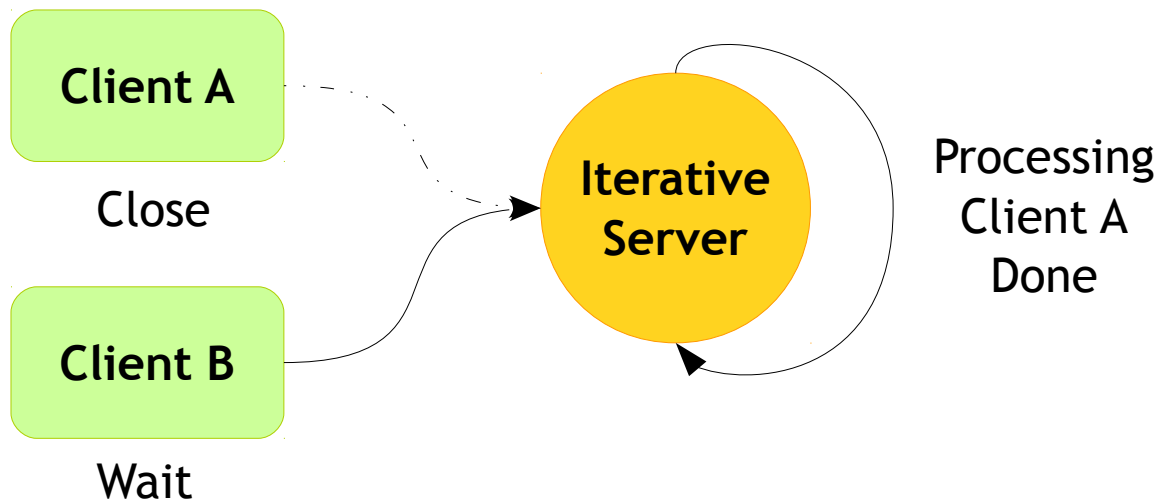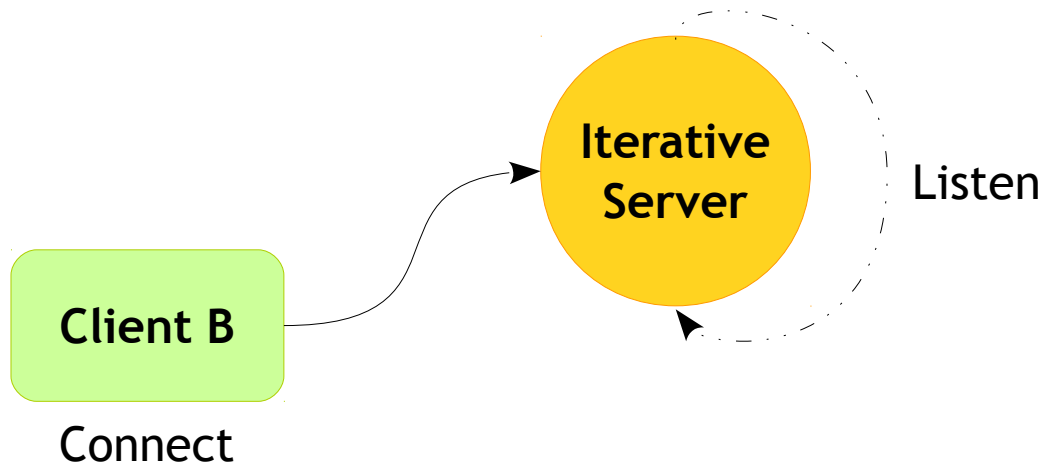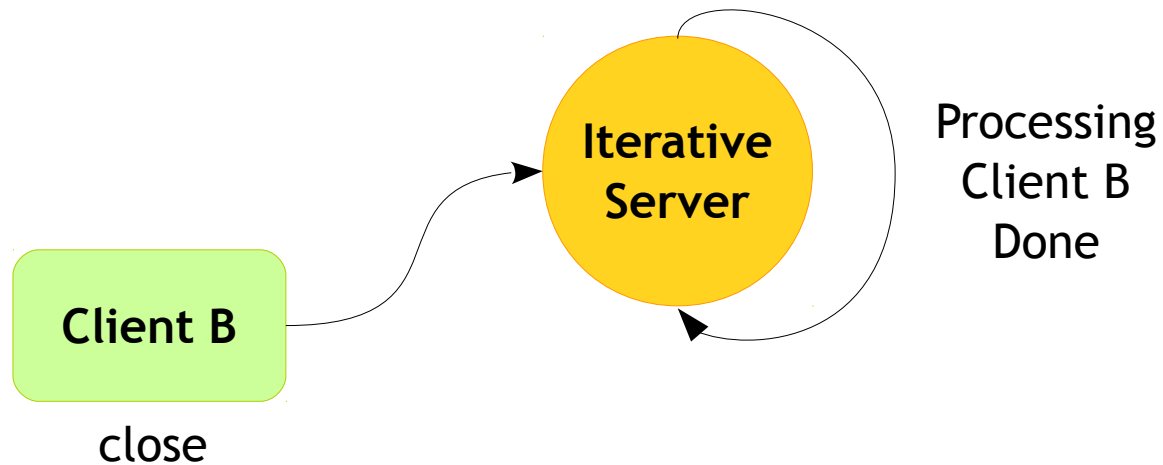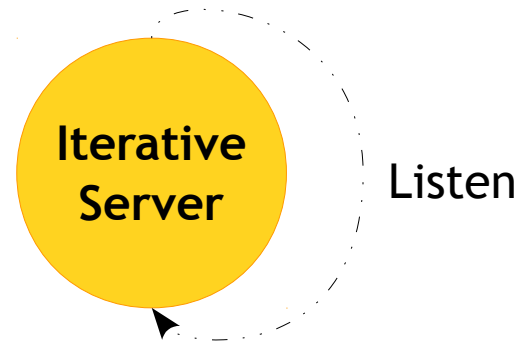
Client – Server Models
Iterative Model – Pros and Cons

- Pros:

  - Simple

  - Reduced network overhead

  - Less CPU intensive

  - Higher single-threaded transaction throughput

- Cons

  - Severely limits concurrent access

  - Server is locked while dealing with one client

Client – Server Models
Concurrent Model – The Flow

- Create a Listening socket

- Bind it to a local address

- Listen (make TCP/IP aware that the socket is available)

- Accept the connection request in loop

- Create a new process and passing the new sockfd

- Do data transaction

- Close (Both process depending on the implementation)

Concurrent Server

Listen

Client – Server Models
Concurrent Model – The Flow

Client – Server Models
Concurrent Model – The Flow

Client – Server Models
Concurrent Model – The Flow

Client – Server Models
Concurrent Model – The Flow

Client – Server Models
Concurrent Model – The Flow

Client – Server Models
Concurrent Model – The Flow

Concurrent Server

Listen
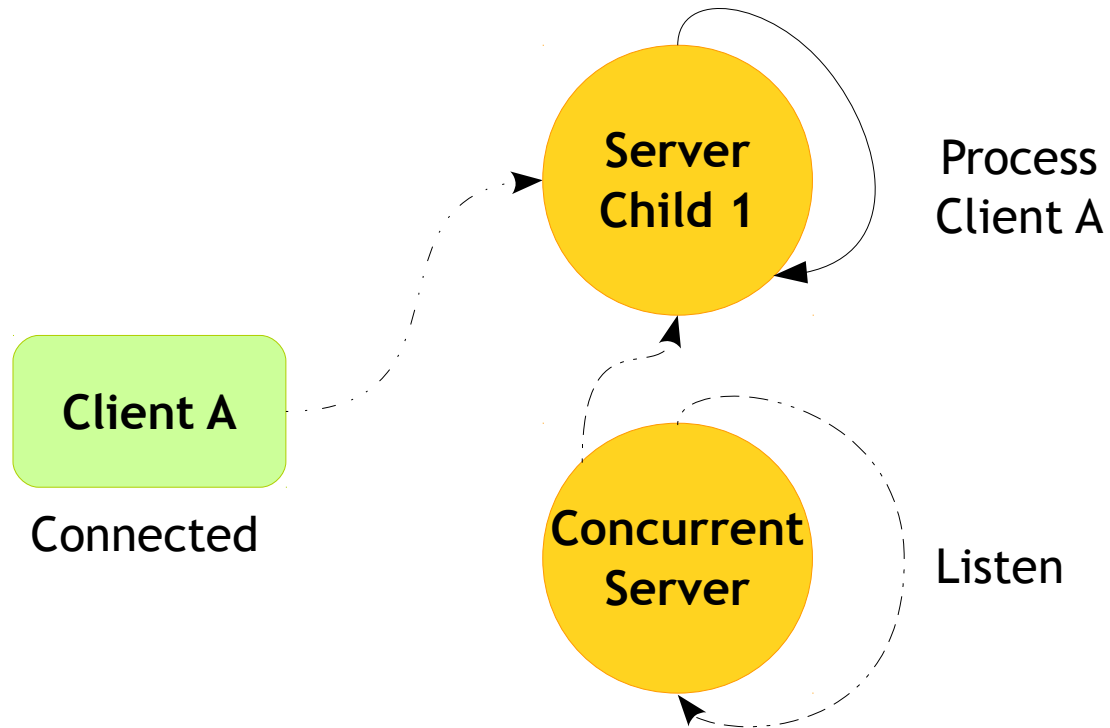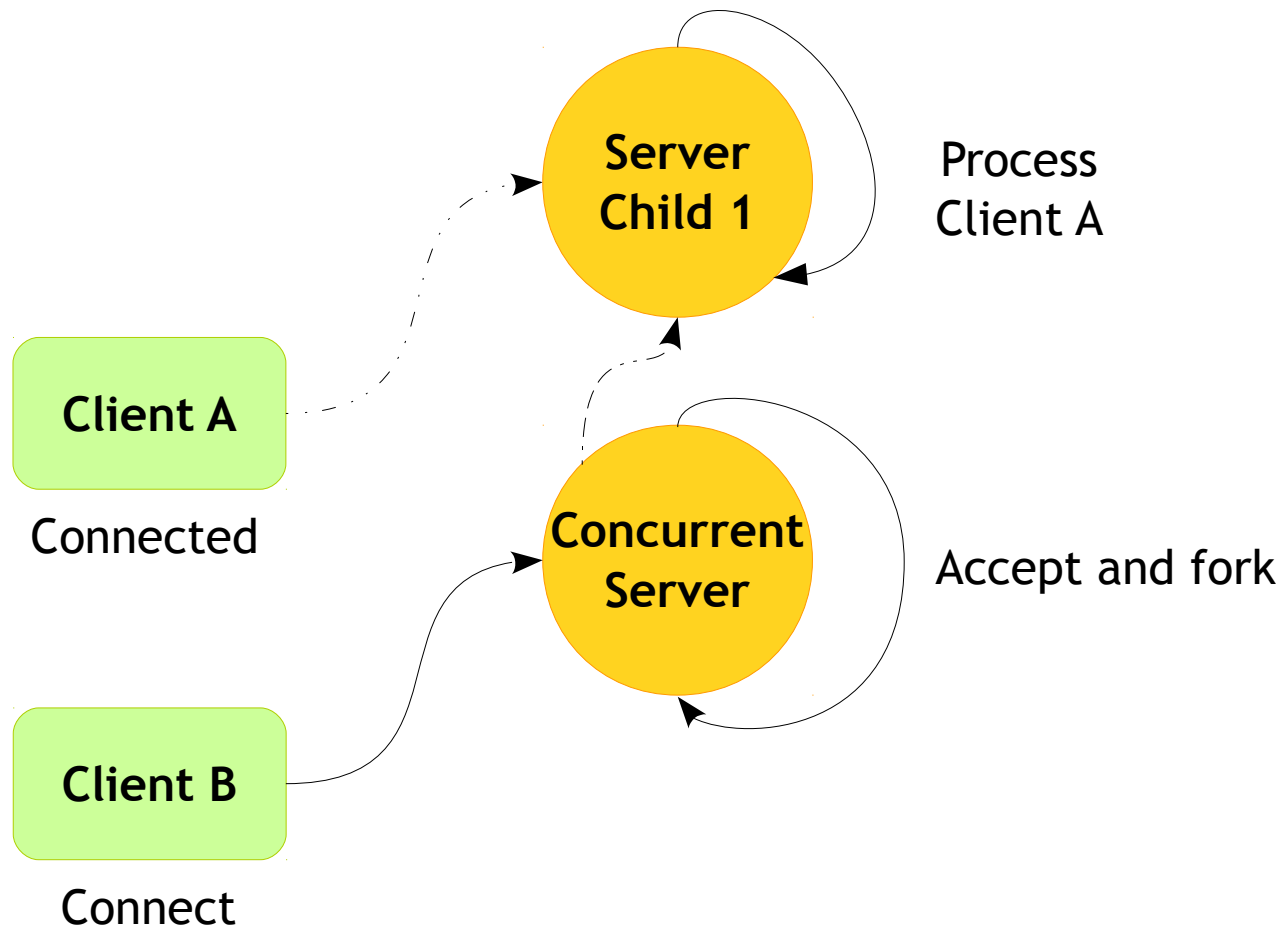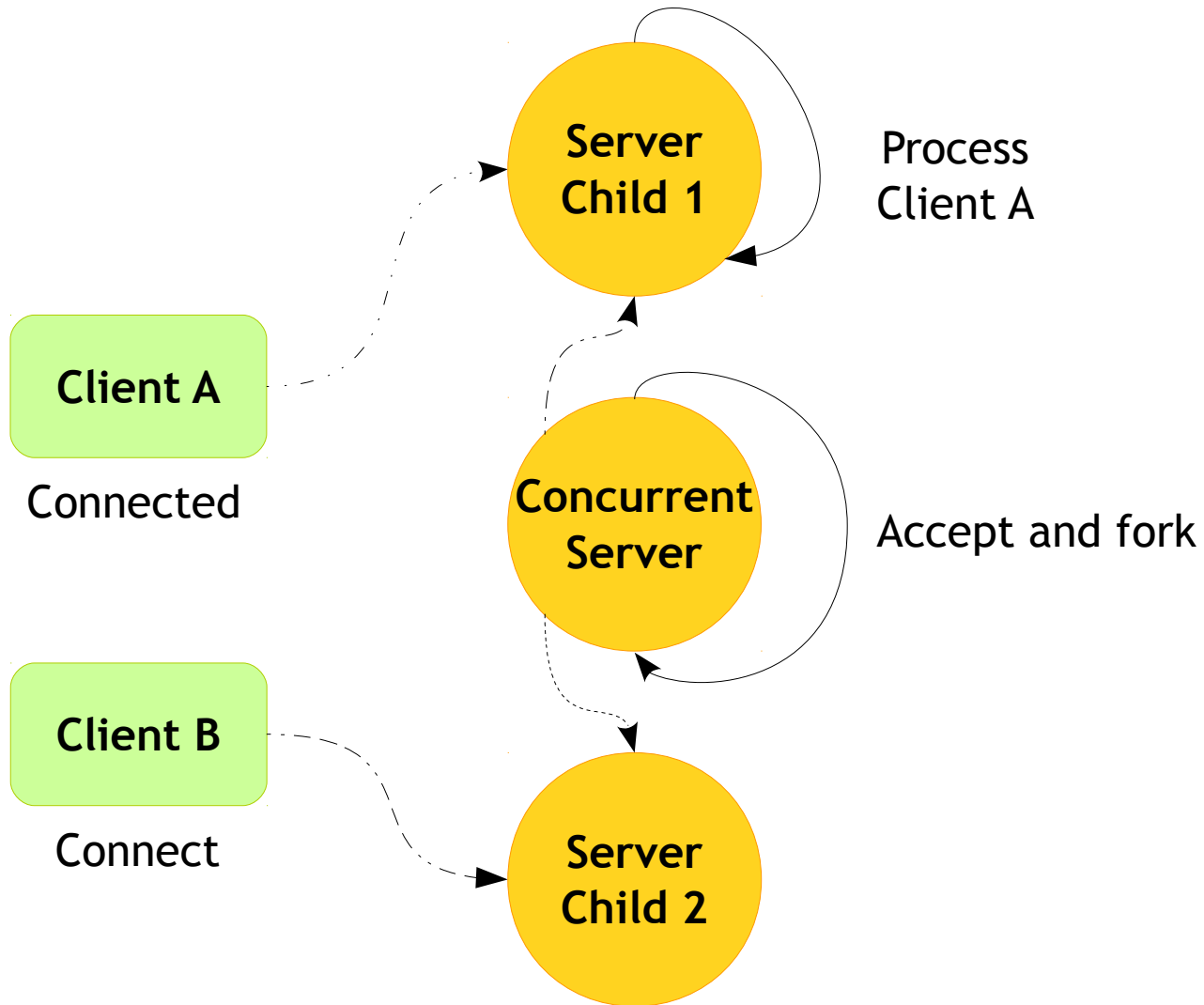
Client – Server Models
Concurrent Model – Pros and Cons

- Pros:

  - Concurrent access

  - Can run longer since no one is waiting for completion

  - Only one listener for many clients

- Cons

  - Increased network overhead

  - More CPU and resource intensive

Inter Process Communications
Summary

- We have covered

  Data exchange

Communication

  - Pipes
  - FIFO
  - Shared memory
  - Signals
  - Sockets

  Resource usage/access/control

Synchronization

  - Semaphores

Threads

Threads

- Threads, like processes, are a mechanism to allow a program to do more than one thing at a time

- As with processes, threads appear to run concurrently

- The Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute

- Threads are a finer-grained unit of execution than processes

- That thread can create additional threads; all these threads run the same program in the same process

- But each thread may be executing a different part of the program at any given time

Threads
Single and Multi threaded Process

code	data	files
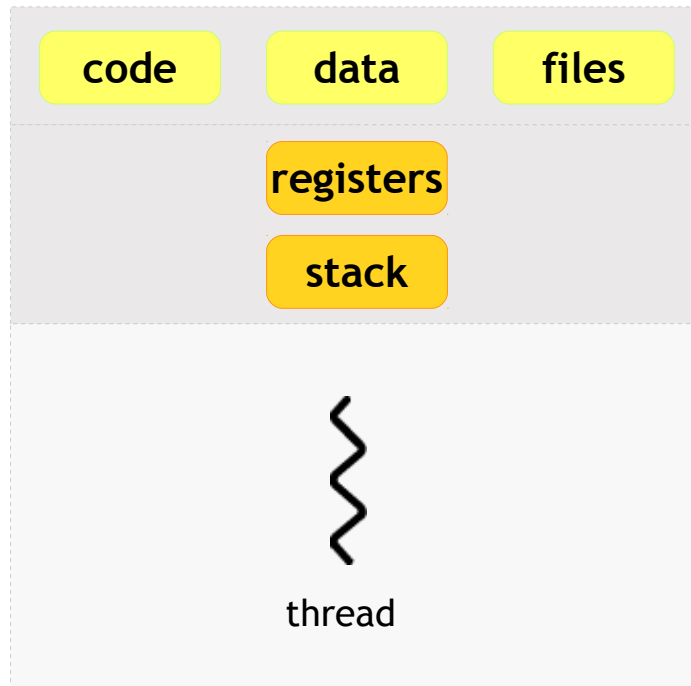
Single Threaded Process

code	data	files
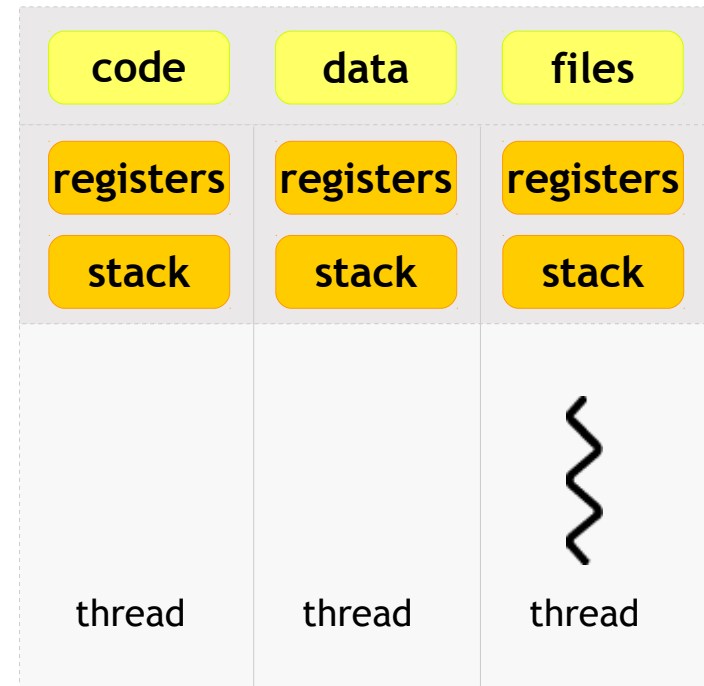
Multi Threaded Process

Threads are similar to handling multiple functions in parallel. Since they share same code & data segments, care to be taken by programmer to avoid issues.

Threads
Advantages

- Takes less time to create a new thread in an existing process than to create a brand new process

- Switching between threads is faster than a normal context switch

- Threads enhance efficiency in communication between different executing programs

- No kernel involved

Threads
pthread API's

- GNU/Linux implements the POSIX standard thread API (known as *pthreads*)

- All thread functions and data types are declared in the header file <pthread.h>

- The pthread functions are not included in the standard C library

- Instead, they are in **libpthread**, so you should add **-lpthread** to the command line when you link your program

> Using libpthread is a very good example to understand differences between functions, library functions and system calls

Threads
Compilation

- Use the following command to compile the programs using thread libraries

  `$ gcc -o <output_file> <input_file.c> -lpthread`

Threads
Creation

- The **pthread_create** function creates a new thread

Function	Meaning
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)	✓ A pointer to a pthread_t variable, in which the thread ID of the new thread is stored ✓ A pointer to a thread attribute object. If you pass NULL as the thread attribute, a thread will be created with the default thread attributes ✓ A pointer to the thread function. This is an ordinary function pointer, of this type: void* (*) (void*) ✓ A thread argument value of type void *. Whatever you pass is simply passed as the argument to the thread function when thread begins executing

Threads
Creation

- A call to **pthread_create** returns immediately, and the original thread continues executing the instructions following the call

- Meanwhile, the new thread begins executing the thread function

- Linux schedules both threads asynchronously

- Programs must not rely on the relative order in which instructions are executed in the two threads

Threads
Joining

- It is quite possible that output created by a thread needs to be integrated for creating final result

- So the main program may need to wait for threads to complete actions

- The pthread_join() function helps to achieve this purpose

Function	Meaning
int pthread_join(pthread_t thread, void **value_ptr)	✓ Thread ID of the thread to wait ✓ Pointer to a void* variable that will receive thread finished value ✓ If you don't care about the thread return value, pass NULL as the second argument.

Threads
Passing Data

- The thread argument provides a convenient method of passing data to threads

- Because the type of the argument is **void***, though, you can't pass a lot of data directly via the argument

- Instead, use the thread argument to pass a pointer to some structure or array of data

- Define a structure for each thread function, which contains the "parameters" that the thread function expects

- Using the thread argument, it's easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data

- If the second argument you pass to **pthread_join** is non-null, the thread's return value will be placed in the location pointed to by that argument

- The thread return value, like the thread argument, is of type **void***

- If you want to pass back a single int or other small number, you can do this easily by casting the value to **void*** and then casting back to the appropriate type after calling **pthread_join**

Threads
Attributes

- Thread attributes provide a mechanism for fine-tuning the behaviour of individual threads

- Recall that **pthread_create** accepts an argument that is a pointer to a thread attribute object

- If you pass a null pointer, the default thread attributes are used to configure the new thread

- However, you may create and customize a thread attribute object to specify other values for the attributes

Threads
Attributes

- There are multiple attributes related to a

- particular thread, that can be set during creation

- Some of the attributes are mentioned as follows:

  - Detach state

  - Priority

  - Stack size

  - Name

  - Thread group

  - Scheduling policy

  - Inherit scheduling

Threads
Joinable and Detached

- A thread may be created as a *joinable thread* (the default) or as a *detached thread*

- A joinable thread, like a process, is not automatically cleaned up by GNU/Linux when it terminates

- Thread's exit state hangs around in the system (kind of like a zombie process) until another thread calls **pthread_join** to obtain its return value. Only then are its resources released

- A detached thread, in contrast, is cleaned up automatically when it terminates

- Because a detached thread is immediately cleaned up, another thread may not synchronize on its completion by using **pthread_join** or obtain its return value

Threads
Creating a Detached Thread

- In order to create a detached thread, the thread attribute needs to be set during creation

- Two functions help to achieve this

Function	Meaning
int pthread_attr_init(pthread_attr_t *attr)	✓ Initializing thread attribute ✓ Pass pointer to pthread_attr_t type ✓ Reurns integer as pass or fail
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);	✓ Pass the attribute variable ✓ Pass detach state, which can take • PTHREAD_CREATE_JOINABLE • PTHREAD_CREATE_DETACHED

Threads
ID

- Occasionally, it is useful for a sequence of code to determine which thread is executing it.

- Also sometimes we may need to compare one thread with another thread using their IDs

- Some of the utility functions help us to do that

Function	Meaning
pthread_t pthread_self()	✓ Get self ID
int pthread_equal(pthread_t threadID1, pthread_t threadID2);	✓ Compare threadID1 with threadID2 ✓ If equal return non-zero value, otherwise return zero

Threads
Cancellation

- It is possible to cancel a particular thread

- Under normal circumstances, a thread terminates normally or by calling **pthread_exit**.

- However, it is possible for a thread to request that another thread terminate. This is called *cancelling* a thread

Function	Meaning
int pthread_cancel(pthread_t thread)	✓ Cancel a particular thread, given the thread ID

Thread cancellation needs to be done carefully, left-over resources will create issue. In order to clean-up properly, let us first understand what is a "critical section"?

Synchronization - Concepts

Synchronization
why?

- In a multi-tasking system the most critical resource is CPU. This is shared between multiple tasks / processes with the help of 'scheduling' algorithm

- When multiple tasks are running simultaneously:

    - Either on a single processor, or on

    - A set of multiple processors

- They give an appearance that:

    - For each process, it is the only task in the system.

    - At a higher level, all these processes are executing efficiently.

    - Process sometimes exchange information:

    - They are sometimes blocked for input or output (I/O).

- Whereas multiple processes run concurrently in a system by communicating, exchanging information with others all the time. They also have very close dependency with various I/O devices and peripherals.

Synchronization
why?

- Considering resources are lesser and processes are more, there is a contention going between multiple processes
- Hence resource needs to be shared between multiple processes. This is called as 'Critical section'
- Access / Entry to critical section is determined by scheduling, however exit from critical section needs to be done when activity is completed properly
- Otherwise it will lead to a situation called 'Race condition'

P2

P1

P3

Shared Resource
(Critical Section)
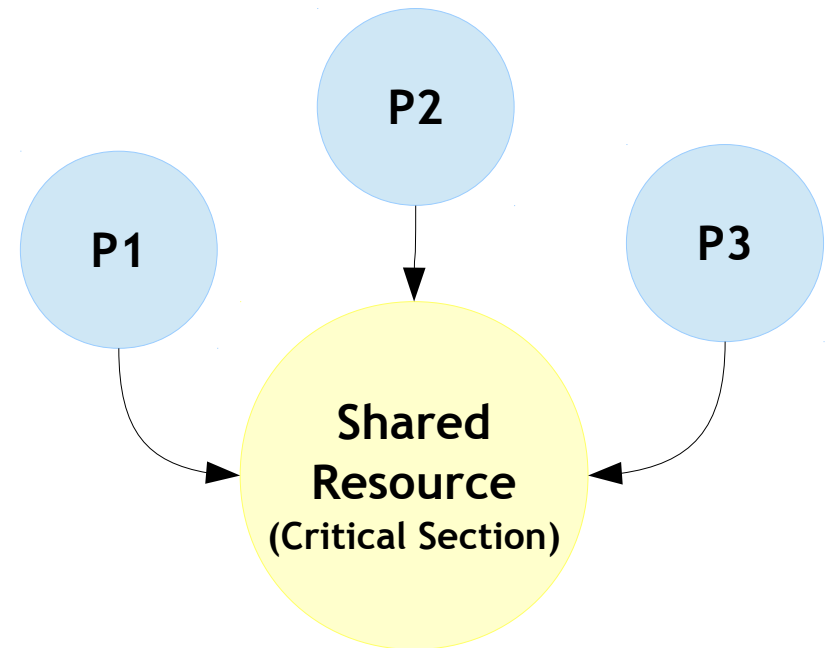
Synchronization
why?

- Synchronization is defined as a mechanism which ensures that two or more concurrent processes do not simultaneously execute some particular program segment known as critical section

- When one process starts executing the critical section (serialized segment of the program) the other process should wait until the first process finishes

- If not handled properly, it may cause a race condition where, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes

- If any critical decision to be made based on variable values (ex: real time actions – like medical system), synchronization problem will create a disaster as it might trigger totally opposite action than what was expected

Synchronization
Race Condition in Embedded Systems

- Embedded systems are typically lesser in terms of resources, but having multiple processes running. Hence they are more prone to synchronization issues, thereby creating race conditions

- Most of the challenges are due to shared data condition. Same pathway to access common resources creates issues

- Debugging race condition and solving them is a very difficult activity because you cannot always easily re-create the problem as they occur only in a particular timing sequence

- Asynchronous nature of tasks makes race condition simulation and debugging as a challenging task, often spend weeks to debug and fix them

Synchronization
Critical Section

- The way to solve race condition is to have the critical section access in such a way that only one process can execute at a time

- If multiple process try to enter a critical section, only one can run and the others will sleep (means getting into blocked / waiting state)
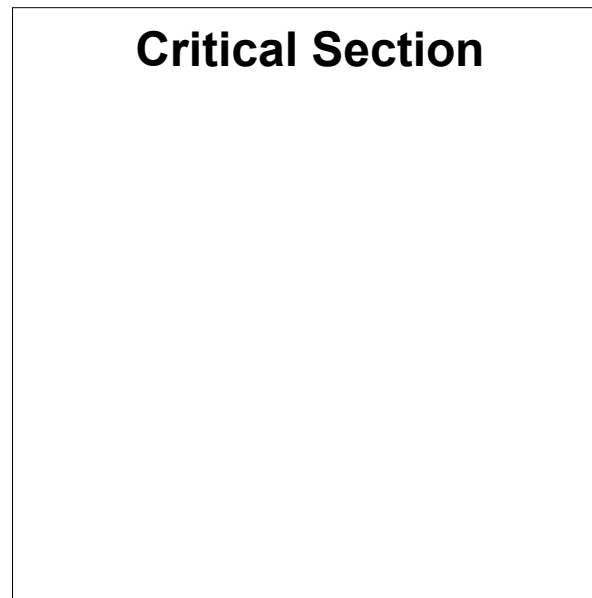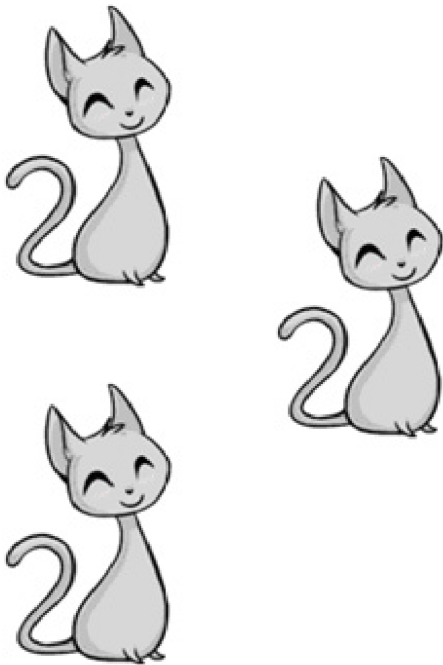
Critical Section

Synchronization
Critical Section

- Only one process can enter the critical section; the other two have to sleep. When a process sleeps, its execution is paused and the OS will run some other task
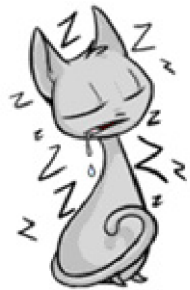
Critical Section

Synchronization
Critical Section

- Once the process in the critical section exits, another process is woken up and allowed to enter the critical section. This is done based on the existing scheduling algorithm

- It is important to keep the code / instructions inside a critical section as small as possible (say similar to ISR) to handle race conditions effectively
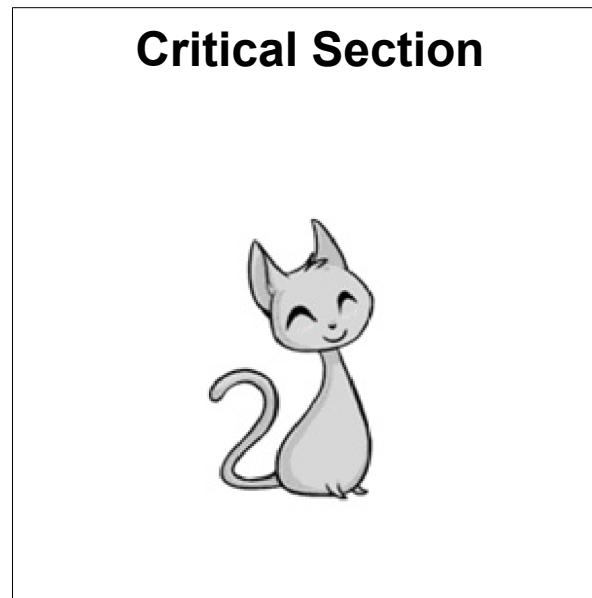
Critical Section

Synchronization
Priority Inversion

- One of the most important aspect of critical section is to ensure whichever process is inside it, has to complete the activities at one go. They should not be done across multiple context switches. This is called Atomicity

- Assume a scenario where a lower priority process is inside the critical section and higher priority process tries to enter

- Considering atomicity the higher priority process will be pushed into blocking state. This creates some issue with regular priority algorithm

- In this juncture if a medium priority tasks gets scheduled, it will enter into the critical section with higher priority task is made to wait. This scenario is further creating a change in priority algorithm

- This is called as 'Priority Inversion' which alters the priority schema
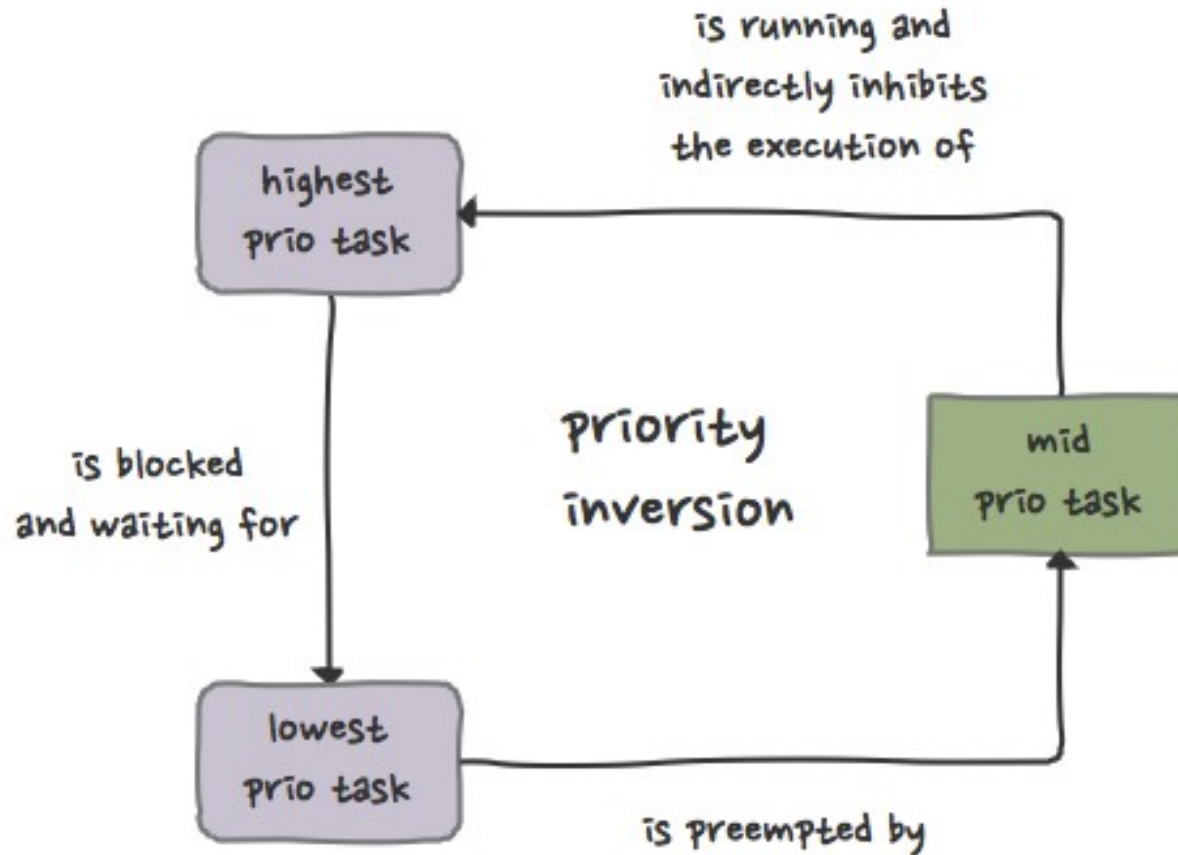
Synchronization
Priority Inversion
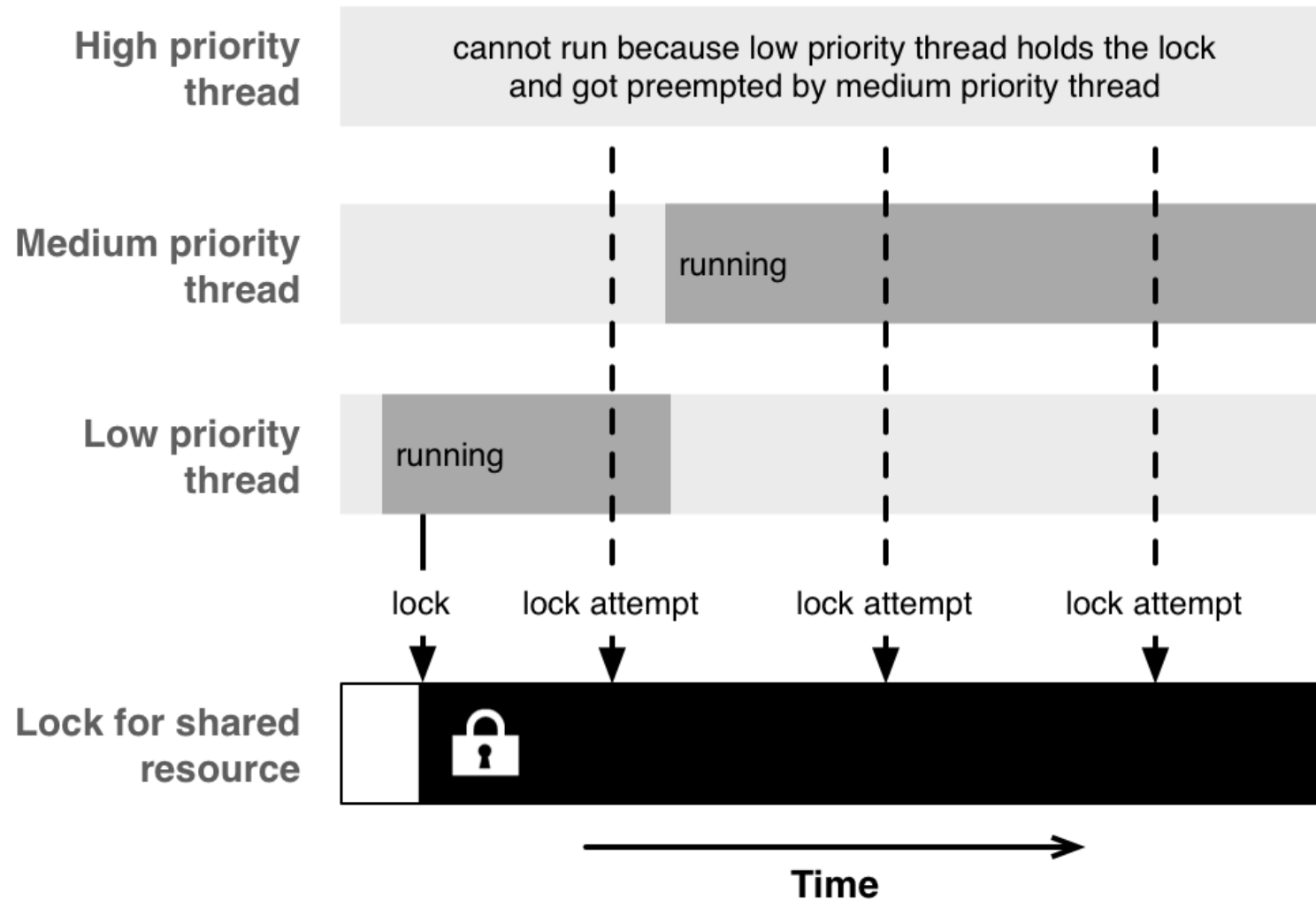
Quick refresher

- Before moving onto exploring various solutions for critical section problem, ensure we understand these terminologies / definitions really well.

    – Difference between scheduling & Synchronization

    – Shared data problem

    – Critical section

    – Race condition

    – Atomicity

    – Priority inversion

ƩMERTXE

Critical section - Solutions

Critical Section
Solutions

Solution to critical section should have following three aspects into it:

- **Mutual Exclusion:** If process P is executing in its critical section, then no other processes can be executing in their critical sections

- **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

- **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
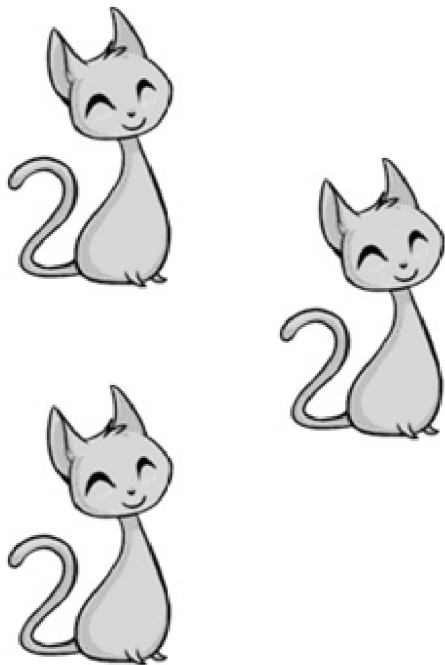
Critical Section
Solutions

- There are multiple algorithms (ex: Dekker's algorithm) to implement solutions that is satisfying all three conditions. From a programmers point of view they are offered as multiple solutions as follows:

  - Locks / Mutex

  - Readers–writer locks

  - Recursive locks

  - Semaphores

  - Monitors

  - Message passing

  - Tuple space

- Each of them are quite detailed in nature, in our course two varieties of solutions are covered they are Mutex and Semaphores

- Let us look into them in detail!

ƩMERTXE

- A Mutex works in a critical section while granting access

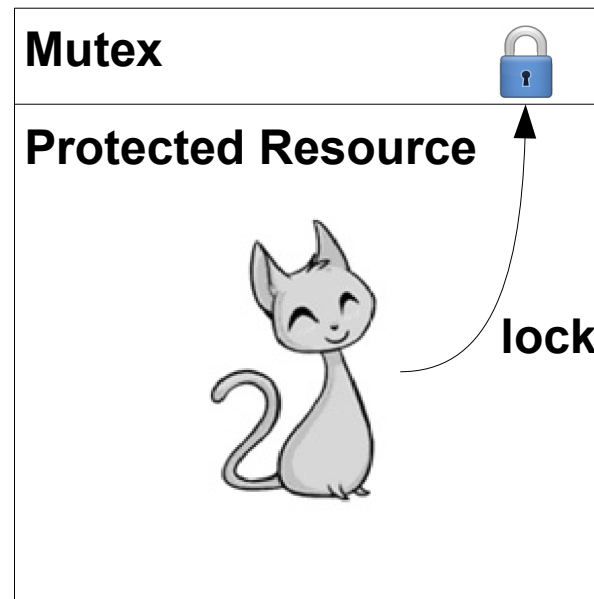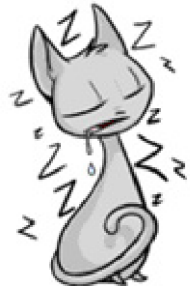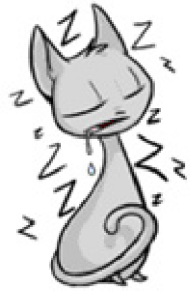- You can think of a Mutex as a token that must be grabbed before execution can continue.

open

Mutex

Protected Resource
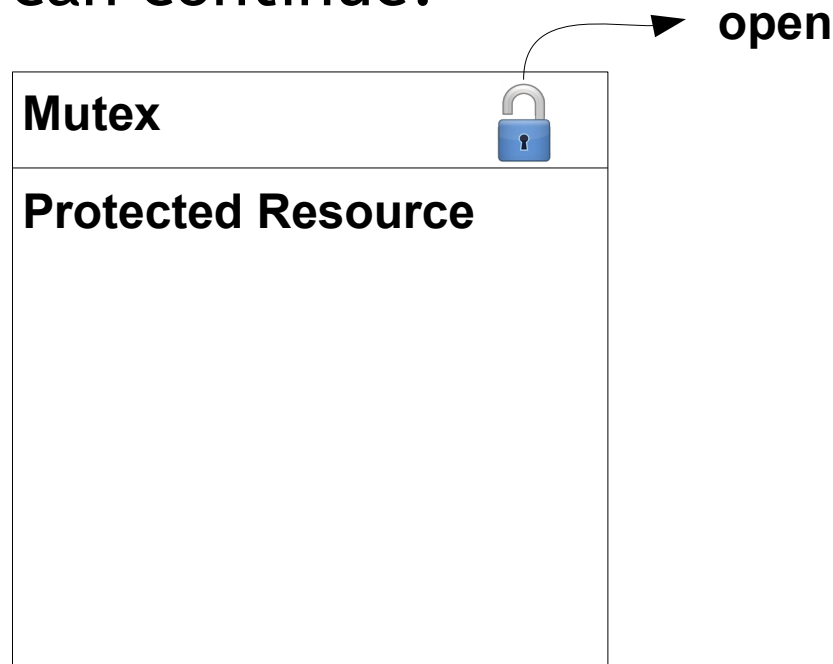
Critical Section
Solutions - Mutual Exclusion

- During the time that a task holds the mutex, all other tasks waiting on the mutex sleep.

- Once a task has finished using the shared resource, it releases the mutex. Another task can then wake up and grab the mutex.

Mutex

Protected Resource

release

Critical Section
Solutions - Mutual Exclusion – Locking / Blocking

- A process may attempt to get a Mutex by calling a **lock** method. If the Mutex was unlocked (means already available), it becomes locked (unavailable) and the function returns immediately

- If the Mutex was locked by another process, the locking function **blocks** execution and returns only eventually when the Mutex is **unlocked** by the other process

- More than one process may be blocked on a locked Mutex at one time

- When the Mutex is unlocked, only one of the blocked process is unblocked and allowed to lock the Mutex. Other tasks stay blocked.

Critical Section
Semaphores

- A semaphore is a counter that can be used to synchronize multiple processes. Typically semaphores are used where multiple units of a particular resources are available

- Each semaphore has a counter value, which is a non-negative integer. It can take any value depending on number of resources available

- The 'lock' and 'unlock' mechanism is implemented via 'wait' and 'post' functionality in semaphore. Where the wait will decrement the counter and post will increment the counter

- When the counter value becomes zero that means the resources are no longer available hence remaining processes will get into blocked state

Critical Section
Semaphores - Sleeping barber problem

A lazy barber who sleeps and gets up on his own will :)

Critical Section
Semaphores – 2 basic operations

- ## Wait operation:

  - Decrements the value of the semaphore by 1

  - If the value is already zero, the operation blocks until the value of the semaphore becomes positive

  - When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns

- ## Post operation:

  - Increments the value of the semaphore by 1

  - If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore

  - One of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero)
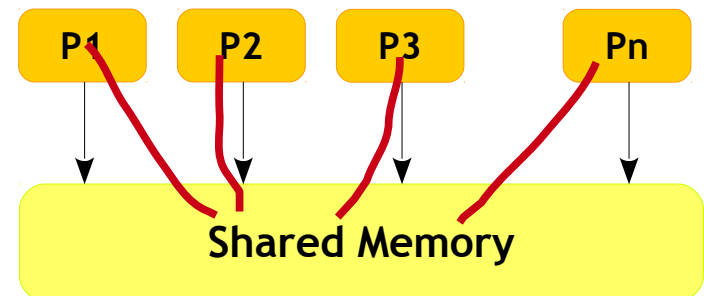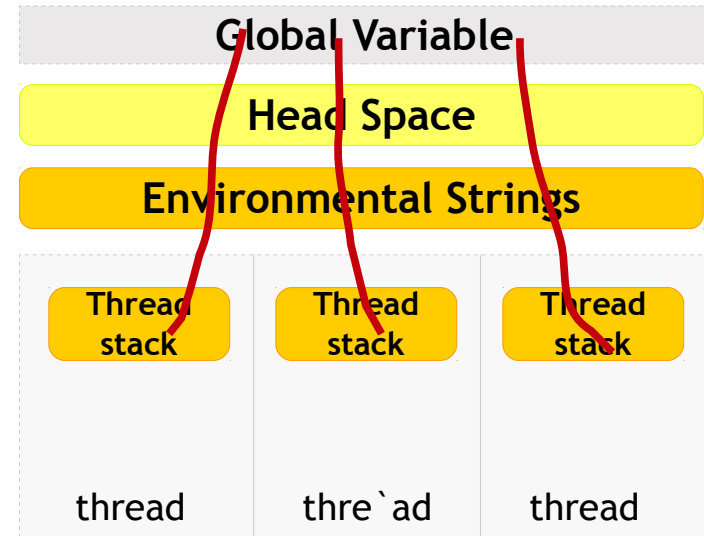
Critical Section
Mutex & Semaphores

- Semaphores which allow an arbitrary resource count (say 25) are called **counting semaphores**

- Semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores**

- A Mutex is essentially the same thing as a binary semaphore, however the differences between them are in how they are used

- While a binary semaphore may be used as a Mutex, a Mutex is a more specific use-case, in that only the process that locked the Mutex is supposed to unlock it

- This constraint makes it possible to implement some additional features in Mutexes

Critical Section
Practical Implementation

- The problem is critical section / race condition is common in multi-threading and multi-processing environment. Since both of them offer concurrency & common resource facility, it will raise to race conditions

- However the common resource can be different. In case of multiple threads a common resource can be a data segment / global variable which is a shared resource between multiple threads

- In case of multiple processes a common resource can be a shared memory

Global Variable

Head Space

Environmental Strings

Thread stack	Thread stack	Thread stack
thread	thre`ad	thread

P1   P2   P3   Pn

Shared Memory

Synchronization
Treads - Mutex

- pthread library offers multiple Mutex related library functions
- These functions help to synchronize between multiple threads

Function	Meaning
int pthread_mutex_init(pthread_mutex_t *mutex const pthread_mutexattr_t *attribute)	✓ Initialize mutex variable ✓ mutex: Actual mutex variable ✓ attribute: Mutex attributes ✓ RETUR: Success (0)/Failure (Non zero)
int pthread_mutex_lock(pthread_mutex_t *mutex)	✓ Lock the mutex ✓ mutex: Mutex variable ✓ RETURN: Success (0)/Failure (Non-zero)
int pthread_mutex_unlock(pthread_mutex_t *mutex)	✓ Unlock the mutex ✓ Mutex: Mutex variable ✓ RETURN: Success (0)/Failure (Non-zero)
int pthread_mutex_destroy(pthread_mutex_t *mutex)	✓ Destroy the mutex variable ✓ Mutex: Mutex variable ✓ RETURN: Success (0)/Failure (Non-zero)

ΣMERTXE

Synchronization

- ## Wait operation:

  - Decrements the value of the semaphore by 1

  - If the value is already zero, the operation blocks until the value of the semaphore becomes positive

  - When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns

- ## Post operation:

  - Increments the value of the semaphore by 1

  - If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore

  - One of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero)

Synchronization
Treads - Semaphores

- pthread library offers multiple Semaphore related library functions

- These functions help to synchronize between multiple threads

Function	Meaning
int sem_init (sem_t *sem, int pshared, unsigned int value)	✓ sem: Points to a semaphore object ✓ pshared: Flag, make it zero for threads ✓ value: Initial value to set the semaphore ✓ RETURN: Success (0)/Failure (Non zero)
int sem_wait(sem_t *sem)	✓ Wait on the semaphore **(Decrements count)** ✓ sem: Semaphore variable ✓ RETURN: Success (0)/Failure (Non-zero)
int sem_post(sem_t *sem)	✓ Post on the semaphore **(Increments count)** ✓ sem: Semaphore variable ✓ RETURN: Success (0)/Failure (Non-zero)
int sem_destroy(sem_t *sem)	✓ Destroy the semaphore ✓ No thread should be waiting on this semaphore ✓ RETURN: Success (0)/Failure (Non-zero)

Inter Process Communications
Synchronization - Semaphores

- Semaphores are similar to counters

- Process semaphores synchronize between multiple processes, similar to thread semaphores

- The idea of creating, initializing and modifying semaphore values remain same in between processes also

- However there are different set of system calls to do the same semaphore operations

Inter Process Communications
Synchronization – Semaphore Functions

Function	Meaning
int semget(key_t key, int nsems, int flag)	✓ Create a process semaphore ✓ key: Seed input ✓ nsems: Number of semaphores in a set ✓ flag: Permission (similar to file) ✓ RETURN: Semaphore ID / Failure
int semop(int semid, struct sembuf *sops, unsigned int nsops)	✓ Wait and Post operations ✓ semid: Semaphore ID ✓ sops: Operation to be performed ✓ nsops: Length of the array ✓ RETURN: Operation Success / Failure
semctl(semid, 0, IPC_RMID)	✓ Semaphores need to be explicitly removed ✓ semid: Semaphore ID ✓ Remove and NULL

ƩMERTXE

Inter Process Communications
Summary

- We have covered

Data exchange

Communication

- Pipes
- FIFO
- Shared memory
- Signals
- Sockets

Resource usage/access/control

Synchronization

- Semaphores

Process Management - Concepts
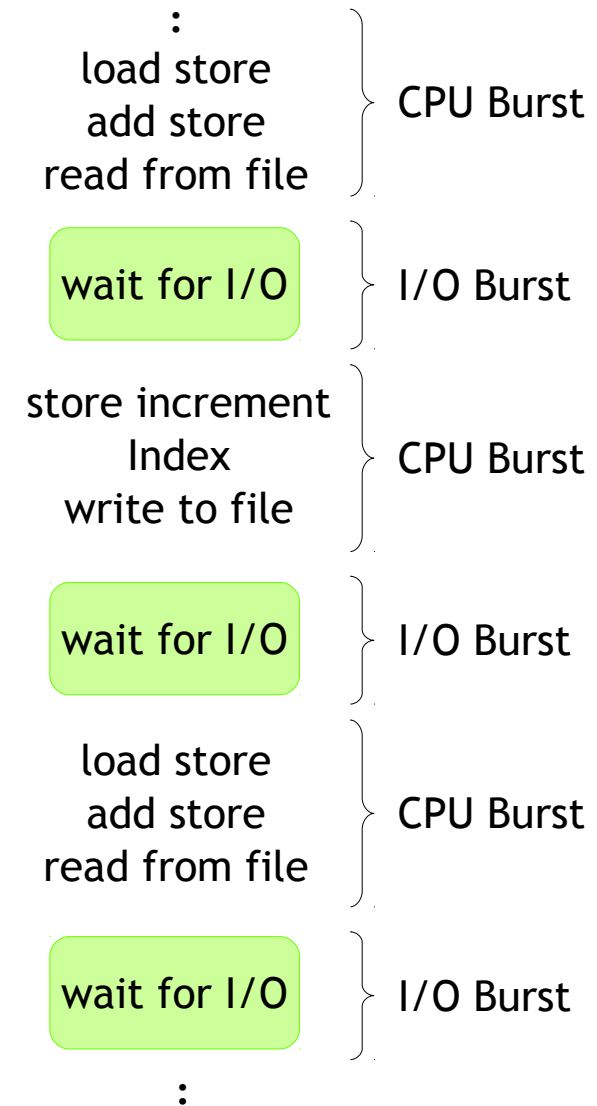
Process Management - Concepts
Scheduling

- It is a mechanism used to achieve the desired goal of multitasking

- This is achieved by SCHEDULER which is the heart and soul of operating System

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU

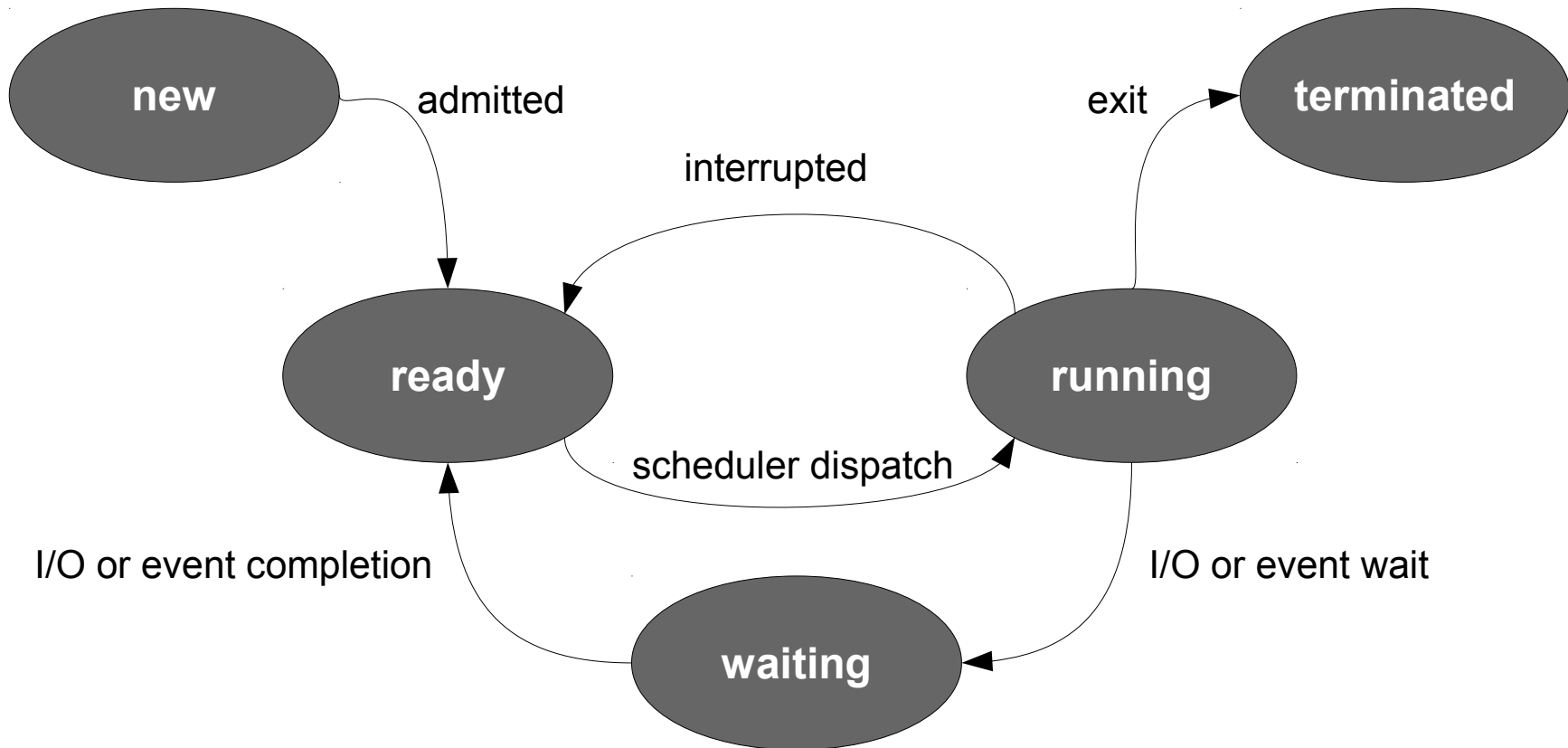When a scheduler schedules tasks and gives a predictable response, they are called as "Real Time Systems"

- Maximum CPU utilization obtained with multi programming

- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait

```
         :
    load store
    add store          CPU Burst
    read from file
```

```
    wait for I/O       I/O Burst
```

```
    store increment
        Index          CPU Burst
    write to file
```

```
    wait for I/O       I/O Burst
```

```
    load store
    add store          CPU Burst
    read from file
```

```
    wait for I/O       I/O Burst
         :
```

Process Management - Concepts
States

Process Management - Concepts
States

- A process goes through multiple states ever since it is created by the OS

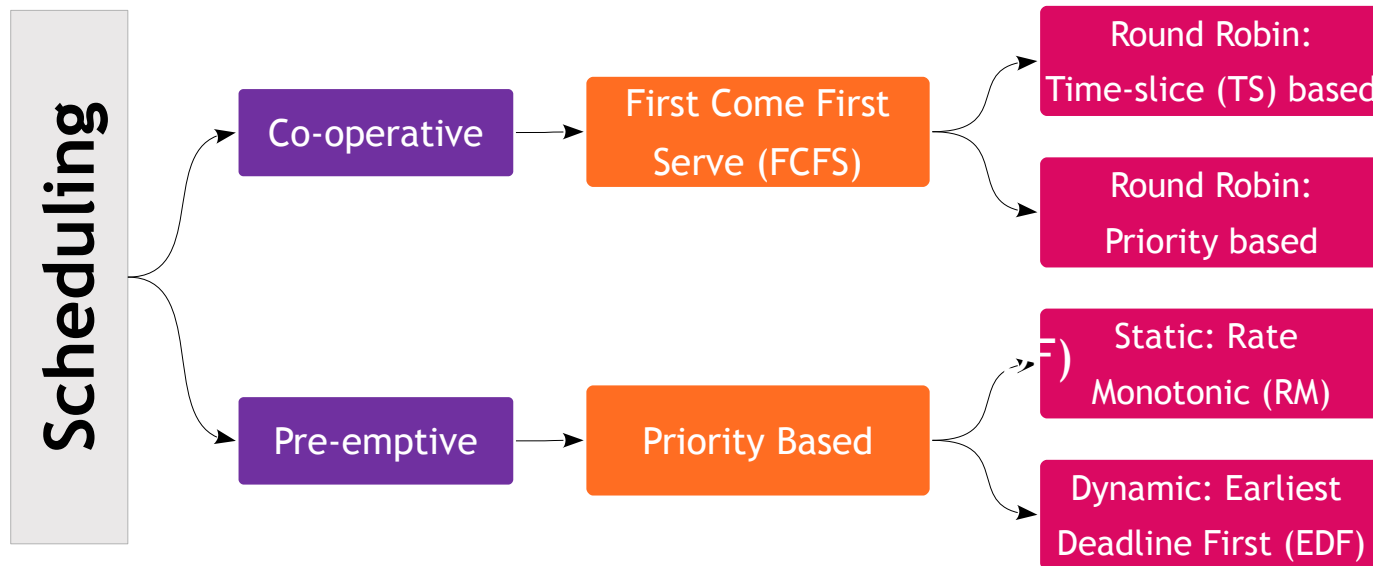| State	Description
New	The process is being created
Running	Instructions are being executed
Waiting	The process is waiting for some event to occur
Ready	The process is waiting to be assigned to processor
Terminated	The process has finished execution

Process Management - Concepts
Schedulers

- Selects from among the processes in memory that are ready to execute

- Allocates the CPU to one of them

- CPU scheduling decisions may take place when a process:

  - Switches from running to waiting state
  - Switches from running to ready state
  - Switches from waiting to ready
  - Terminates

- Scheduling under 1 and 4 is non-preemptive

- All other scheduling is preemptive

Scheduling

- Co-operative → First Come First Serve (FCFS)
  - Round Robin: Time-slice (TS) based
  - Round Robin: Priority based
- Pre-emptive → Priority Based
  - Static: Rate Monotonic (RM)
  - Dynamic: Earliest Deadline First (EDF)

- In Co-operative scheduling, process co-operate in terms of sharing processor timing. The process voluntarily gives the kernel a chance to perform a process switch

- In Preemptive scheduling, process are preempted a higher priority process, thereby the existing process will need to relinquish CPU

- First Come First Served (FCFS) is a Non-Preemptive scheduling algorithm. FIFO (First In First Out) strategy assigns priority to process in the order in which they request the processor. The process that requests the CPU first is allocated the CPU first. This is easily implemented with a FIFO queue for managing the tasks. As the process come in, they are put at the end of the queue. As the CPU finishes each task, it removes it from the start of the queue and heads on to the next task.

Process Management - Concepts
Scheduling – Types – FCFS

Process	Burst time
P1	20
P2	5
P3	3

- Suppose processes arrive in the order: P1, P2, P3 , The Gantt Chart for the schedule is

P1	P2	P3

0   20   25   28

- Processes are scheduled based on time-slice, but they are time-bound

- This time slicing is similar to FCFS except that the scheduler forces process to give up the processor based on the timer interrupt

- It does so by preempting the current process (i.e. the process actually running) at the end of each time slice

- The process is moved to the end of the priority level

Process	Burst time
P1	20
P2	5
P3	3

- Suppose processes arrive in the order: P1, P2, P3 , The Gantt Chart for the schedule is

P1	P2	P3	P1	P1	P1

0   4   9   12   17   22
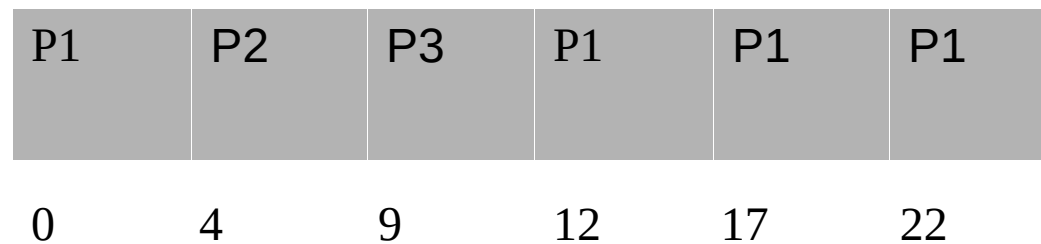
- Processes are scheduled based on RR, but priority attached to it

- While processes are allocated based on RR (with specified time), when higher priority task comes in the queue, it gets pre-empted
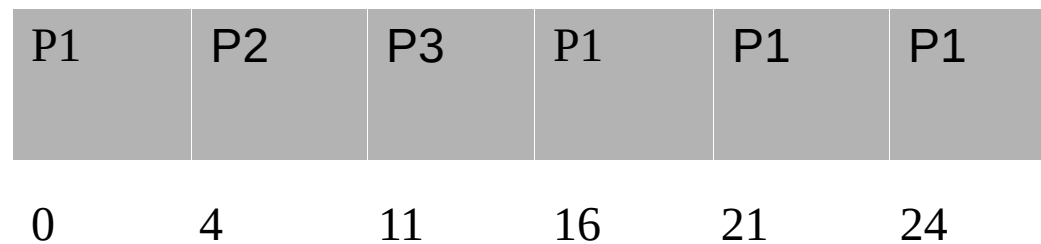
- The time slice remain the same

Process	Burst time
P1	24
P2	10
P3	15

- Suppose processes arrive in the order: P1, P2, P3 and assume P2 have high priority, The Gantt Chart for the schedule is

P1	P2	P3	P1	P1	P1

0   4   11   16   21   24

- Pre-emption means while a lower priority process is executing on the processor another process higher in priority than comes up in the ready queue, it preempts the lower priority process.

- Rate Monotonic (RM) scheduling:

  - The highest Priority is assigned to the Task with the Shortest Period

  - All Tasks in the task set are periodic

  - The relative deadline of the task is equal to the period of the Task

  - Smaller the period, higher the priority

- Earliest Deadline First (EDF) scheduling:

  - This kind of scheduler tries to give execution time to the task that is most quickly approaching its deadline

  - This is typically done by the scheduler changing priorities of tasks on-the-fly as they approach their individual deadlines
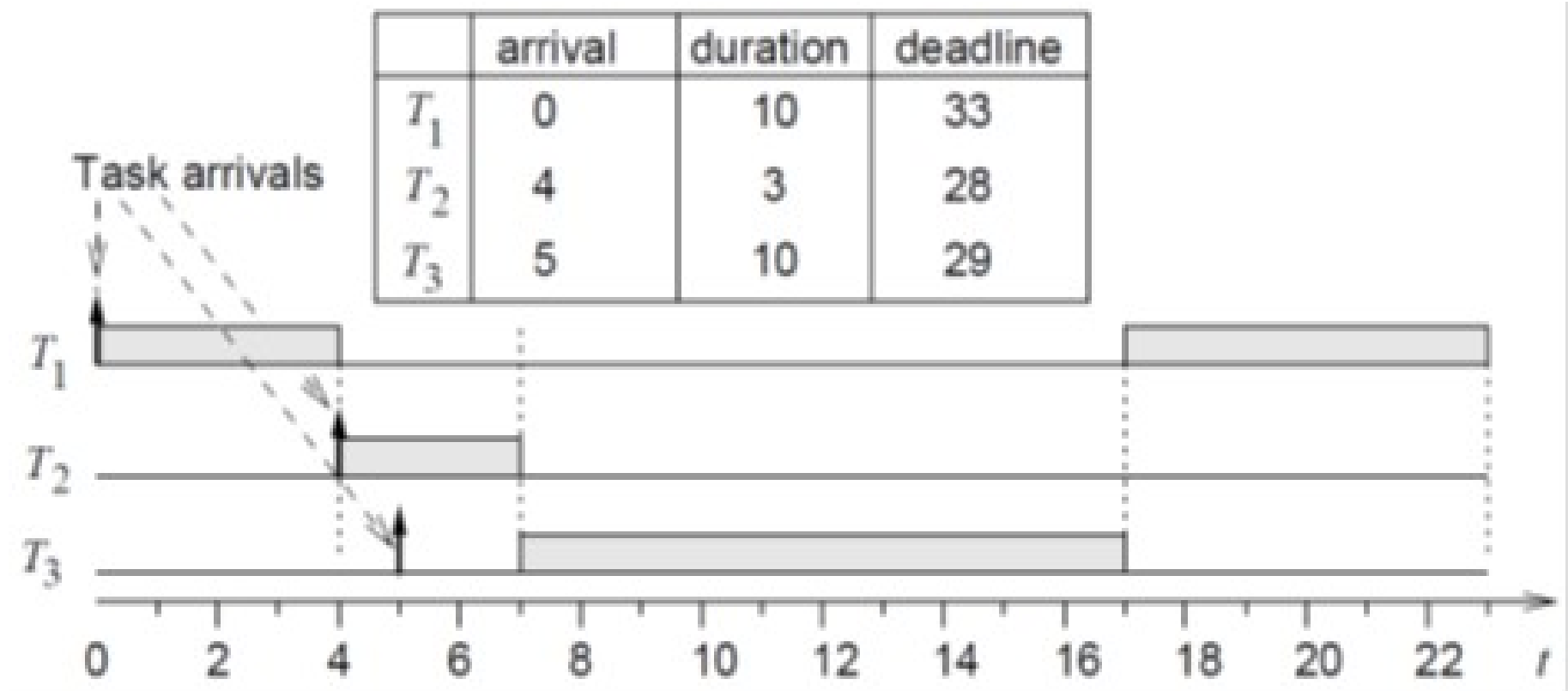
- T1 preempts T2 and T3.

- T2 and T3 do not preempt each other.
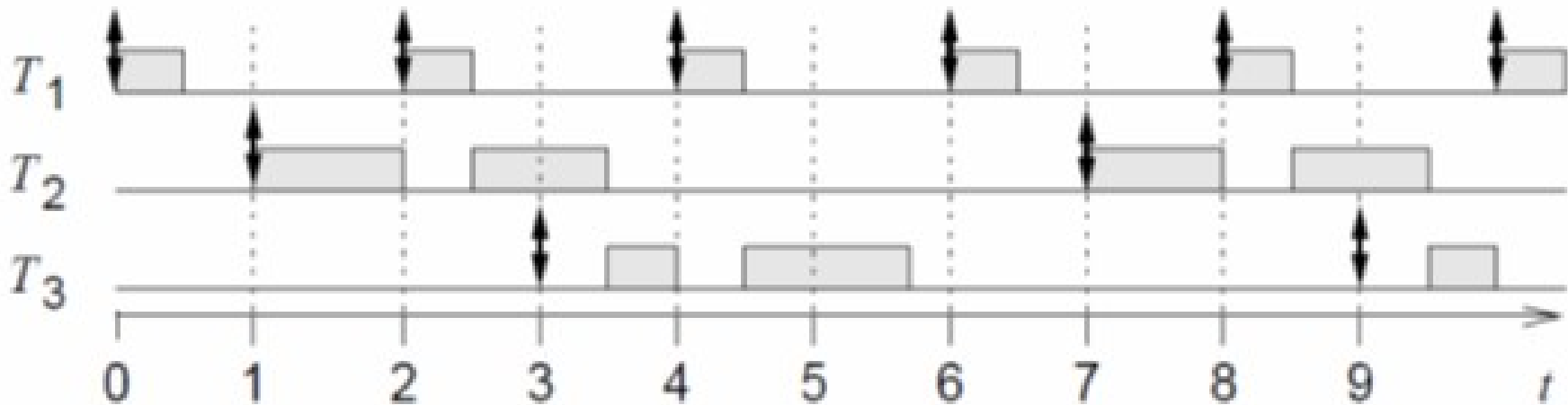
	arrival	duration	deadline
T_1	0	10	33
T_2	4	3	28
T_3	5	10	29

Task arrivals

Introduction to RTOS

Real Time Systems

- Characteristics:

  – Capable of guaranteeing timing requirements of the processes under its control

  – Fast – low latency

  – Predictable – able to determine task's completion time with certainty

  – Both time-critical and non time-critical tasks to coexist

- Types:

  – Hard real time system

    - Guarantees that real-time tasks be completed within their required deadlines.
    - Requires formal verification/guarantees of being to always meet its hard deadlines (except for fatal errors).
    - Examples: air traffic control , vehicle subsystems control, medical systems.

  – Soft real time system

    - Provides priority of real-time tasks over non real-time tasks.
    - Also known as "best effort" systems. Example – multimedia streaming, computer games

Real Time OS

- Operating system is a program that runs on a super loop

- Consist of Scheduler, Task, Memory, System call interface, File systems etc.

- All of these components are very much part of Embedded and Real-time systems

- Some of the parameters need to be tuned/changed in order to meet the needs of these systems

- Real time & Embedded systems – Coupling v/s De-coupling

Real Time OS
Characteristics

- Real-time systems are typically single-purpose (Missing: Support for variety of peripherals)

- Real-time systems often do not require interfacing with a user (Missing: Sophisticated user modes & permissions)

- High overhead required for protected memory and for switching modes (Missing: User v/s Kernel mode)

- Memory paging increases context switch time (Missing: Memory address translation between User v/s Kernel)

- User control over scheduler policy & configuration

Real Time OS
Properties

- Reliability

- Predictability

- Performance

- Compactness

- Scalability

- User control over OS Policies

- Responsiveness

  - Fast task switch

  - Fast interrupt response

Real Time OS
Examples

- LynxOS

- OSE

- QNX

- VxWorks

- Windows CE

- RT Linux

Memory Management - Concepts

Memory Management - Concepts

Introduction

- Overall memory sub-division:

  - OS

  - Application

- Uni-programming vs. Multi-programming

- Memory Management is task of OS, called MMU

- May involve movement between:

  - Primary (Hard disk / Flash)

  - Secondary (RAM)

Memory Management - Concepts
Requirements

- Relocation

- Protection

- Sharing

- Logical Organization

- Physical Organization

Memory Management - Concepts
Requirements - Relocation

- Programmer does not know where the program will be placed in memory when it is executed

- Before the program is loaded, address references are usually relative addresses to the entry point of program

- These are called logical addresses, part of logical address space

- All references must be translated to actual addresses

- It can be done at compile time, load time or execution

- Mapping between logical to physical address mechanism is implemented as "Virtual memory"

- Paging is one of the memory management schemes where the program retrieves data from the secondary storage for use in main memory

Memory Management - Concepts
Virtual memory – Why?

- If programs access physical memory we will face three problems

  - Don't have enough physical memory.

  - Holes in address space (fragmentation).

  - No security (All program can access same memory)

- These problems can be solved using virtual memory.

  - Each program will have their own virtual memory.

  - They separately maps virtual memory space to physical memory.

  - We can even move to disk if we run out of memory (Swapping)

Memory Management - Concepts
Virtual memory – Paging & page table

- Virtual memory divided into small chunks called pages.

- Similarly physical memory divided into frames.

- Virtual memory and physical memory mapped using page table.
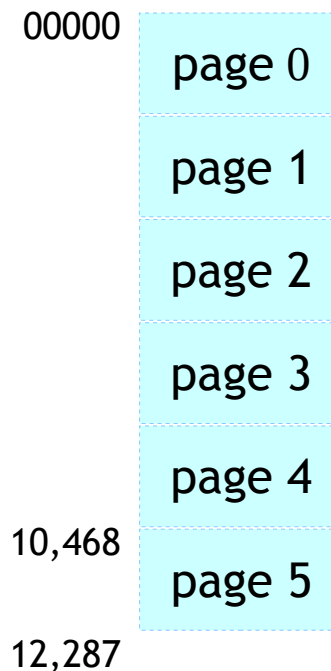
frame number   valid-invalid bit
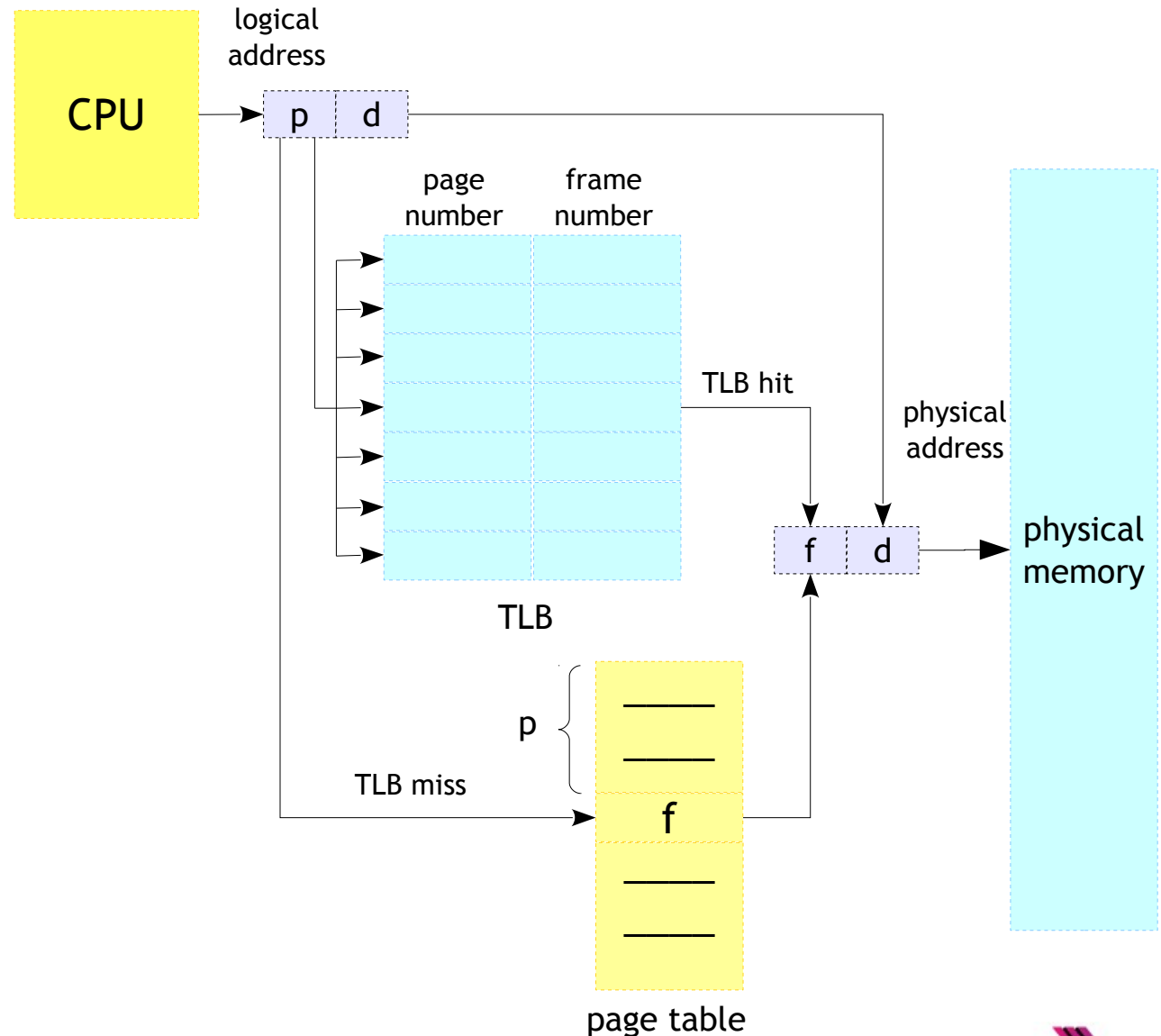
2	v
3	v
4	v
7	v
8	v
9	v
0	i
0	i

page table

00000

page 0

page 1

page 2

page 3

page 4

10,468

page 5

12,287

0

1

2 page 0

3 page 1

4 page 2

5

6

7 page 3

8 page 4

9 page 5

page n
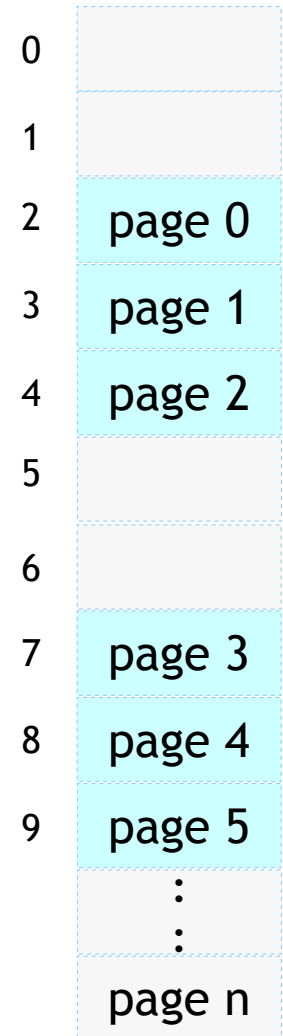
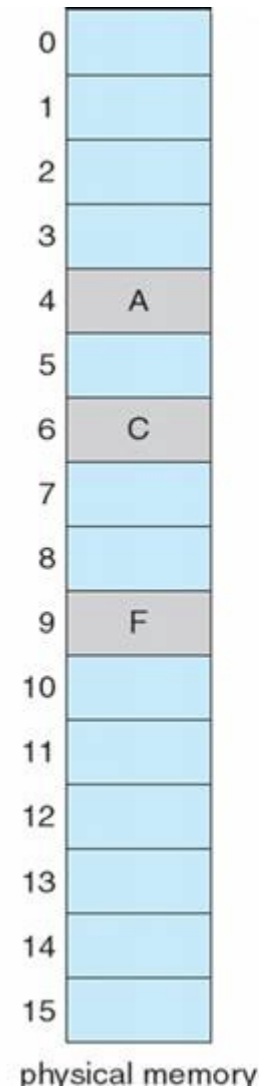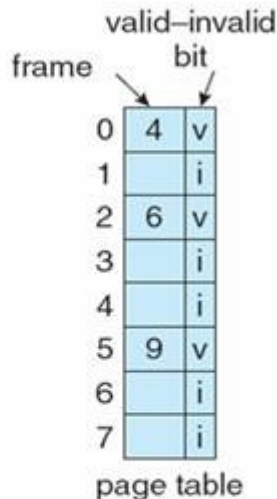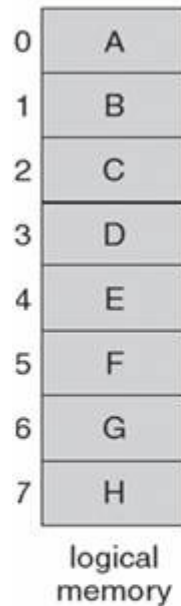# Memory Management - Concepts
Virtual memory – TLB

- For faster access page table will be stored in CPU cache memory.

- But limited entries only possible.

- If page entry available in TLB (Hit), control goes to physical address directly (Within one cycle).

- If page entry not available in TLB (Miss), it use page table from main memory and maps to physical address(Takes more cycles compared to TLB).

- When a process try access a frame using a page table and that frame moved to swap memory, generates an interrupt called page fault.
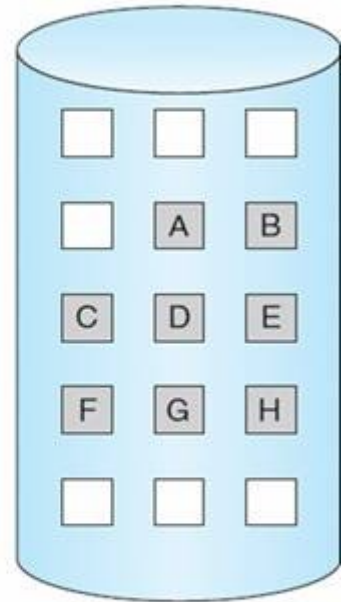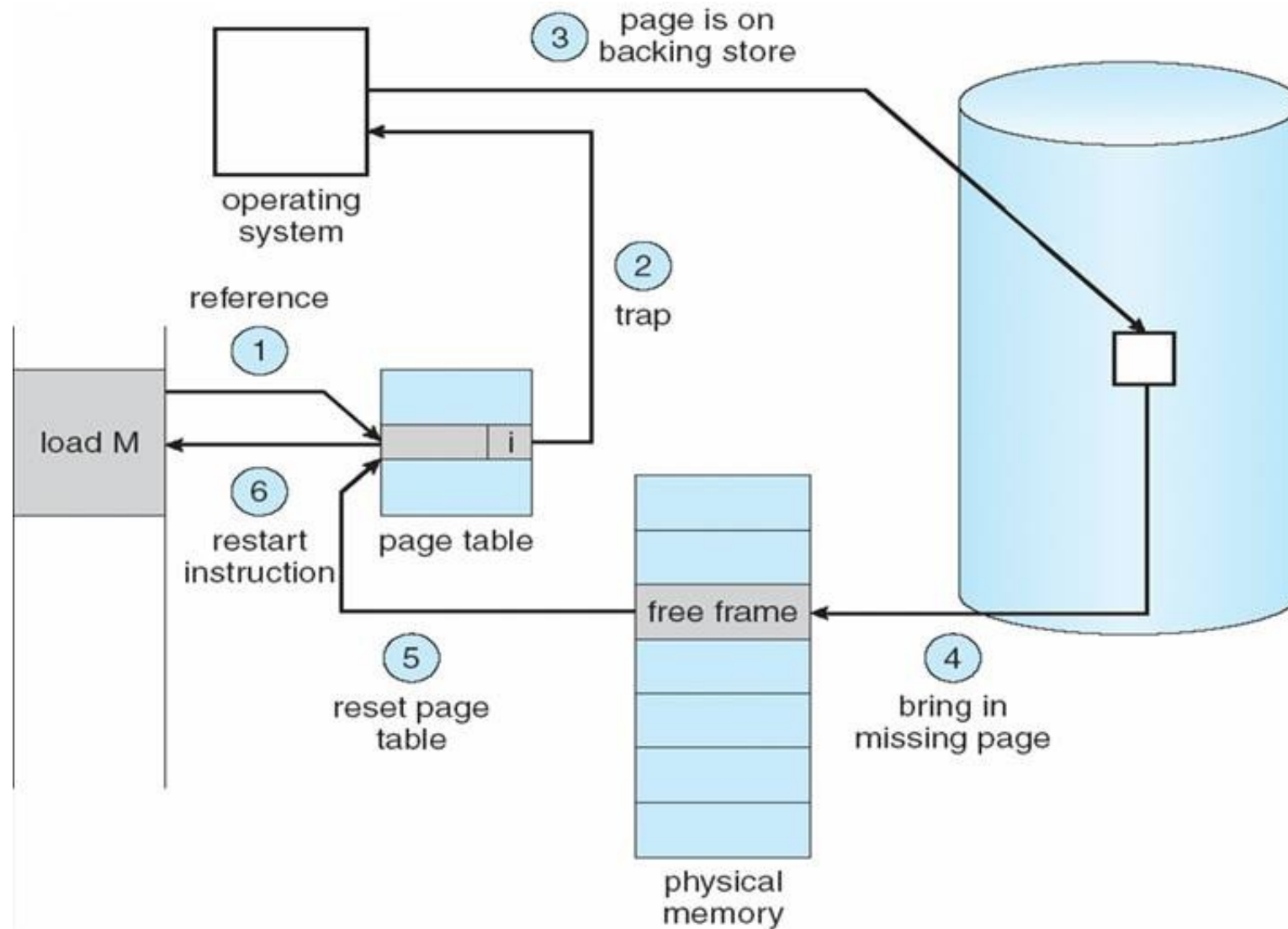
Memory Management - Concepts
Page fault – Handling

1. Check an internal table for this process, to determine whether the reference was a valid or it was an invalid memory access.

2. If the reference was invalid, terminate the process. If it was valid, but page have not yet brought in, page in the latter.

3. Find a free frame.

4. Schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. Restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.
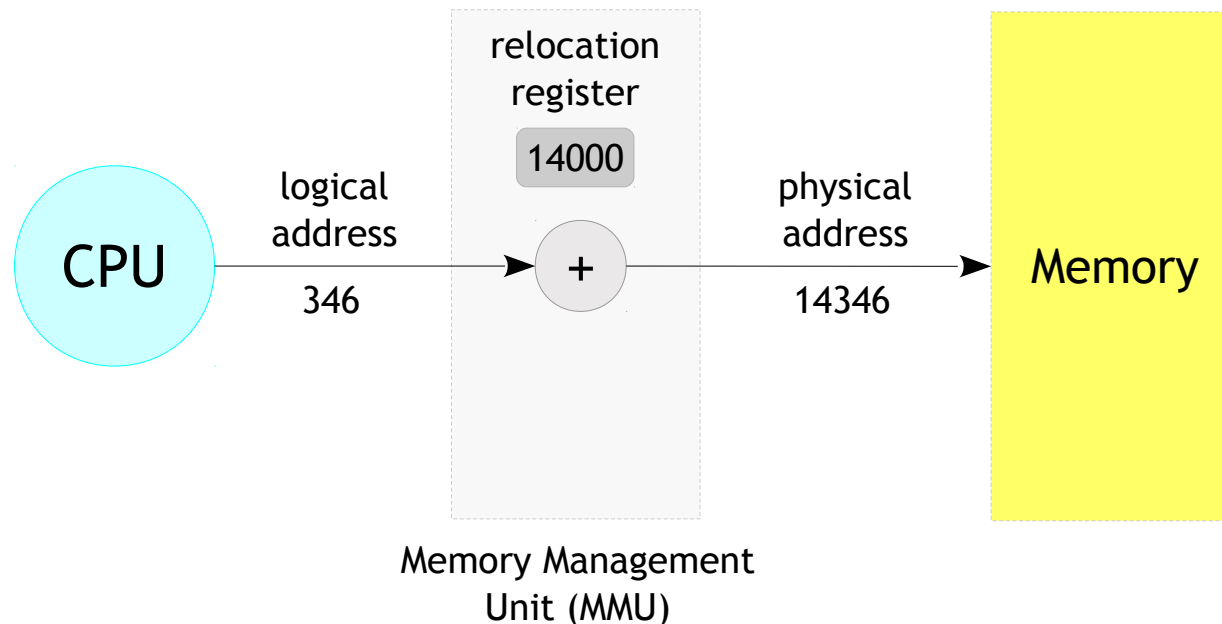
Memory Management - Concepts
MMU

- MMU is responsible for all aspects of memory management. It is usually integrated into the processor, although in some systems it occupies a separate IC (integrated circuit) chip.

- The work of the MMU can be divided into three major categories:

  - Hardware memory management, which oversees and regulates the processor's use of RAM (random access memory) and cache memory.

  - OS (operating system) memory management, which ensures the availability of adequate memory resources for the objects and data structures of each running program at all times.

  - Application memory management, which allocates each individual program's required memory, and then recycles freed-up memory space when the operation concludes.

- The logical address of a memory allocated to a process is the combination of base register and limit register. When this logical address is added to the relocation register, it gives the physical address.

relocation register

14000

logical address

346

CPU

+

physical address

14346

Memory

Memory Management Unit (MMU)

Memory Management - Concepts
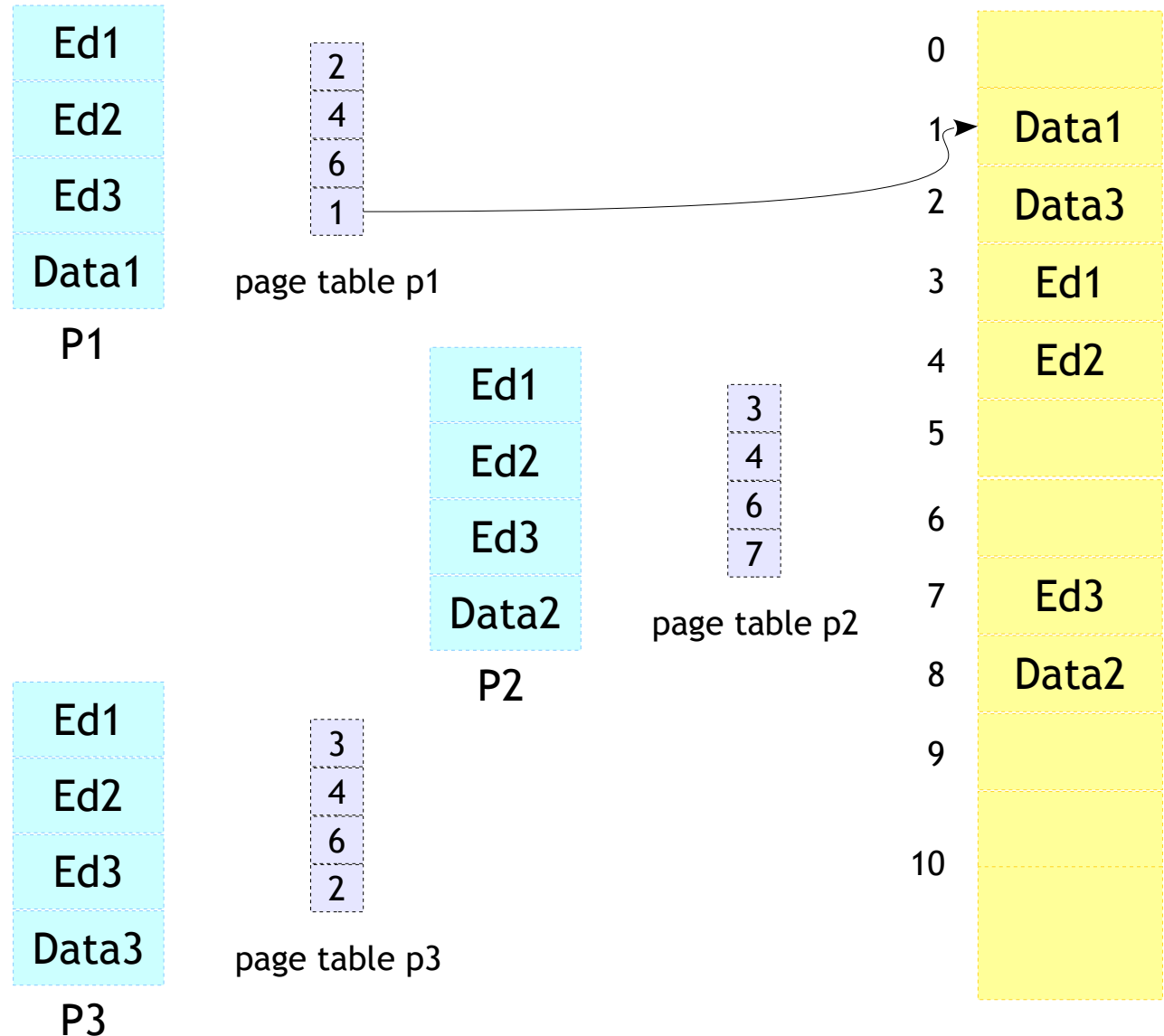## Requirements - Protection

- Processes should not be able to reference memory locations in another process without permission

- Impossible to check absolute addresses in programs since the program could be relocated

- Must be checked during execution

frame number   valid-invalid bit

2	v
3	v
4	v
7	v
8	v
9	v
0	i
0	i

page table
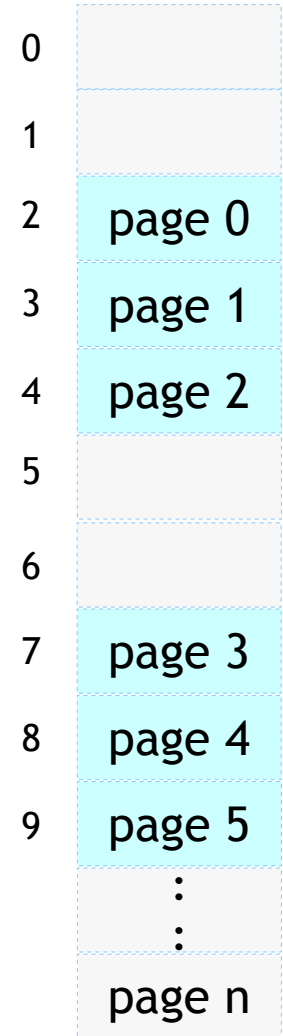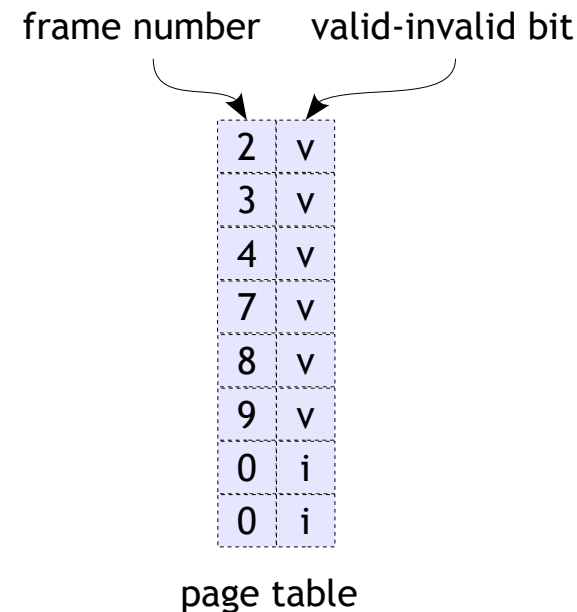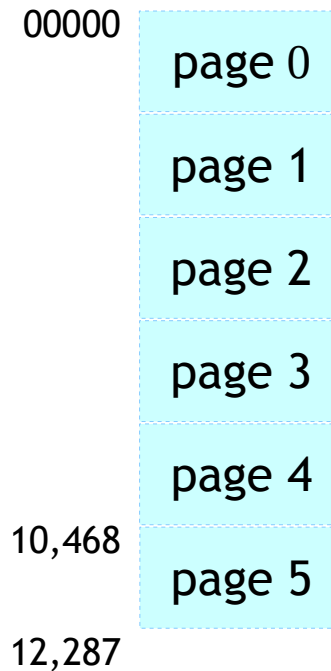
Requirements - Sharing

- Allow several processes to access the same portion of memory

- For example, when using shared memory IPC, we need two processes to share the same memory segment

P1		page table p1		

| Ed1 |
| Ed2 |
| Ed3 |
| Data1 |

P1

page table p1: 2, 4, 6, 1

| Ed1 |
| Ed2 |
| Ed3 |
| Data2 |

P2

page table p2: 3, 4, 6, 7

| Ed1 |
| Ed2 |
| Ed3 |
| Data3 |

P3

page table p3: 3, 4, 6, 2

0	
1	Data1
2	Data3
3	Ed1
4	Ed2
5	
6	
7	Ed3
8	Data2
9	
10	

Memory Management - Concepts
Requirements – Logical Organization

- Logical Organization Memory is organized linearly (usually) In contrast, programs are organized into modules.

- Modules can be written and compiled independently

- Different degrees of protection can be given to different modules (read-only, execute-only)

- Modules can be shared among processes Segmentation helps here

- In Linux, Code Segment has a read-only attribute

Memory Management - Concepts
Requirements – Physical Organization

- Processes in the user space will be leaving & getting in

- Each process needs the memory to execute

- So, the memory needs to be partitioned between processes

  – Fixed Partitioning

  – Dynamic Partitioning

ΣMERTXE

Networking - Concepts

Networking – concept
Types of networks

There are several different types of computer networks. Computer networks can be characterized by their size as well as their purpose.

The size of a network can be expressed by the geographic area they occupy and the number of computers that are part of the network. Networks can cover anything from a handful of devices within a single room to millions of devices spread across the entire globe.

Some of the different networks based on size are:

Personal area network, or PAN

Local area network, or LAN

Metropolitan area network, or MAN

Wide area network, or WAN

Networking – concept

Types of networks

- LAN
    - → Mostly same physical technology
    - → No switching elements
    - → Channel contention problem
    - → Range: 10 m to 1 km
- WAN
    - → Different physical layer technologies
    - → Switching elements
    - → Hosts and routers
    - → Internet

Networking – concept

Types of networks

- WPN

    ➔ Consumer home networking

    ➔ Connected home 'experience'

    ➔ Connect everything

    ➔ Low range, high power

    ➔ Office on the move
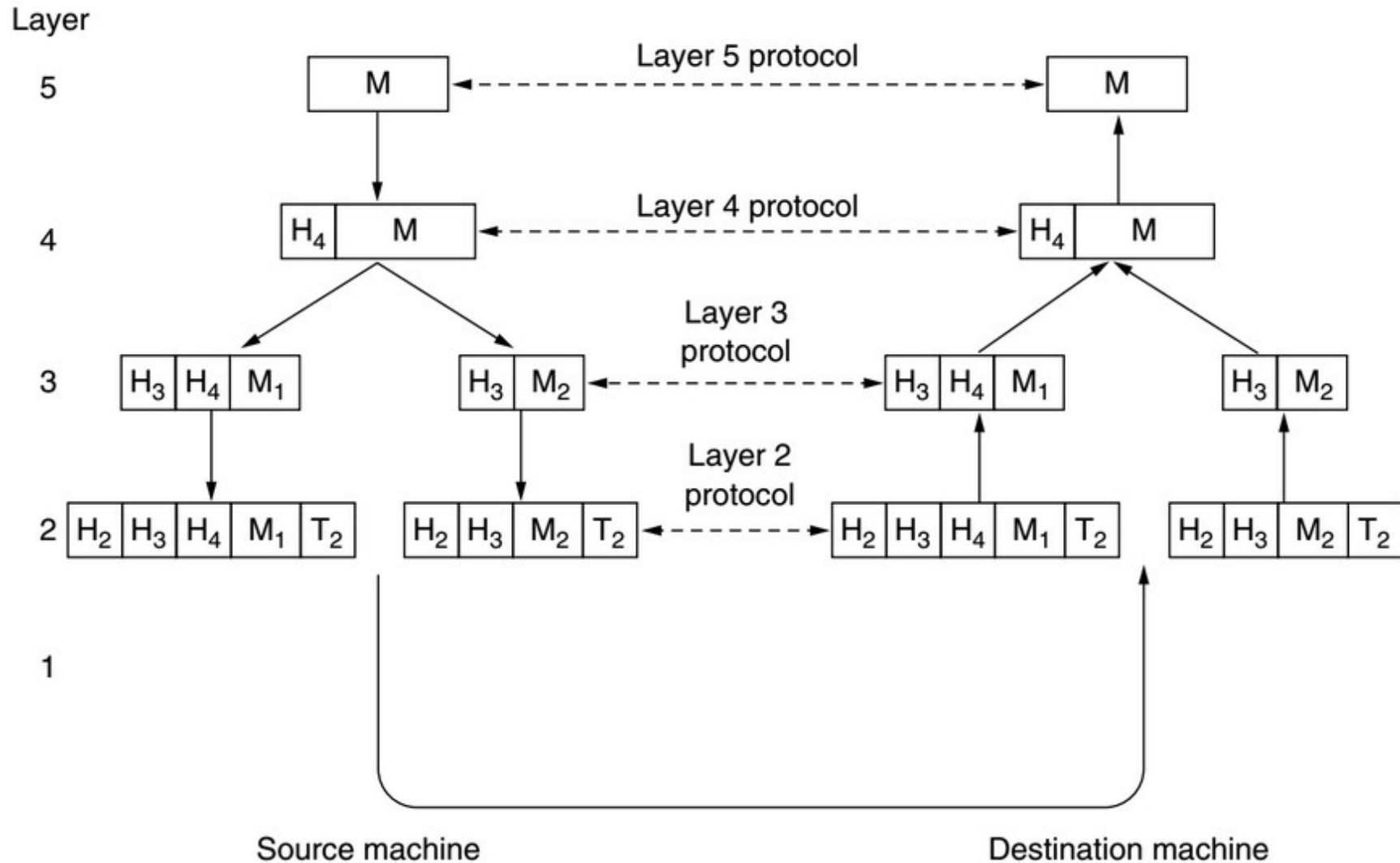
Networking – concept
Protocols

A protocol is a set of rules and standards that basically define a language that devices can use to communicate. There are a great number of protocols in use extensively in networking, and they are often implemented in different layers.
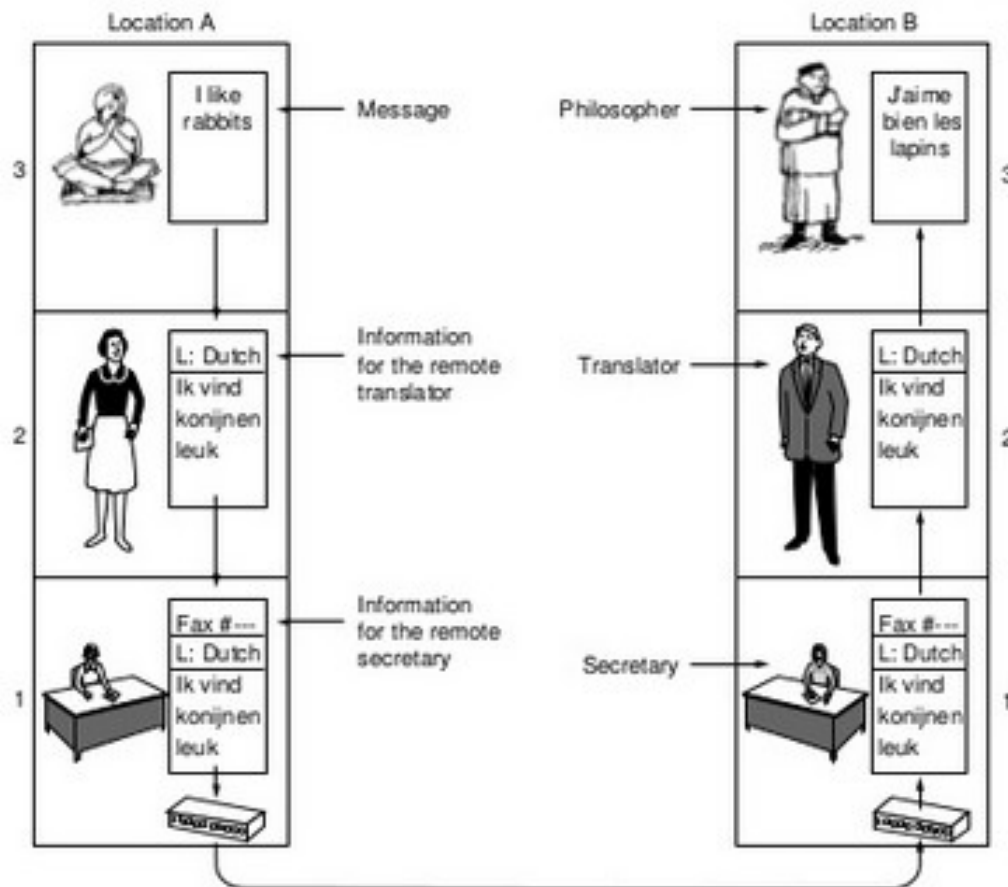
Networking – concept
Protocols

Networking – concept
Protocols- Examples

- Media Access Control

   Media access control is a communications protocol that is used to distinguish specific devices. Each device is supposed to get a unique MAC address during the manufacturing process that differentiates it from every other device on the internet.

- ICMP

   ICMP stands for internet control message protocol. It is used to send messages between devices to indicate the availability or error conditions. These packets are used in a variety of network diagnostic tools, such as ping and traceroute.

ΣMERTXE

Networking – concept
Protocols- Examples

- HTTP

  HTTP stands for hypertext transfer protocol. It is a protocol defined in the application layer that forms the basis for communication on the web.

- FTP

  FTP stands for file transfer protocol. It is also in the application layer and provides a way of transferring complete files from one host to another.

- DNS

  DNS stands for domain name system. It is an application layer protocol used to provide a human-friendly naming mechanism for internet resources. It is what ties a domain name to an IP address and allows you to access sites by name in your browser.

Networking – concept
Protocols- Examples

- SSH

  SSH stands for secure shell. It is an encrypted protocol implemented in the application layer that can be used to communicate with a remote server in a secure way. Many additional technologies are built around this protocol because of its end-to-end encryption and ubiquity.

- DHCP

  Dynamic Host Configuration Protocol (DHCP) is a client/server protocol that automatically provides an Internet Protocol (IP) host with its IP address and other related configuration information such as the subnet mask and default gateway.

- ARP

  A host wishing to obtain a physical address broadcastsan ARP request onto the TCP/IP network. The host on the network that has the IP address in the request then replies with its physical hardware address.

Networking – concept

Some questions from protocols

1) What type of cable or transmission media is used to connect hosts on the network?

2) How is data transmitted on the transmission media?

3) How do the hosts on the network know when to transmit data?

4) How does each host know how much data can be transmitted at a time?

5) How can hosts using different operating systems communicate?

6) How can a host check the data received for transmissions?

ΣMERTXE

Stay Connected

About us: Emertxe is India's one of the top IT finishing schools & self learning kits provider. Our primary focus is on Embedded with diversification focus on Java, Oracle and Android areas
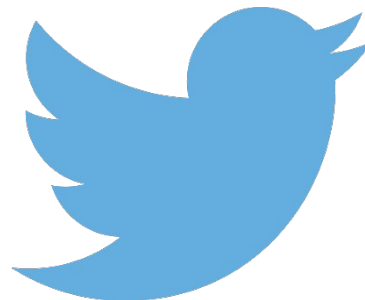
Emertxe Information Technologies,

No-1, 9th Cross, 5th Main,
Jayamahal Extension,
Bangalore, Karnataka 560046

T: +91 80 6562 9666
E: training@emertxe.com

https://www.facebook.com/Emertxe   https://twitter.com/EmertxeTweet   https://www.slideshare.net/EmertxeSlides

Thank You