Linux Internals & Networking

System programming using Kernel interfaces

Team Emertxe

Inter Process Communications (IPC)

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

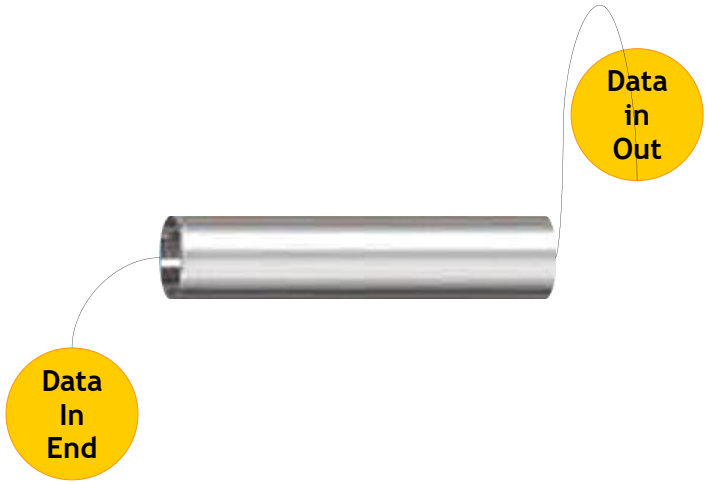
Some are due to “UNIX war” reasons also

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

Helps in portability as well

Inter Process Communications
Pipes

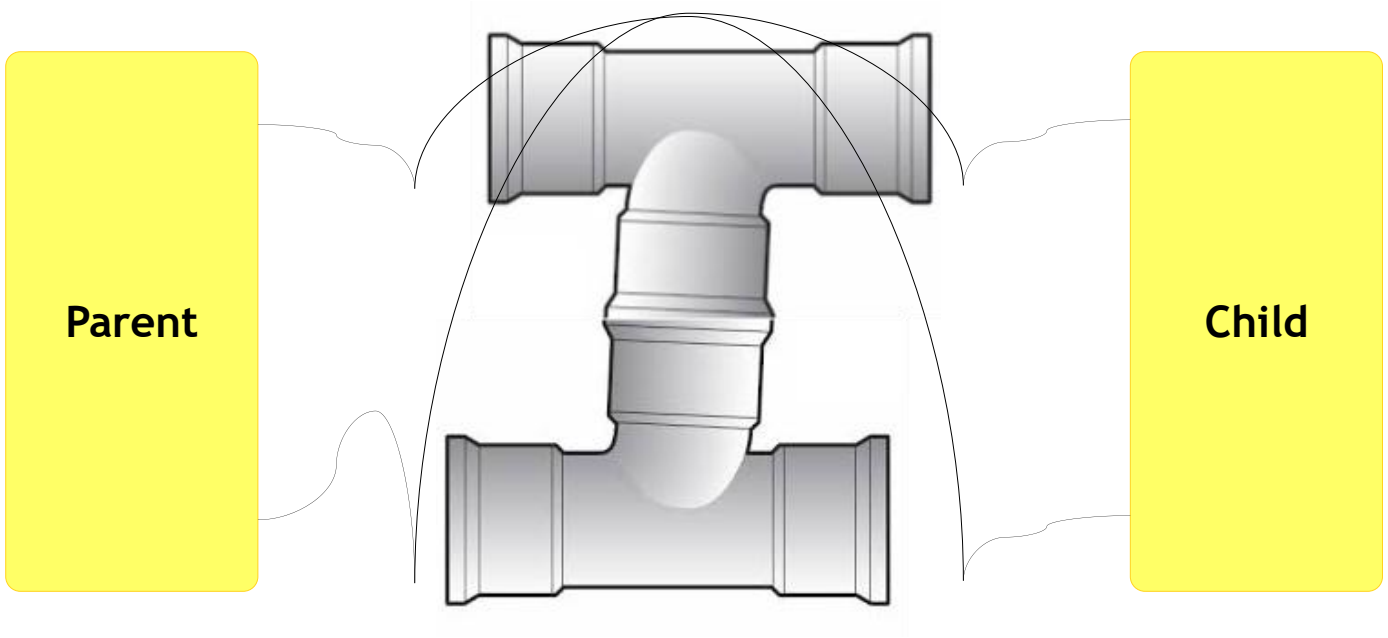
- A pipe is a communication device that permits unidirectional communication
- Data written to the “write end” of the pipe is read back from the “read end”
- Pipes are serial devices; the data is always read from the pipe in the same order it was written



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 - Pros & Cons

PROS

- Naturally synchronized
- Simple to use and create
- No extra system calls required to communicate (read/write)

CONS

- Less memory size (4K)
- Only related process can
- communicate.
- Only two process can communicate
- One directional communication
- Kernel is involved

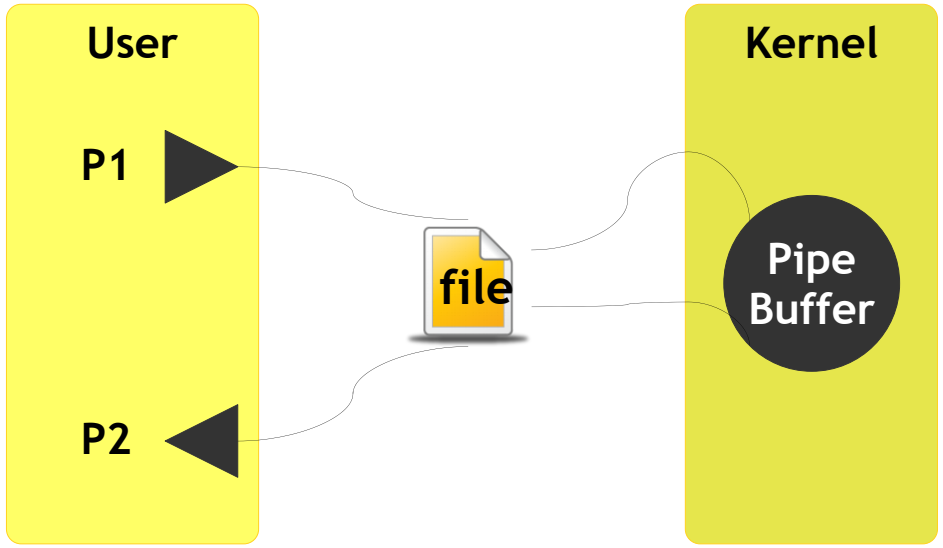
Inter Process Communications

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 – Working



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

Inter Process Communications
FIFO - Access

- 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
```

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 - Pros & Cons

PROS

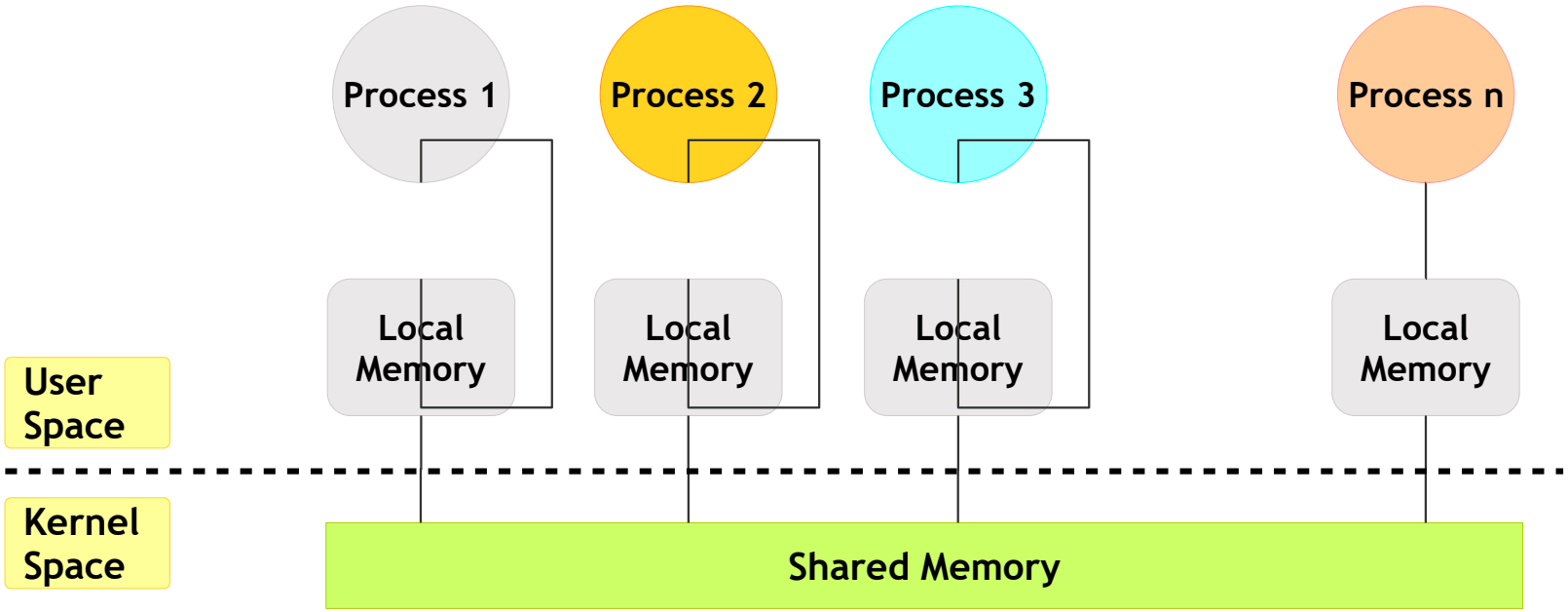
- Naturally synchronized
- Simple to use and create
- Unrelated process can
- communicate.
- No extra system calls required to communicate (read/write)
- Work like normal file

CONS

- Less memory size (4K)
- Only two process can communicate
- One directional communication
- Kernel is involved

Inter Process Communications

Shared vs Local Memory



Inter Process Communications

Shared Memories - Procedure

- Create
- Attach
- Read/Write
- Detach
- Remove

} **95%**

Inter Process Communications

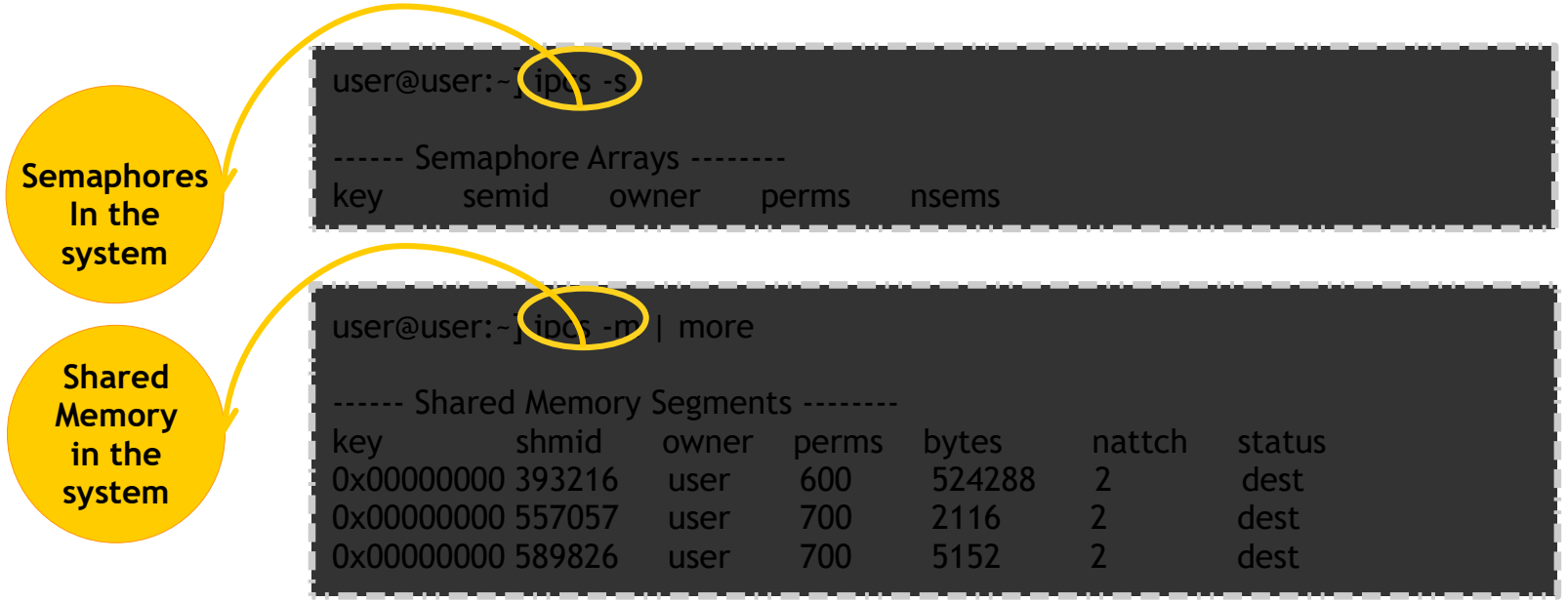
Shared Memories – Function calls

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:



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'

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

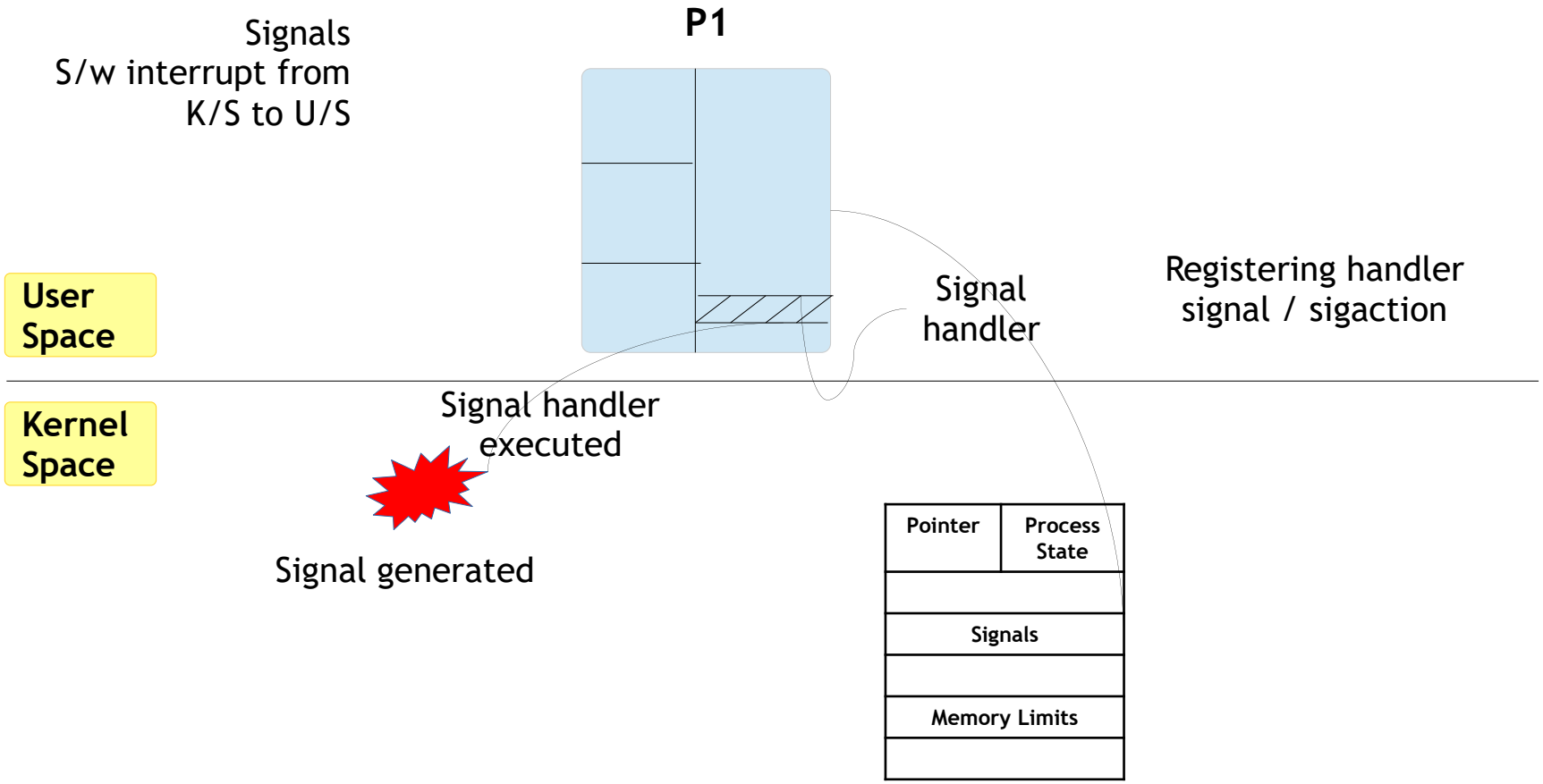
- When a process receives a signal, it processes
- Immediate handling
- For all possible signals, the system defines a default disposition or action to take when a signal occurs
- There are four possible default dispositions:
  - Exit: Forces process to exit
  - Core: Forces process to exit and create 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
Handling



Signals
Handler

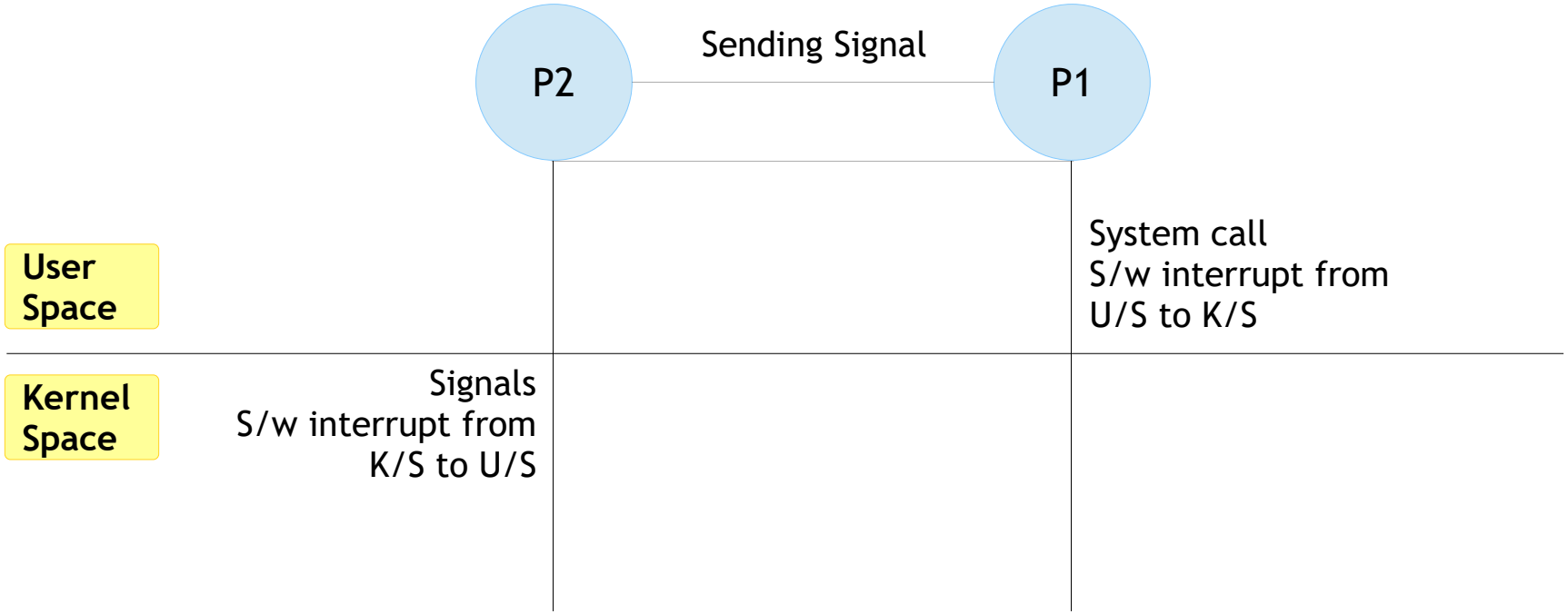
- A signal handler should perform the minimum work necessary to respond to the signal
- The control will return to the main program (or terminate the program)
- In most cases, this consists simply of recording the fact that a signal occurred or some minimal handling
- The main program then checks periodically whether a signal has occurred and reacts accordingly
- Its called as **asynchronous handling**

Signals
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
vs system calls



Signals
Self Signaling

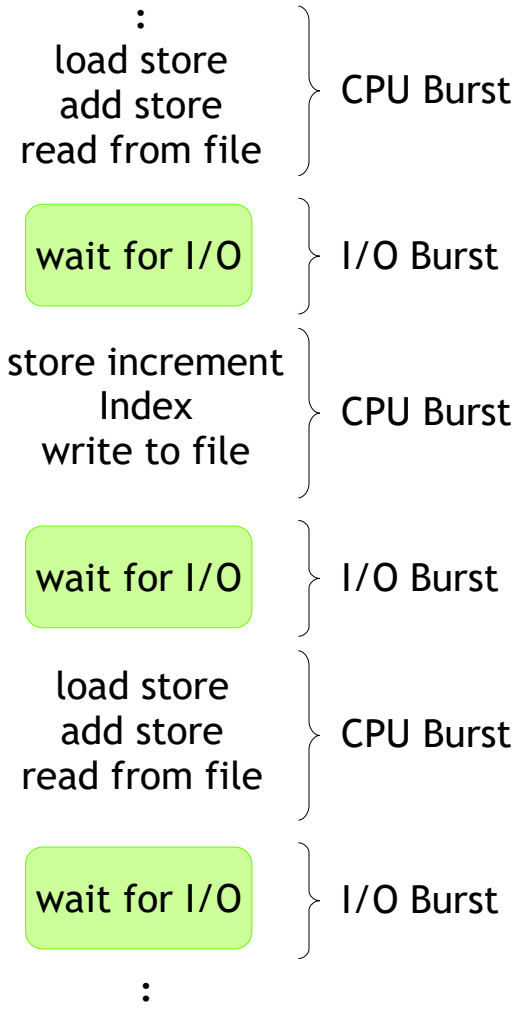
- A process can send or detect signals to itself
- This is another method of sending signals
- There are three functions available for this purpose
- This is another method, apart from ‘kill’

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

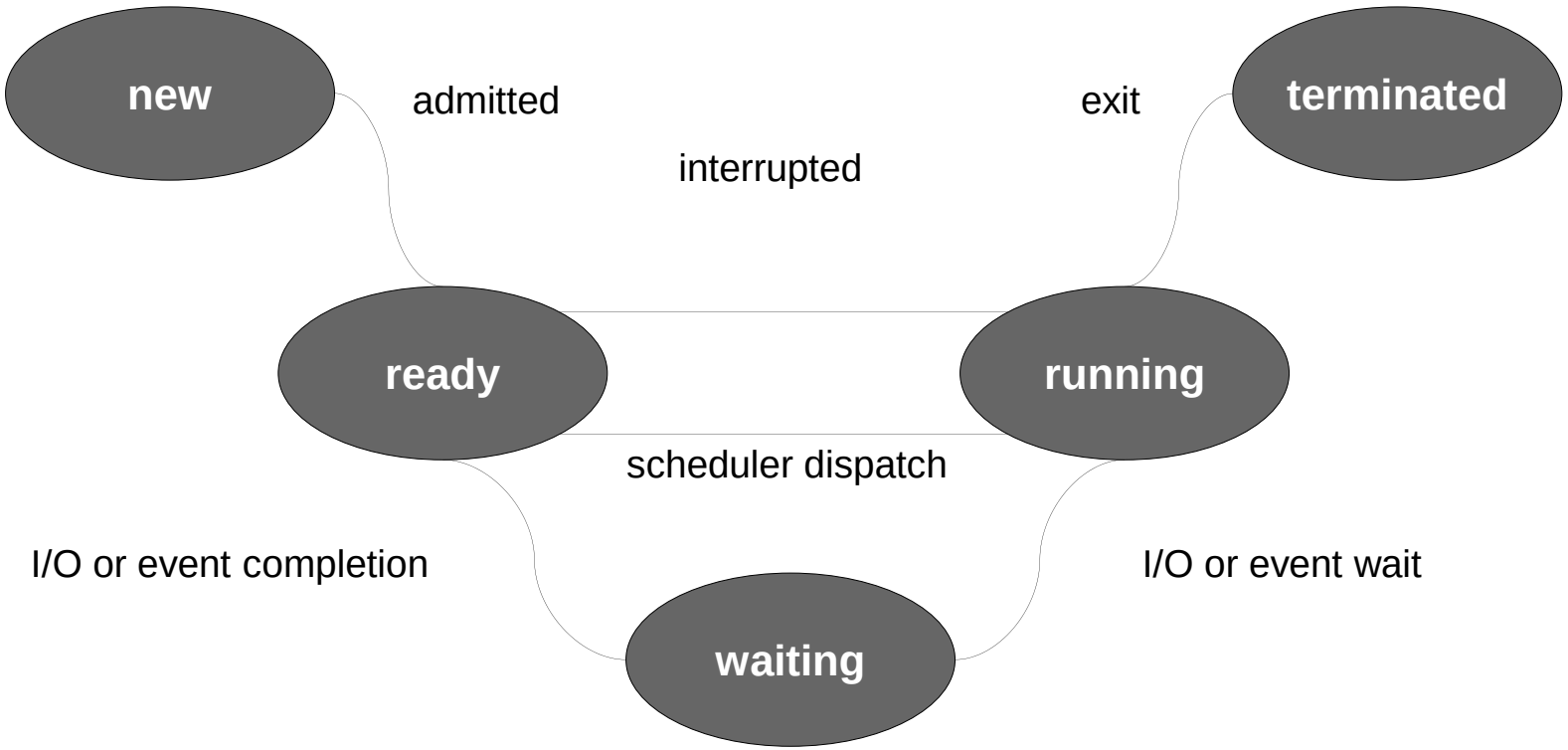
Process Management - Concepts

Process Management - Concepts
CPU Scheduling

- 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



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

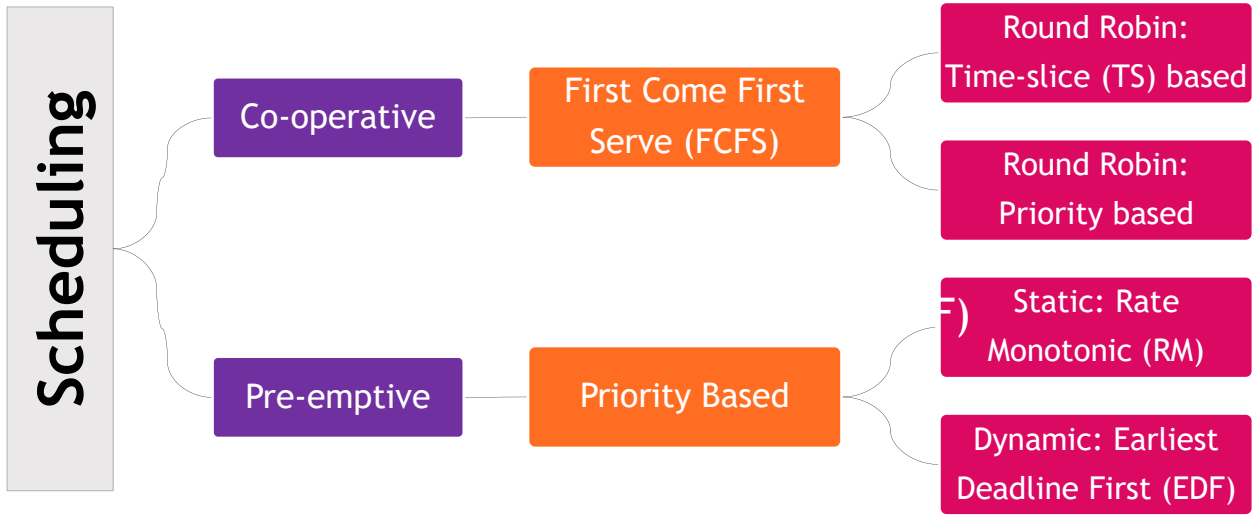
Process Management - Concepts

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

Process Management - Concepts

Scheduling - Types



Process Management - Concepts

Scheduling – Types – Co-operative vs Pre-emptive

- 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

Process Management - Concepts

Scheduling – Types – FCFS

- 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

Process Management - Concepts

Scheduling – Types – RR: Time Sliced

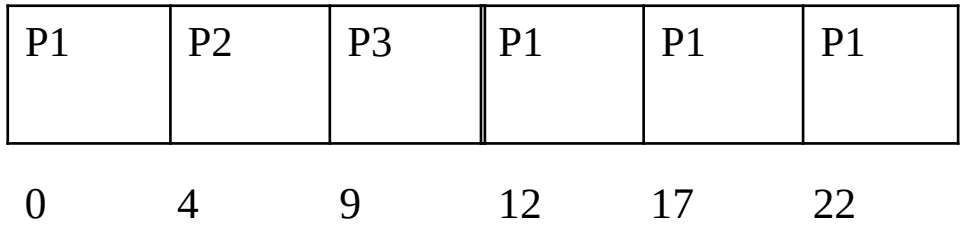
- Processes are scheduled based on time-slice, but they are time-bound
- This time slicing is similar to FCFS except that the scheduler forces process to give up the processor based on the timer interrupt
- It does so by preempting the current process (i.e. the process actually running) at the end of each time slice
- The process is moved to the end of the priority level

Process Management - Concepts

Scheduling – Types – RR: Time Sliced

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



Process Management - Concepts

Scheduling – Types – RR: Priority

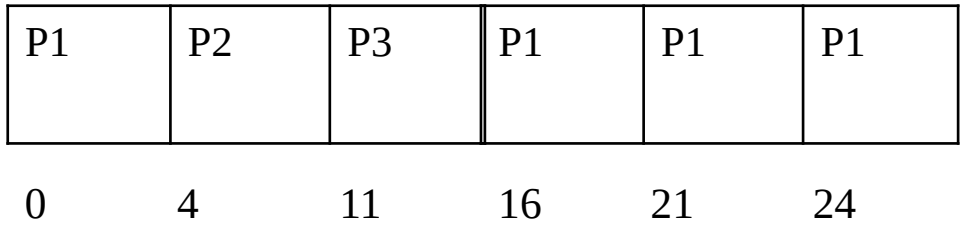
- Processes are scheduled based on RR, but priority attached to it
- While processes are allocated based on RR (with specified time), when higher priority task comes in the queue, it gets pre-empted
- The time slice remain the same

Process Management - Concepts

Scheduling – Types – RR: Time Sliced

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



Process Management - Concepts

Scheduling – Types – Pre-emptive

- 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

Process Management - Concepts

Scheduling – Types – Pre-emptive

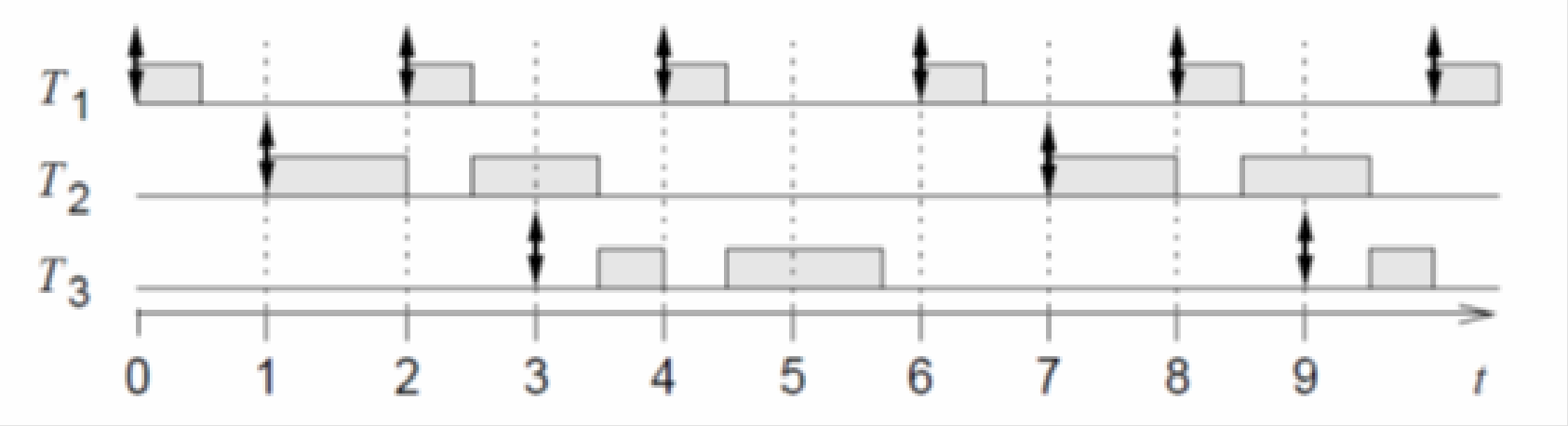
- Earliest Deadline First (EDF) scheduling:
  - This kind of scheduler tries to give execution time to the task that is most quickly approaching its deadline
  - This is typically done by the scheduler changing priorities of tasks on-the-fly as they approach their individual deadlines

Process Management - Concepts

Scheduling – Types – Rate Monotonic (RM)

.T1 preempts T2 and T3.

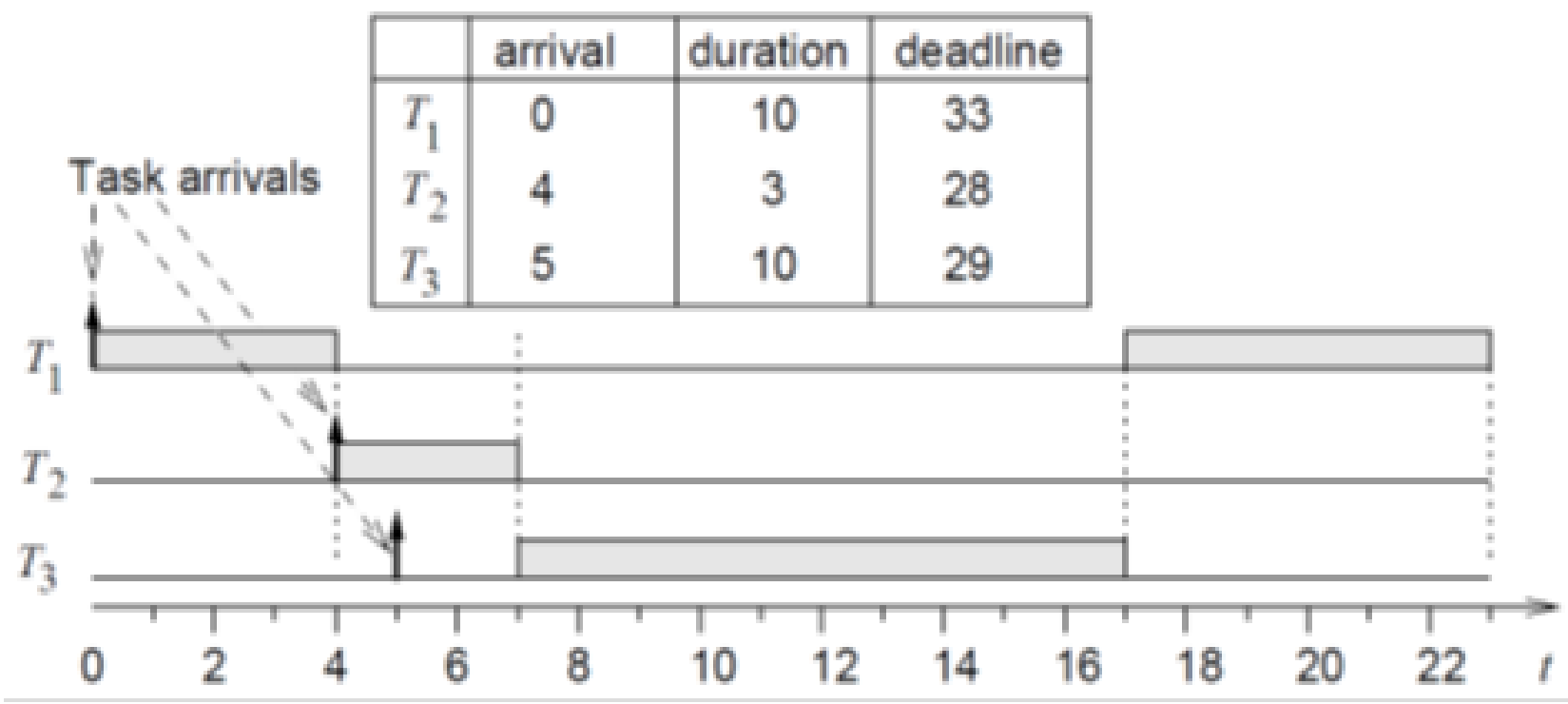
.T2 and T3 do not preempt each other.



Process Management - Concepts

Scheduling – Types – Earliest Deadline First (EDF)

	arrival	duration	deadline
T_1	0	10	33
T_2	4	3	28
T_3	5	10	29



Thank You