Linux Internals & Networking

System programming using Kernel interfaces

Team Emertxe

Contents

Linux Internals & Networking

Contents

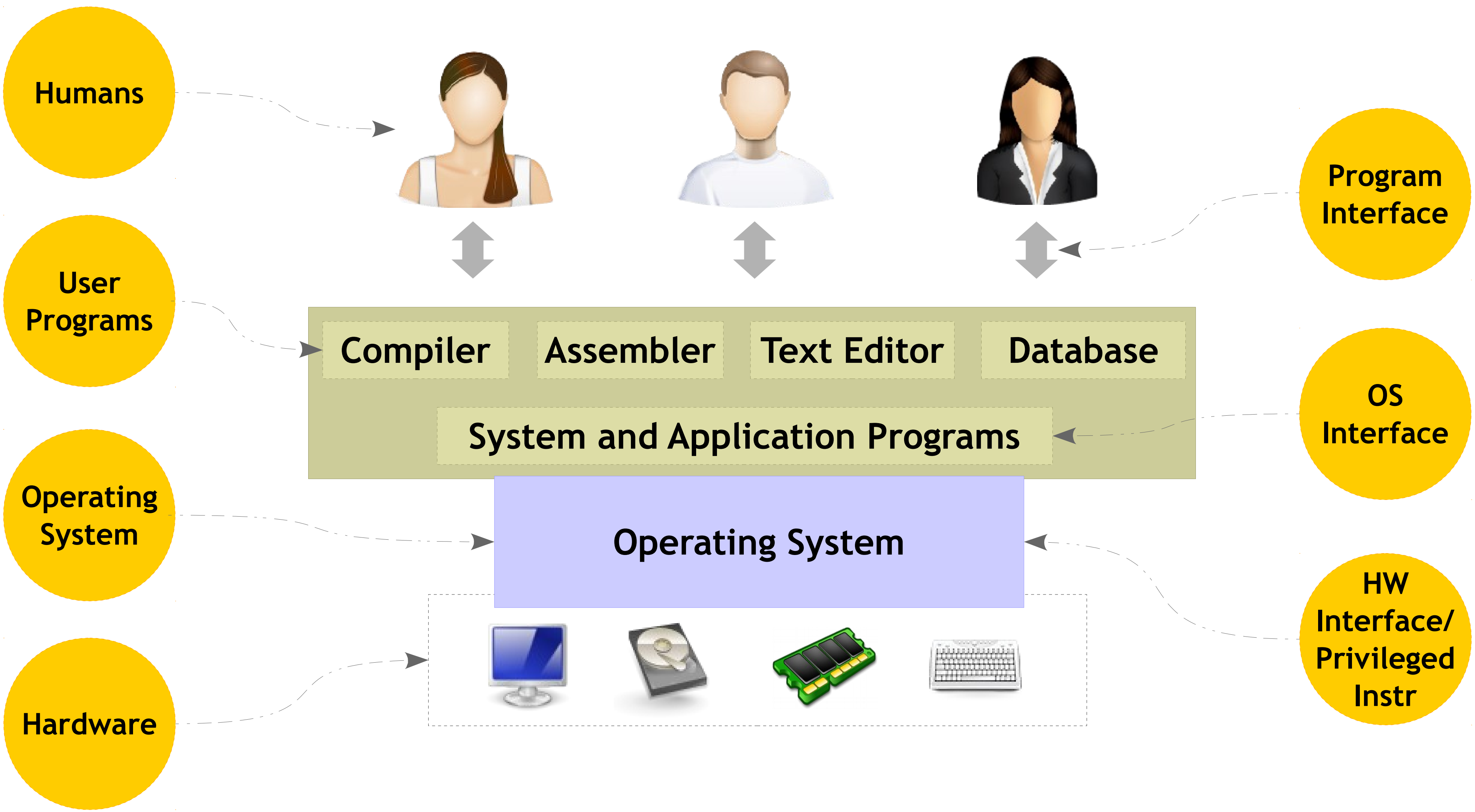
Introduction

Introduction

Let us ponder ...

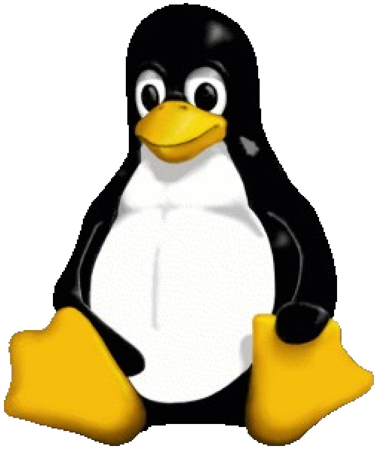
- What exactly is an Operating System (OS)?
- Why do we need OS?
- How would the OS would look like?
- Is it possible for a team of us (in the room) to create an OS of our own?
- Is it necessary to have an OS running in a Embedded System?
- Will the OS ever stop at all?

Introduction
Operating System



Introduction
What is Linux?

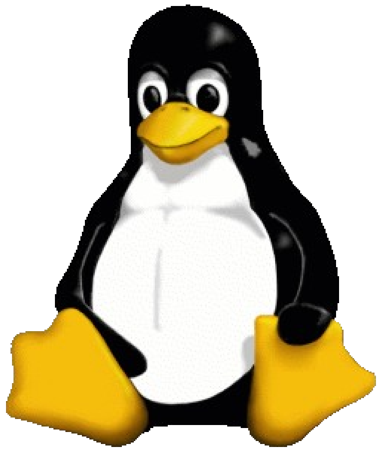
- Linux is a free and open source operating system that is causing a revolution in the computer world
- Originally created by Linus Torvalds with the assistance of developers called community
- This operating system in only a few short years is beginning to dominate markets worldwide



Introduction

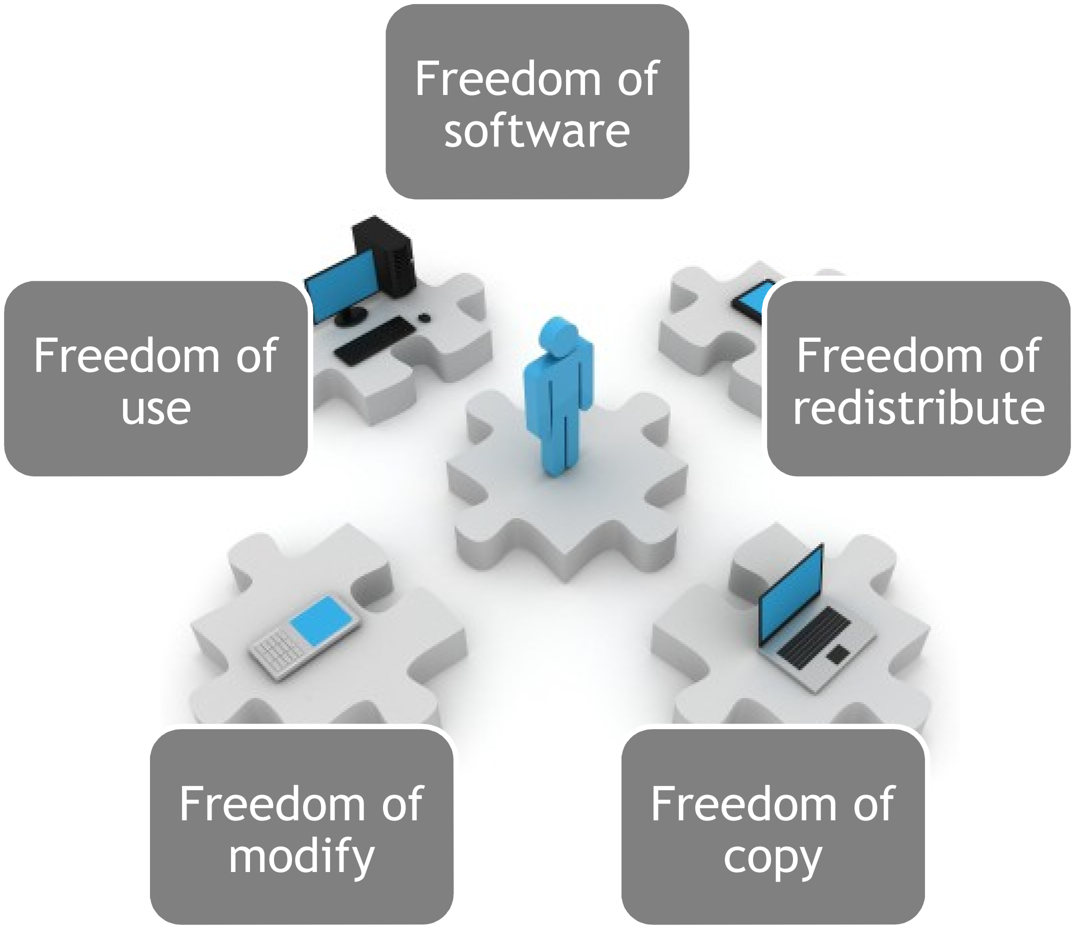
Why use Linux?

- 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

What is Open Source?



Introduction

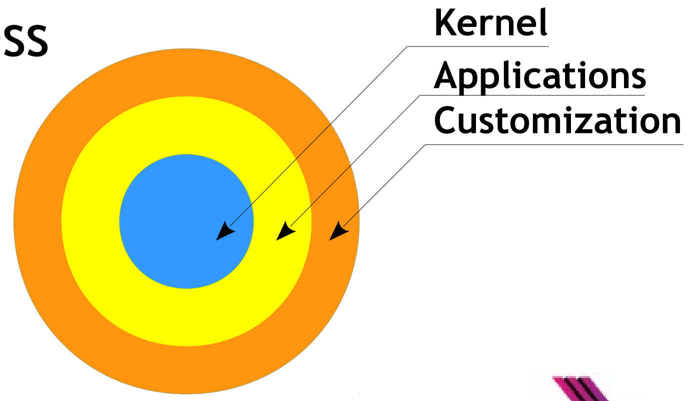
Open Source - How it all started?

- With GNU (GNU is not UNIX)
- Richard Stallman made the initial announcement in 1983, Free Software Foundation (FSF) got formed during 1984
- Volunteer driven GNU started developing multiple projects, but making it as an operating system was always a challenge
- During 1991 a Finnish Engineer Linus Torvalds developed core OS functionality, called it as “Linux Kernel”
- Linux Kernel got licensed under GPL, which laid strong platform for the success of Open Source
- Rest is history!

Introduction

Open Source - How it evolved?

- Multiple Linux distributions started emerging around the Kernel
- Some applications became platform independent
- Community driven software development started picking up
- Initially seen as a “geek-phenomenon”, eventually turned out to be an engineering marvel
- Centered around Internet
- Building a business around open source started becoming viable
- Redhat set the initial trend in the OS business



Introduction

Open Source - Where it stands now?

OS
- CANONICAL
- redhat
- android
- Novell

Databases
- EnterpriseDB — The Enterprise PostgreSQL Company
- MySQL
- Hortonworks
- VoltDB

Server/Cloud
- openNMS
- openstack
- cloudstack — open source cloud computing
- OPSCODE

Enterprise
- pentaho — Powerful Analytics Made Easy
- Alfresco
- SUGARCRM
- XWIKI

Consumer
- Apache OpenOffice
- LibreOffice

Education
- canvas by Instructure
- docebo
- moodle

CMS
- Joomla!
- AUTOMATTIC
- MediaWiki

eCommerce
- Magento — eCommerce Platform for Growth
- spree
- opencart

Introduction
GPL

- Basic rights under the GPL – access to source code, right to make derivative works
- Reciprocity/Copy-left
- Purpose is to increase amount of publicly available software and ensure compatibility
- Licensees have right to modify, use or distribute software, and to access the source code

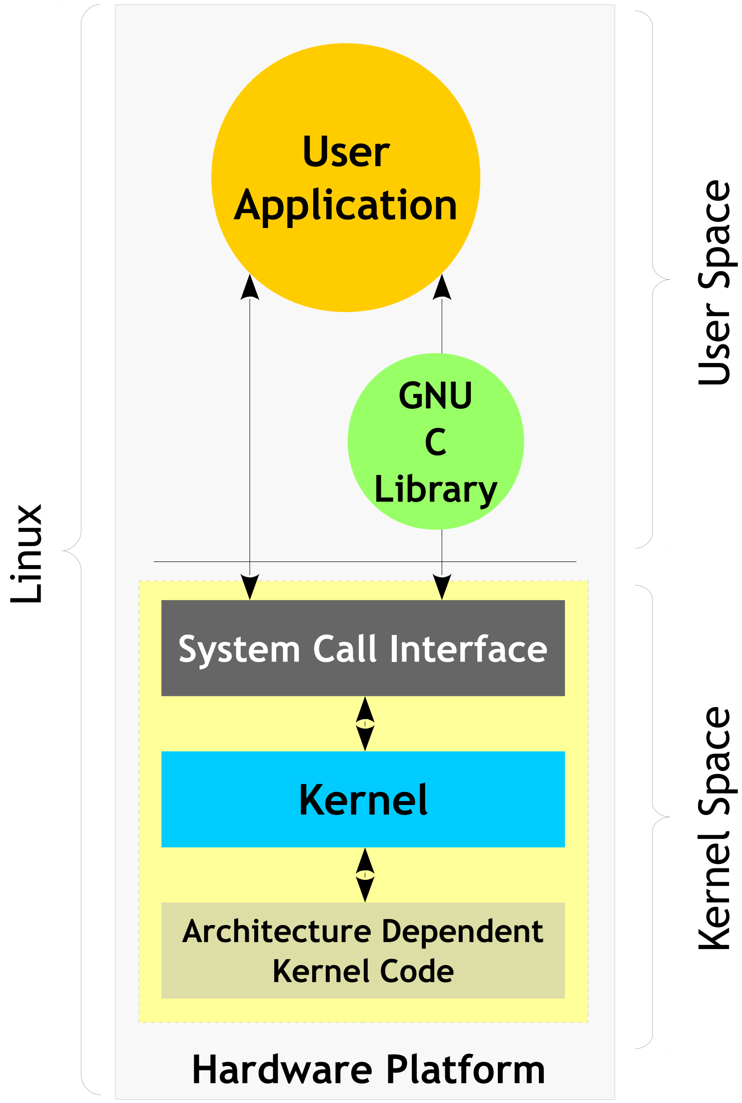
Introduction

GPL - Issues

- Linking to GPL programs
- No explicit patent grant
- Does no discuss trademark rights
- Does not discuss duration
- Silent on sub-licensing
- Relies exclusively on license law, not contract

Introduction

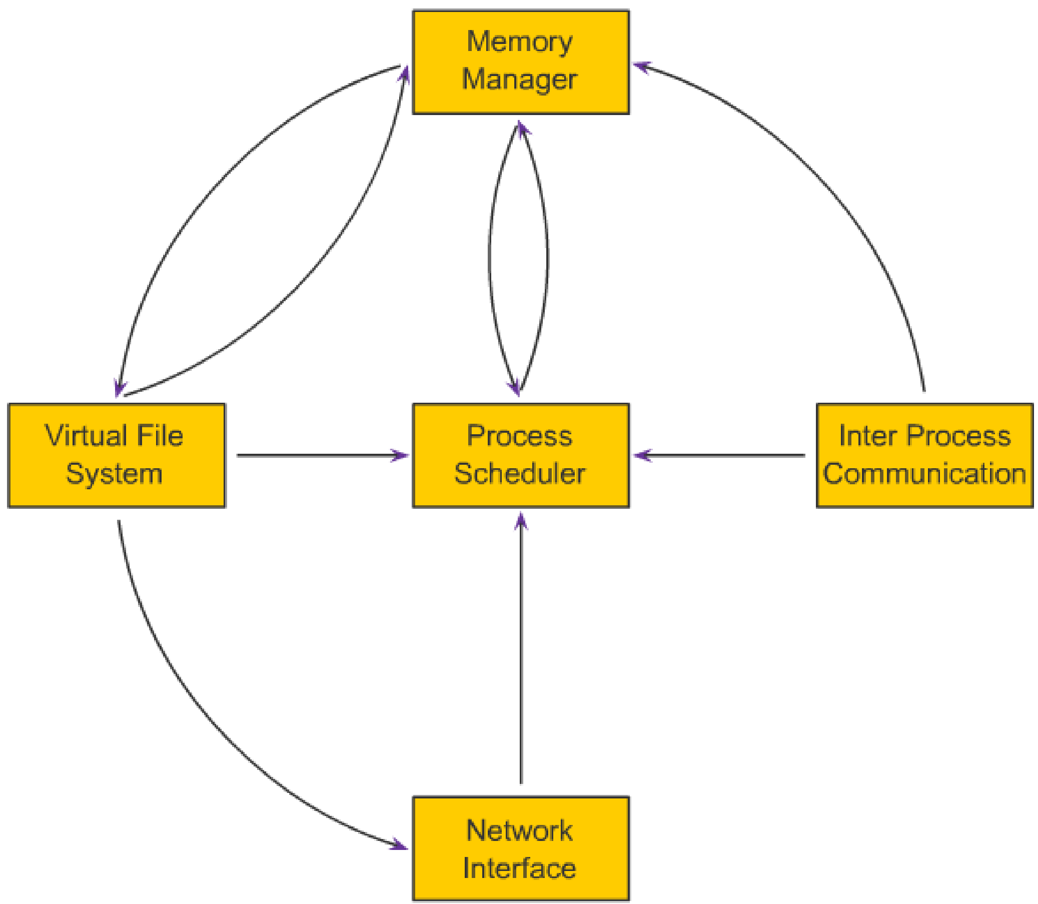
Linux Components



- **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)

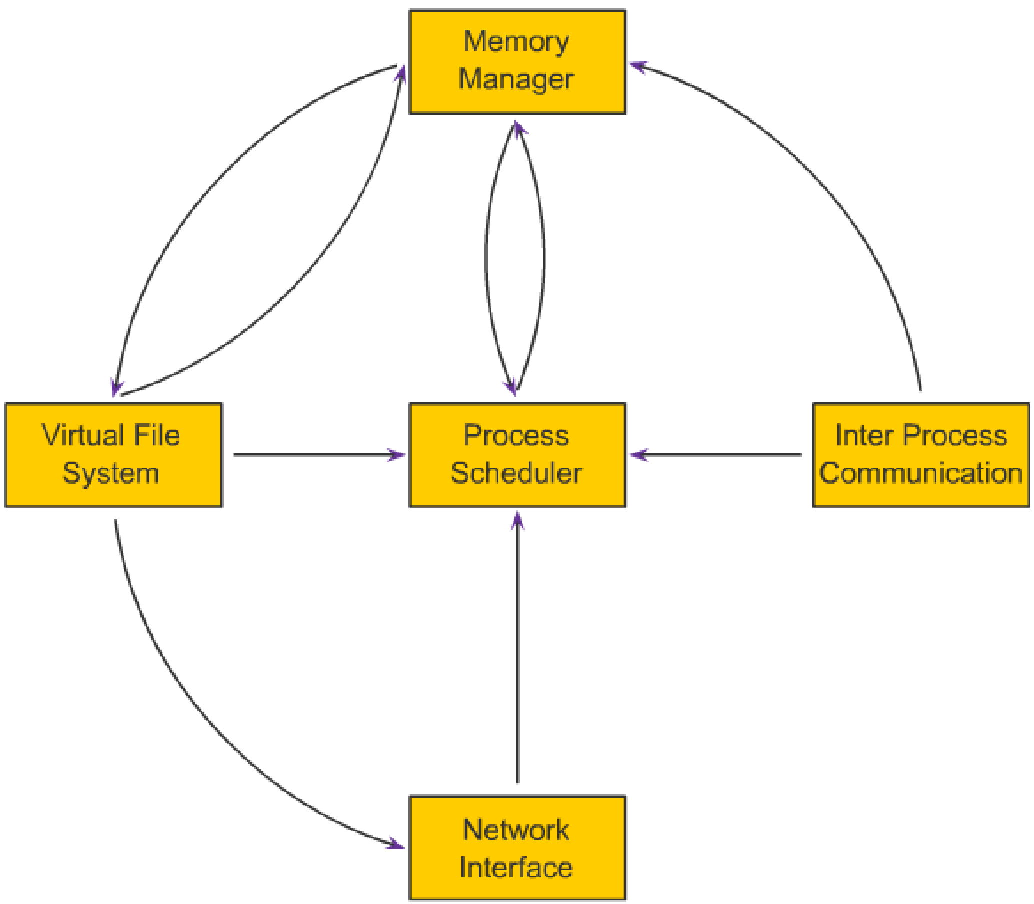
Introduction

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



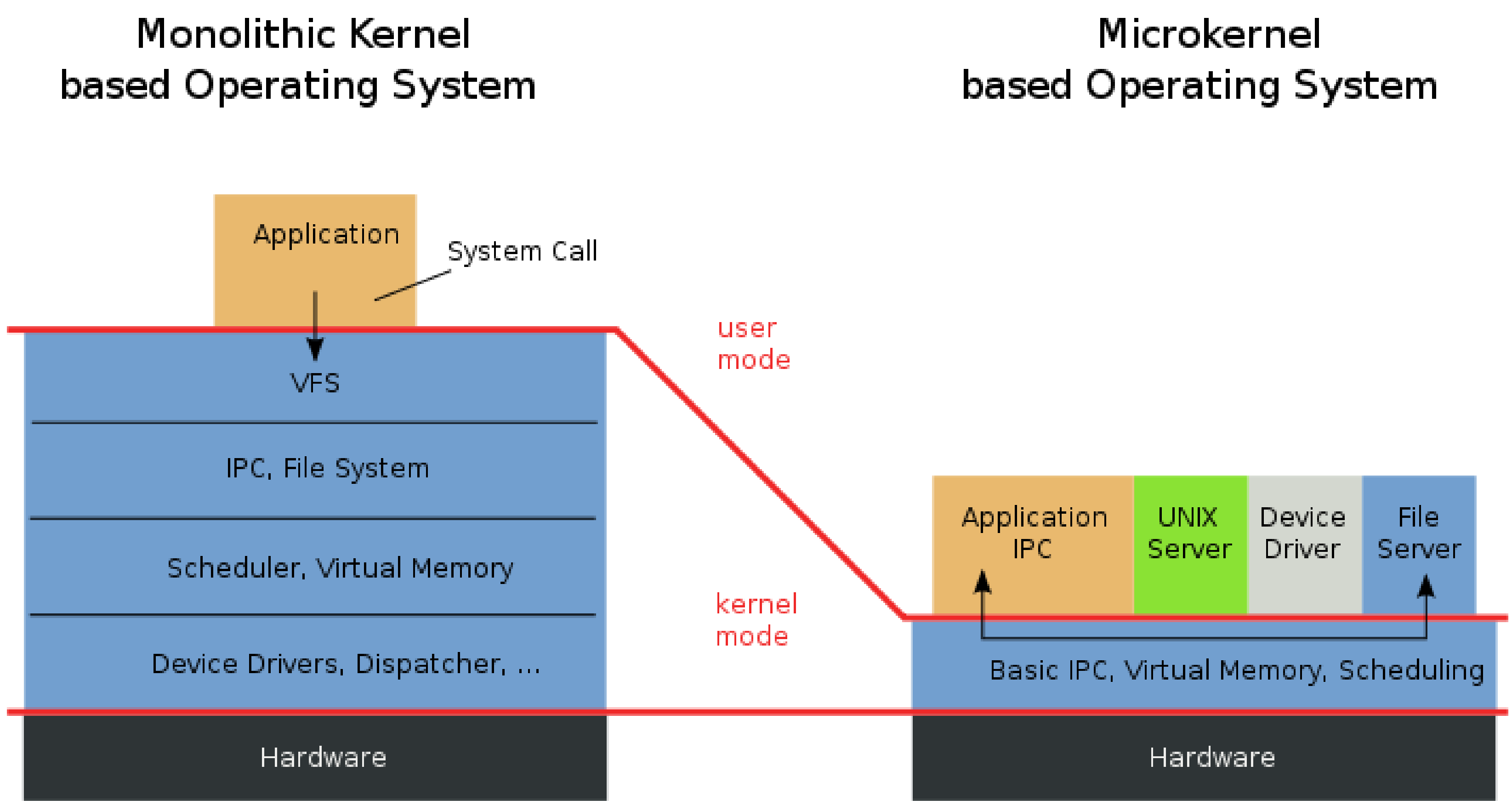
- Network Interface (NET):
  - provides access to several networking standards and a variety of network hardware
- Inter Process Communications (IPC):
  - supports several mechanisms for process-to-process communication on a single Linux system

Introduction

Linux Kernel Architecture

- 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
Linux Kernel Architecture



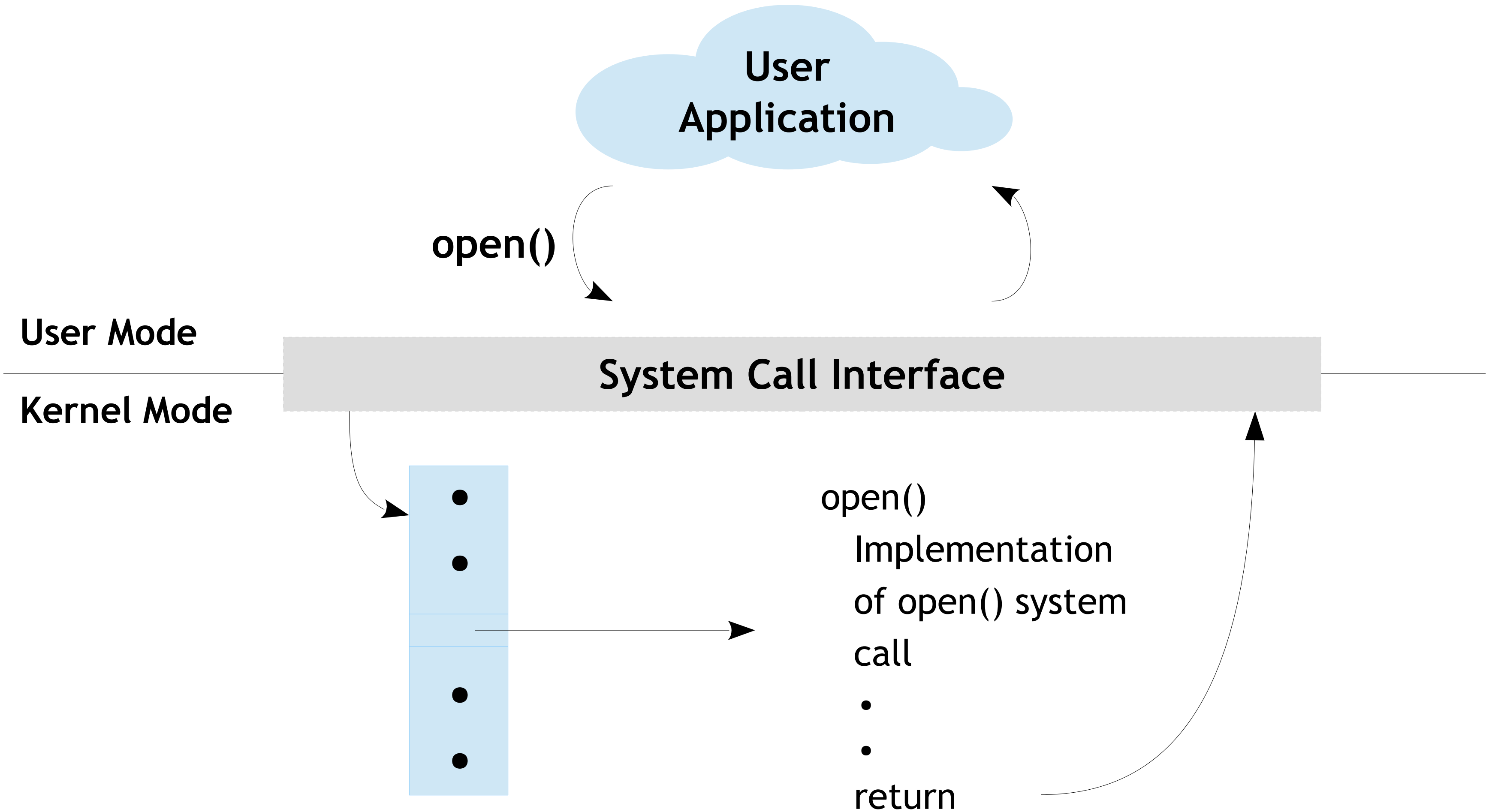
System Calls

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

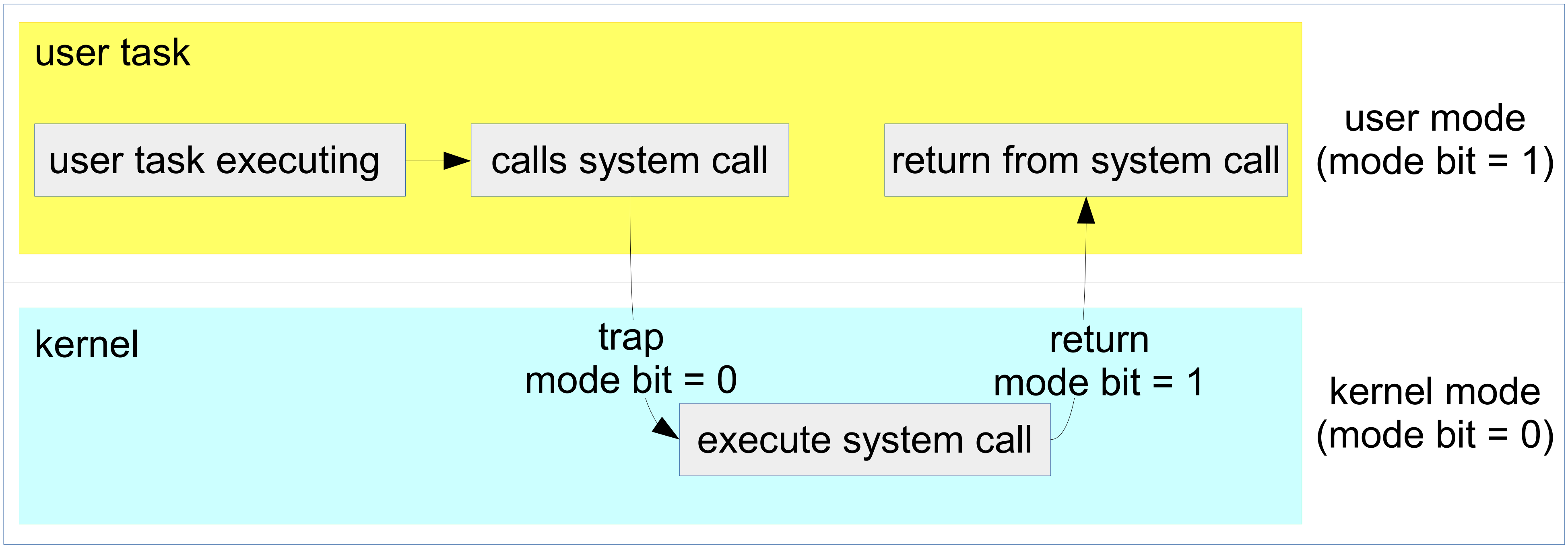
For a OS programmer, calling a system call is no different from a normal function call. But the way system call is executed is way different.

System calls



System Call

Calling Sequence



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’

System Call
vs Library Function

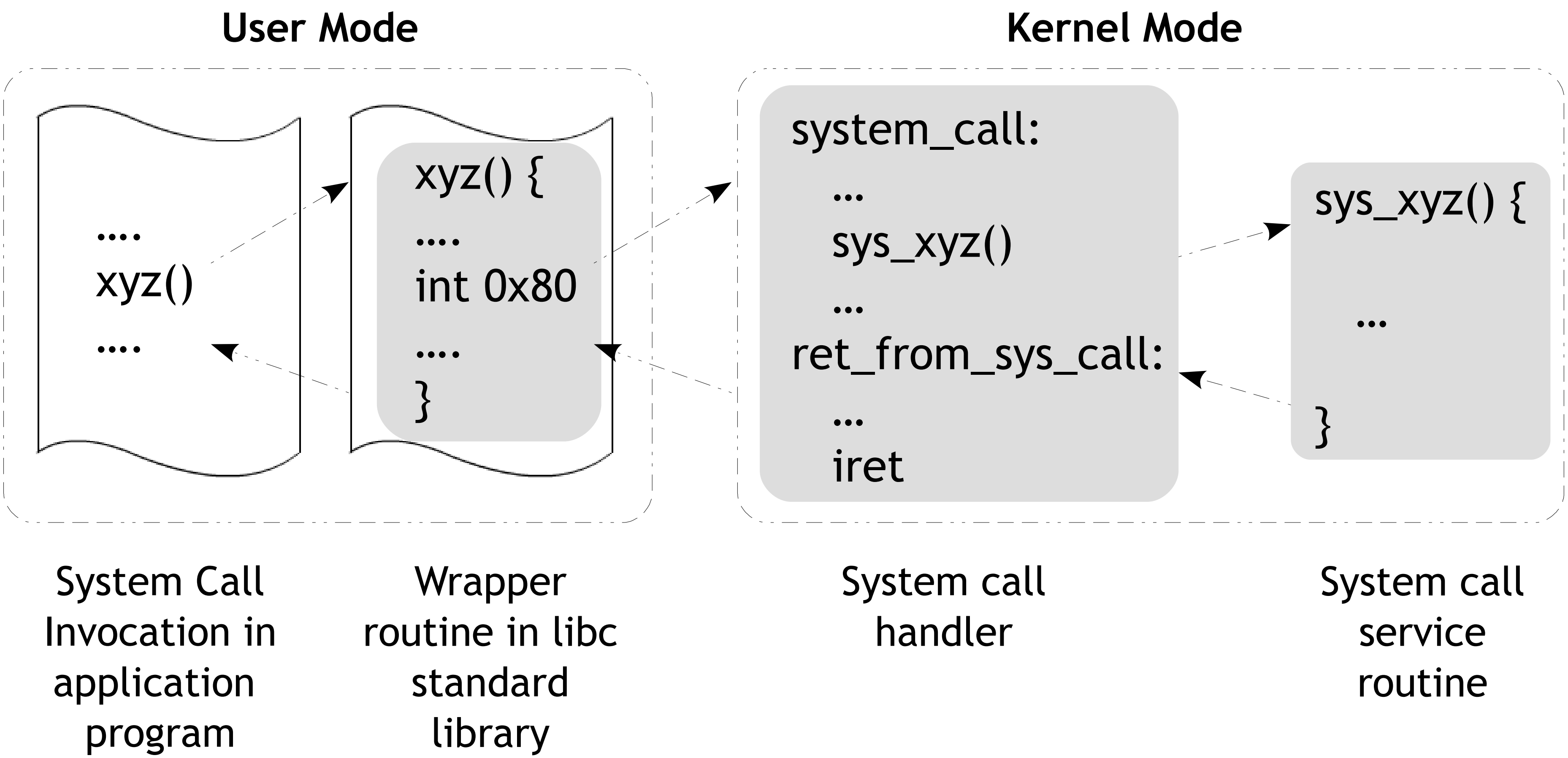
- A library function is an ordinary function that resides in a library external to your program. A call to a library function is just like any other function call
- A system call is implemented in the Linux kernel and a special procedure is required in to transfer the control to the kernel
- Usually, 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



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

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..

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

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

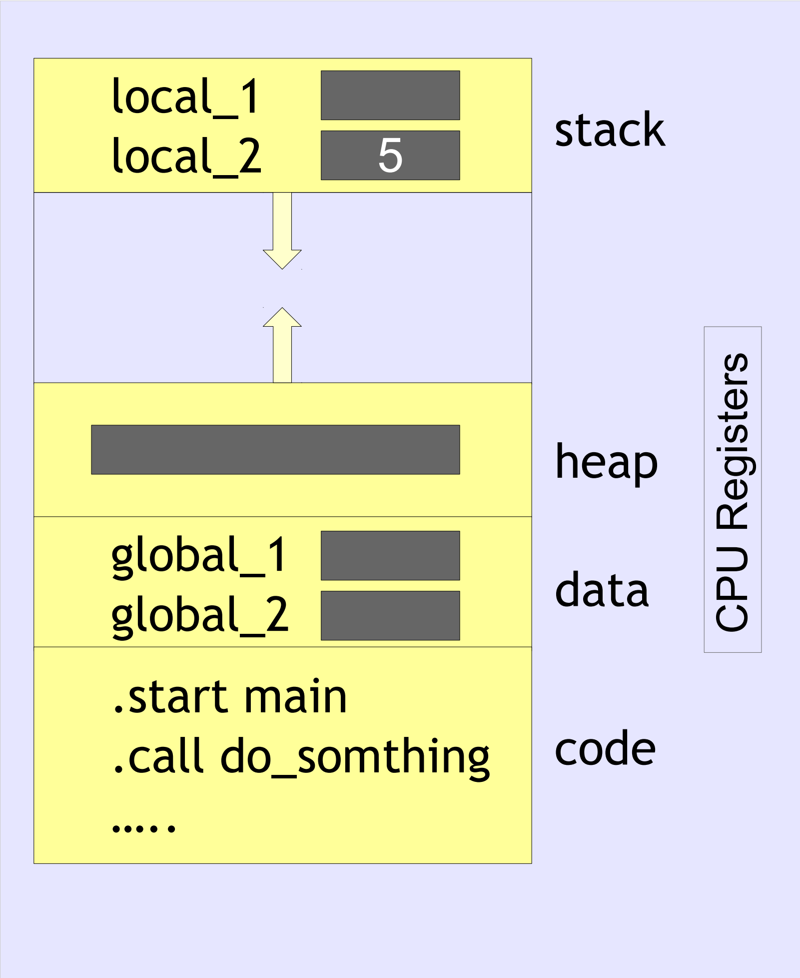
Factor	Process	Program
Storage	Dynamic Memory	Secondary Memory
State	Active	Passive

Process
vs Program

Program

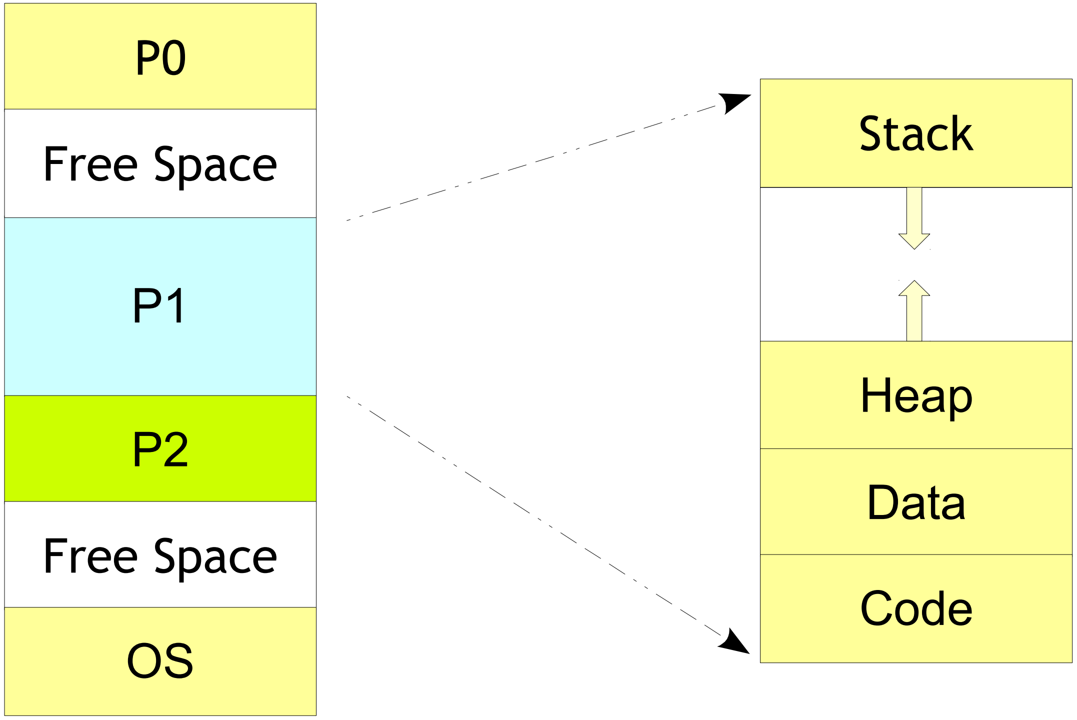
```
int global_1 = 0;
int global_2 = 0;

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

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

        do_somthing();
        …..
}
```

Task



Process

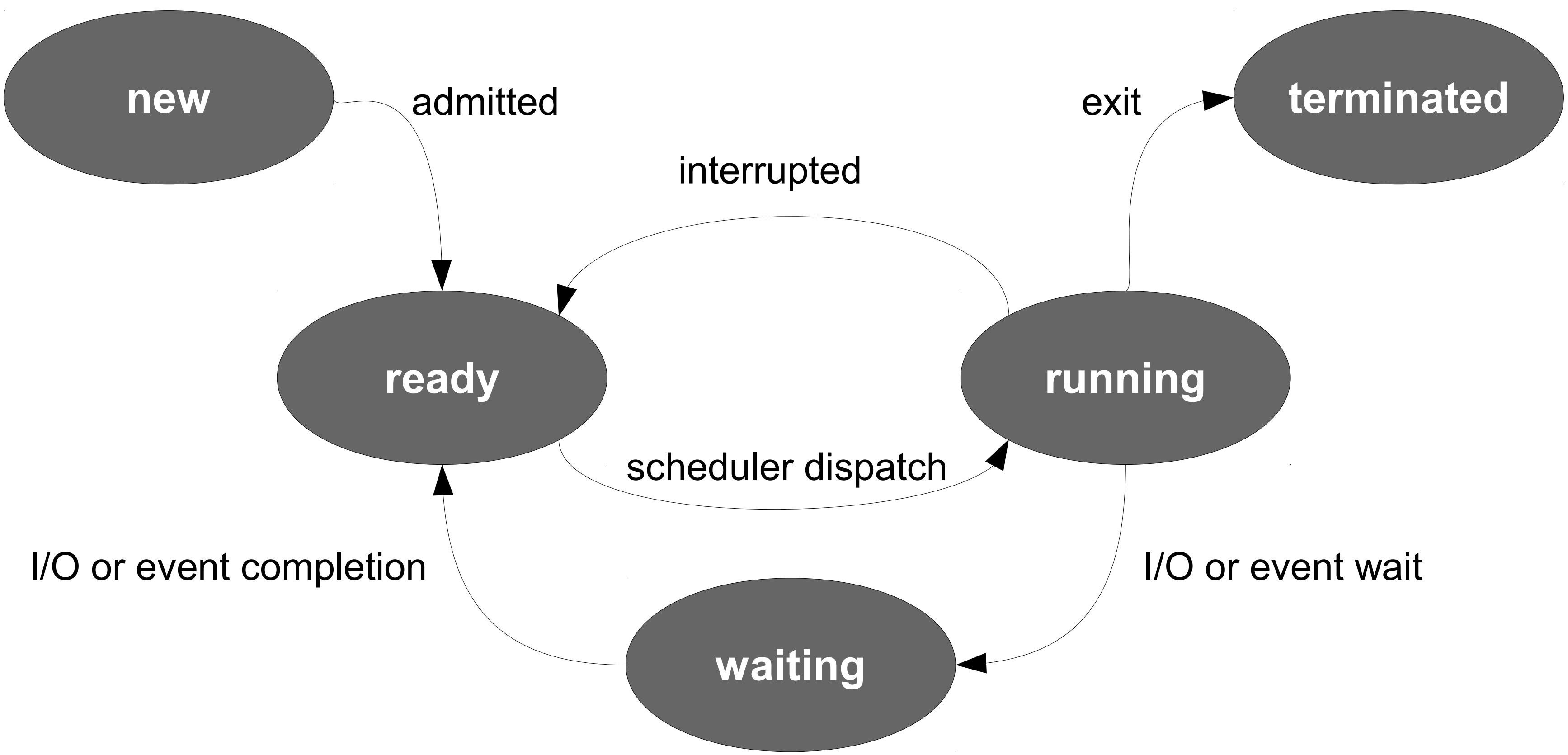
More processes in memory!



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

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

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

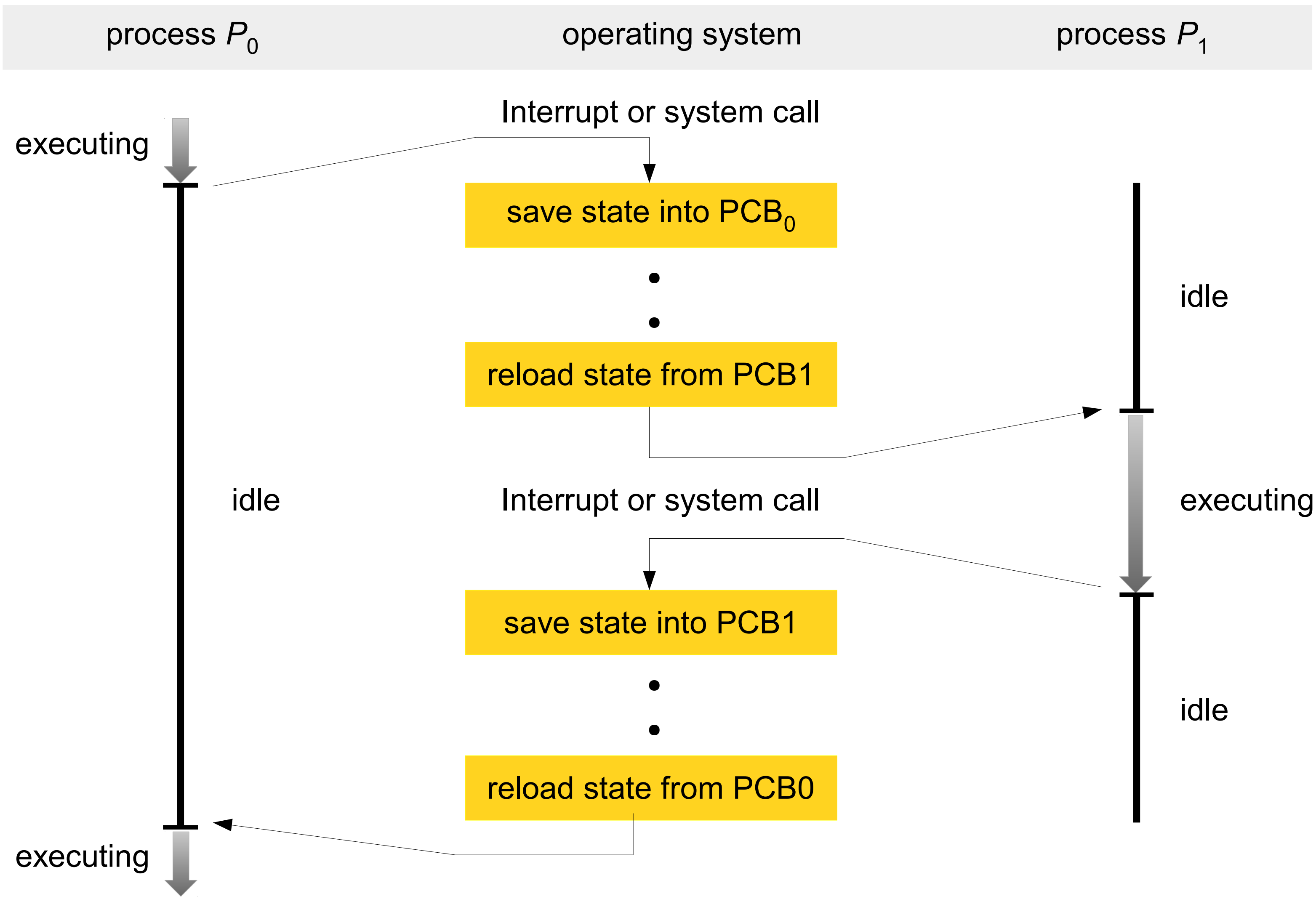
Parent Process ID

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

Process
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()

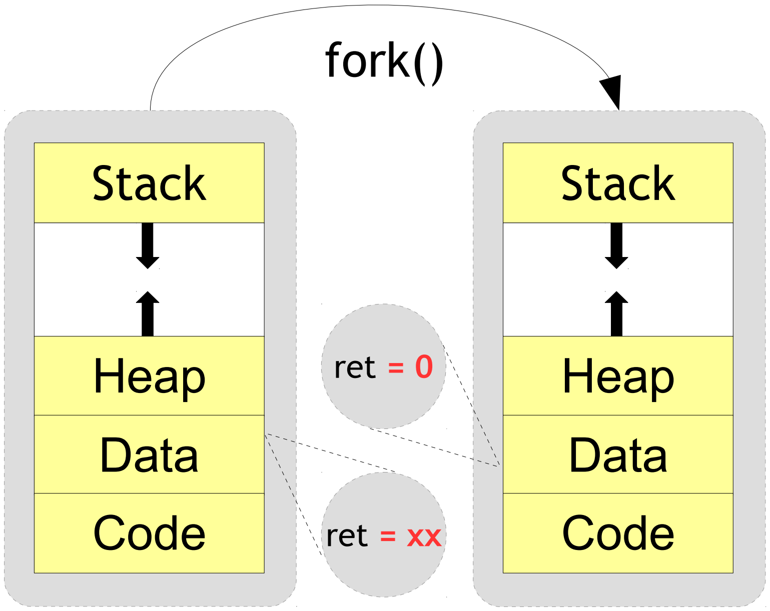
- fork makes a child process that is an exact copy of its parent process
- When a program calls fork, a duplicate process, called the child process, is created
- The parent process continues executing the program from the point that fork was called
- The child process, too, executes the same program 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

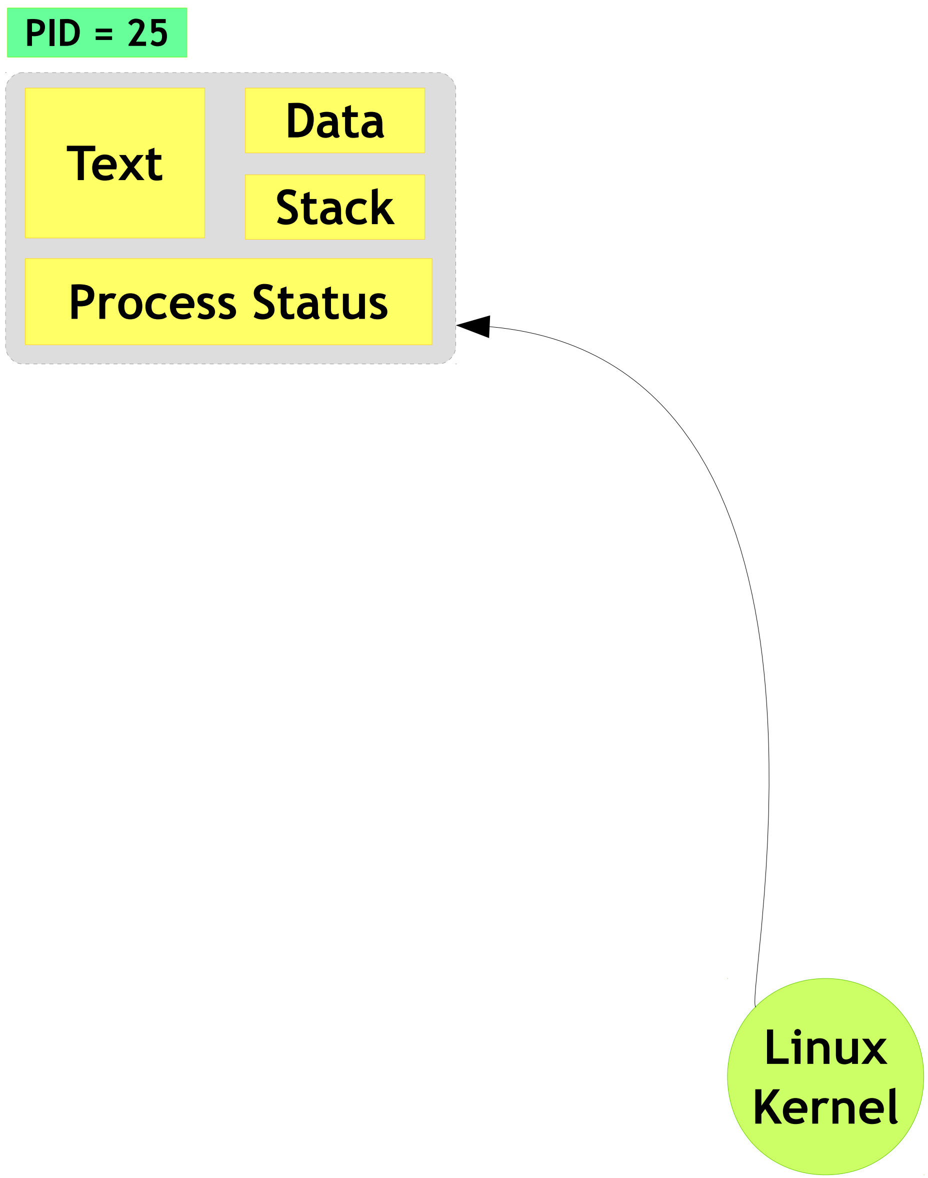
Creation - fork()

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

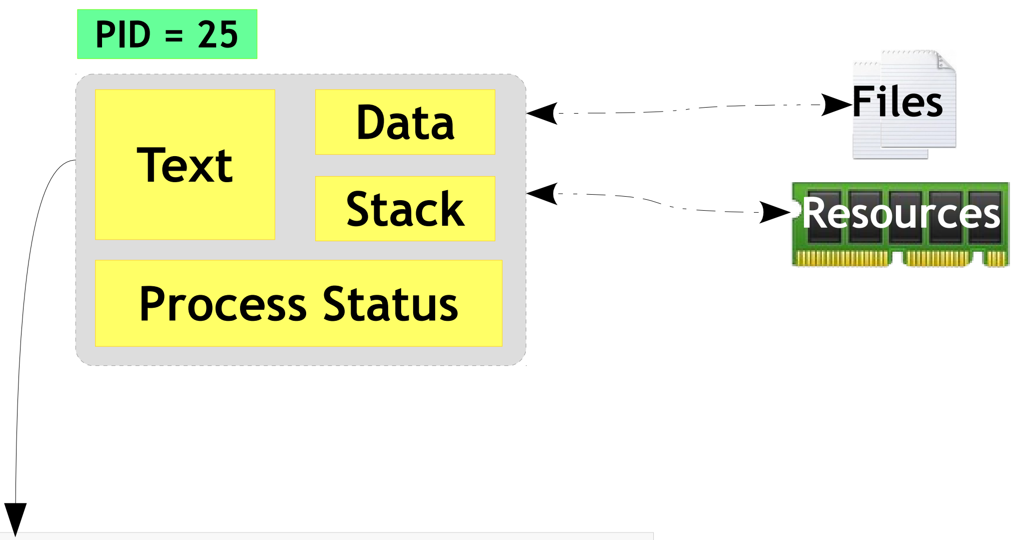
```
int child_pid;
int child_status;

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);
     }
}
```



Process
fork() - The Flow



Process
fork() - The Flow

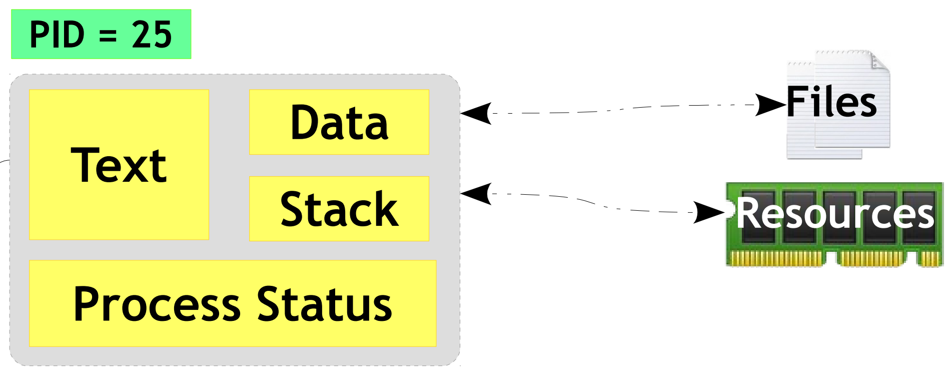



```
ret = fork();
switch (ret)
{
     case -1:
          perror("fork");
          exit(1);
     case 0:
          <code for child>
          exit(0);
     default:
          <code for parent>
          wait(&child_status);
}
```

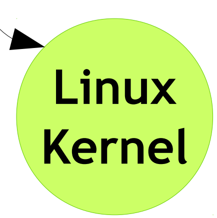
Linux Kernel

Process

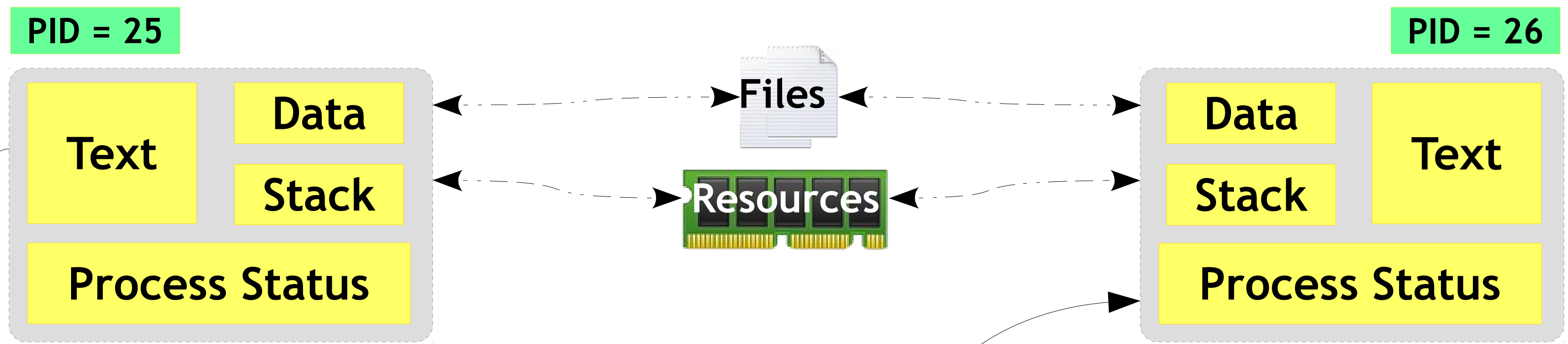
fork() - The Flow




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



Process
fork() - The Flow

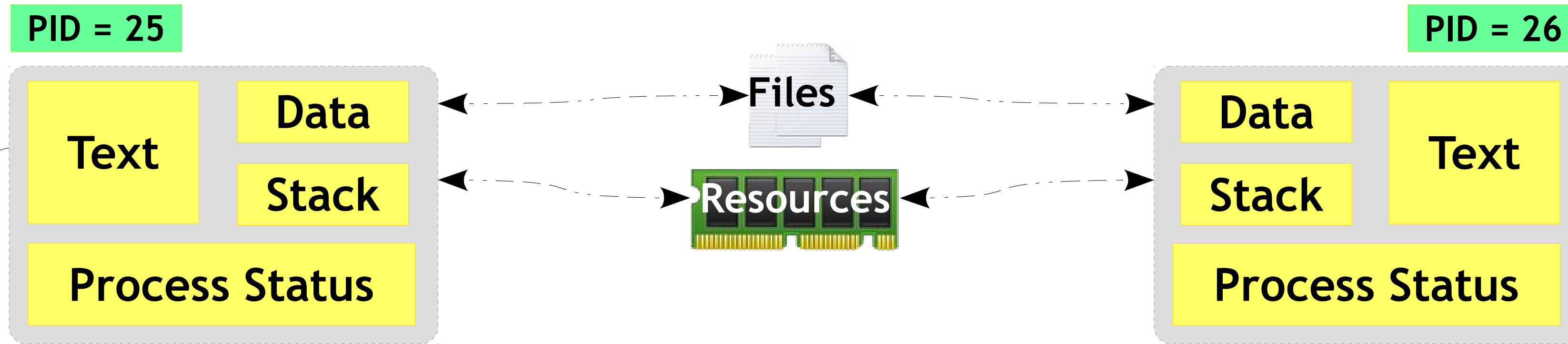



```
ret = fork();          ret = 26
switch (ret)
{
     case -1:
          perror("fork");
          exit(1);
     case 0:
          <code for child>
          exit(0);
     default:
          <code for parent>
          wait(&child_status);
}
```

Linux Kernel

Process

fork() - The Flow

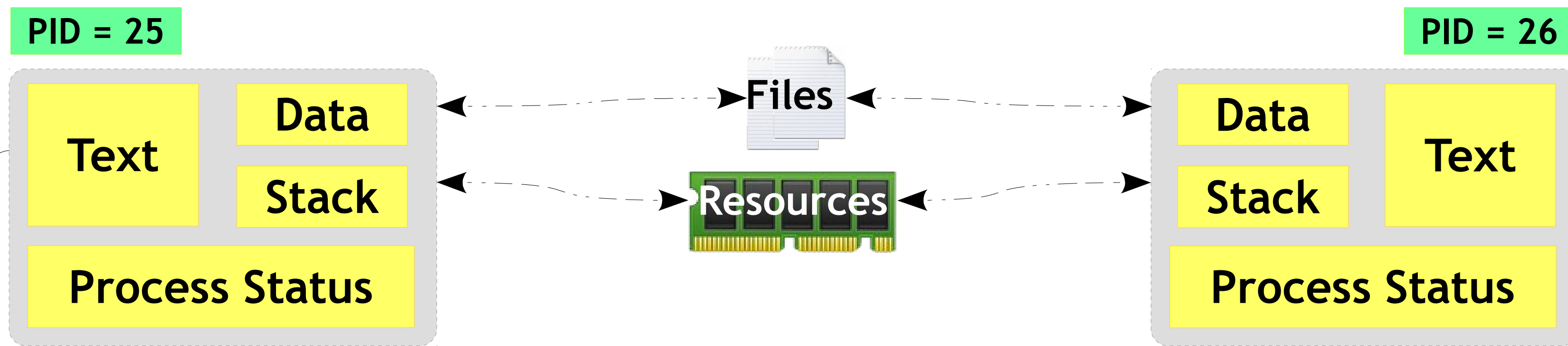



```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

```
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



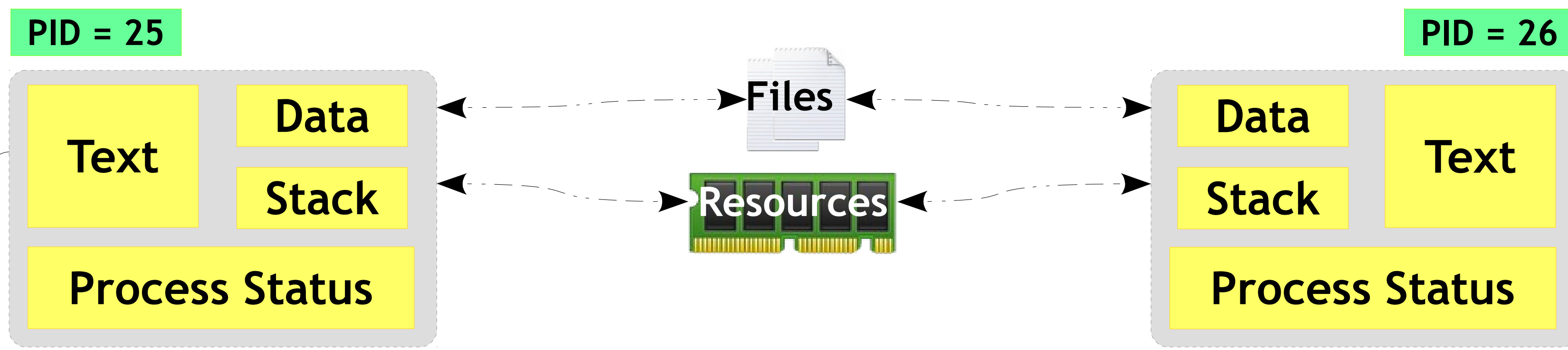

```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

```
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

Process

fork() - The Flow



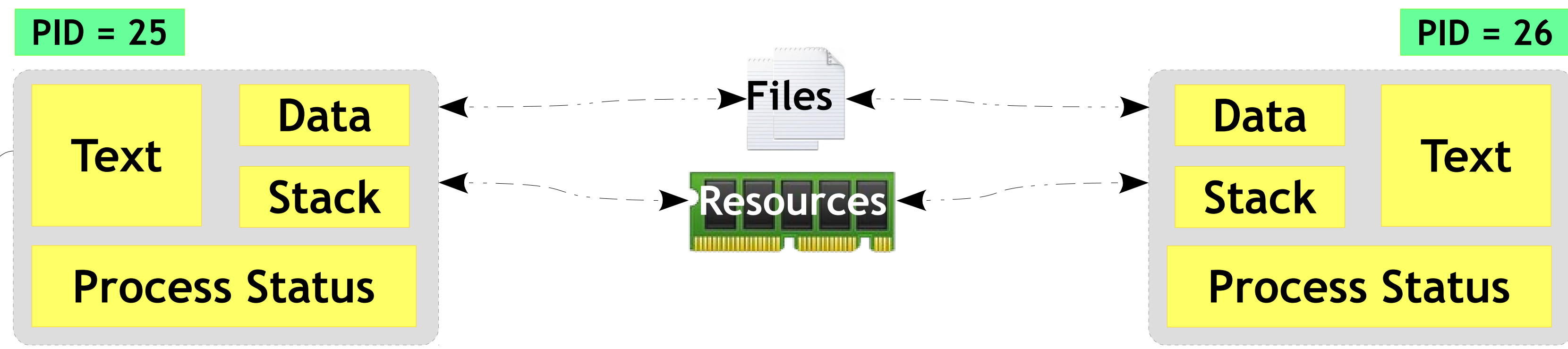

```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux
Kernel

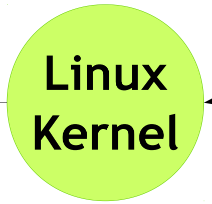
```
ret = fork();          ret = 0
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Process

fork() - The Flow

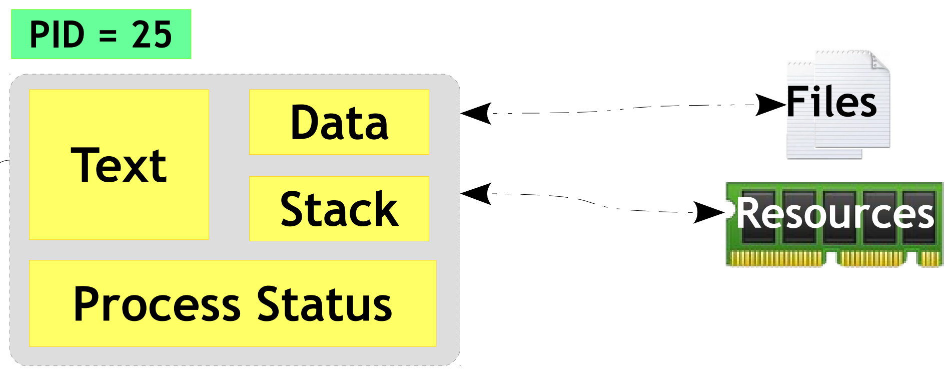



```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

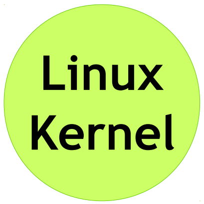



```
ret = fork();          ret = 0
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Process
fork() - The Flow




```
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);
}
```



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

Process

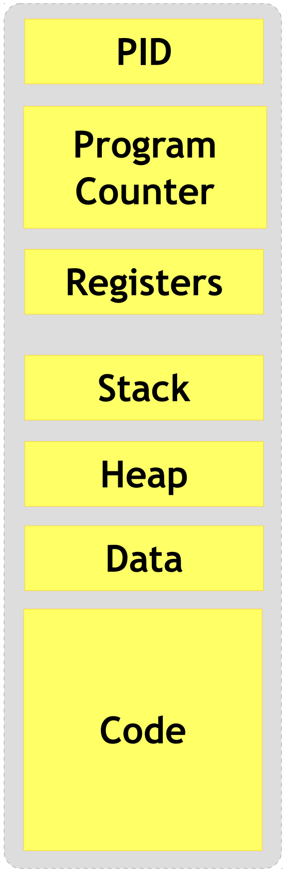
Overlay - exec()

- 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);
}
```

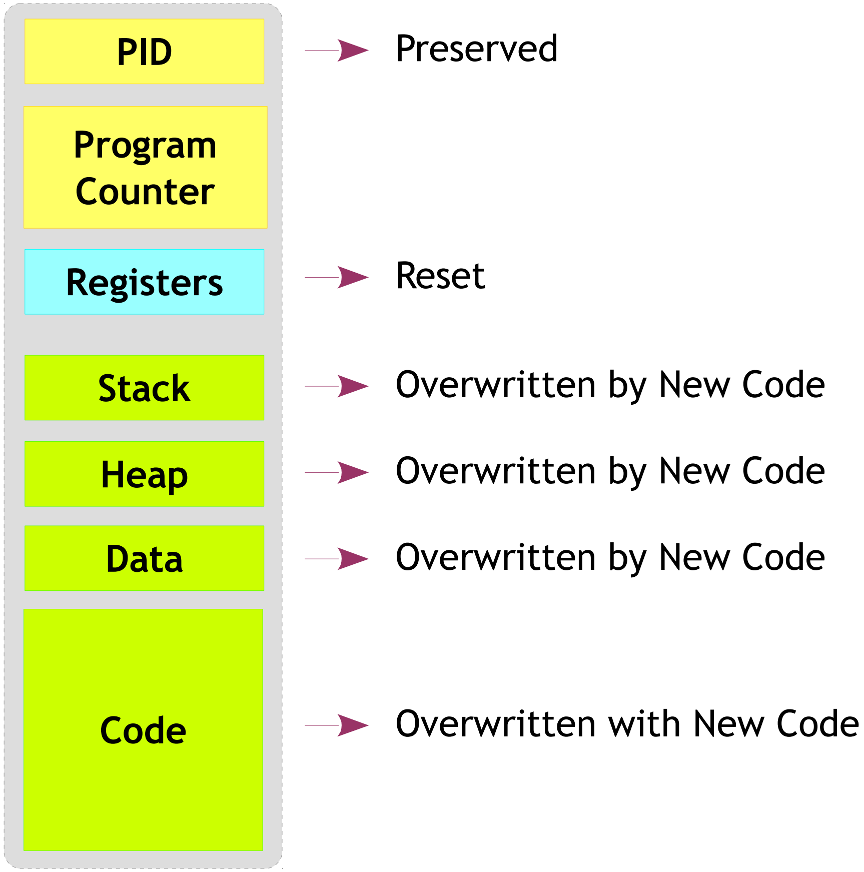


Process

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);
}
```



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

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

Process
Resource Structure

```
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 */
};
```

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.

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
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
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

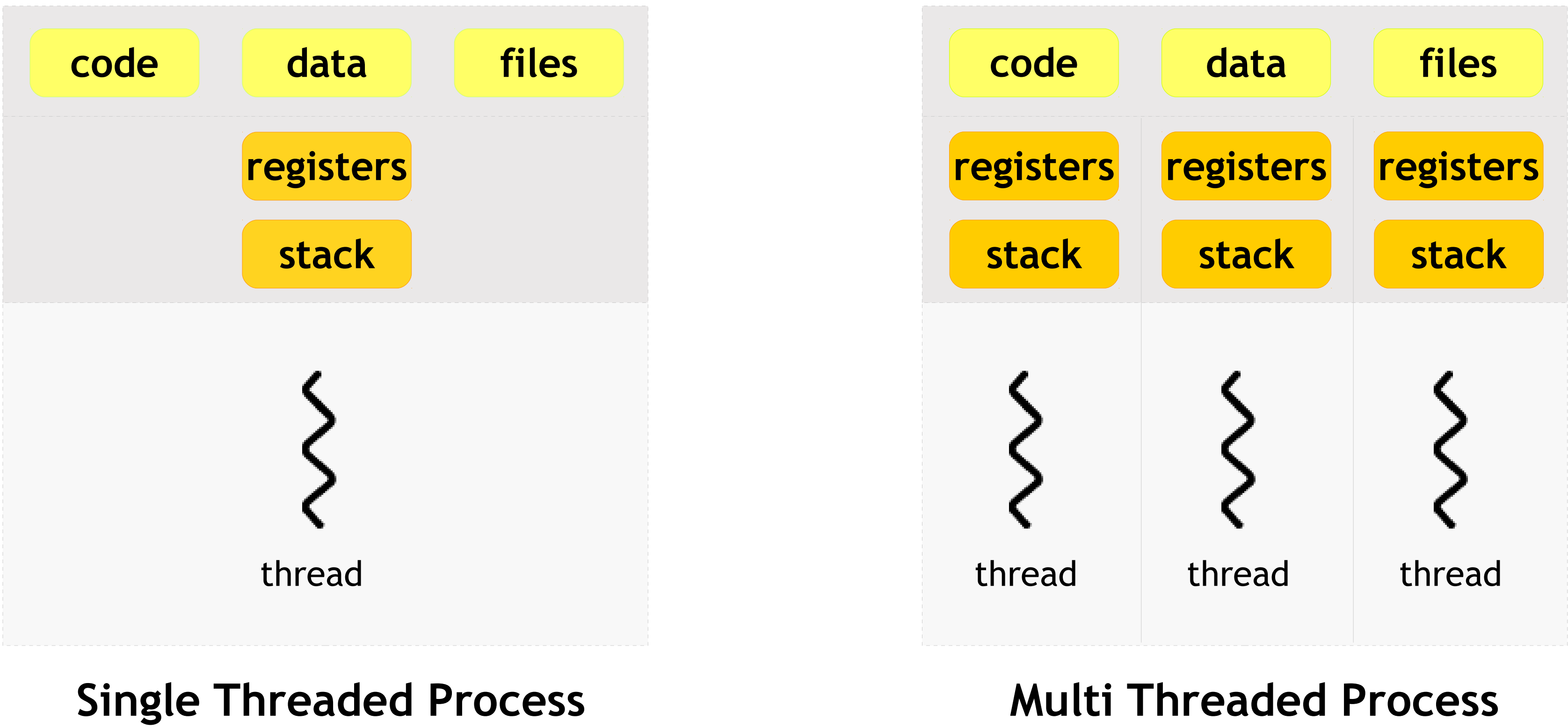
Threads

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



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

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

Threads

Return Values

- 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

Synchronization

why?

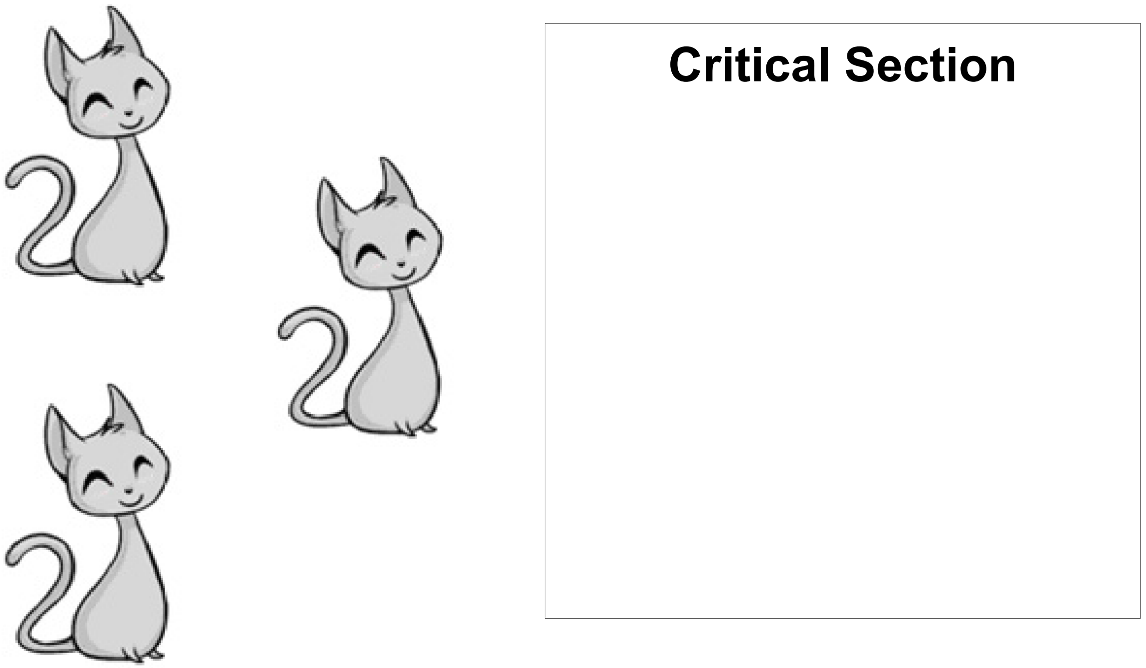
- When multiple tasks are running simultaneously:
  - either on a single processor, or on
  - a set of multiple processors
- They give an appearance that:
  - For each task, it is the only task in the system
  - At a higher level, all these tasks are executing efficiently
  - Tasks sometimes exchange information:
    - They are sometimes blocked for input or output (I/O)
- This asynchronous nature of scheduled tasks gives rise to ***race conditions***

Synchronization
Race Condition

- In Embedded Systems, most of the challenges are due to shared data condition
- Same pathway to access common resources creates issues
- These bugs are called ***race conditions***; the tasks are racing one another to change the same data structure
- Debugging a muti-tasking application is difficult because you cannot always easily reproduce the behavior that caused the problem
- Asynchronous nature of tasks makes race condition simulation and debugging as a challenging task

Synchronization
Critical Section

- A piece of code that only one task can execute at a time.
- If multiple tasks try to enter a critical section, only one can run and the others will sleep.

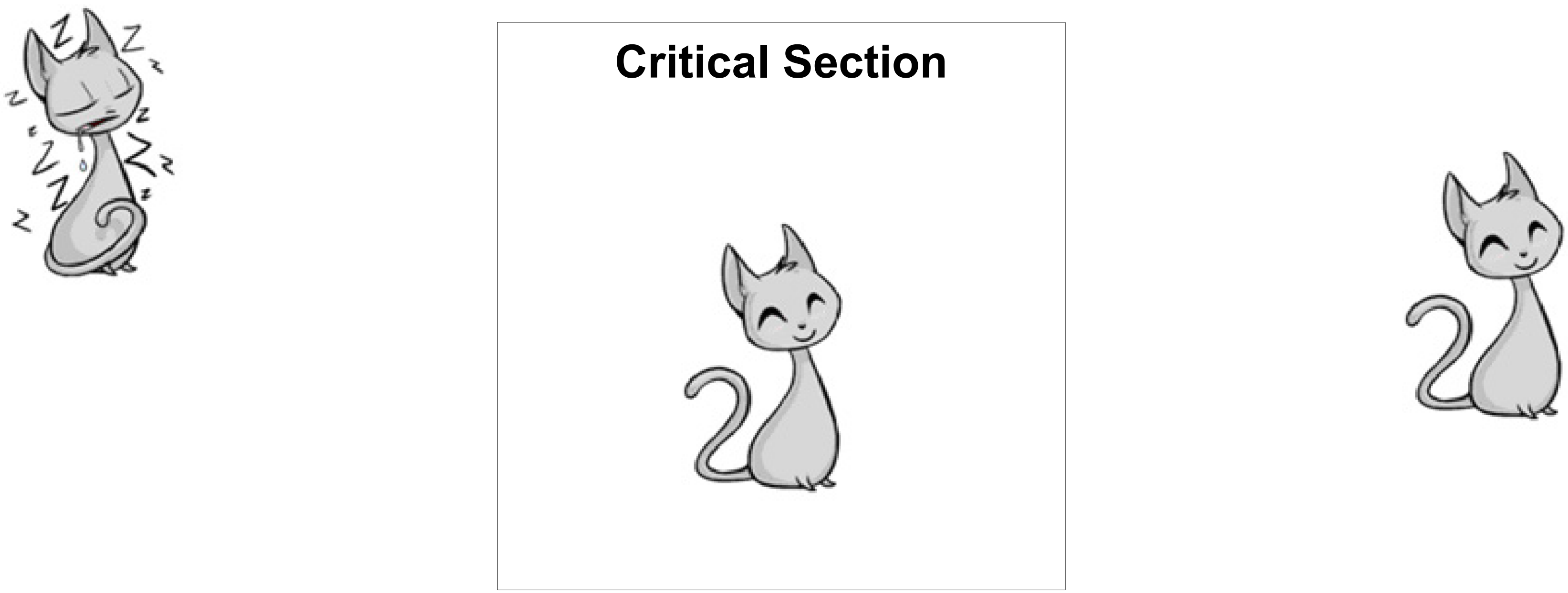


Synchronization
Critical Section

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

Synchronization
Critical Section

- Once the thread in the critical section exits, another thread is woken up and allowed to enter the critical section.
- It is important to keep the code inside a critical section as small as possible



Synchronization
Mutual Exclusion

- A mutex works like a critical section.
- You can think of a mutex as a token that must be grabbed before execution can continue.

Synchronization
Mutual Exclusion

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

Synchronization
Mutual Exclusion

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

Synchronization
Practical Implementation

- The concept of synchronization is common across various OS
- Implementation methods to solve the race condition varies
- Typically it uses 'lock' and 'unlock' mechanisms in an atomic fashion during implementation
- The mutual exclusion when it is implemented with two processes is known as ‘Mutex’
- When implemented with multiple processes known as 'semaphores‘
- Semaphores uses counter mechanism
- Mutexes are also known as binary semaphores

Synchronization
Treads

- In multi threaded systems, they need to be brought in sync
- This is achieved by usage of Mutex and Semaphores
- They are provided as a part of pthread library
- The same issue of synchronization exists in multi-processing environment also, which is solved by process level Mutex and Semaphores
- The ‘lock’ and ‘unlock’ concept remain the same

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)

Synchronization
Treads - Semaphores

- A semaphore is a **counter** that can be used to synchronize multiple threads
- As with a mutex, GNU/Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition
- Each semaphore has a counter value, which is a non-negative integer
- The ‘lock’ and ‘unlock’ mechanism is implemented via ‘wait’ and ‘post’ functionality in semaphore
- Semaphores in conjunction with mutex are used to solve synchronization problem across multiple processes

Synchronization

Treads – 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)

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 (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

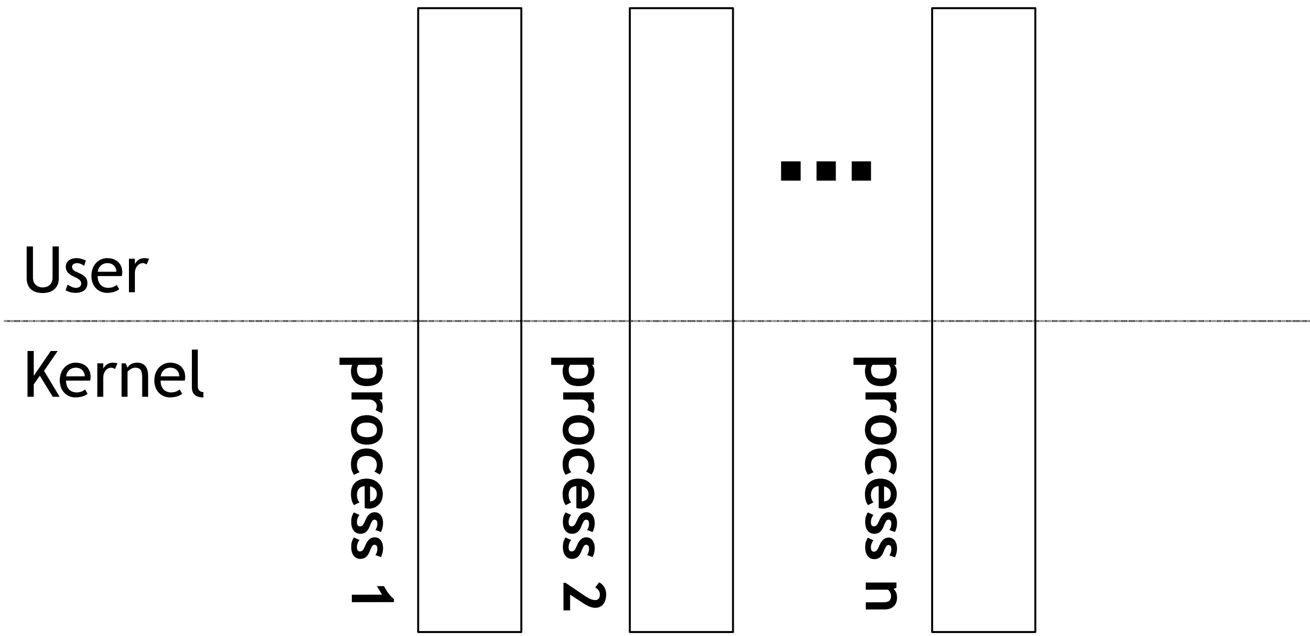
Inter Process Communications
Introduction

- IPC can be categorized broadly into two areas:
  - Communication
  - Synchronization
- Even in case of Synchronization also two processes are talking ☺
- Here are the various IPC mechanisms:
  - Pipes
  - FIFO (or named pipes)
  - Message Queues
  - Shared memory
  - Semaphores (Process level)
  - Sockets

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

Inter Process Communications
User vs Kernel Space

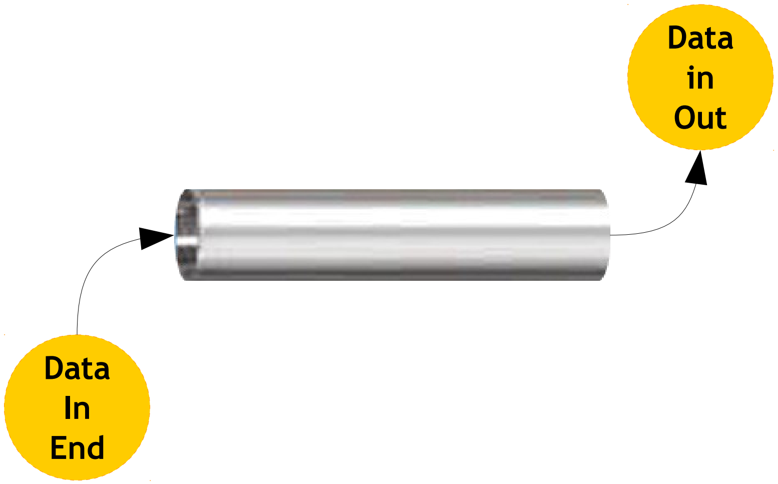
- Protection domains - (virtual address space)



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



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.

ƩMERTXE

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 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 - 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.

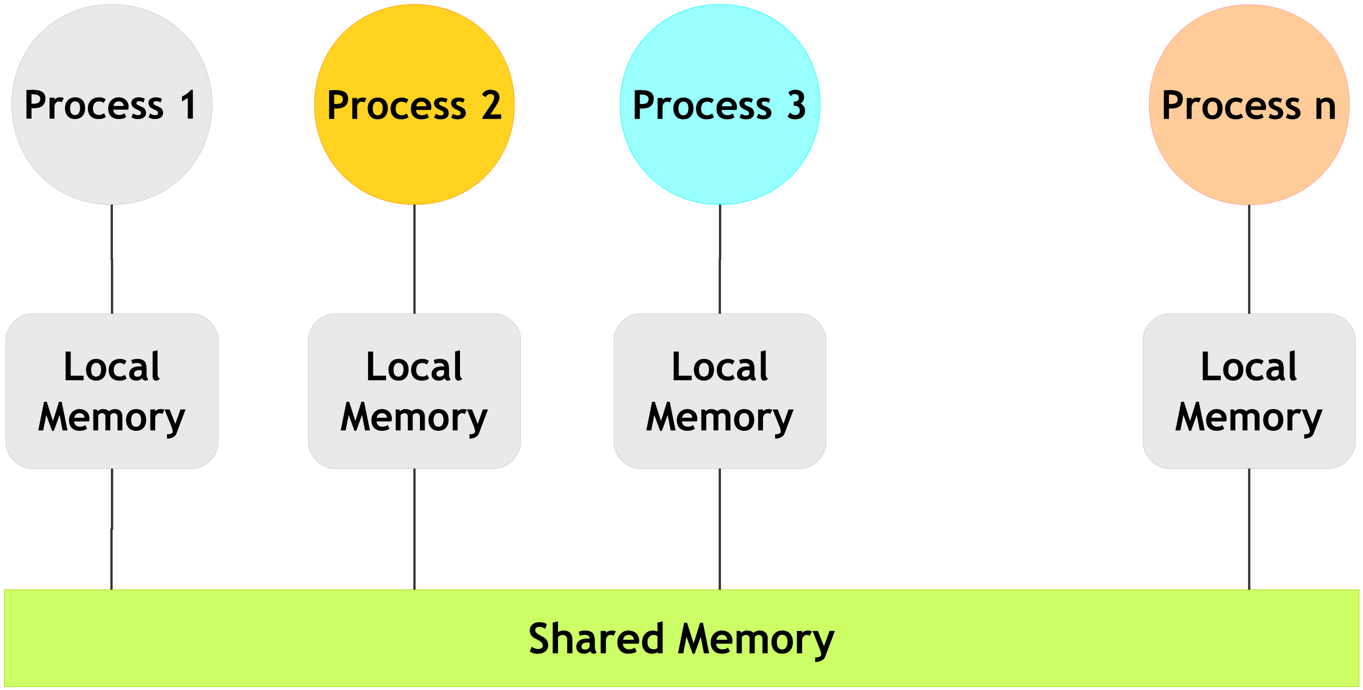
Inter Process Communications

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

- 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

- 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

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
```

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

Networking Fundamentals

TCP / IP Model

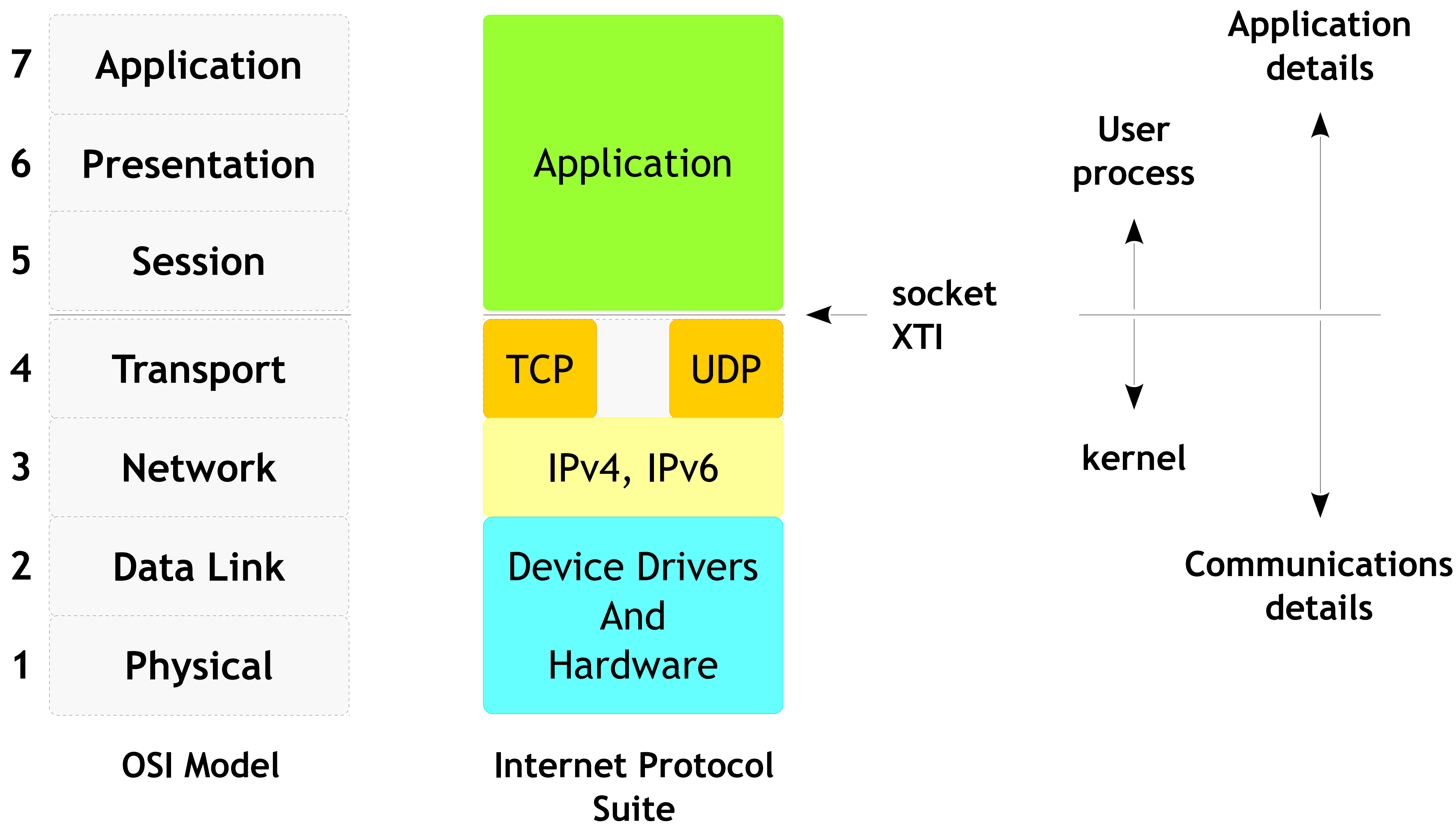
OSI Model		TCP / IP Protocol Layers	Internet Protocol Suite
7	Application	Application	Telnet, DNS
6	Presentation		FTP, RIP
5	Session		SMTP, SNMP
4	Transport	Transport	TCP, UDP
3	Network	Inter-network	IP, ARP, ICMP, IGMP
2	Data Link	Link	Ethernet, Token Ring
1	Physical		ATM, Wireless

OSI Model

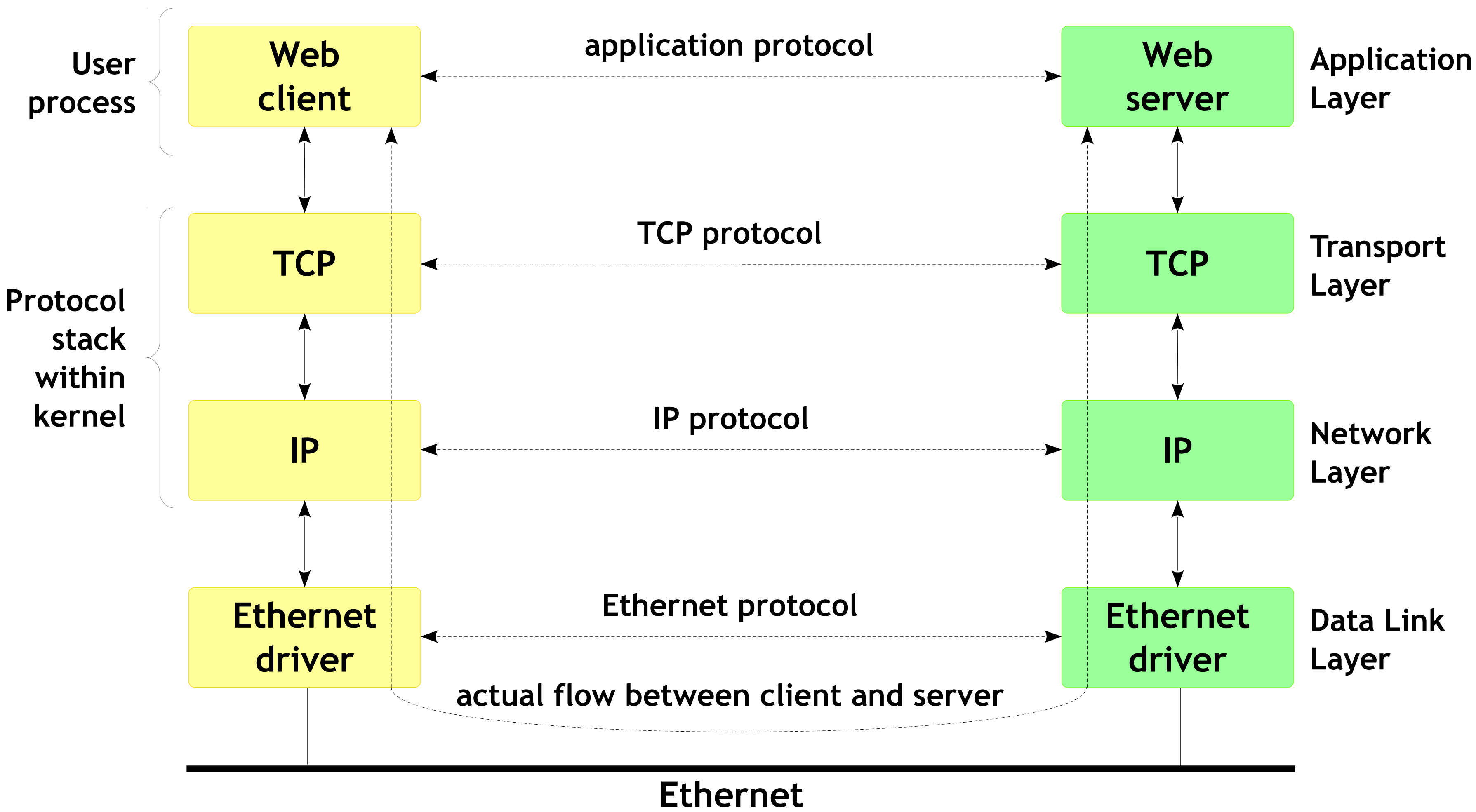
TCP / IP Protocol Layers

Internet Protocol Suite

Networking Fundamentals
TCP / IP Model – Implementation in Linux



Networking Fundamentals
Protocols



Networking Fundamentals
Addressing

- IP layer: IP address
  - Dotted decimal notation (“192.168.1.10”)
  - 32 bit integer is used for actual storage
- TCP/UDP layer: Port numbers
  - Well known ports [ex: HTTP (80), Telnet (23)]
  - User defined protocols and ports
- Source IP, Port and Destination IP, Port are essential for creating a communication channel
- IP address must be unique in a network
- Port number helps in multiplexing and de-multiplexing the messages

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

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)

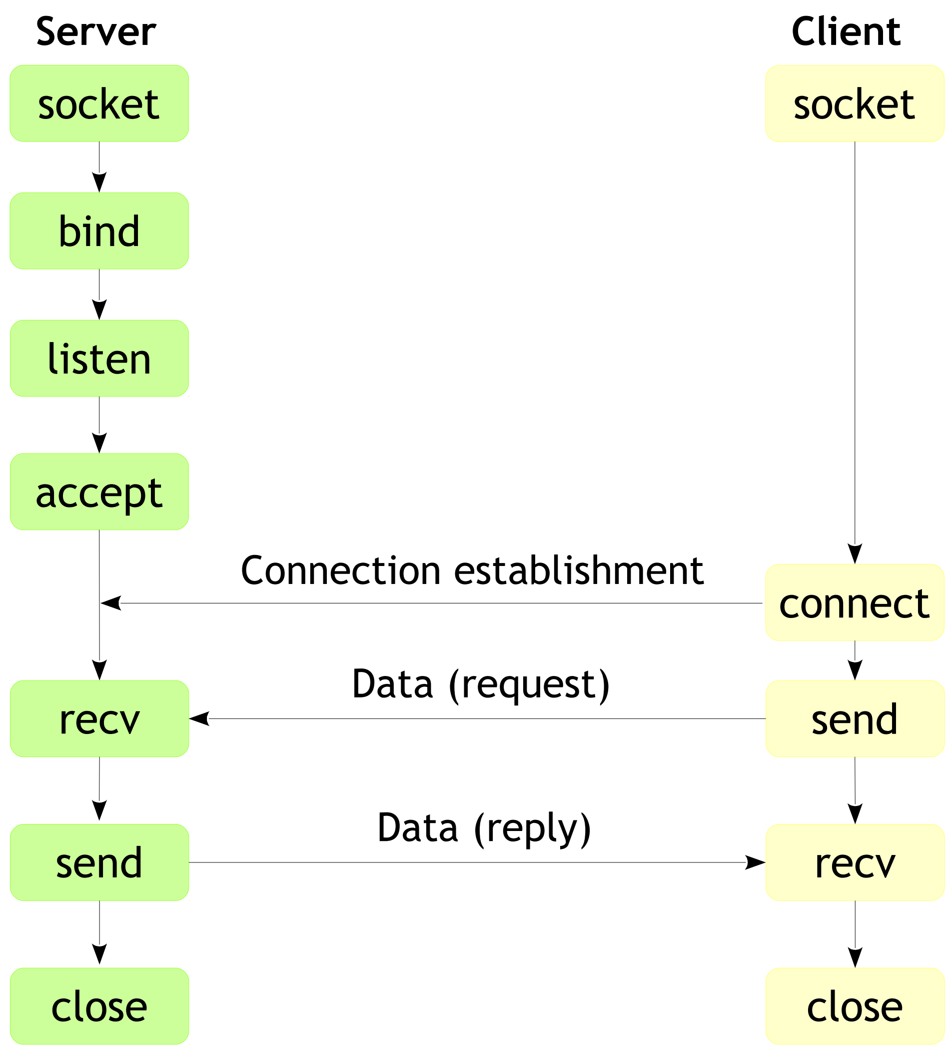
Sockets
Calls – send

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

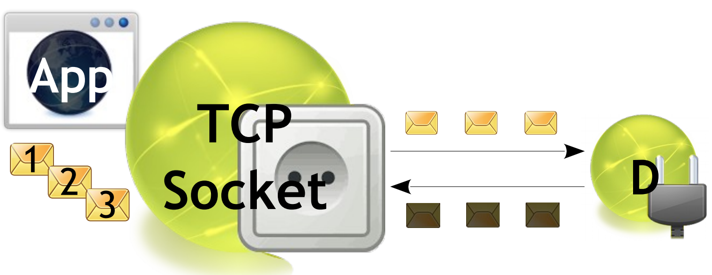
Sockets
TCP - Summary

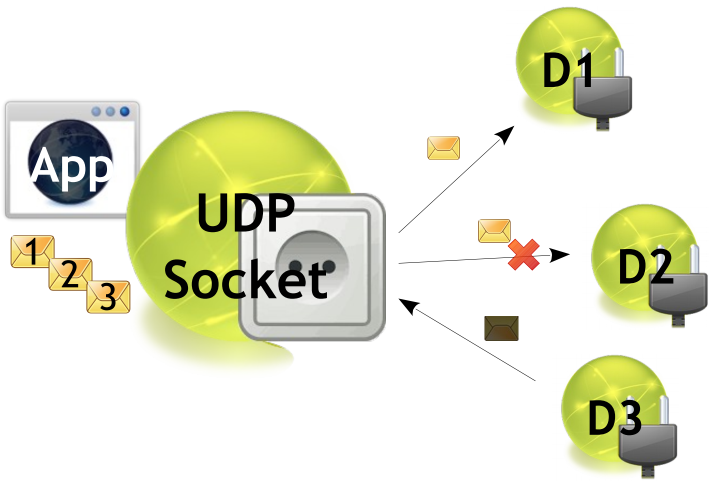


NOTE: Bind() – call is optional from client side

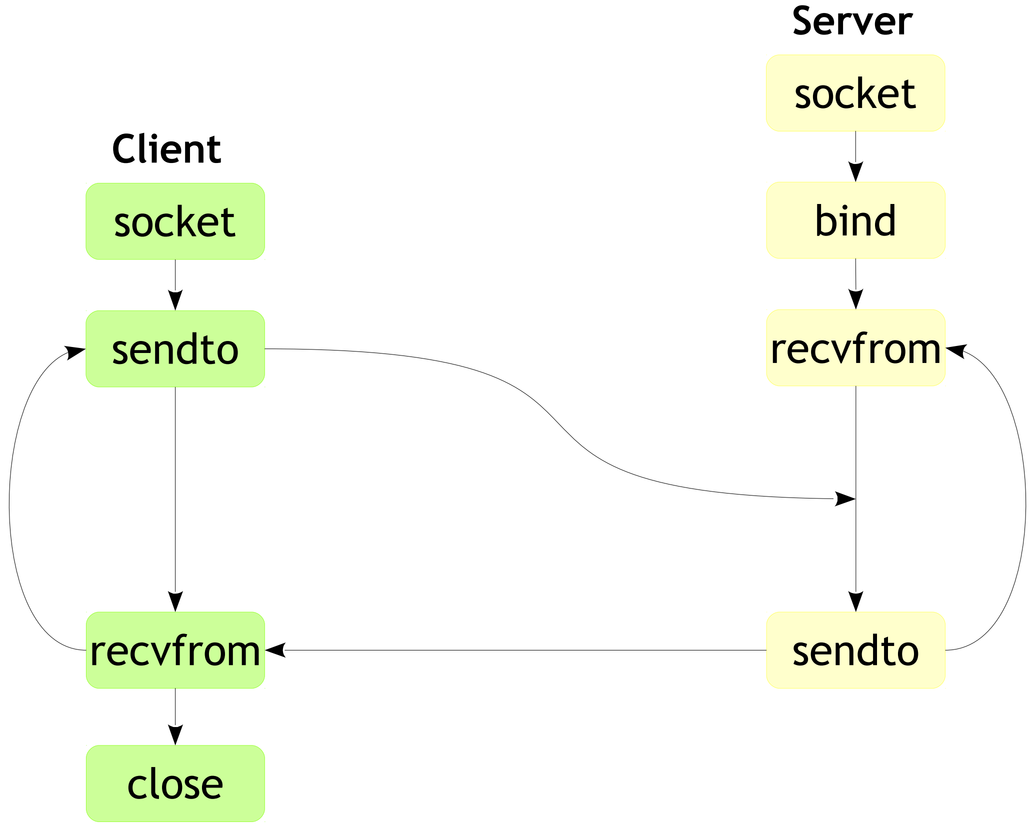
Sockets
TCP vs UDP

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



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

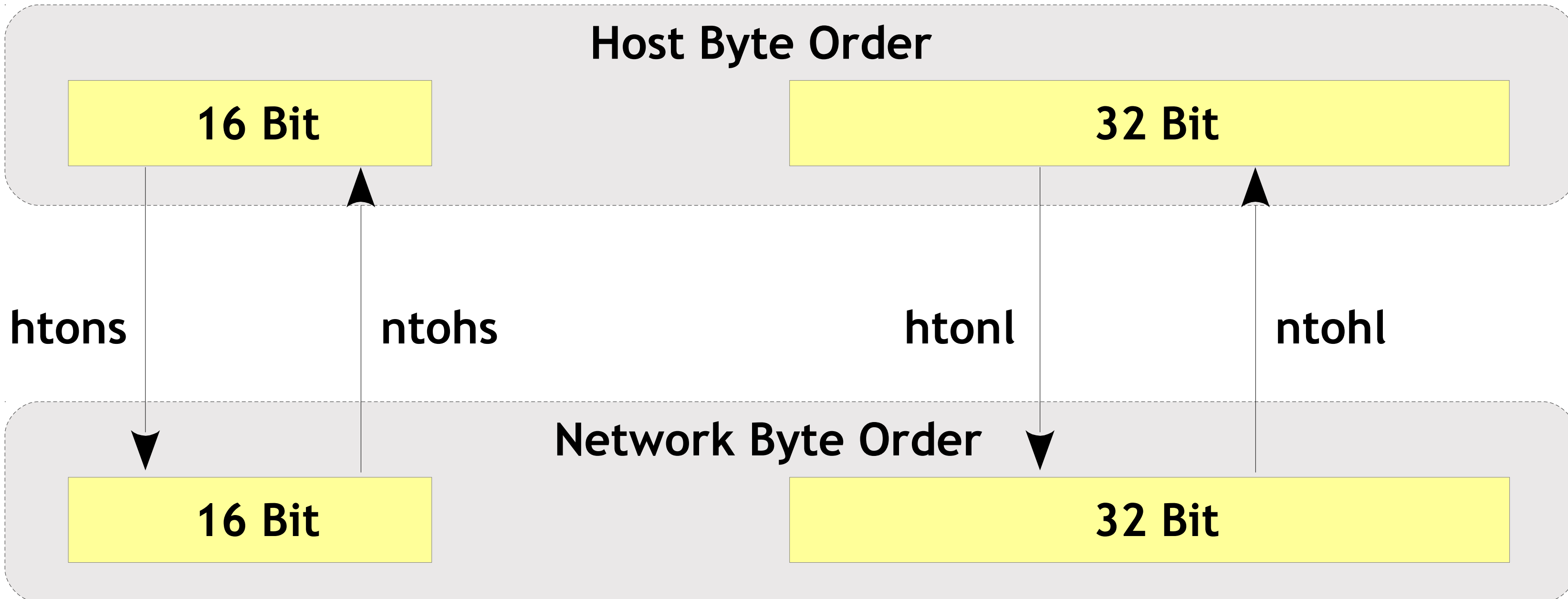
Sockets
UDP – Functions calls

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)

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
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);
```

Process Management - Concepts

Process Management - Concepts
Scheduling

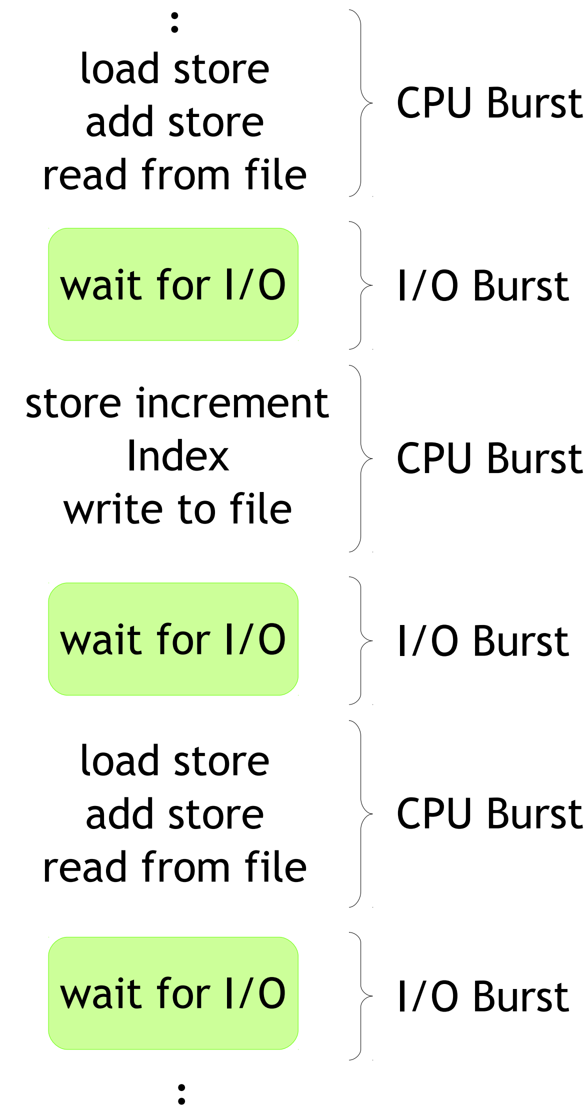
- It is a mechanism used to achieve the desired goal of multitasking
- This is achieved by SCHEDULER which is the heart and soul of operating System
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
- Short-term scheduler (or CPU scheduler) – selects 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”

Process Management - Concepts

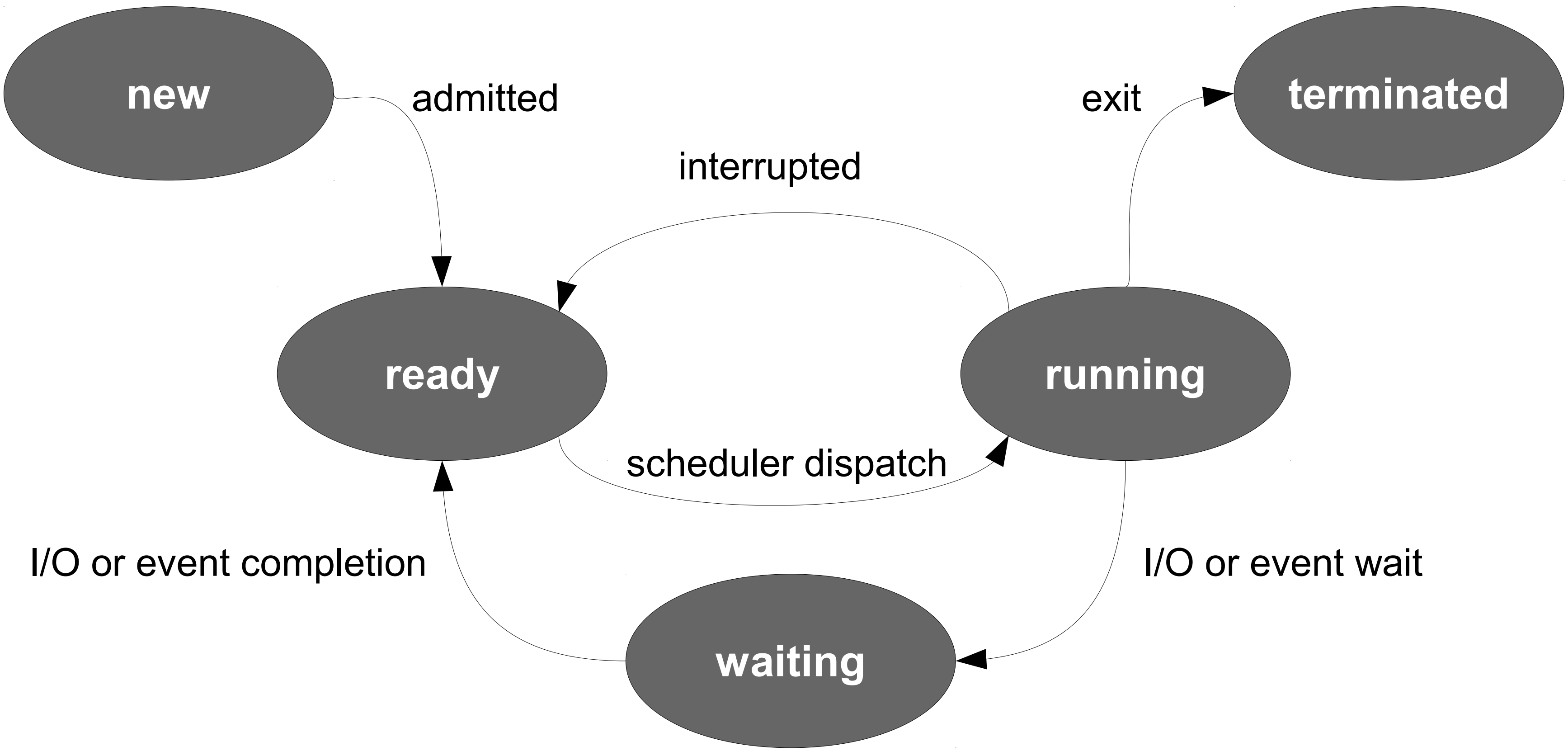
CPU Scheduling

- 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

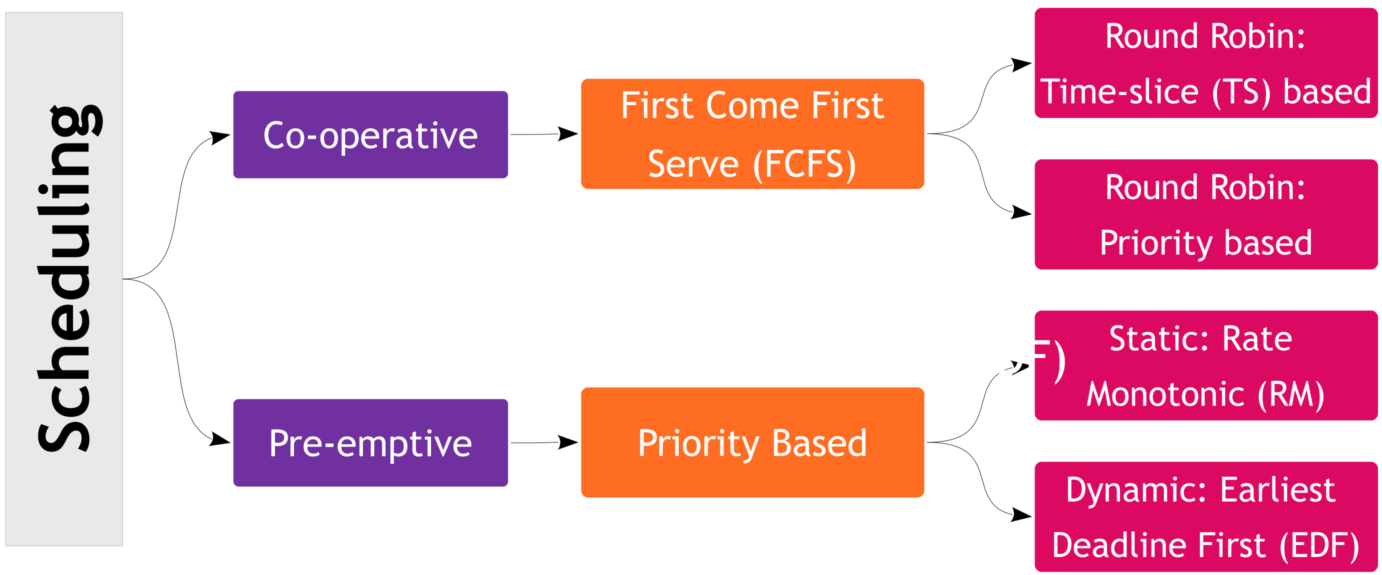
State	Description
New	The process is being created
Running	Instructions are being executed
Waiting	The process is waiting for some event to occur
Ready	The process is waiting to be assigned to processor
Terminated	The process has finished execution

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 – 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: 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 – 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
- 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

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

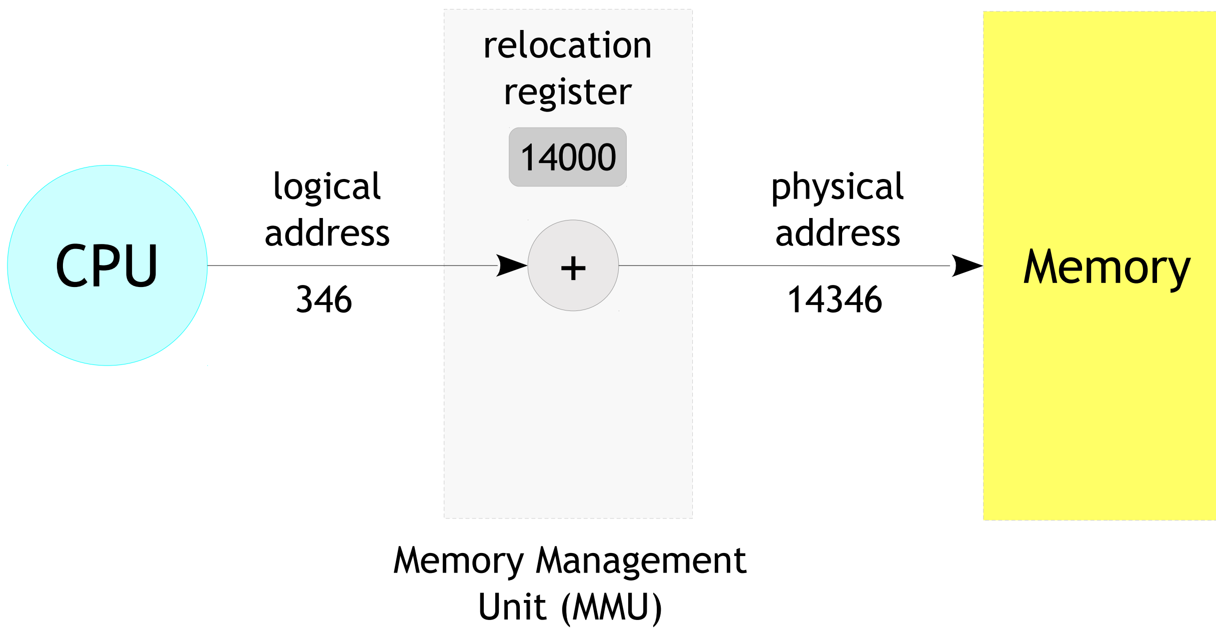
Requirements - Relocation

- 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

Requirements - Relocation

- In paging OS picks data from secondary memory called pages
- Paging allows the physical address space of a process to be non-contiguous



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

Memory Management - Concepts

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

Memory Management - Concepts

Requirements – Logical Organization

- Program is divided into modules, each having a different attribute
- 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

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