Programming in Linux

System programming using Kernel interfaces

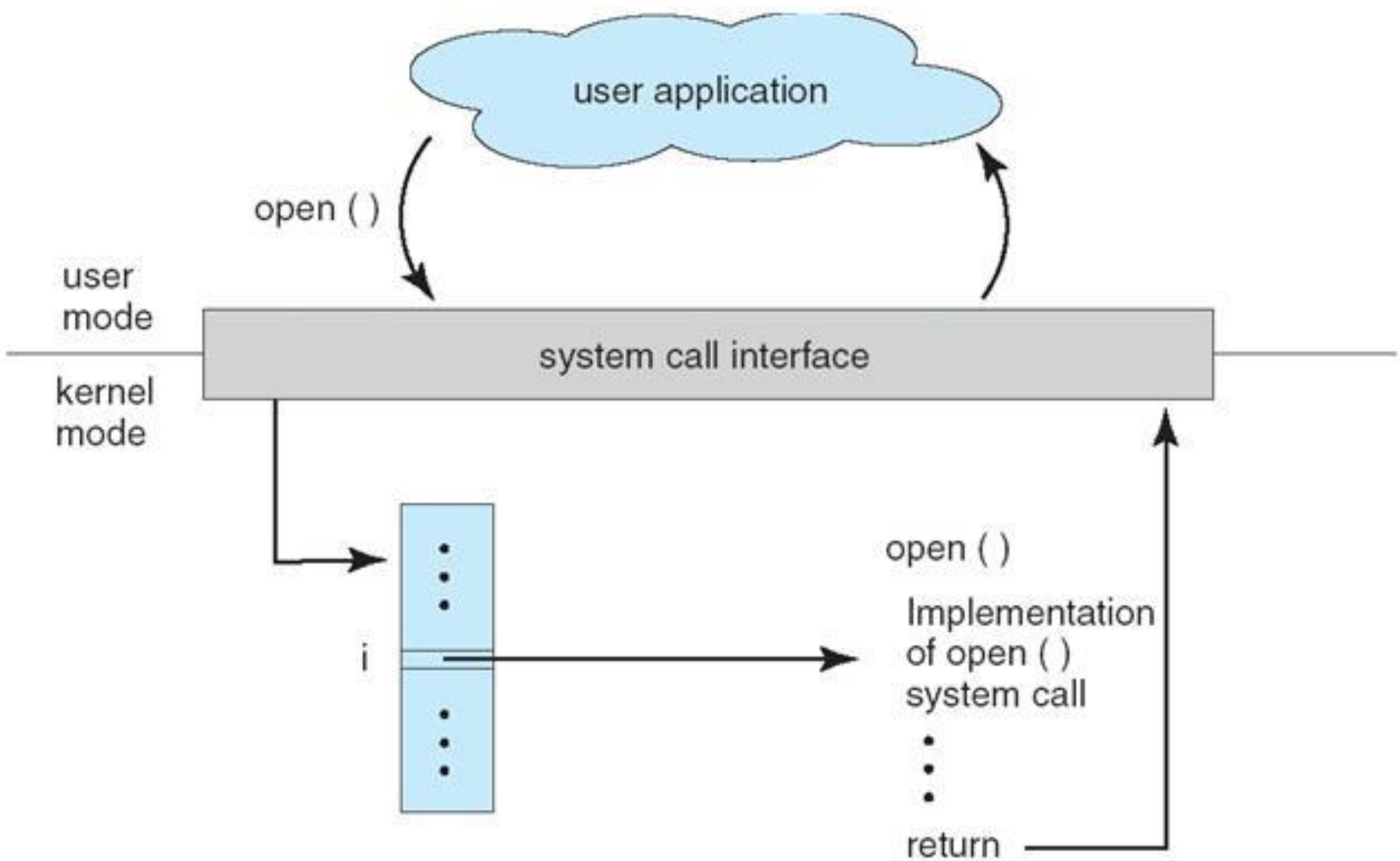
Team Emertxe

System Calls

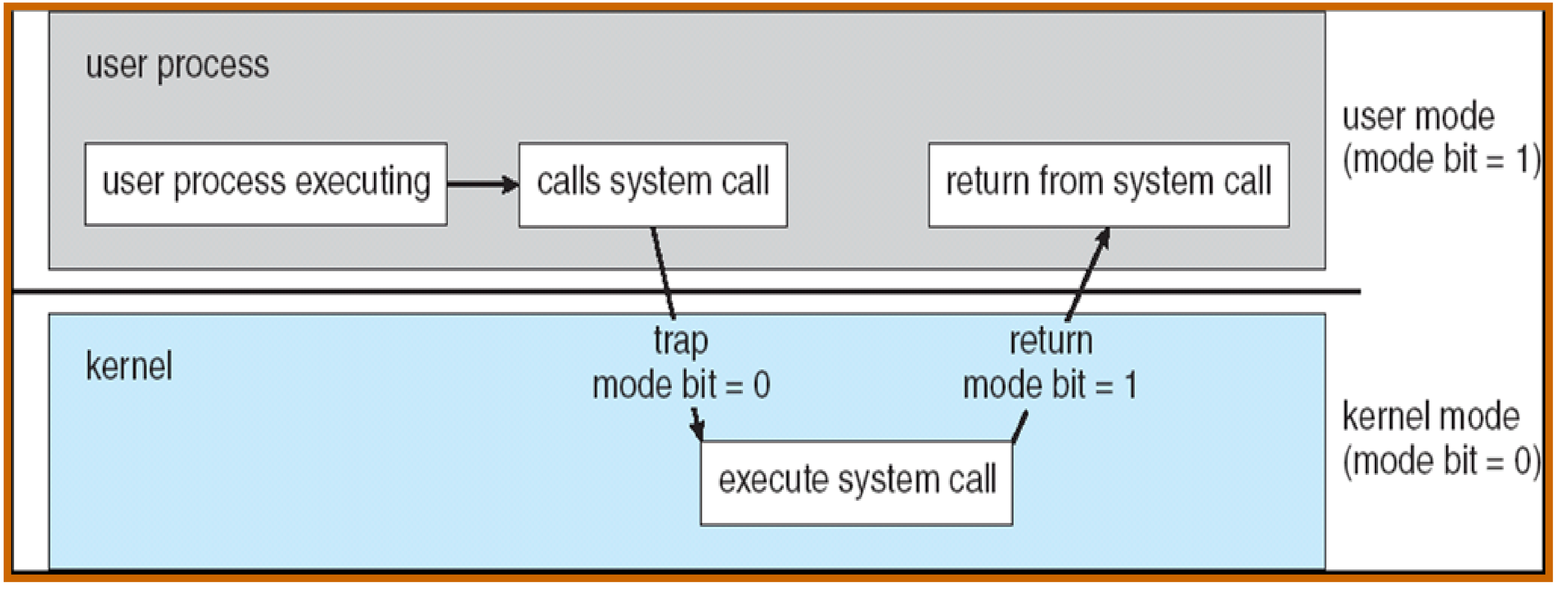
System calls

- ✓ 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

System call


user application
open ( )
user
mode
system call interface
kernel
mode
i
open ( )
Implementation
of open ( )
system call
return

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’

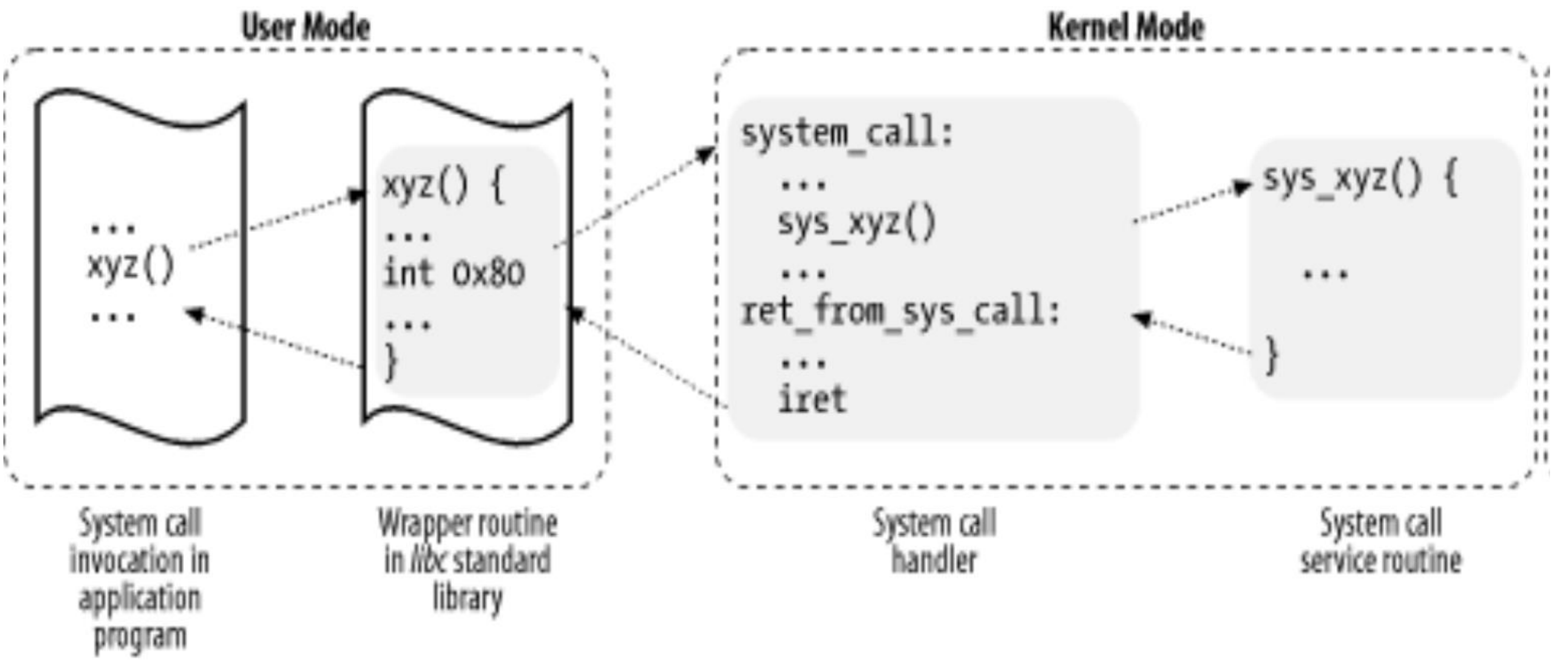
System Call & Library Functions

- ✓ 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

ΣMERTXE

System call Implementation



System call 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

Systemcall - 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_usec field - additional number of usecs

Other System calls

- Exit
- Wait
- Read
- Write
- Open
- Close
- Waitpid
- Getpid
- Sync
- Nice
- Kill etc..

Process

What's a 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 v/s Program

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

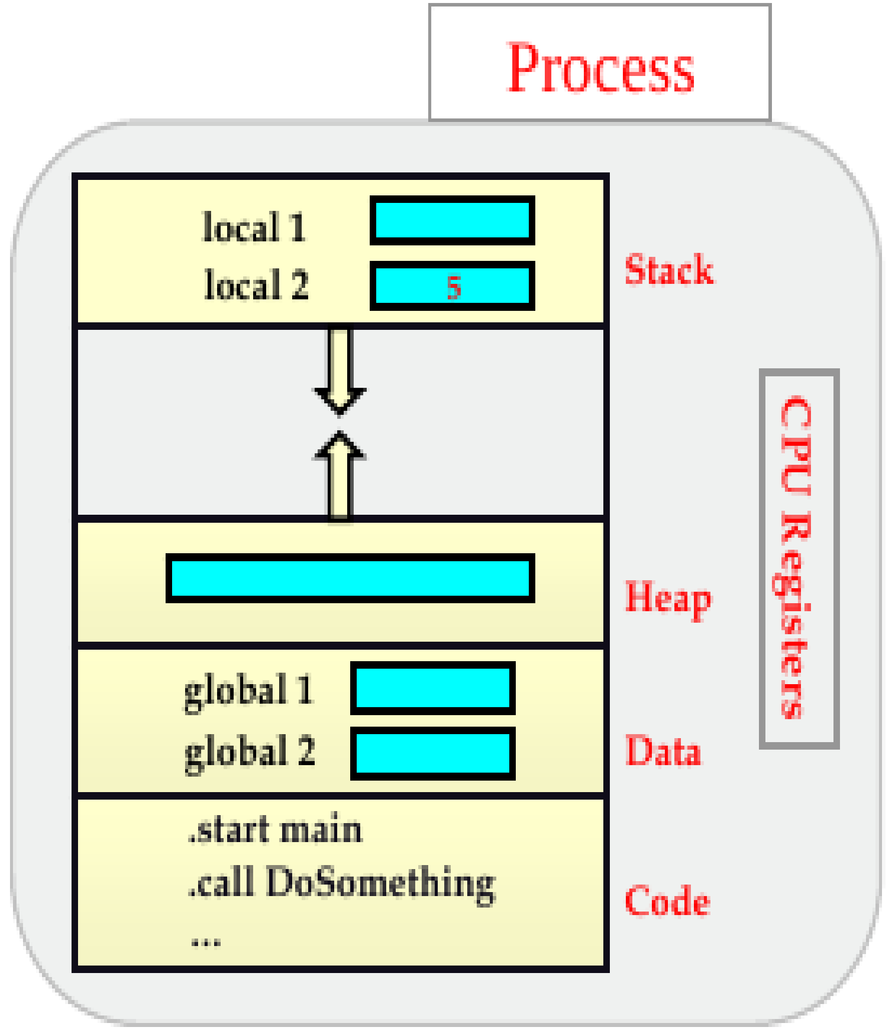
✓ Process is a active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

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

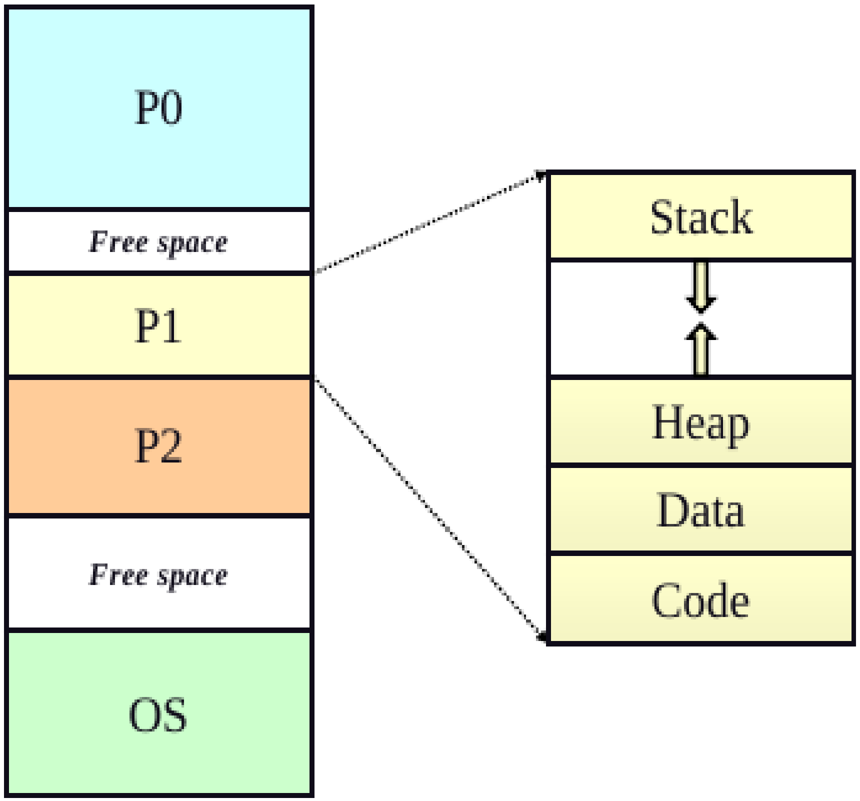
Cont...

Program

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

void DoSomething()  {
        int local2 = 5;
        local2 = local2 + 1;
        ...
}
int main() {
        char *  local1 = malloc(100);
        DoSomething();
         ...
}
```



What if More Process in Memory?



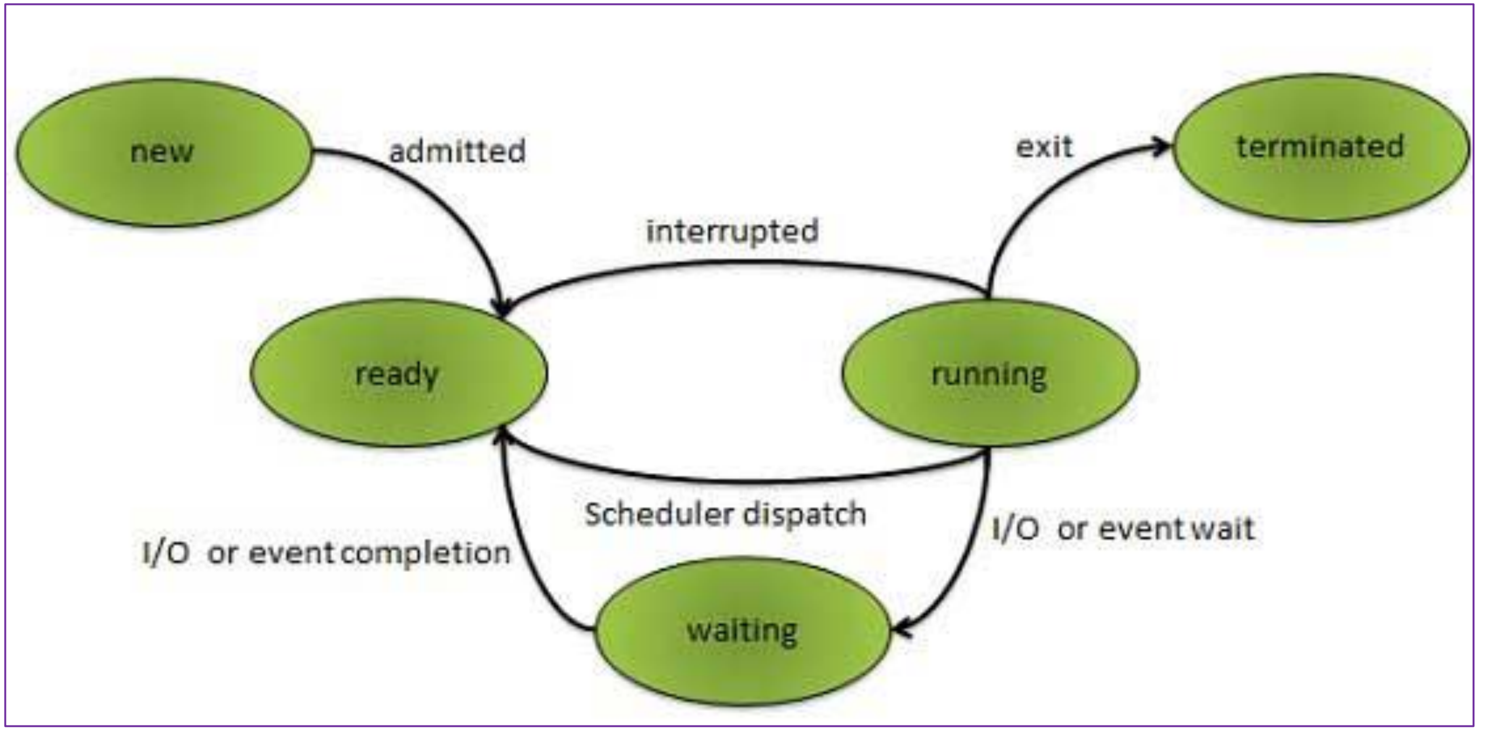
Each process has its own Code, Data, Heap and stack

Process States

✓A task goes through multiple states ever since it is created by the OS

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

Process - State Transition Diagram



Process Structure / Descriptor

✓ To manage tasks:

- Kernel must have a clear idea 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.

✓ This is the role of the task descriptor

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

Cont...

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

Pointer	Task State
Task Number	
Program Counter	
Registers	
Memory Limits	
List of Open Files	
. . .	

EMERTXE

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 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

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”):

```
Jayakumar>ps -aef | more
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 19:37 ?        00:00:01 /sbin/init showopts
root         2     0  0 19:37 ?        00:00:00 [kthreadd]
root               2  0 19:37 ?        00:00:00 [ksoftirqd/0]
root               2  0 19:37 ?        00:00:00 [kworker/0:0]
root         6     2  0 19:37 ?        00:00:00 [migration/0]
root         7     2  0 19:37 ?        00:00:00 [watchdog/0]
root         8     2  0 19:37 ?        00:00:00 [cpuset]
root         9     2  0 19:37 ?        00:00:00 [khelper]
root        10     2  0 19:37 ?        00:00:00 [kdevtmpfs]
root        11     2  0 19:37 ?        00:00:00 [netns]
root        12     2  0 19:37 ?        00:00:00 [sync_supers]
root        13     2  0 19:37 ?        00:00:00 [bdi-default]
root        14     2  0 19:37 ?        00:00:00 [kintegrityd]
root        15     2  0 19:37 ?        00:00:00 [kblockd]
root        16     2  0 19:37 ?        00:00:00 [md]
```

Process ID

Parent Process ID

ΣMERTXE

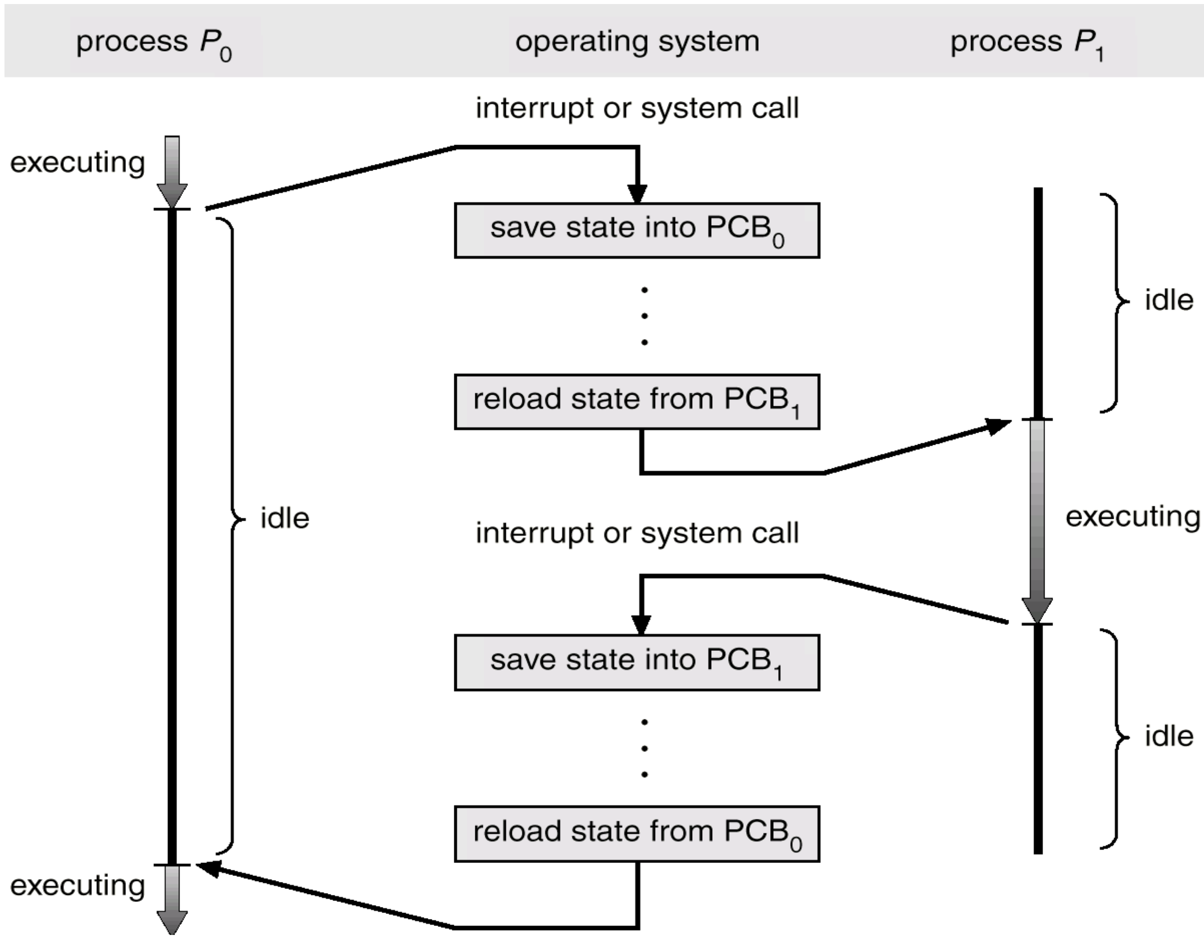
Context switching

✓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



Creating Processes

✓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

Using 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

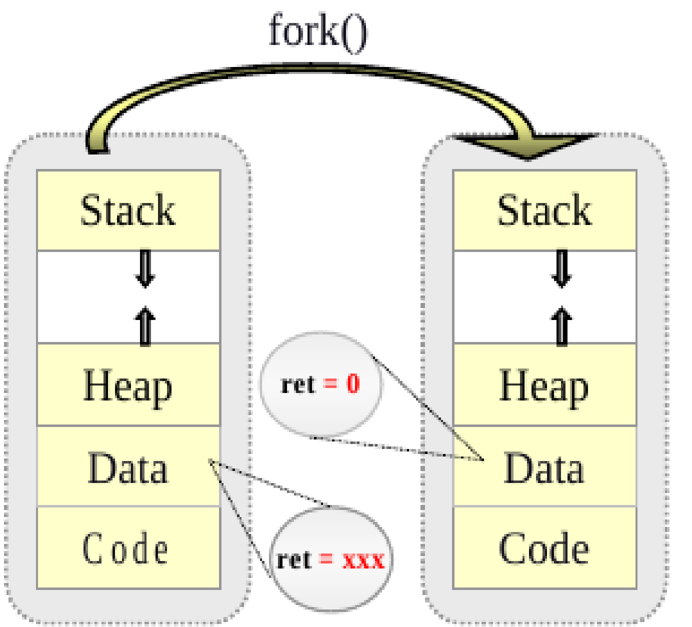
Using fork()

- ✓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

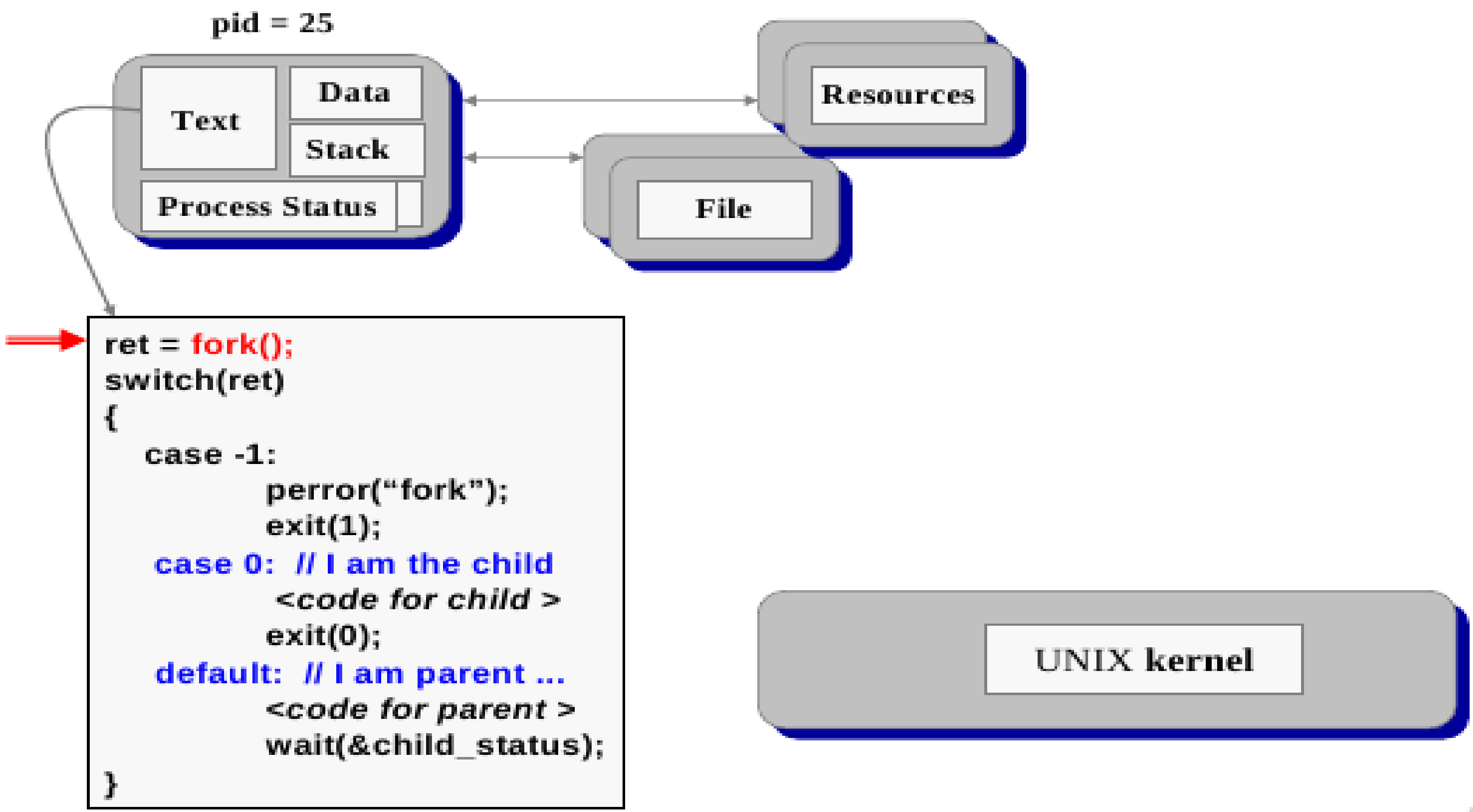
Cont....

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

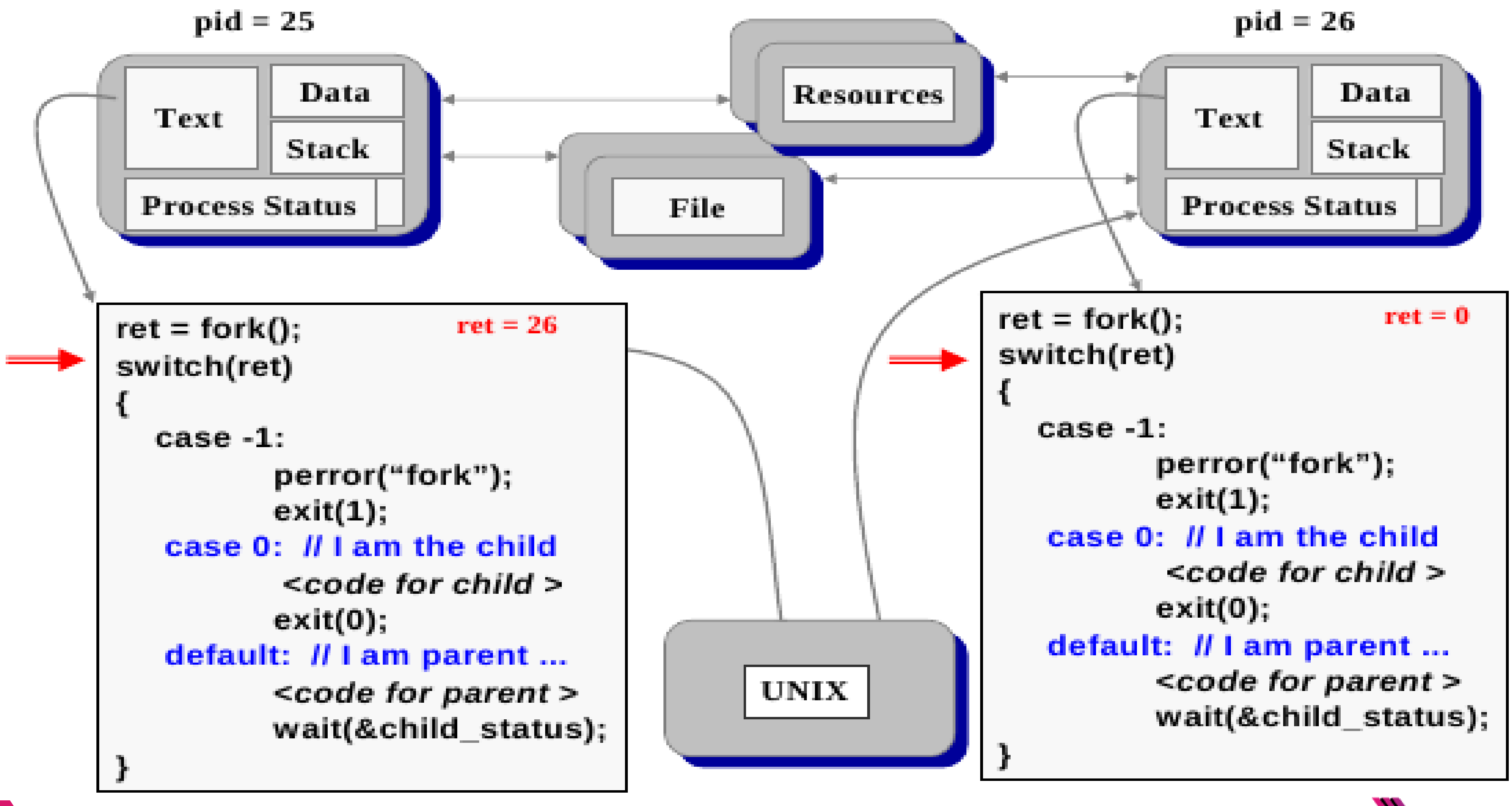


Fork



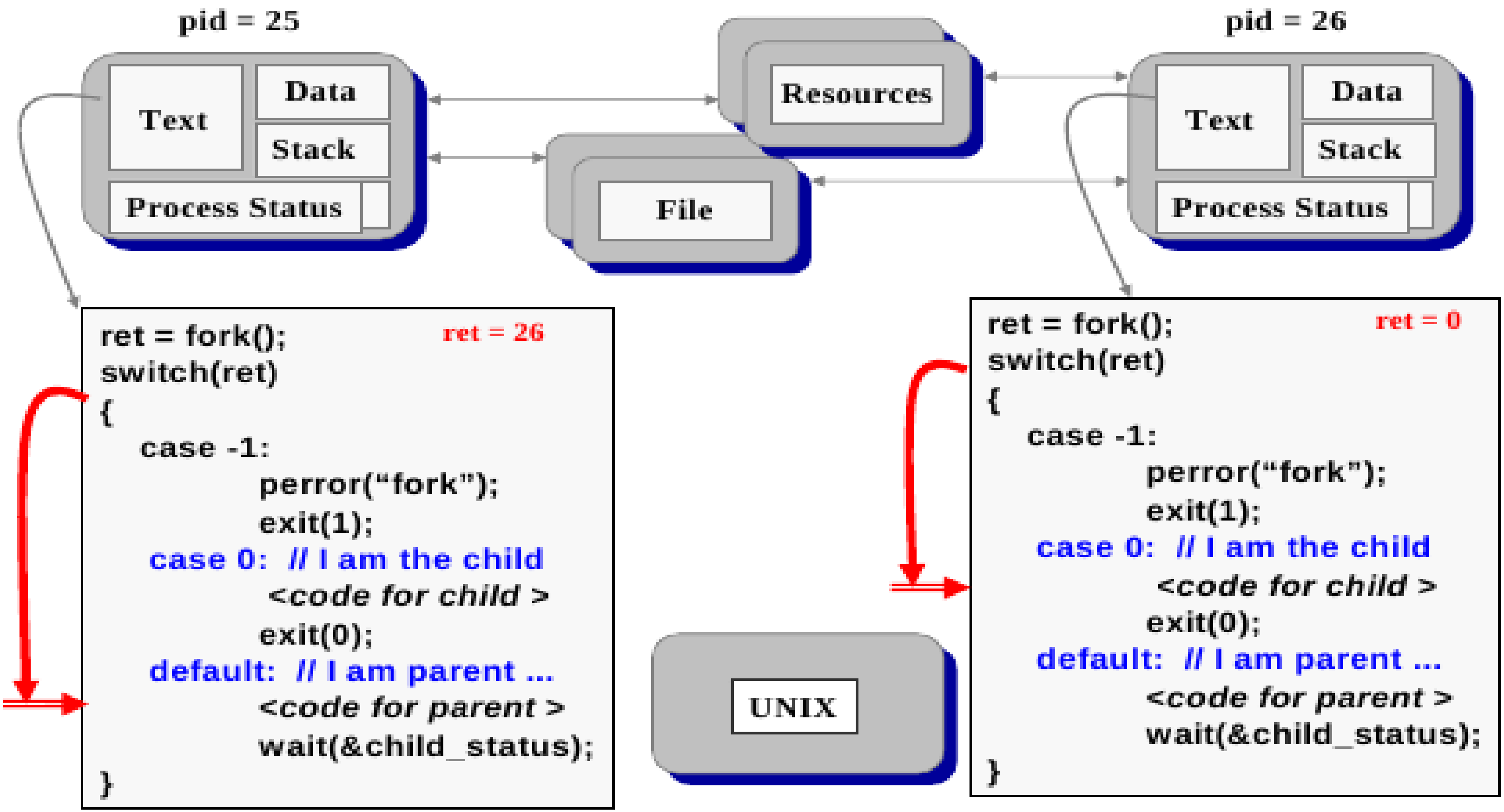
pid = 25
Text
Data
Stack
Process Status
Resources
File
ret = fork();
switch(ret)
{
case -1:
perror(“fork”);
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(&child_status);
}
UNIX kernel

Fork

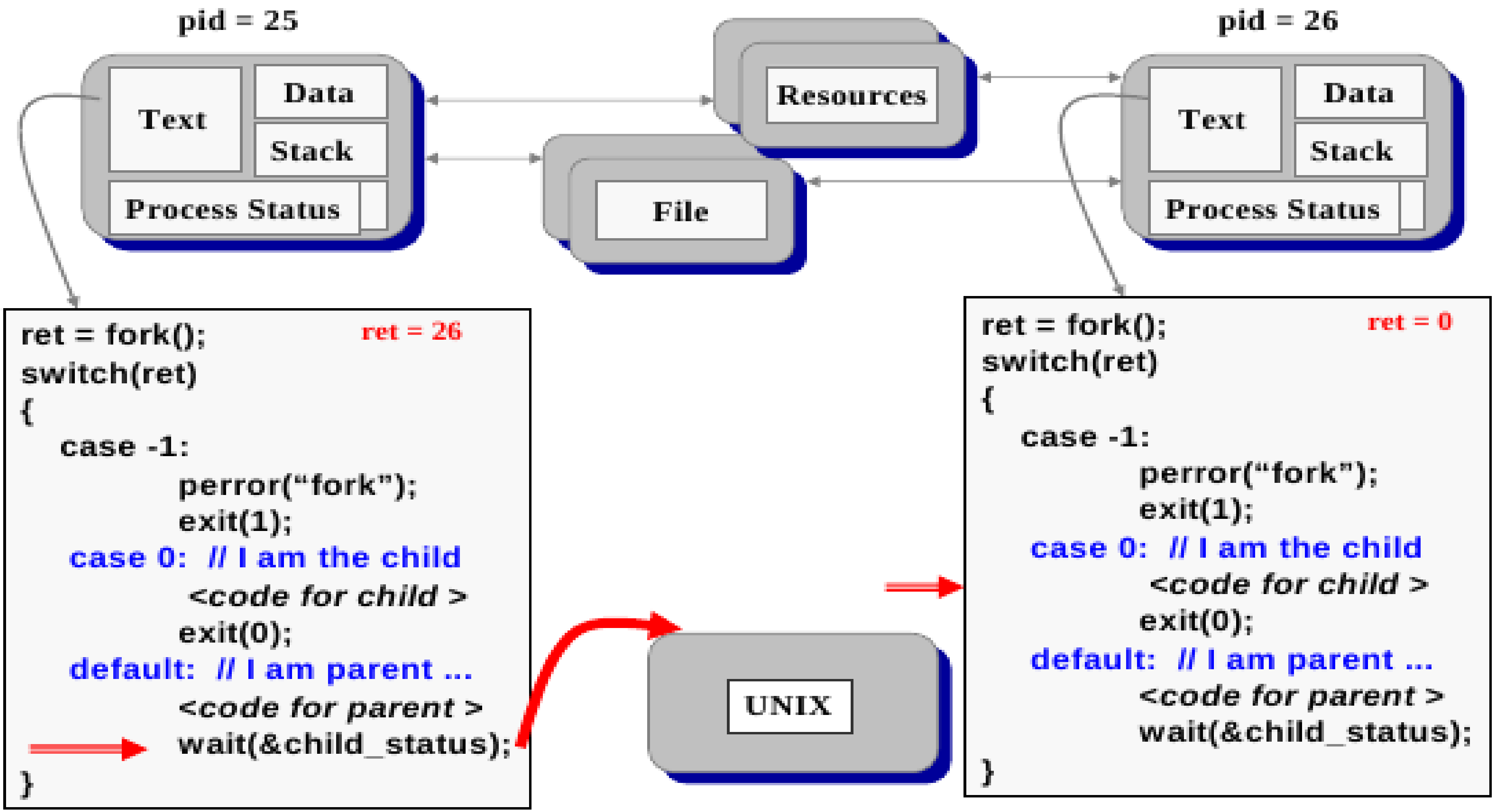


pid = 25
Text
Data
Stack
Process Status
Resources
File
pid = 26
Text
Data
Stack
Process Status
ret = fork();
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(&child_status);
}
ret = 26
UNIX
ret = fork();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

Fork

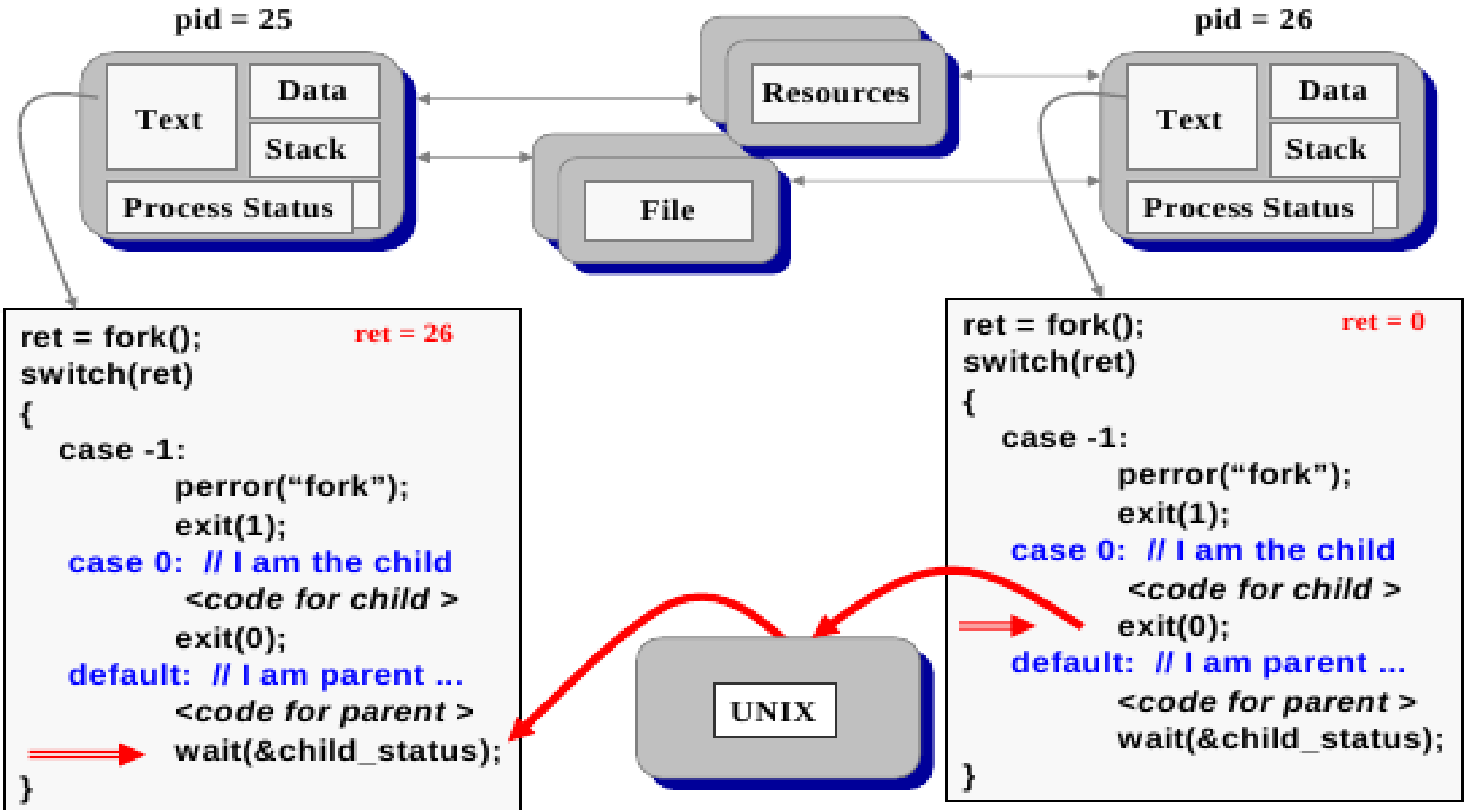

pid = 25
Text
Data
Stack
Process Status
Resources
File
pid = 26
Text
Data
Stack
Process Status
ret = fork();
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(&child_status);
}
ret = 26
UNIX
ret = fork();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

Fork

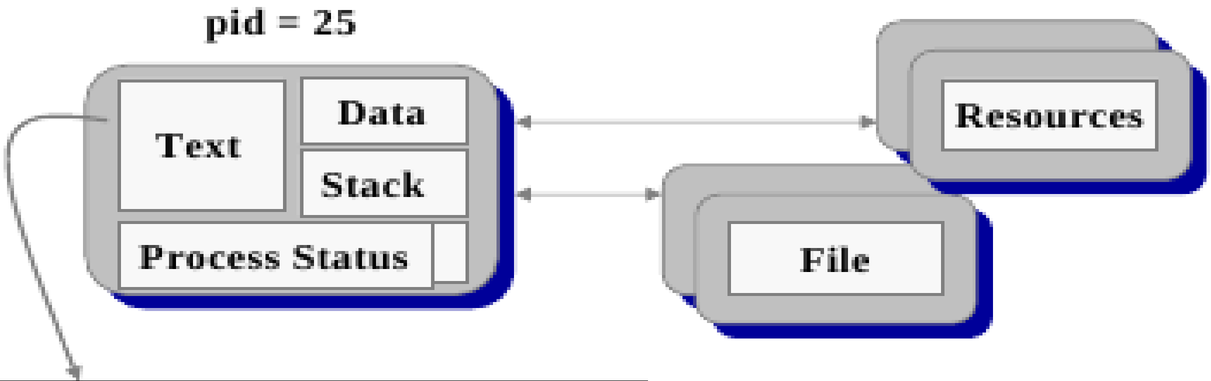


pid = 25
Text
Data
Stack
Process Status
Resources
File
pid = 26
Text
Data
Stack
Process Status
ret = fork();
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent ...
<code for parent >
wait(&child_status);
}
ret = 26
UNIX
ret = fork();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

Fork


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

Fork




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

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

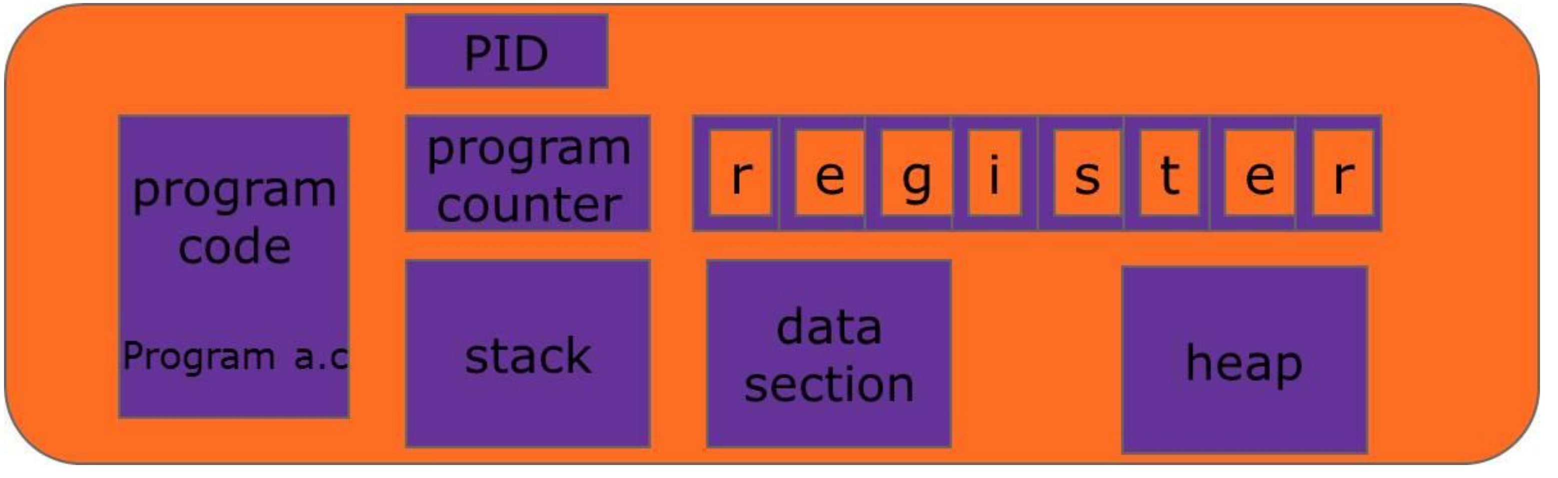
The exec

- ✓ 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

exec working

Example for exec

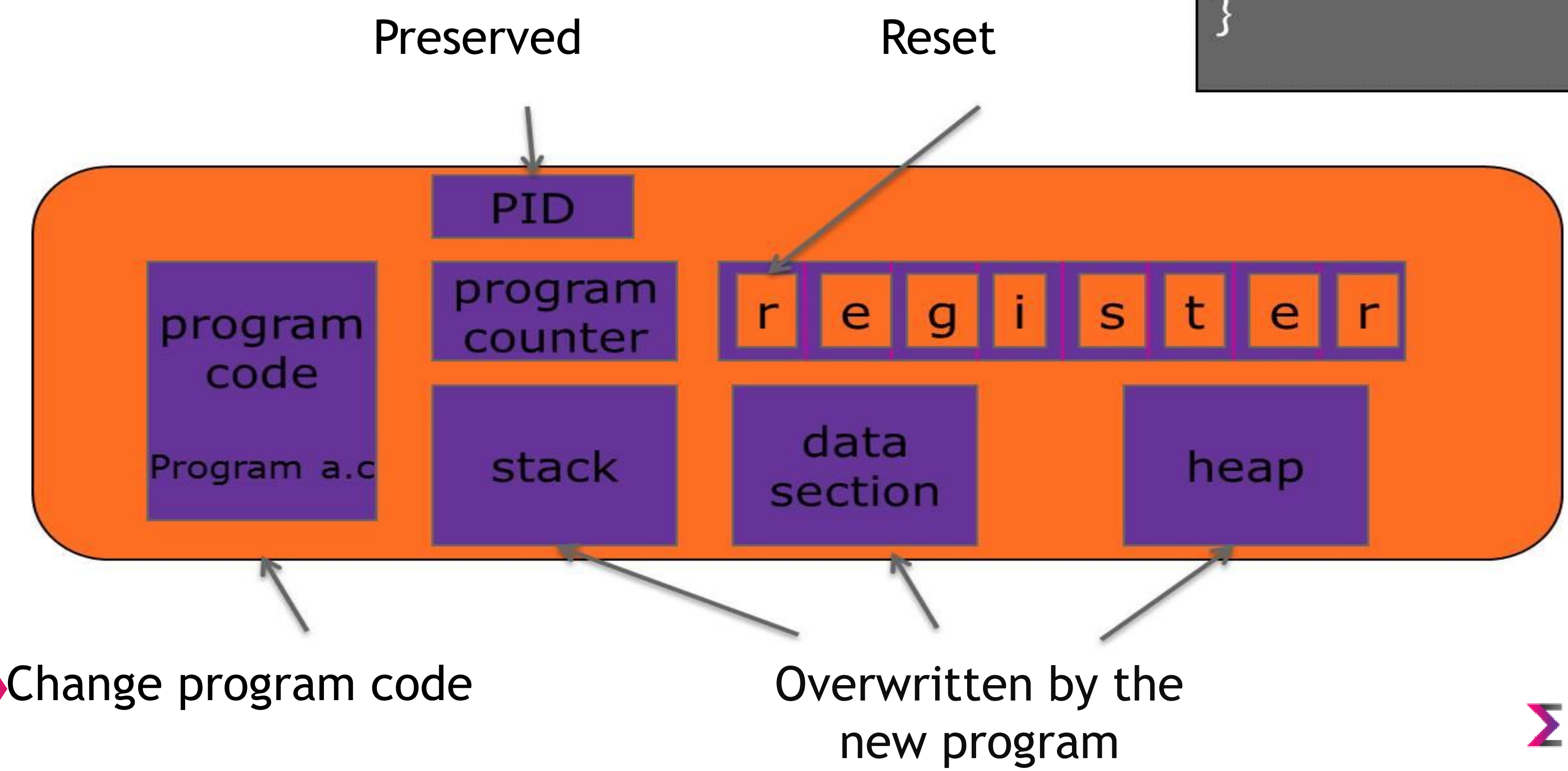
```
//Program a.c

int main()
{
:
//want to execute "ls"
execlp("/bin/ls", "ls", NULL);
:
}
```



exec working

After exec, the parent process is replaced by the newly created one

```
//ProgramA.c

int main()
{
:
//want to execute "ls"
execlp("/bin/ls", "ls", NULL);
:
}
```



exec family

- ✓ 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

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

Copy On Write (COW)

✓ 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

The wait

✓ 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

✓ 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

EMERTXE

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 */
};
```

Zombie Process

✓ 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)

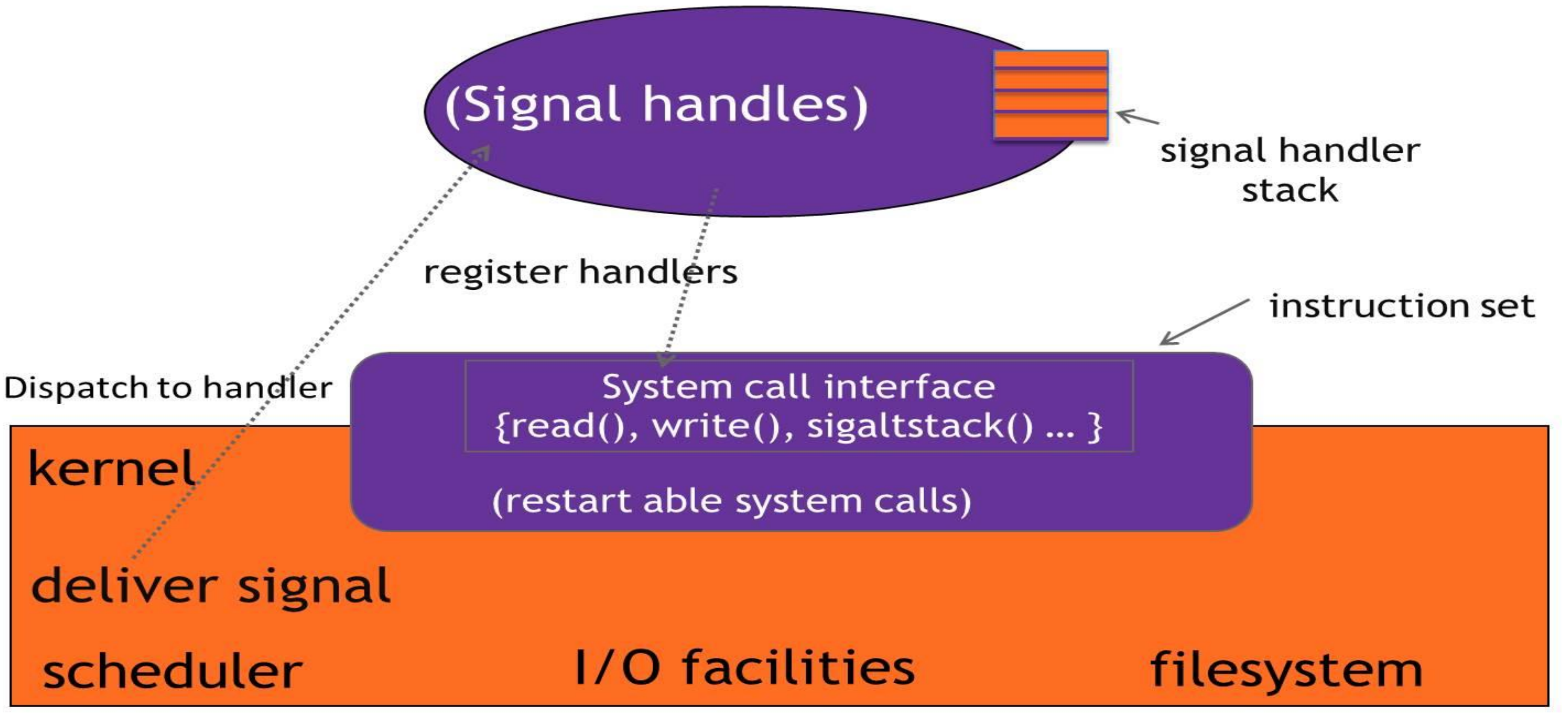
✓ The init process automatically cleans up any zombie child processes that it inherits.

Signals

Introduction

- ✓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



(Signal handles)
signal handler
stack
register handlers
instruction set
Dispatch to handler
System call interface
{read(), write(), sigaltstack() ... }
(restart able system calls)
kernel
deliver signal
scheduler
I/O facilities
filesystem

Signal 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.

Signal - 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 (send kill signal to PID 4481)
  - kill –USR1 4481 (Send user signal to PID 4481)

Signal - 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’

Signal - 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

Signal - 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 & Interrupts

✓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.

Advanced signal 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

EMERTXE

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

Self & Signals

- ✓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’

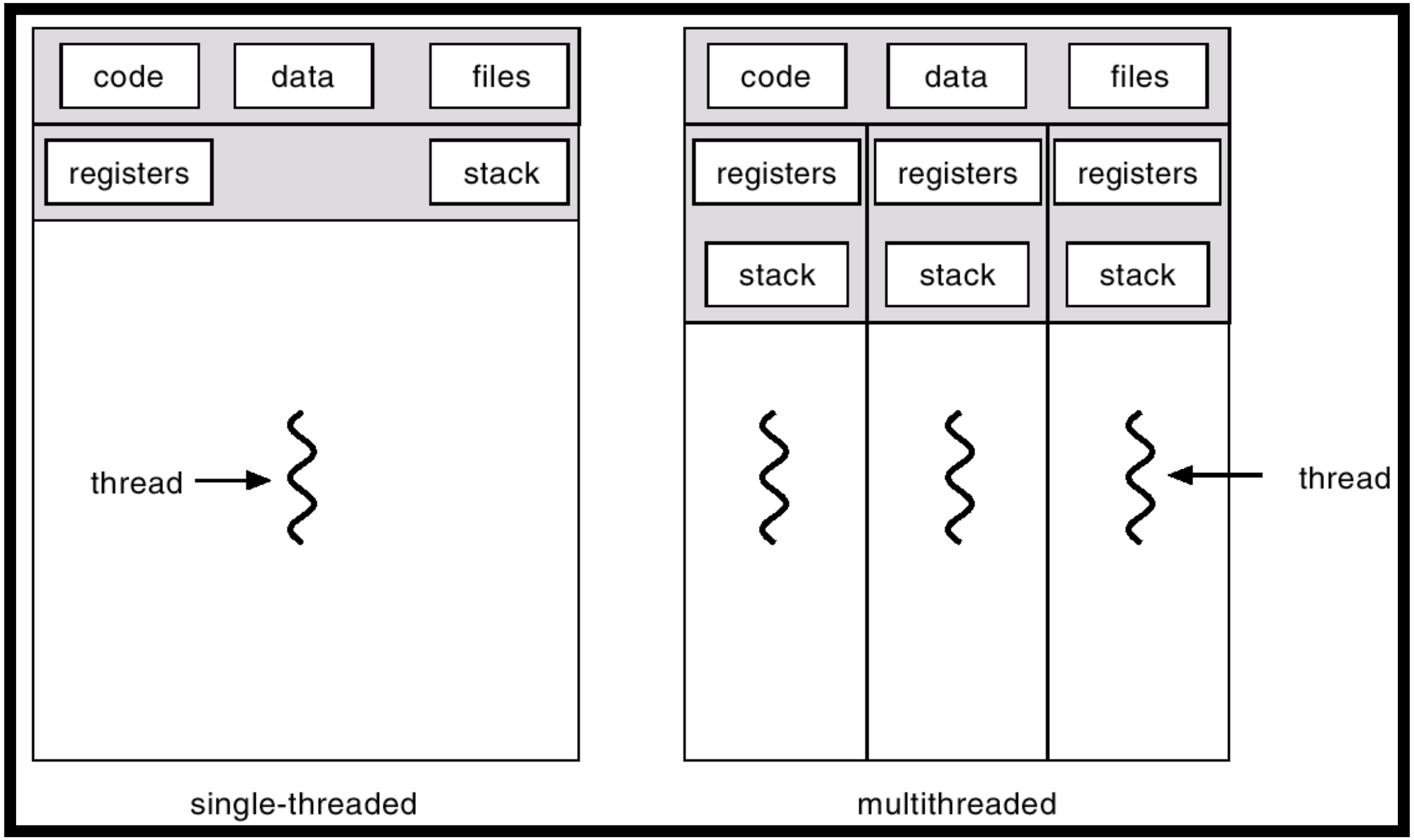
Function	Meaning
raise (int sig)	Raise a signal to currently executing process. Takes signal number as input
alarm (int sec)	Sends an alarm signal (SIGALRM) to currently executing process after specified number of seconds
pause()	Suspends the current process until expected signal is received. This is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

Threads

What is Thread?

- ✓ 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

Single and Multi-threaded Processes



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.

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

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

How To Compile

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

```
# gcc -o <output> <inputfile.c>  -lpthread
```

EMERTXE

Thread 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

Thread 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

Thread 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.

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**

Thread 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

Thread 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

Now let us try to understand more by exploring ATTACH/DETACH attribute

Joinable & Detached threads

- ✓ 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

Creating detached threads

- ✓In order to create a dethatched 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 ✓ Returns 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

Now let us try the basic thread program with a detached thread attribute

Thread 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

EMERTXE

Thread 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

Why Synchronization?

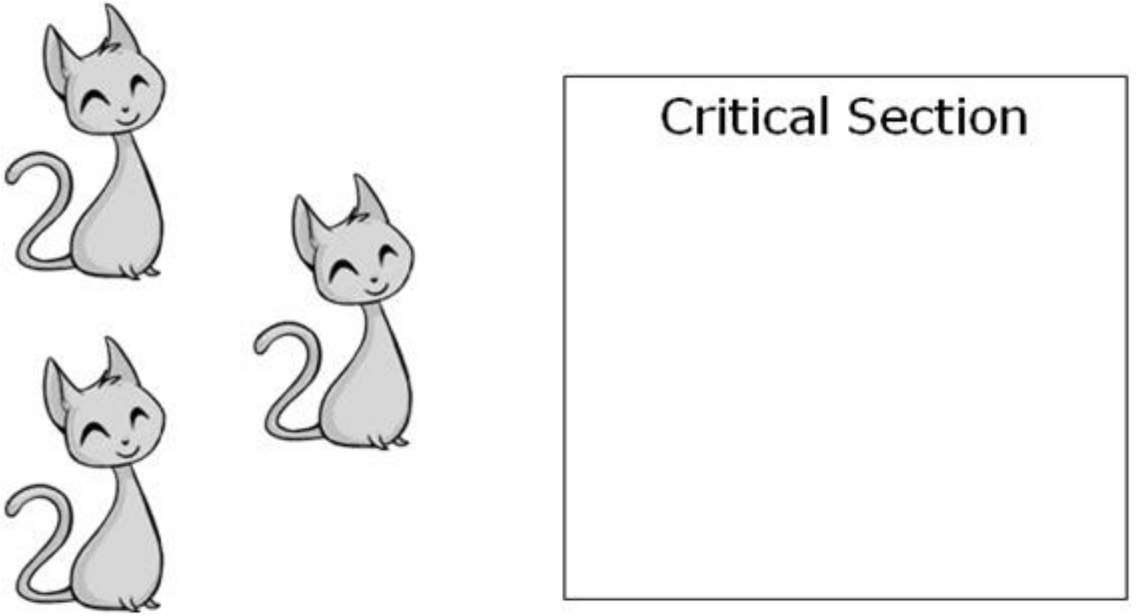
- ✓ When multiple tasks are running **simultaneously**:
  - ✓ either on a single processor, or on
  - ✓ a set of multiple processors
- ✓ They give an appearance that:
  - ✓ For each task, it is the only task in the system.
  - ✓ At a higher level, all these tasks are executing efficiently.
  - ✓ Tasks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

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

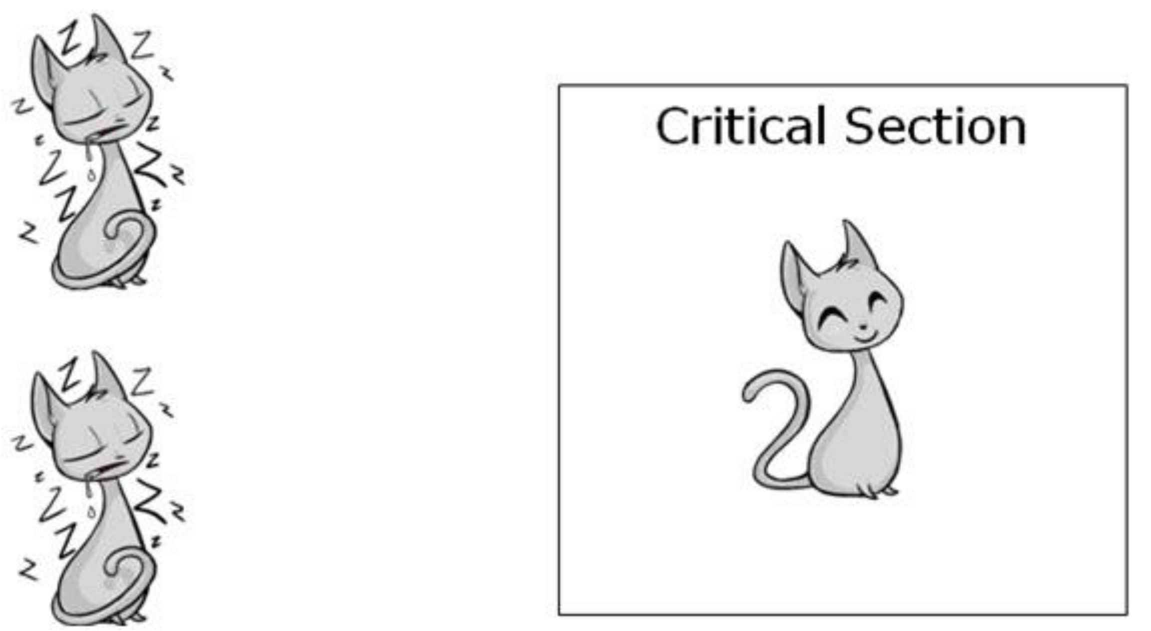
Critical section

- ✓ 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.



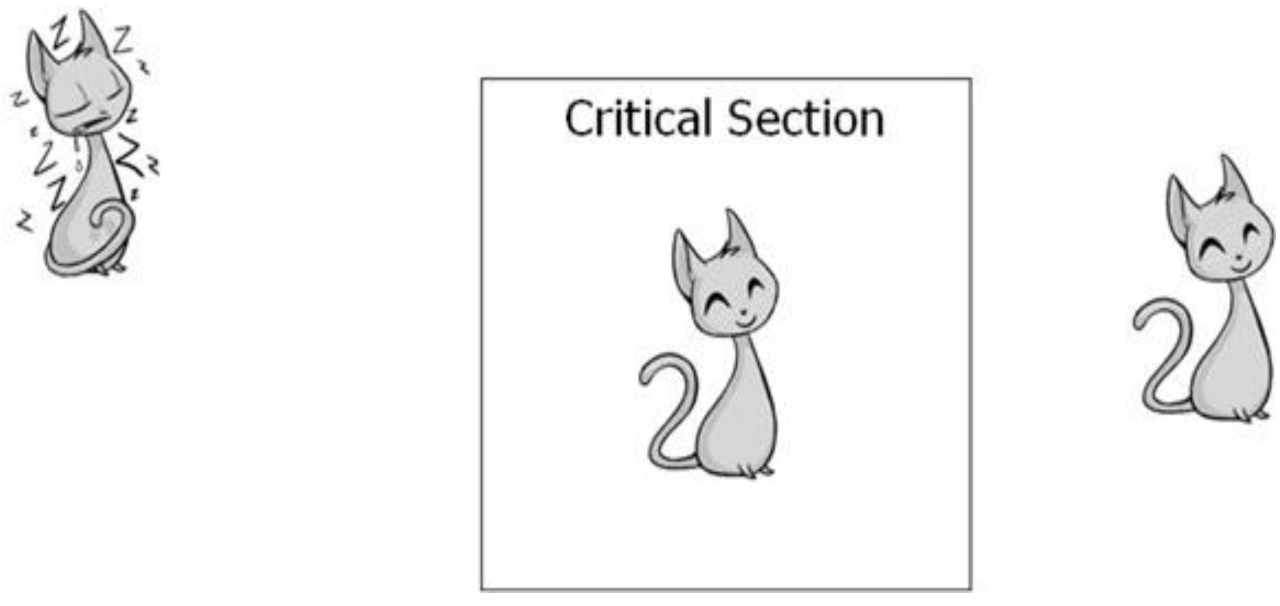
Critical section

- ✓ 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.



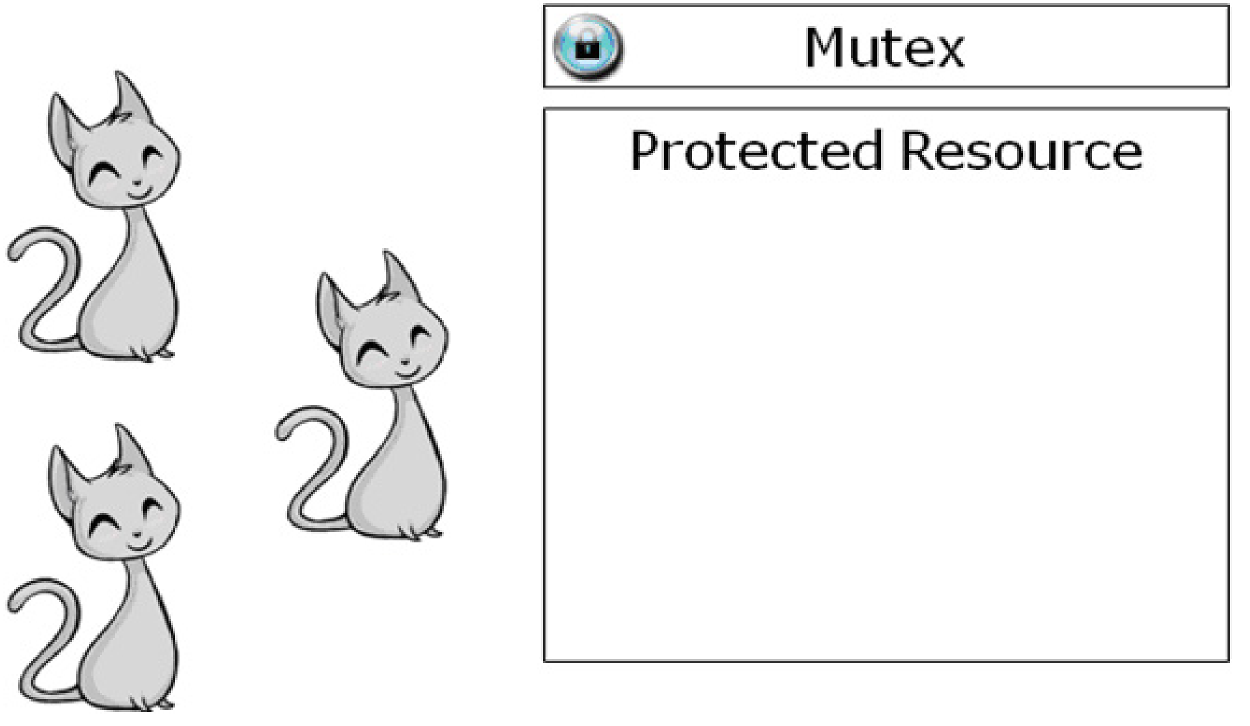
Critical section

- ✓ 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



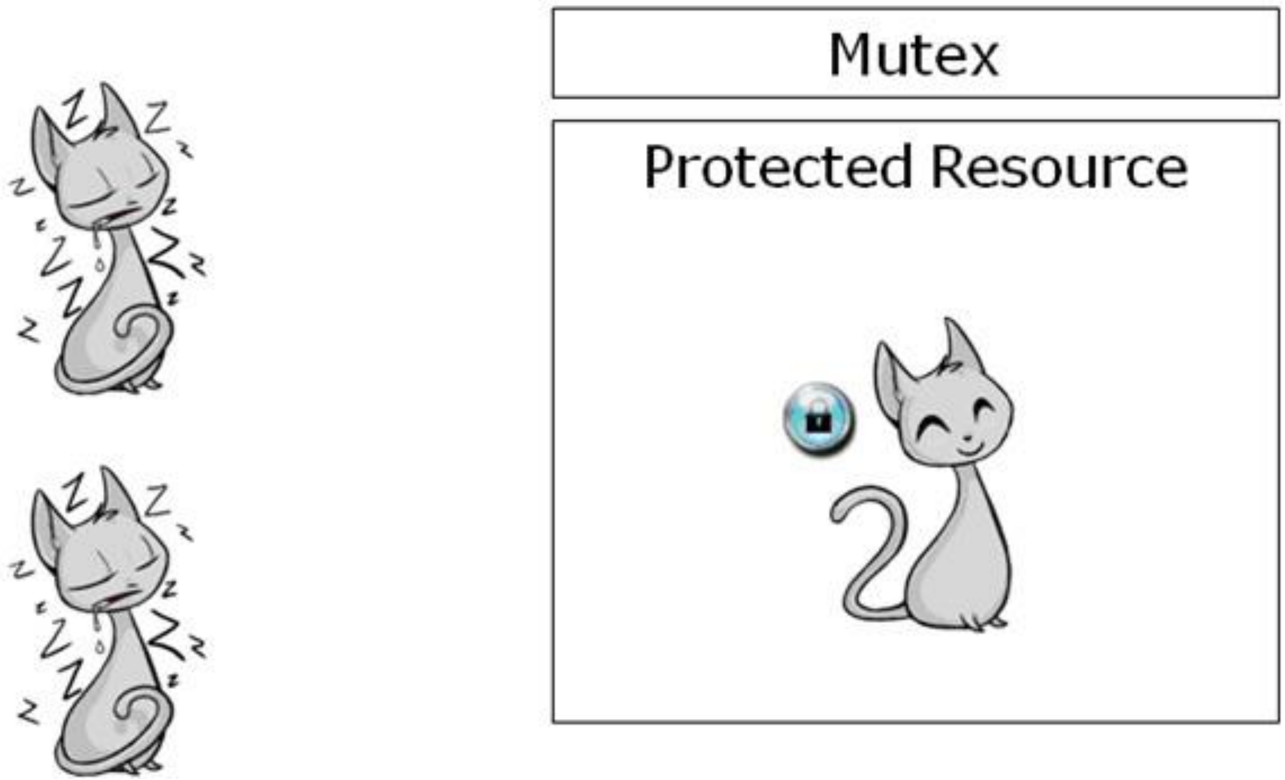
Mutual Exclusion

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



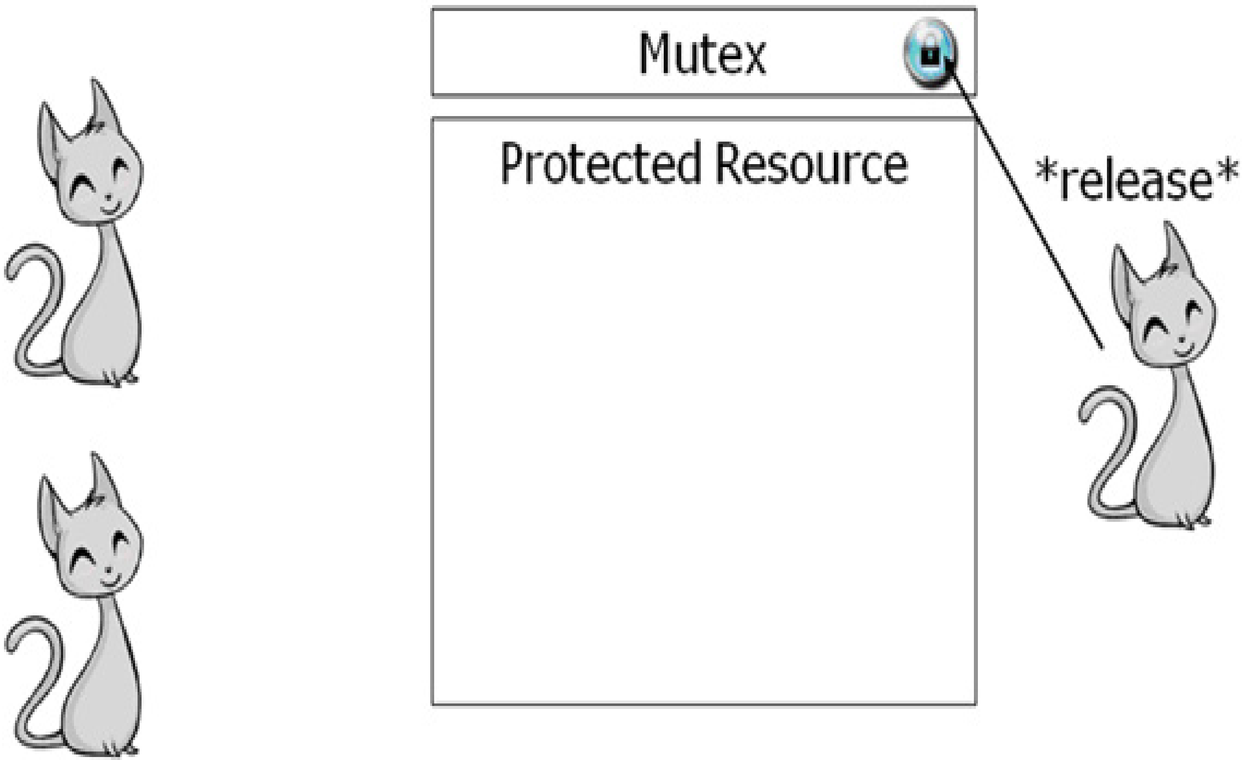
Mutual Exclusion

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



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.



Locking & Blocking

- ✓ A task may attempt to lock a mutex by calling a ***lock*** method on it.
- ✓ If the mutex was unlocked, it becomes locked and the function returns immediately.
- ✓ If the mutex was locked by another task, the *locking function* ***blocks*** execution and returns only eventually when the mutex is ***unlocked*** by the other task.
- ✓ *More than one task* may be blocked on a locked mutex at one time.
- ✓ When the mutex is unlocked, *only one* of the blocked tasks is unblocked and allowed to lock the mutex; the other tasks stay blocked.

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

Thread Synchronization

Thread Synchronization

- ✓ 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

Thread - 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 ✓ RETURN: 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)

Thread - 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

Two 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)

Thread - 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 Communication(IPC)

Introduction

✓*Interprocess 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

IPC - Categories

✓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:

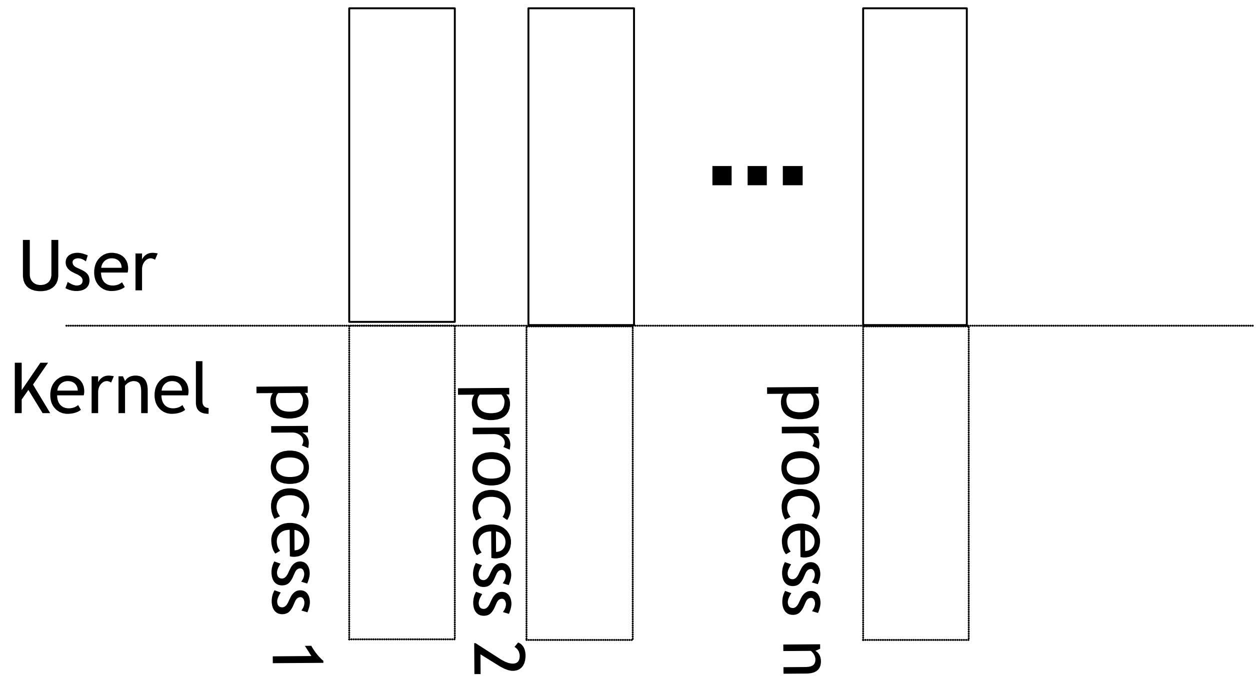
- Pipes
- FIFO (or named pipes)
- Message Queues
- Shared memory
- Semaphores (Process level)
- Sockets

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

EMERTXE

User vs. Kernel space

✓Protection domains - (virtual address space)

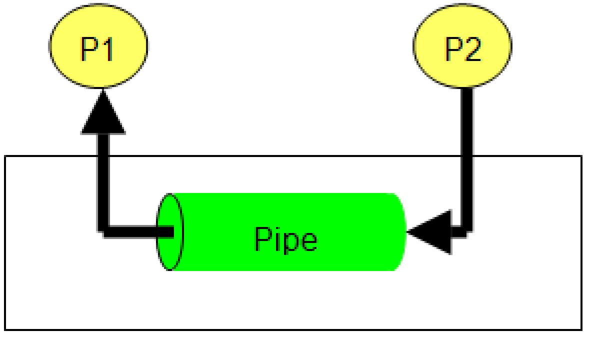


How can processes communicate with each other and the kernel? The answer is nothing but IPC mechanisms

Pipes

Properties

- ✓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



Creating Pipes

- ✓ 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.

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

ΣMERTXE

Pipes Vs FIFO

- ✓ 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

Creating a FIFO

- ✓ 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

Accessing 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.

Shared Memory

Properties

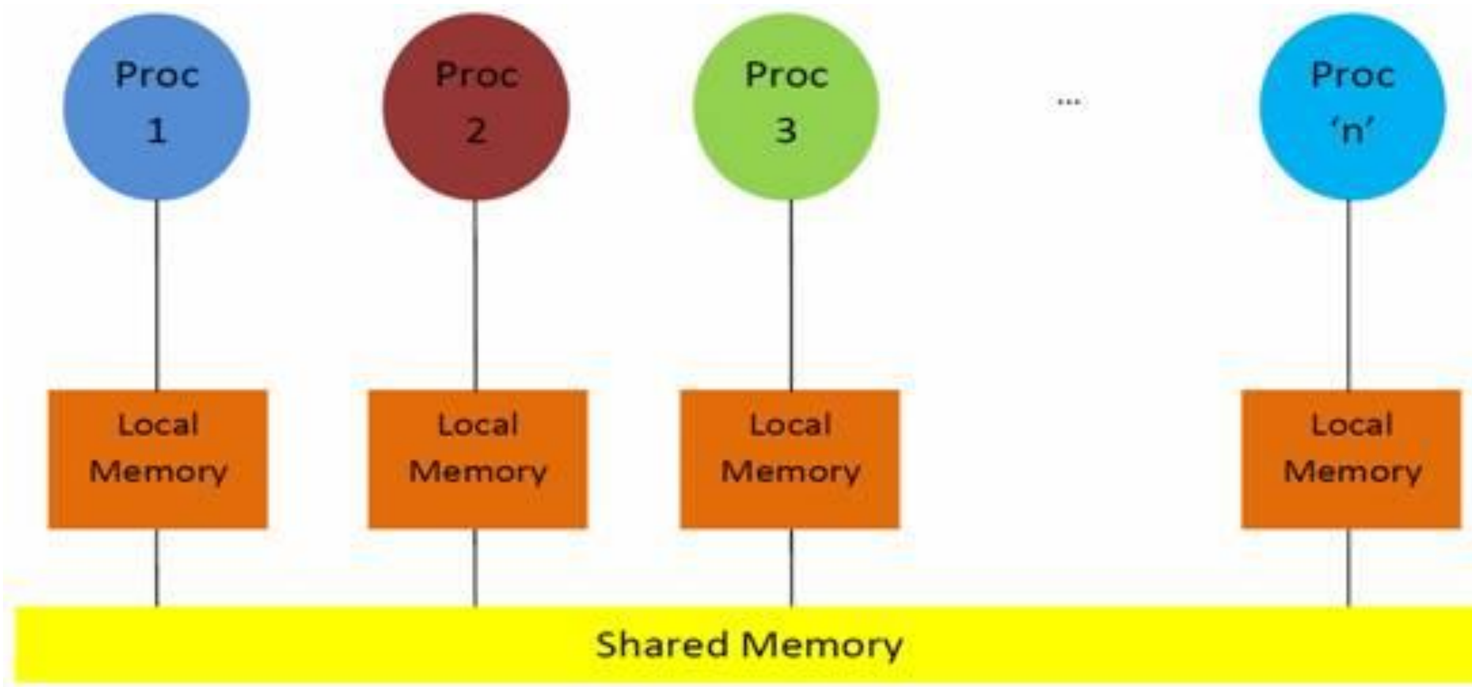
- ✓ Shared memory allows two or more processes to access the same memory
- ✓ When one process changes the memory, all the other processes see the modification
- ✓ Shared memory is the fastest form of Inter process communication because all processes share the same piece of memory
- ✓ It also avoids copying 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 ☺

EMERTXE

Shared vs. Local memory



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.

Shared memory 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

Process Semaphores

Properties

- ✓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

Semaphore calls

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

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 code illustrates that one shared memory segment, numbered 1627649, is in use:

```
Jayakumar>ipcs -s
```

Semaphores in the system

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

```
Jayakumar>ipcs -m | more

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 262144     jayakumar  777        2376       2          dest
0x00000000 294913     jayakumar  777        5016       2          dest
0x00000000 327682     jayakumar  777        4136       2          dest
0x00000000 360451     jayakumar  777        2376       2          dest
0x00000000 393220     jayakumar  777        2400       2          dest
0x00000000 425989     jayakumar  777        12672      2          dest
0x00000000 458758     jayakumar  777        5472       2          dest
0x00000000 491527     jayakumar  777        7680       2          dest
0x00000000 524296     jayakumar  777        33504      2          dest
0x00000000 557065     jayakumar  777        11508      2          dest
0x00000000 589834     jayakumar  777        1920       2          dest
```

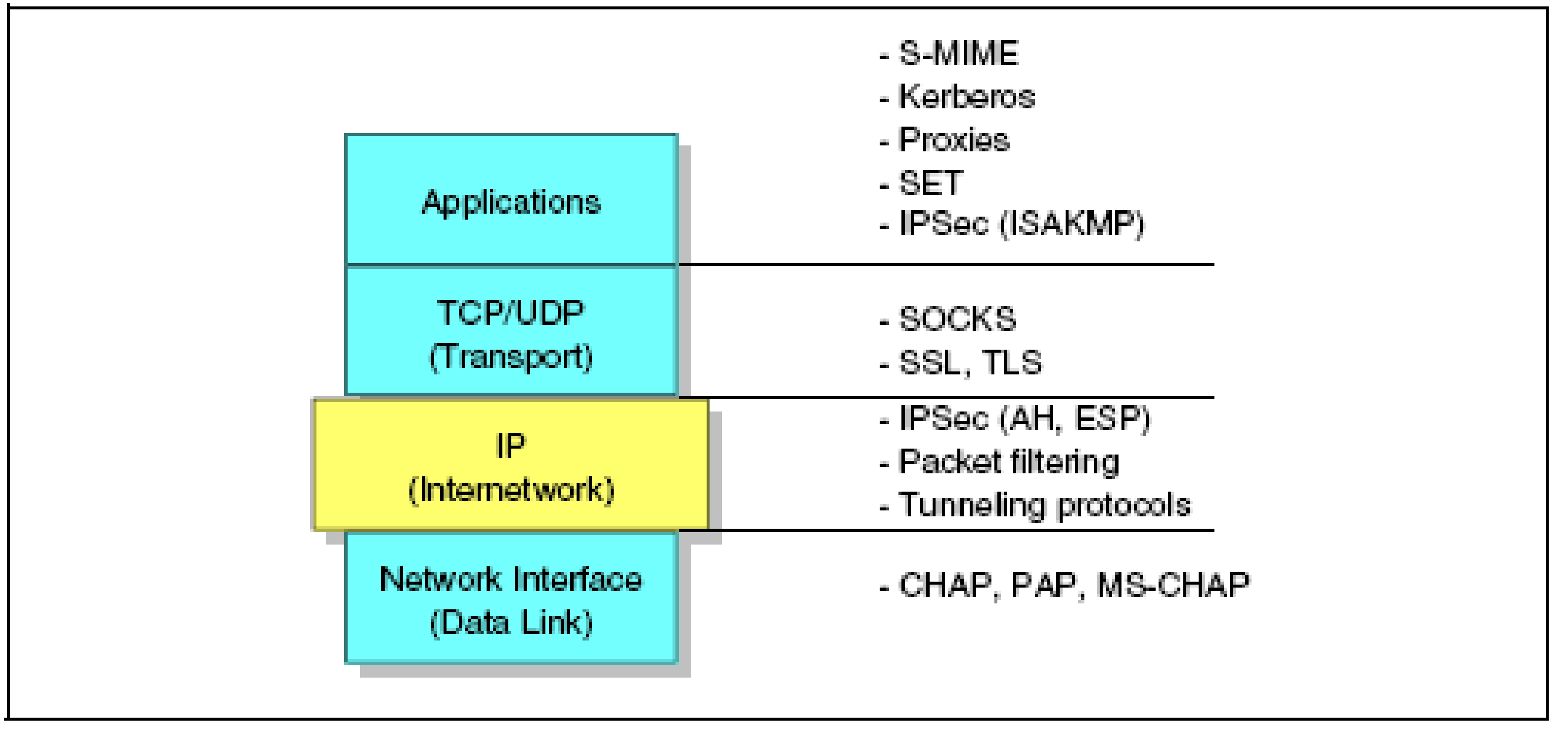
Networking - Fundamentals

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

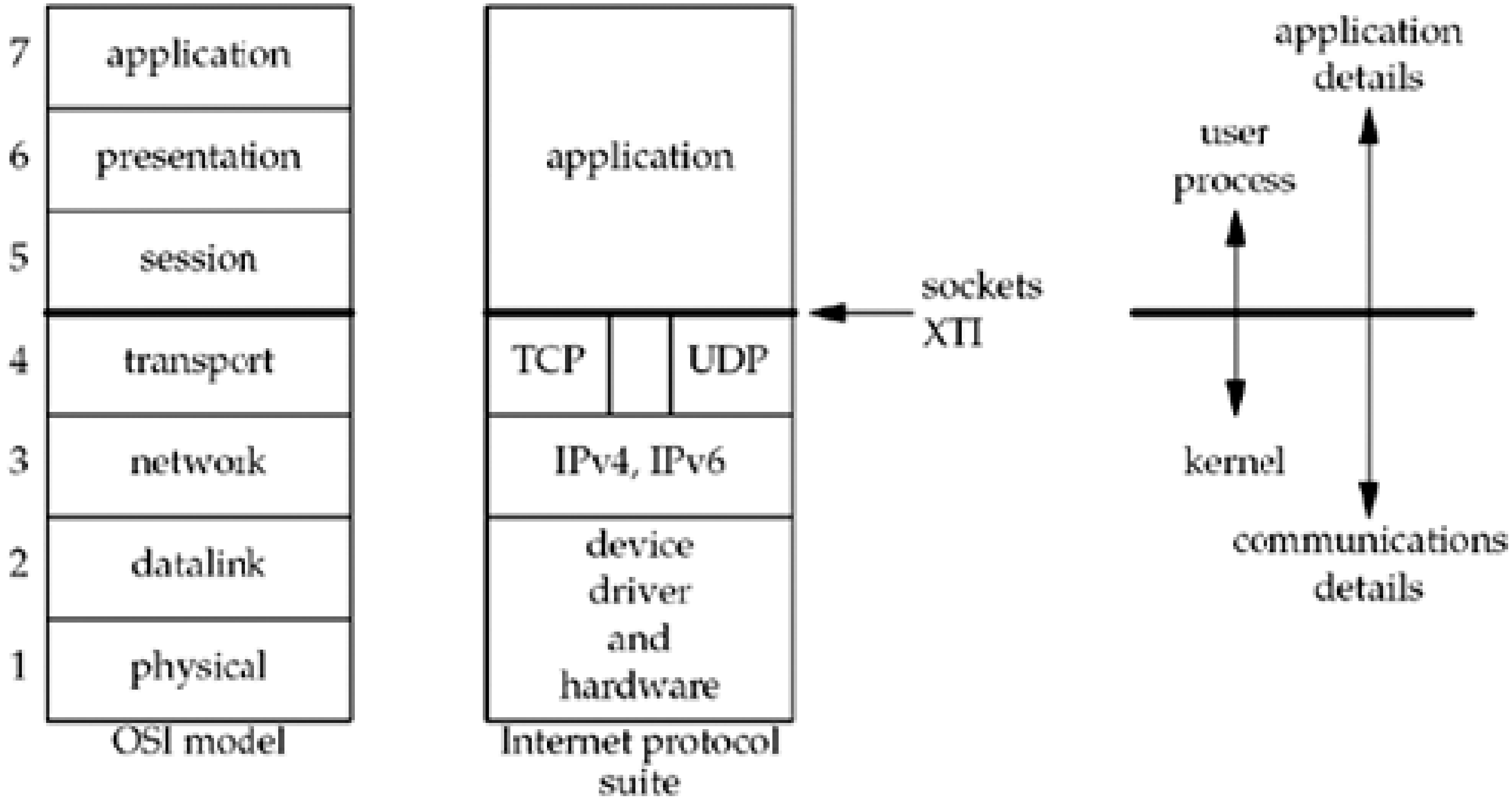
Organizations like IEEE, IETF, ITU etc…work together in creating global standards for interoperability and compliance

TCP/IP model

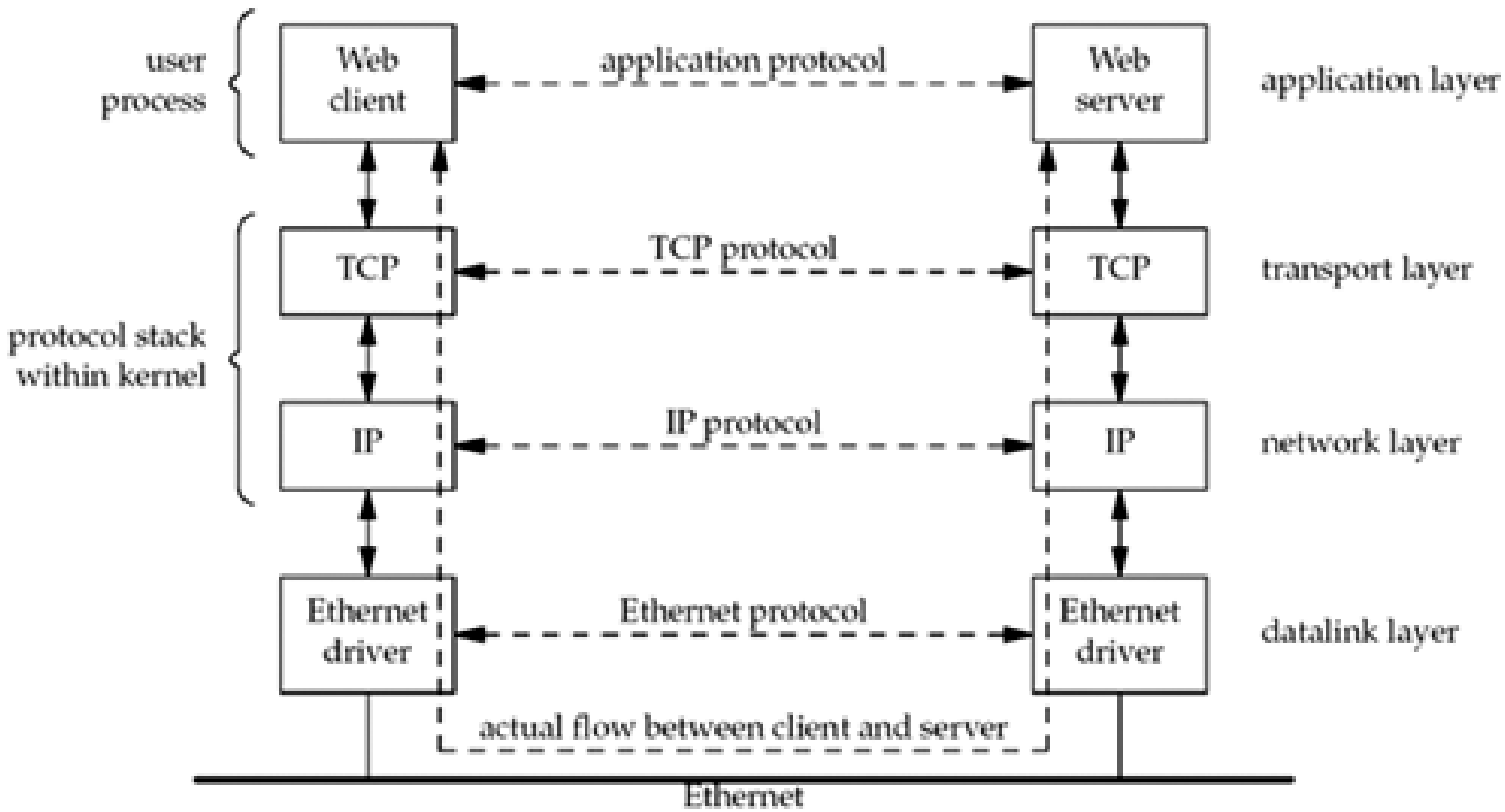


Applications
TCP/UDP
(Transport)
IP
(Internetwork)
Network Interface
(Data Link)
- S-MIME
- Kerberos
- Proxies
- SET
- IPSec (ISAKMP)
- SOCKS
- SSL, TLS
- IPSec (AH, ESP)
- Packet filtering
- Tunneling protocols
- CHAP, PAP, MS-CHAP

TCP/IP Implementation in Linux



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
datalink layer
actual flow between client and server
Ethernet

Network - 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

✓Post number helps in multiplexing and de-multiplexing the messages

Sockets

Properties

✓Sockets is another IPC mechanism, different from other mechanisms are 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

Socket address

✓ In order to attach (called as “bind”) a socket to network address (IP address & phone 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 */
};
```

ΣMERTXE

Socket calls

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 */

Socket calls

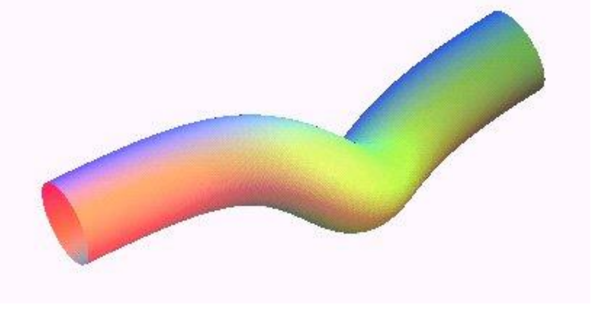
Function	Meaning
int bind(int sockfd, struct sockaddr *my_addr, int addrlen)	✓ Bind a socket to network address ✓ sockfd: Socket descriptor ✓ my_addr: Network address (IP address & port number) ✓ addrlen: Length of socket structure ✓ RETURN: Success or Failure (-1)

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));
```

Socket calls

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 */
```

EMERTXE

Socket calls

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);

Socket calls

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);
```

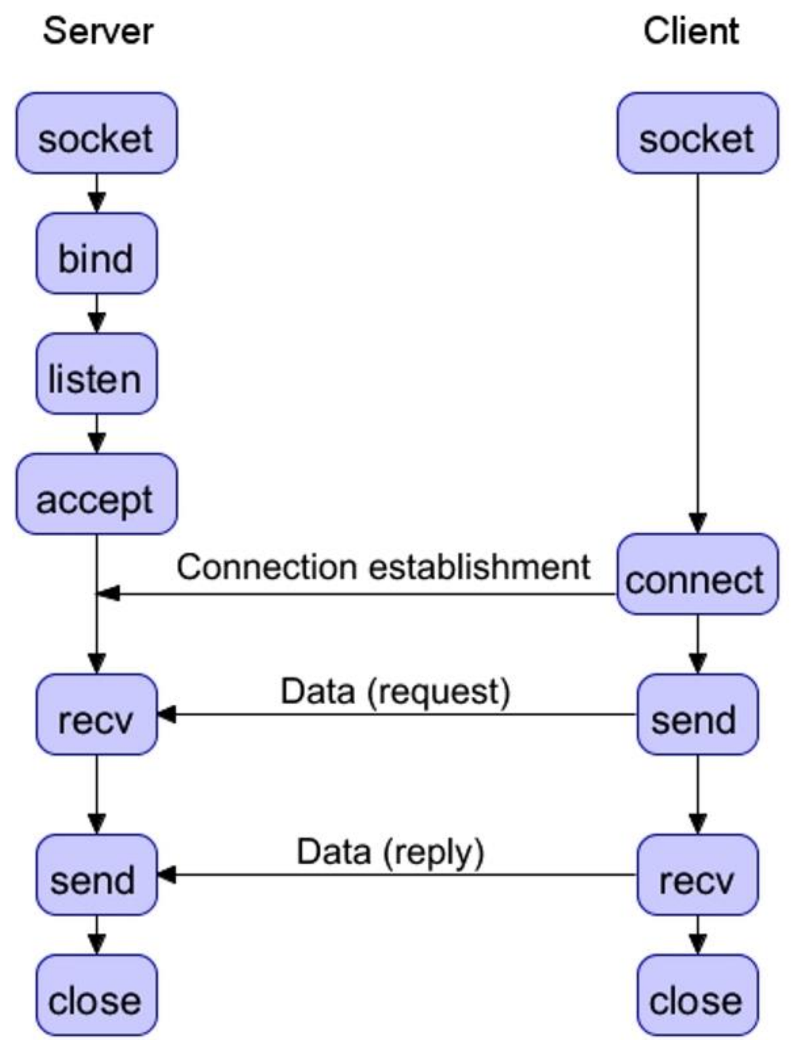
- The accept() returns a new socket ID, mainly to separate control and data sockets
- By having this servers become concurrent
- Further concurrency is achieved by fork() system call

ΣMERTXE

Socket calls

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

TCP sockets:Summary



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

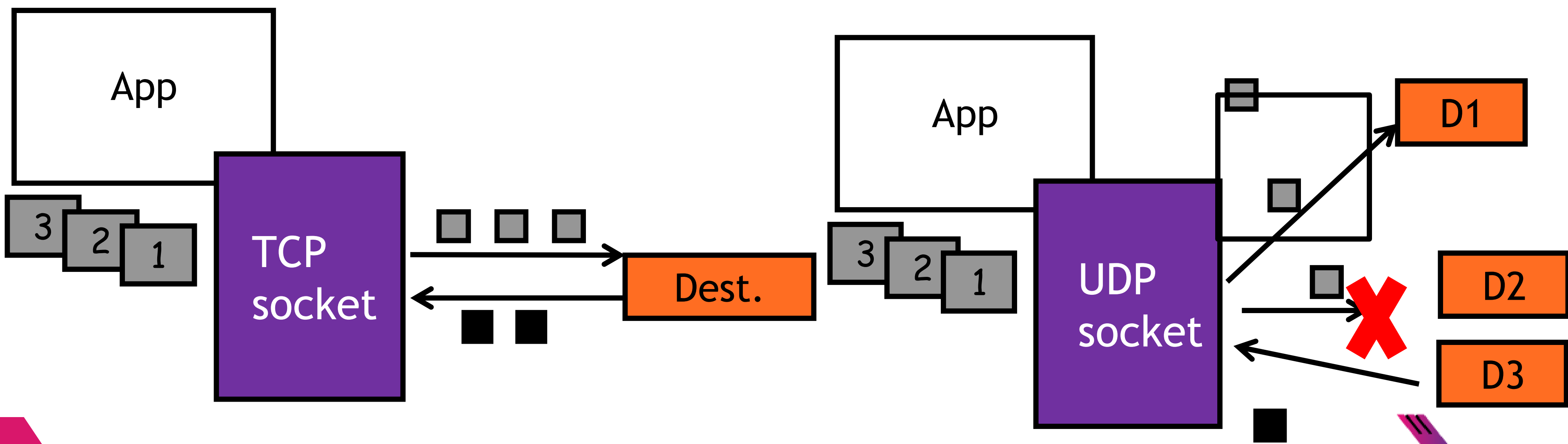
TCP & UDP sockets

TCP socket (SOCK_STREAM)

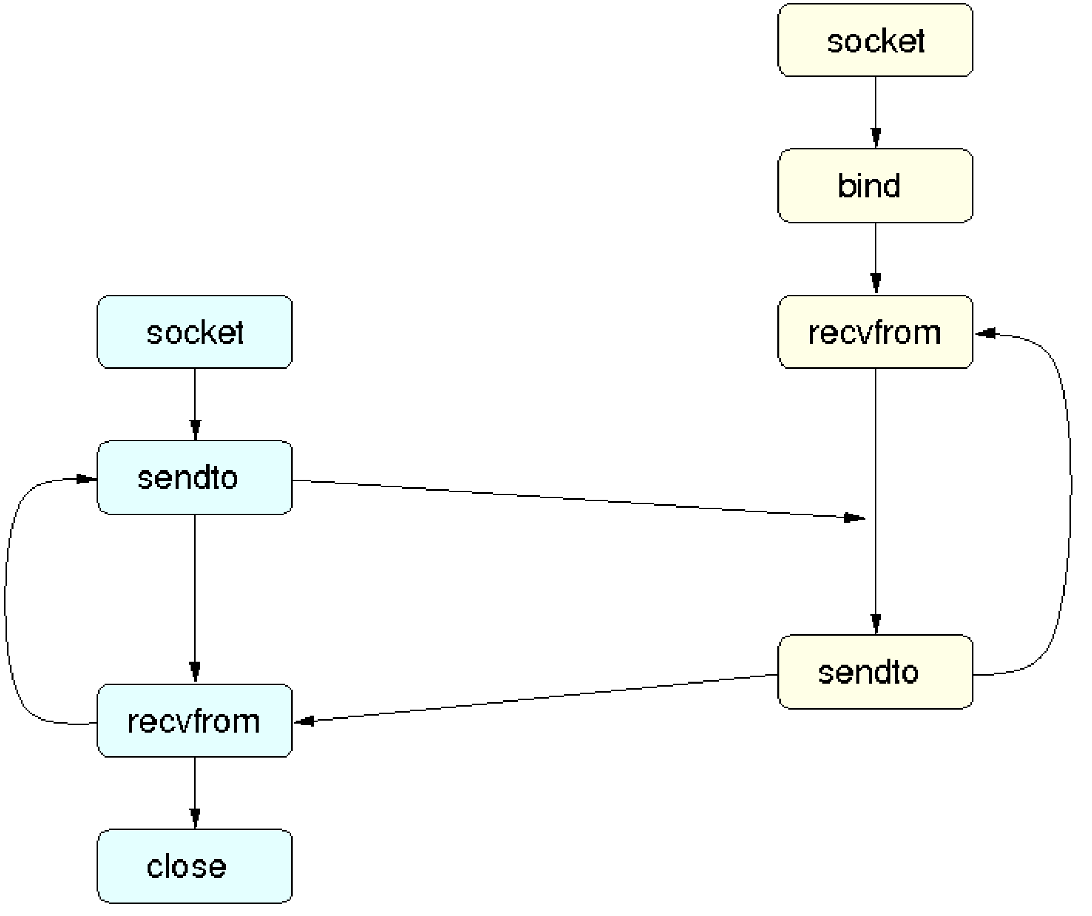
- ✓ Connection oriented TCP
- ✓ Reliable delivery
- ✓ In-order guaranteed
- ✓ Three way handshake
- ✓ More network BW

UDP socket (SOCK_DGRAM)

- ✓ Connectionless UDP
- ✓ Unreliable delivery
- ✓ No-order guarantees
- ✓ No notion of "connection"
- ✓ Less network BW



UDP sockets



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

UDP Socket 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)

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

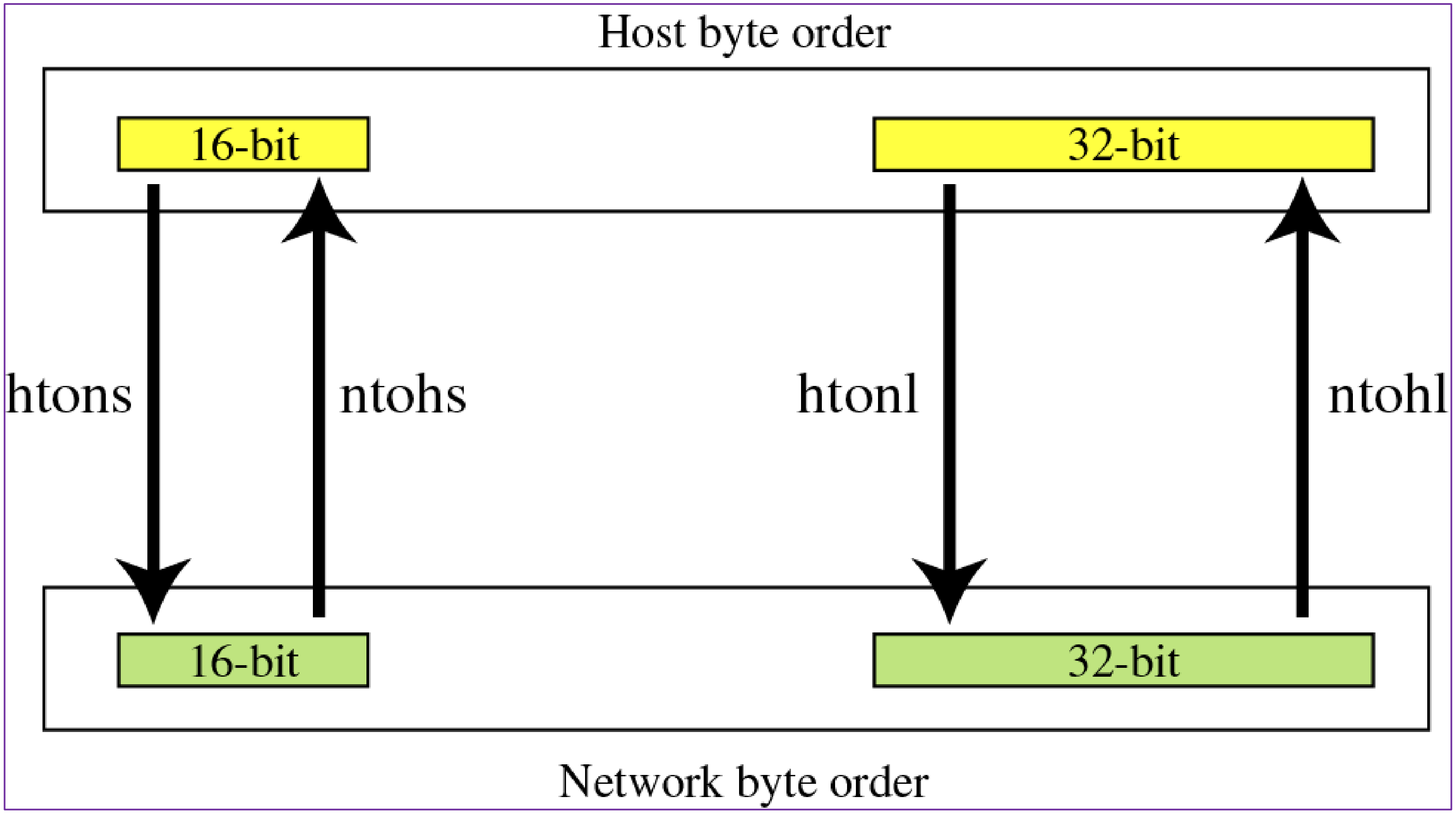
✓In order to ensure consistency network (big endian) byte order to be used as a standard

✓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

Help functions



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);
```

EMERTXE

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

EMERTXE

THANK YOU