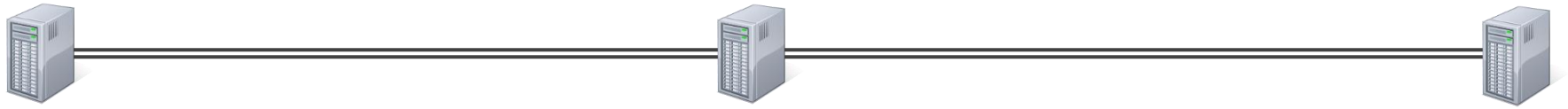
Process

(Operating System)



Deepika H V

C-DAC Bengaluru

(deepikahv@cdac.in)

- **Day 4 (March 17 2022)**
  - Process(T &L)
- **Day 5 (March 19 2022)**
  - Scheduler(T)
  - Deadlocks(T&L)
- **Day 6 (March 21 2022)**
  - Signals(T&L)
  - Threads(T& L)
- **Day 7 (March 22 2022)**
  - IPC(T&L)
- **Lab Exam**
  - KP (March 23 2022)
  - EC (March 24 2022)

Today's Agenda

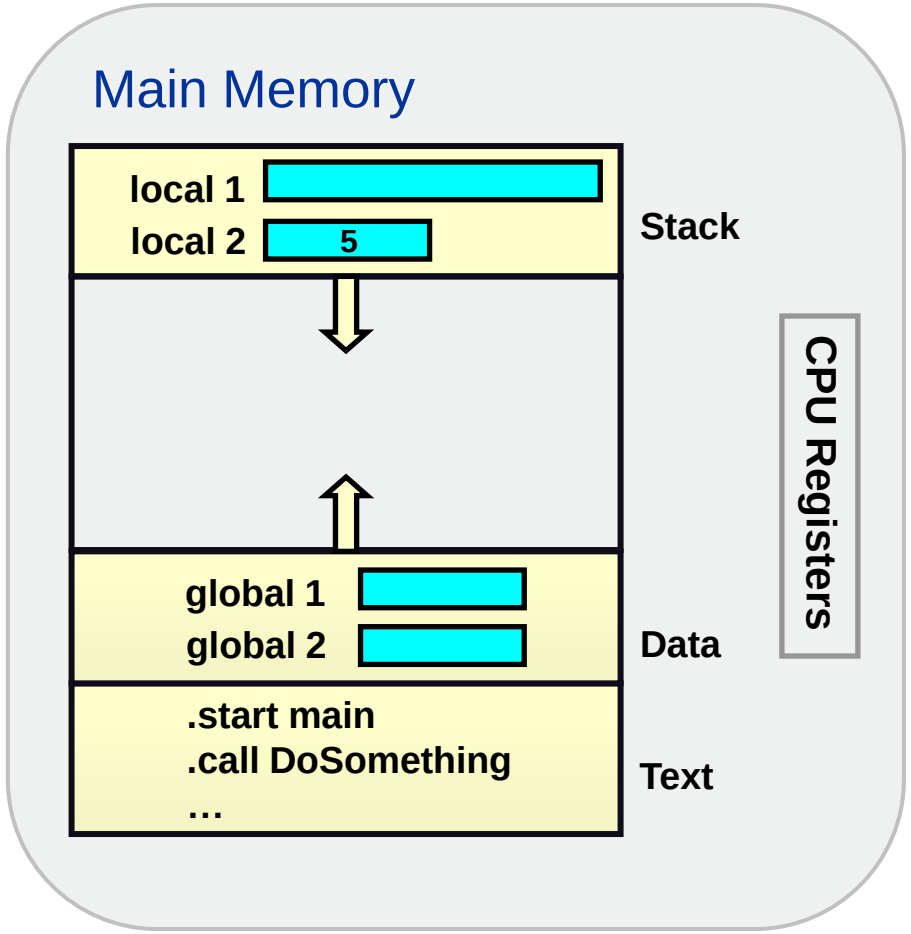
- **Process and Program**
- **Address Spaces**
- **Process Life Cycle and its States**
- **Process Identifier**
- **Process Creation**
  - Fork
  - Exit
  - Wait
  - Exec

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

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

Process contains

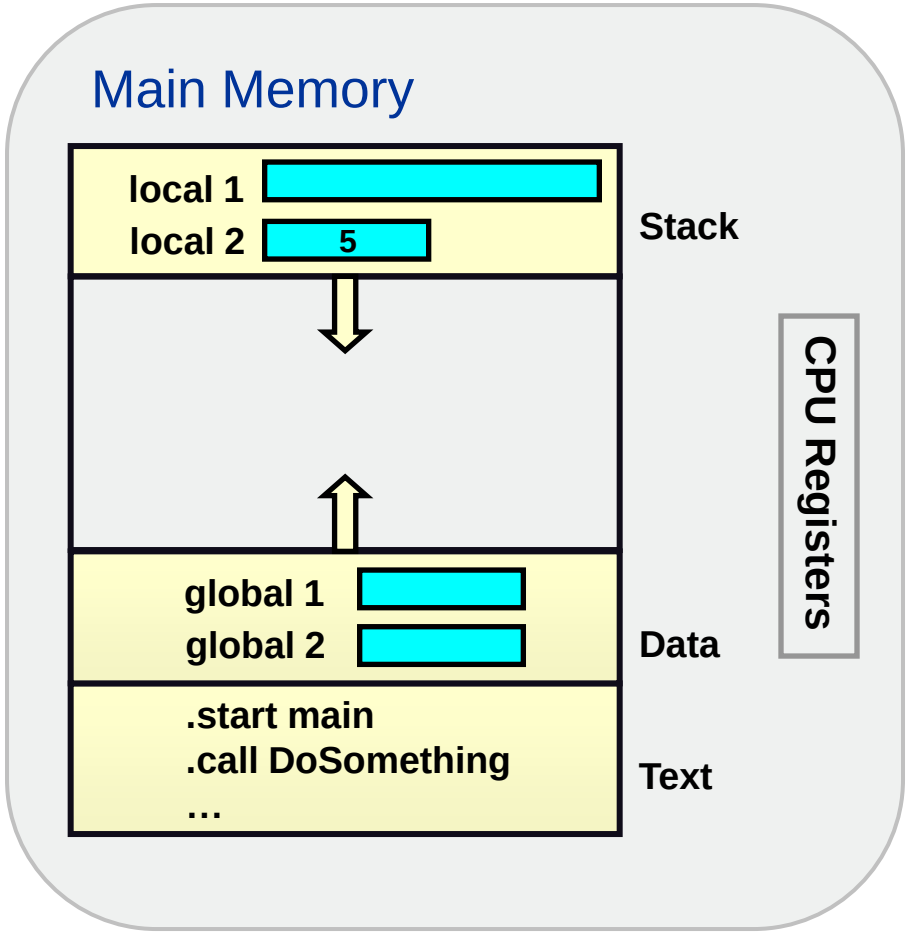
```
int global1 = 0;
int global2 = 0;

void DoSomething()
{
   int local2 = 5;

   local2 = local2 + 1;
   ...
 }

int main()
{
   char local1[10];

   DoSomething();
                ...
}
```

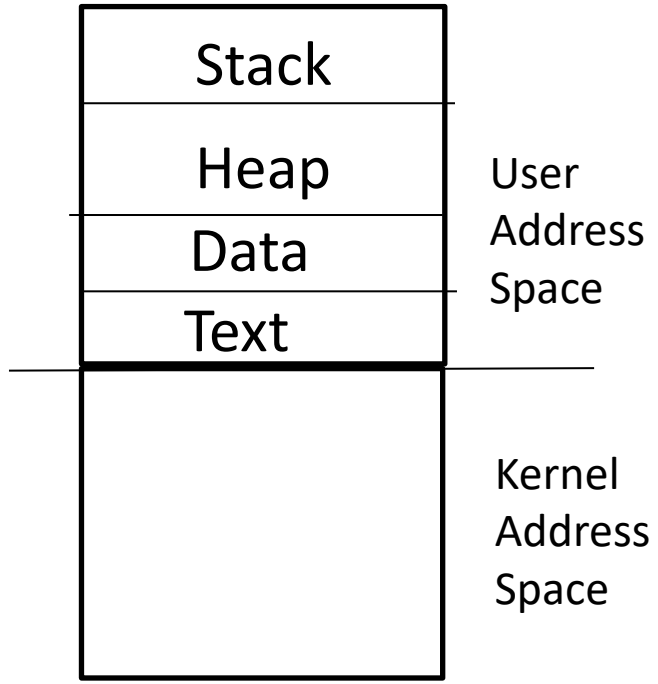


Process contains

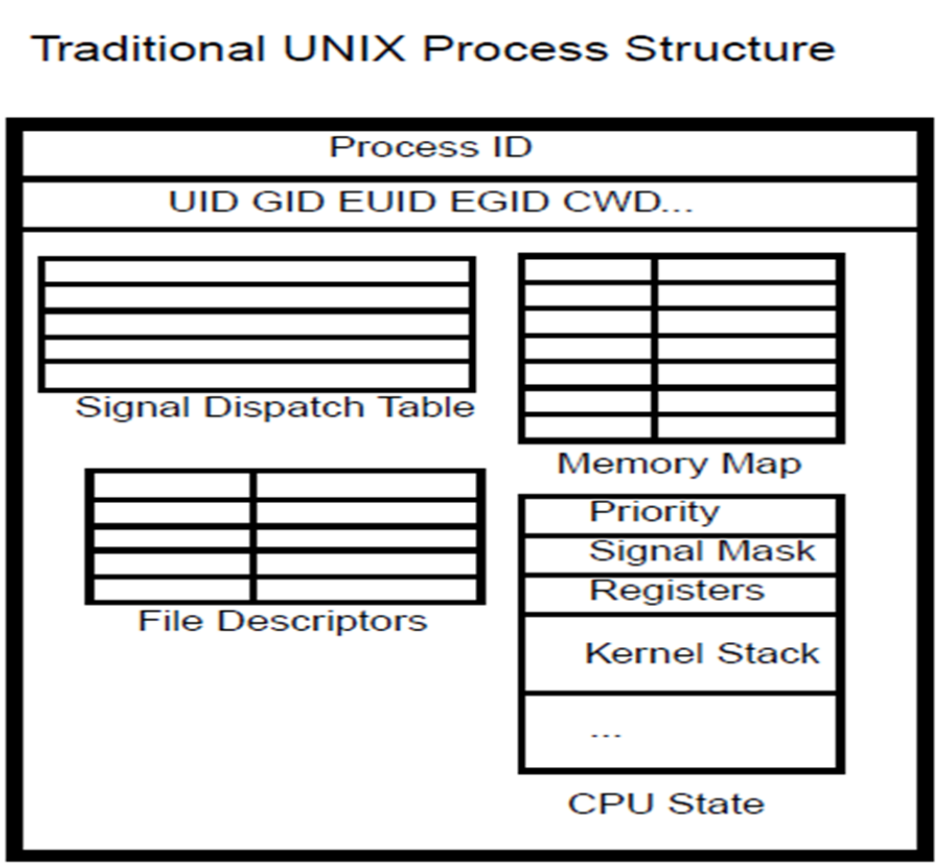
- **User Address Space**

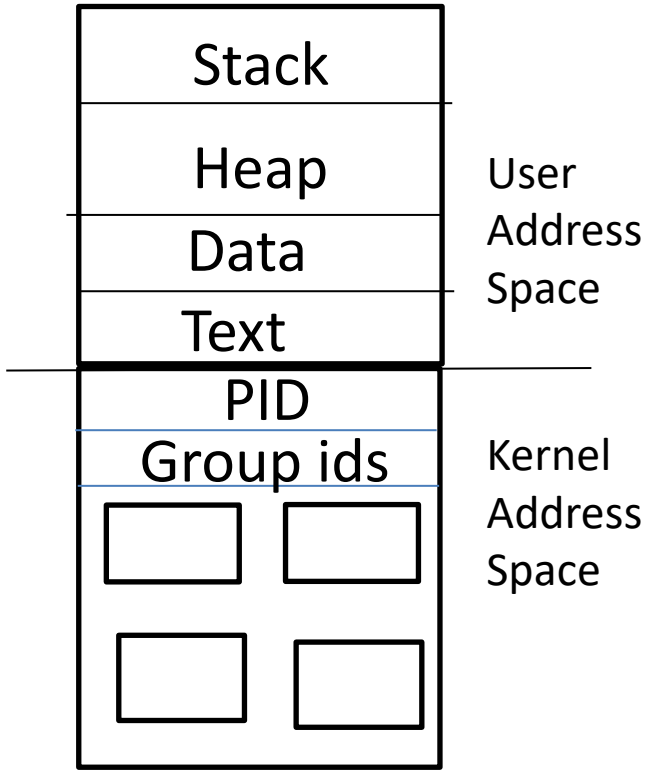
```
int global1 = 0;
int global2 = 0;

void DoSomething()
{
   int local2 = 5;

   local2 = local2 + 1;
   ...
 }

int main()
{
   char local1[10];

   DoSomething();
                ...
}
```





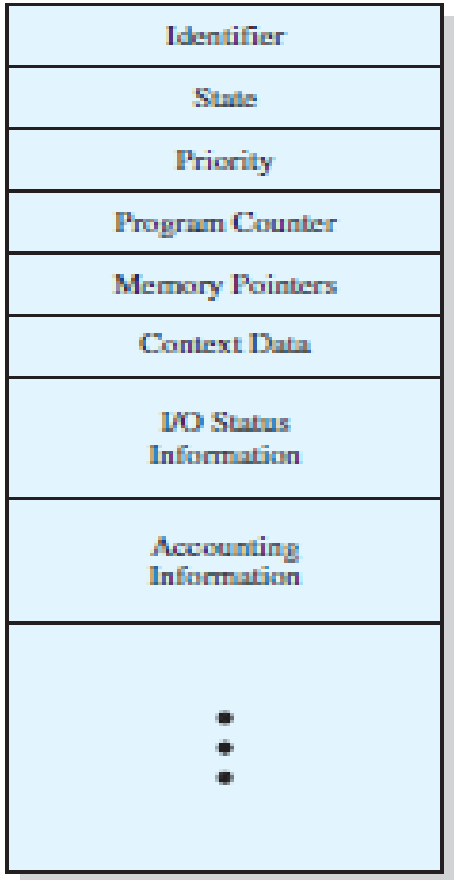
- **Kernel Address Space**



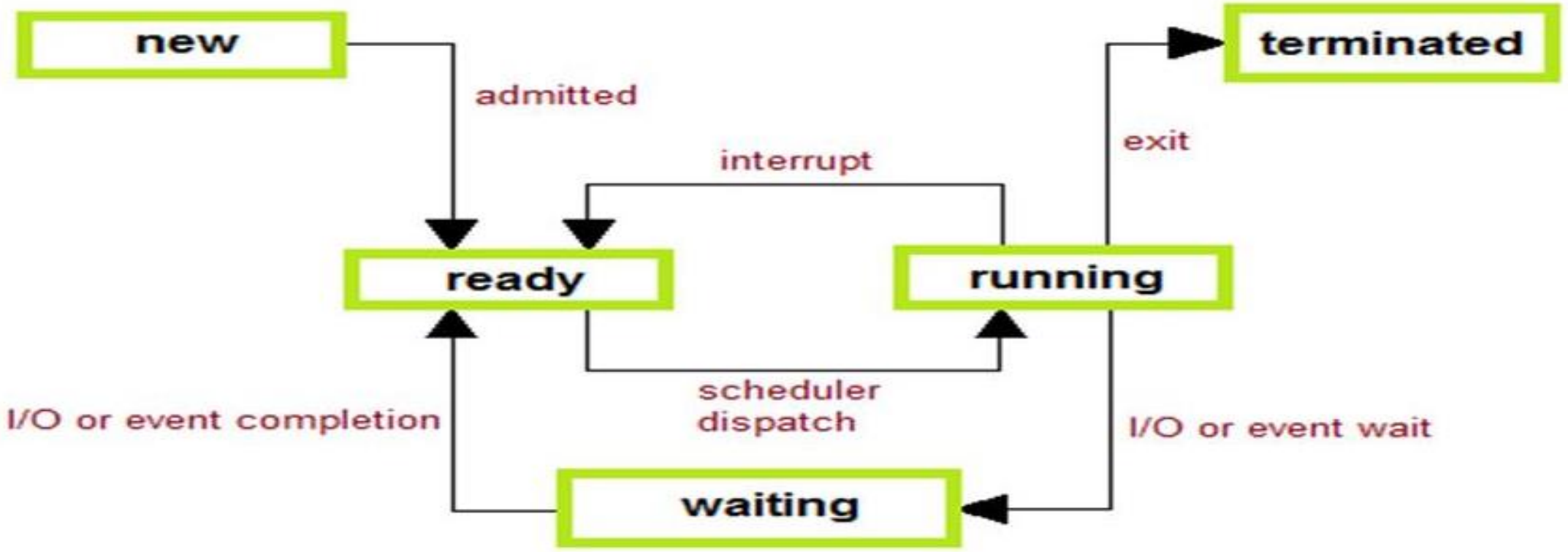


PCB

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



Process Life Cycle



Process Life Cycle

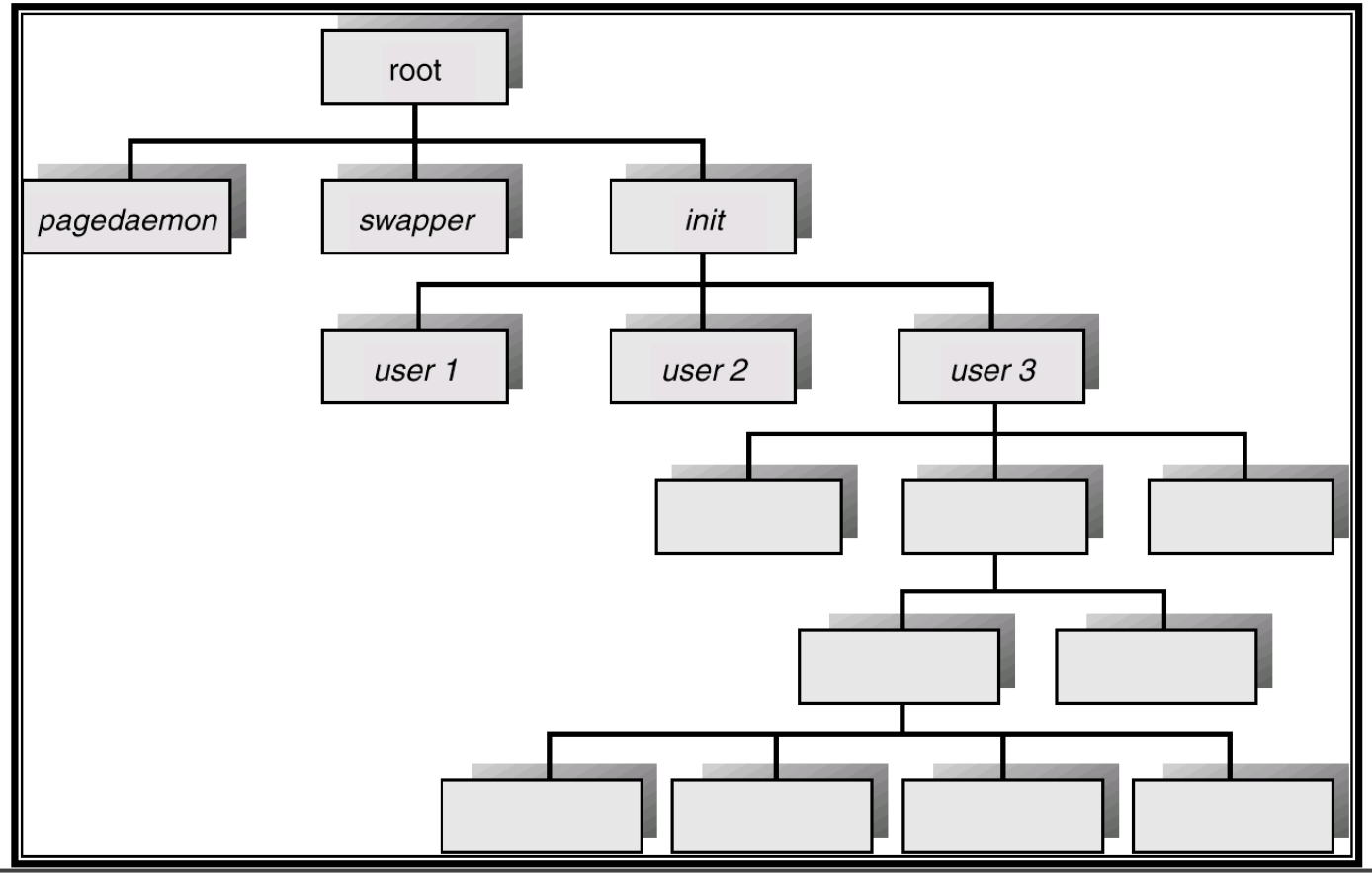
- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running** - The CPU is working on this process's instructions.
- **Waiting** - The process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

Process Identifier

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

Process Creation

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



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

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

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

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

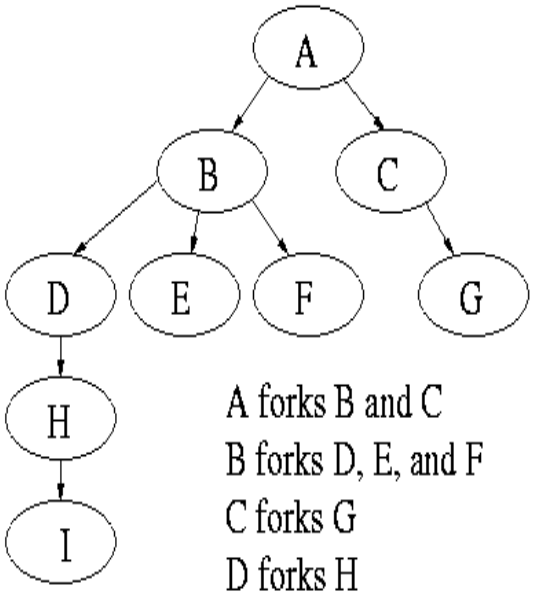
Reasons for process creation

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

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

Create a Process

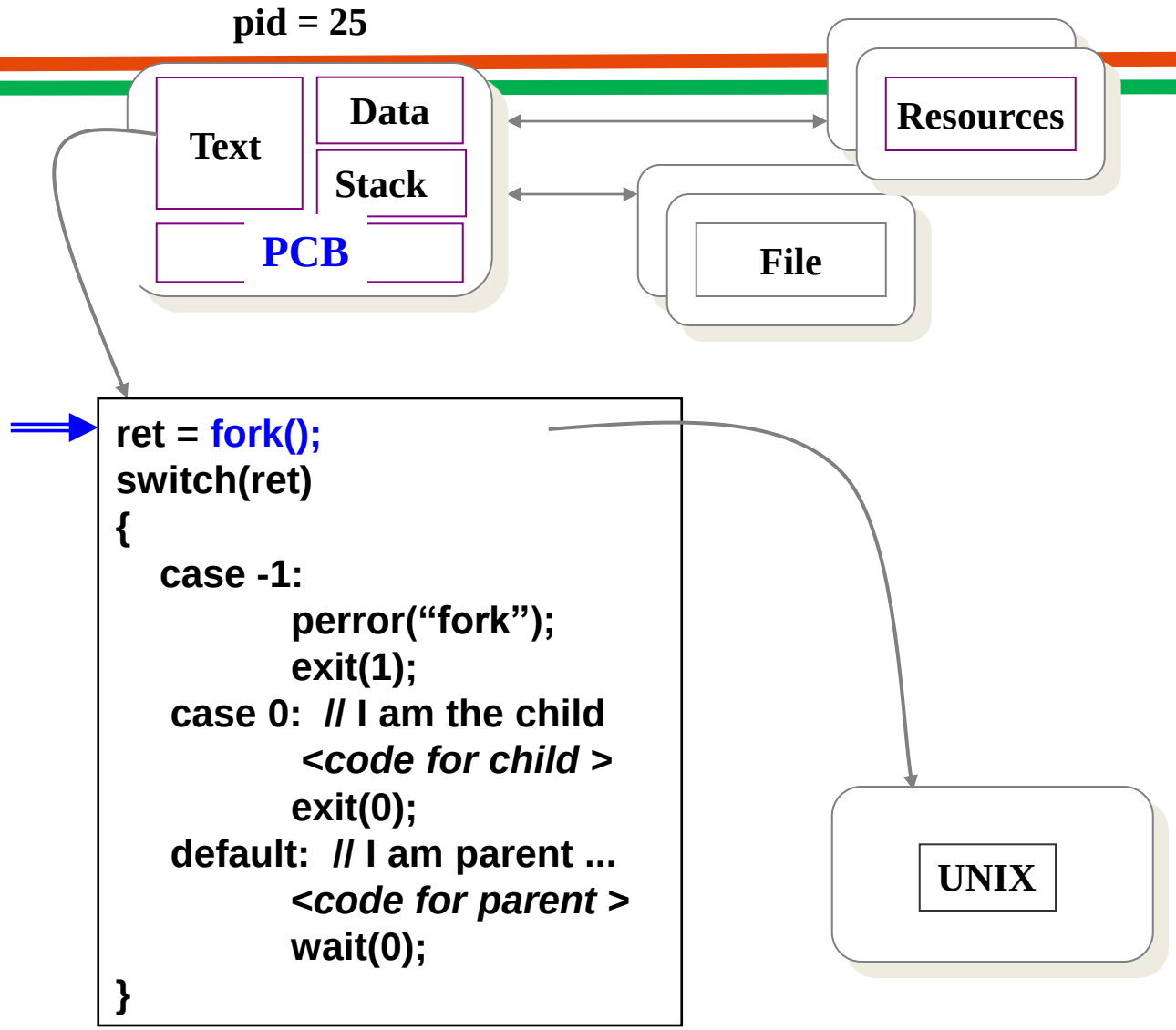
- Linux – fork system call is used to create a process
- Creates a child process by making a copy of the parent process --- an exact duplicate.
- Implicitly specifies code, registers, stack, data, files

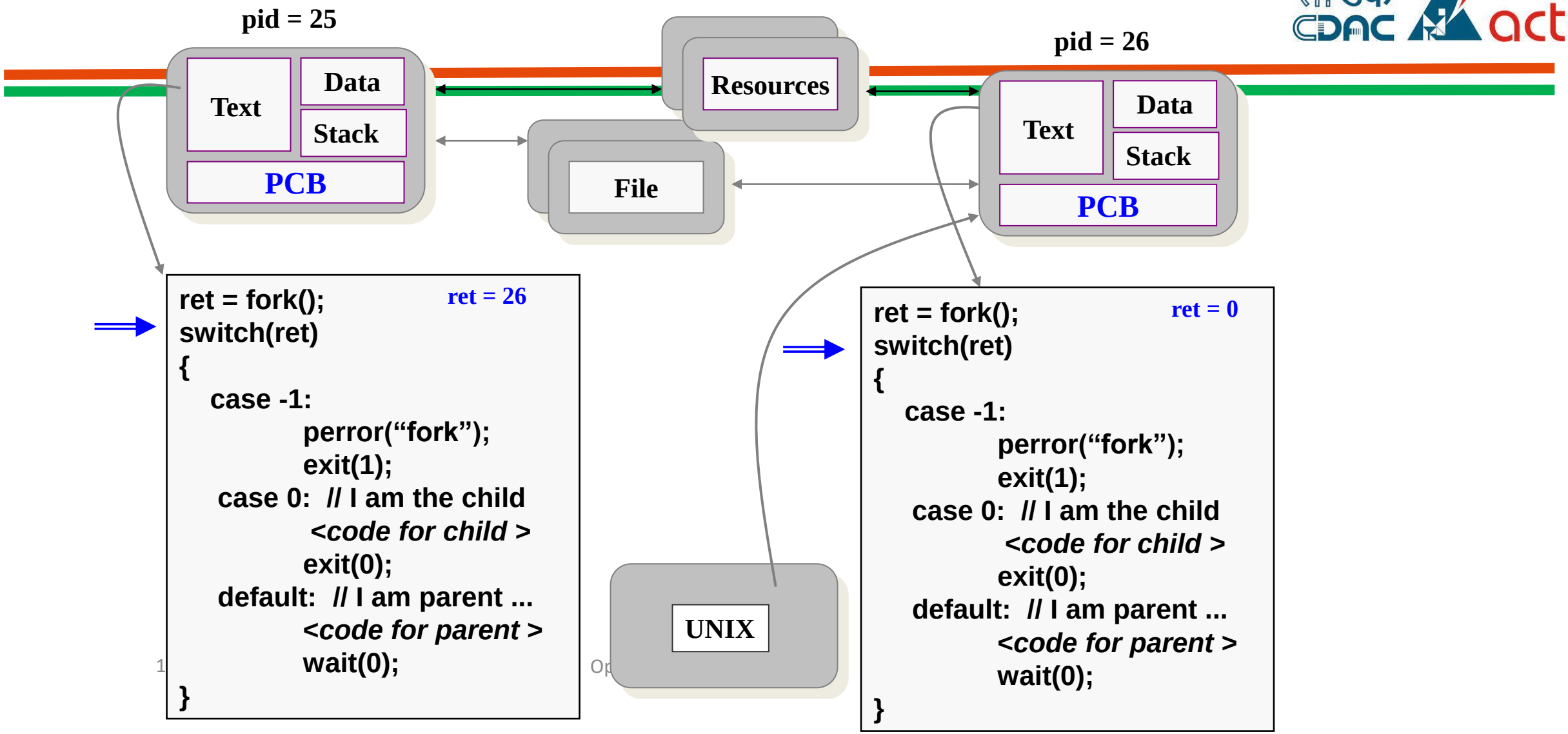


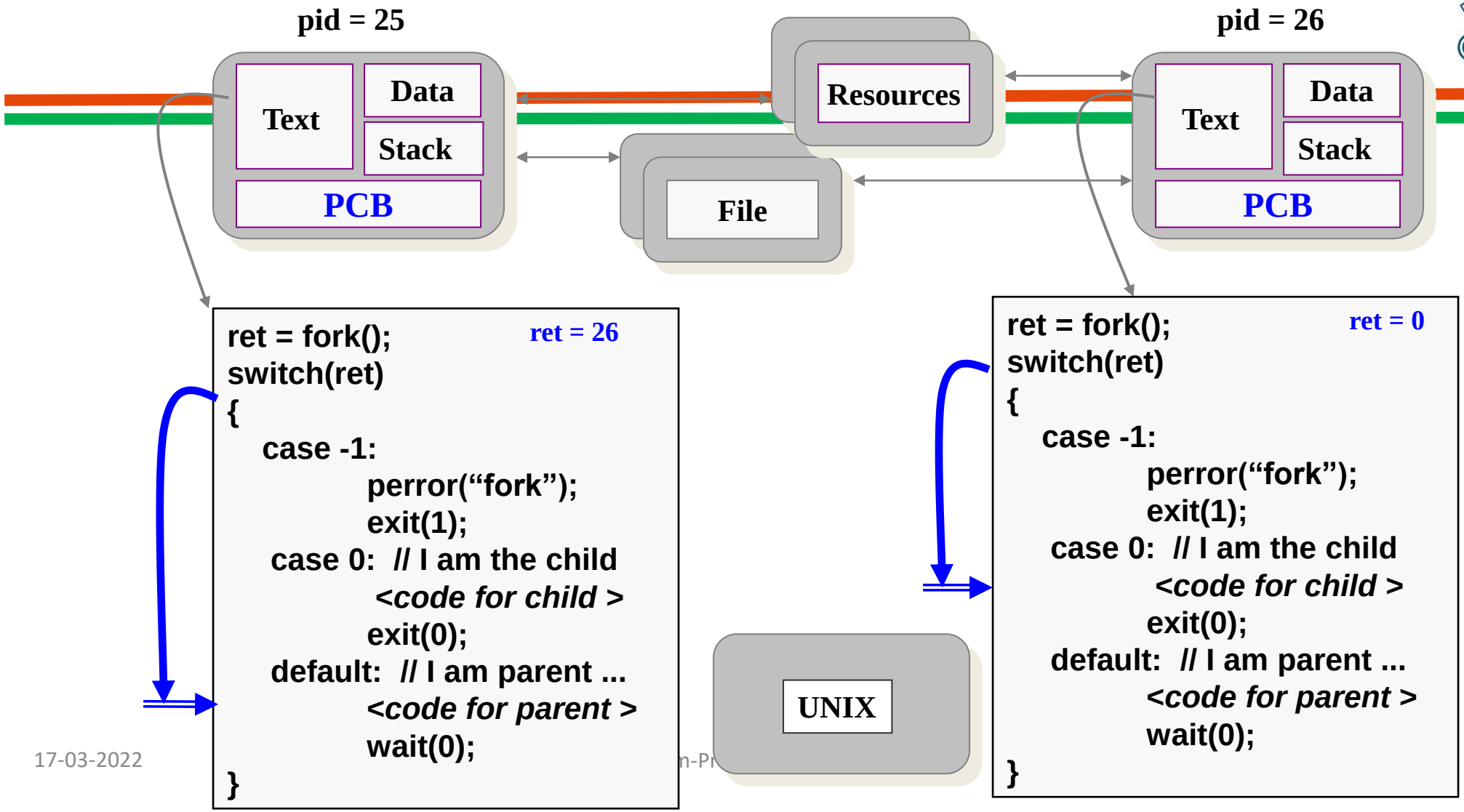
- **Fork()**

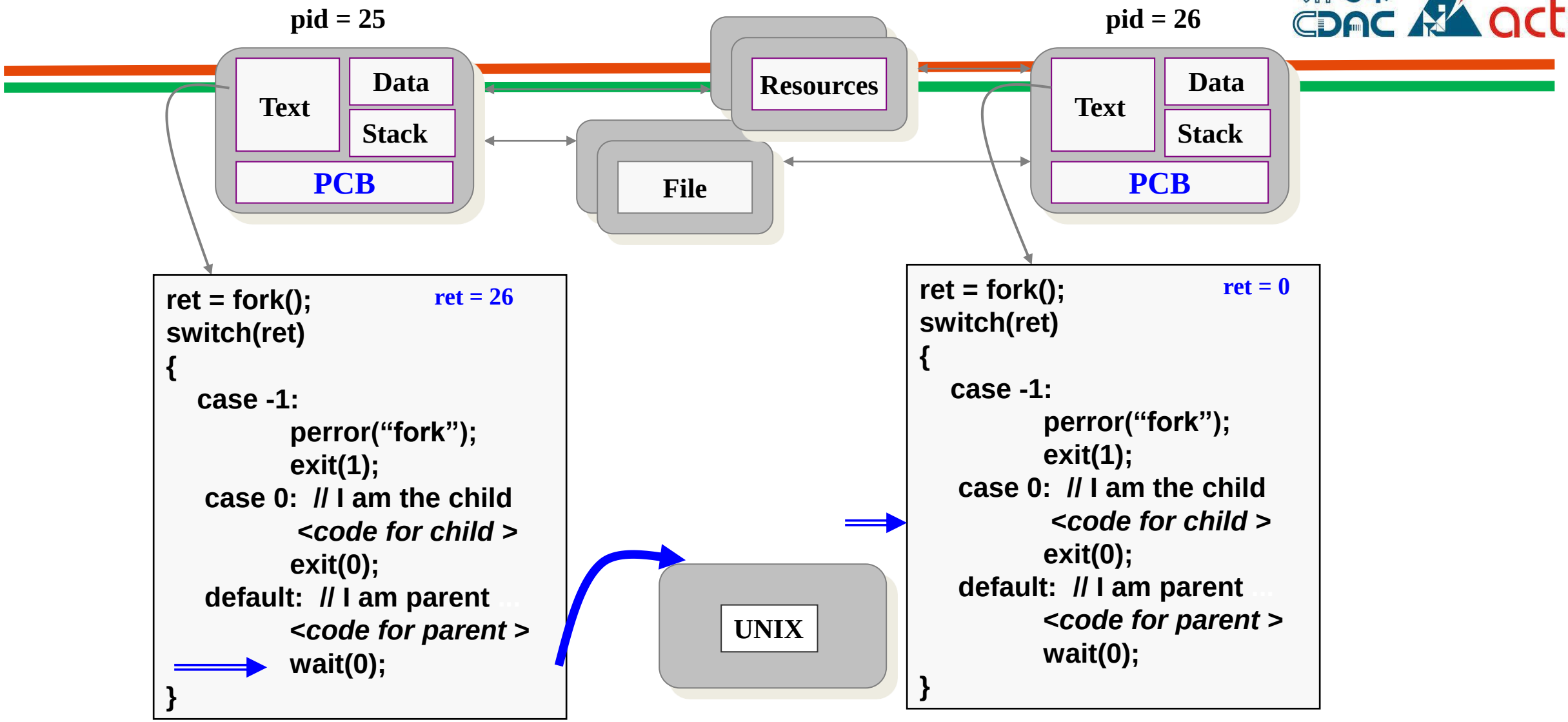
```
#include <sys/types.h>
#include <unistd.h>
pid_t fork( void );
```

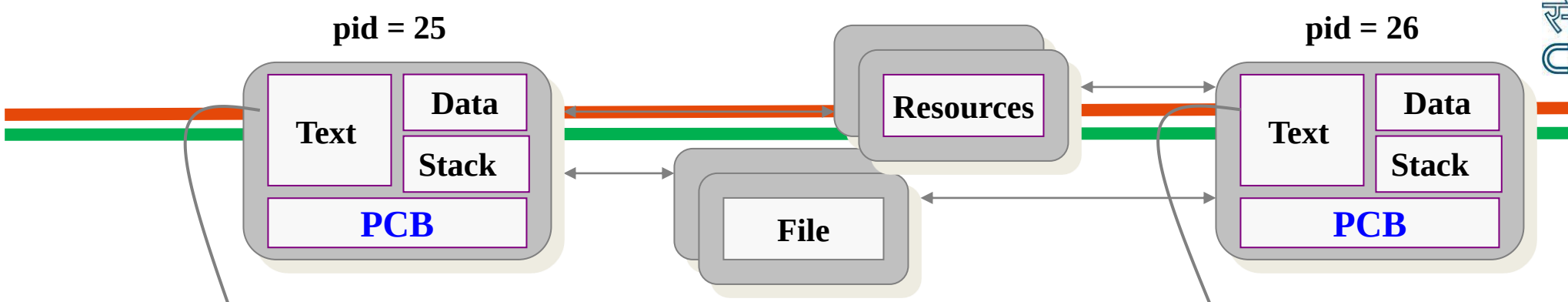
- **Both parent and child continue to execute**
  - In the child: ret == 0;
  - In the parent: ret == the process ID of the child.


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


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


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

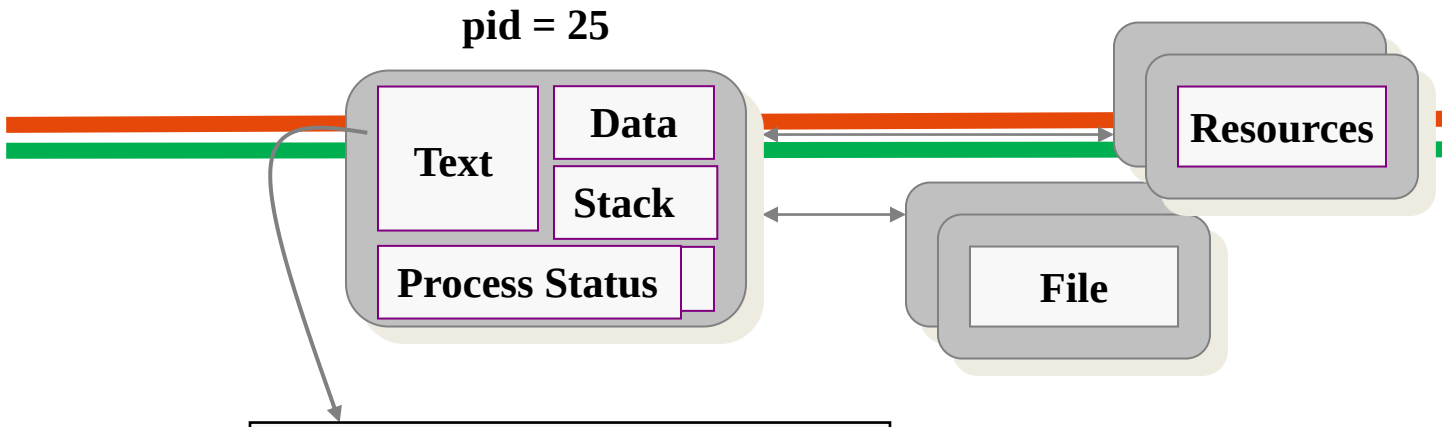

pid = 25
Text
Data
Stack
PCB
Resources
File
pid = 26
Text
Data
Stack
PCB
ret = fork();
ret = 26
switch(ret)
{
case -1:
perror("fork");
exit(1);
case 0: // I am the child
<code for child >
exit(0);
default: // I am parent
<code for parent >
wait(0);
}
UNIX
ret = fork();
ret = 0
switch(ret)
{
case -1: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




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

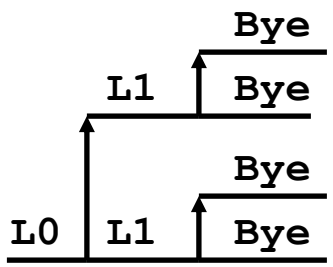
UNIX

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

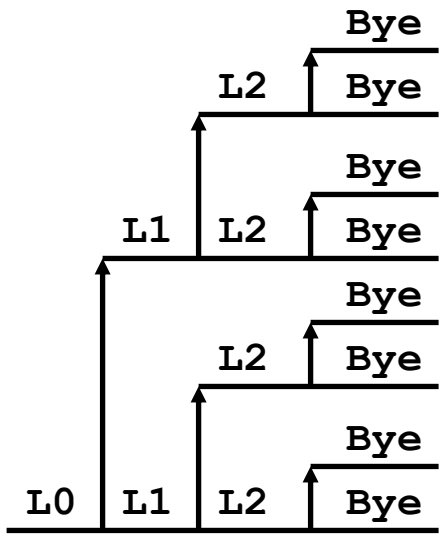



```
ret = fork();                    ret = 26
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
             <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
           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

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




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



Process Termination

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

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

Exit()

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

Wait()

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

pid_t wait(int *status);
```

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

Waitpid()

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

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

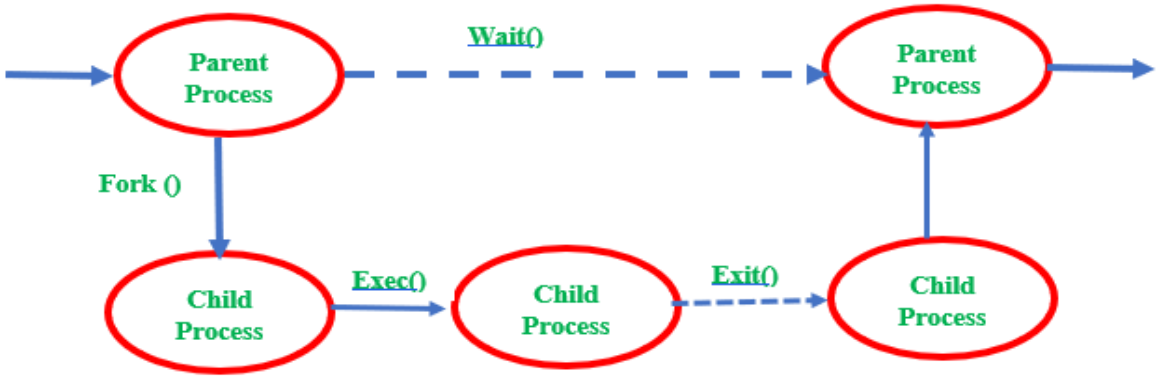
Types of Process

- Parent Process
- Child process

- Foreground process
- Background process

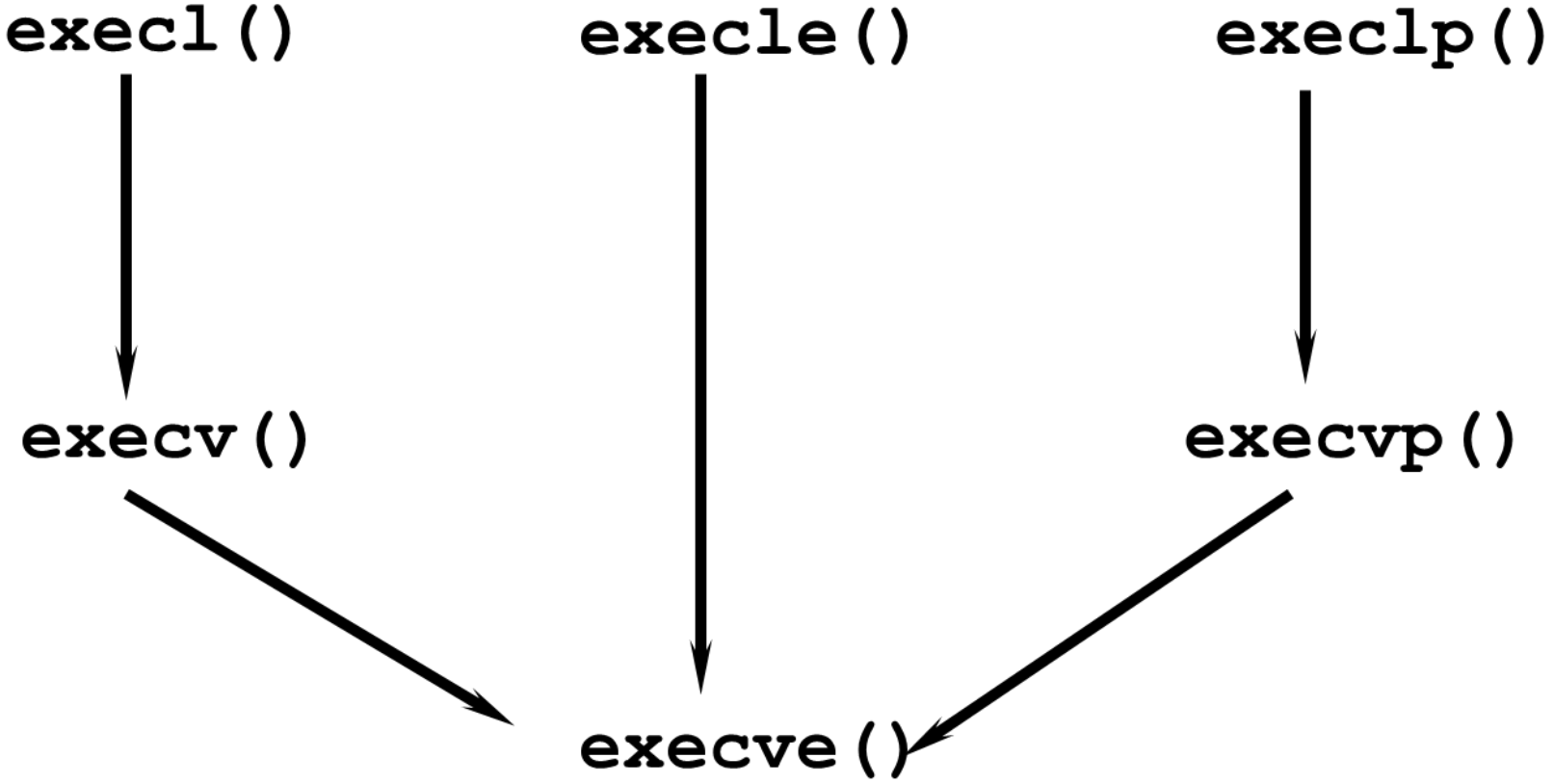
- Interactive process
- Non-interactive process

- Zombie process
- Orphan process
- Daemon Process



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

Exec..() family



Execv()

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

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

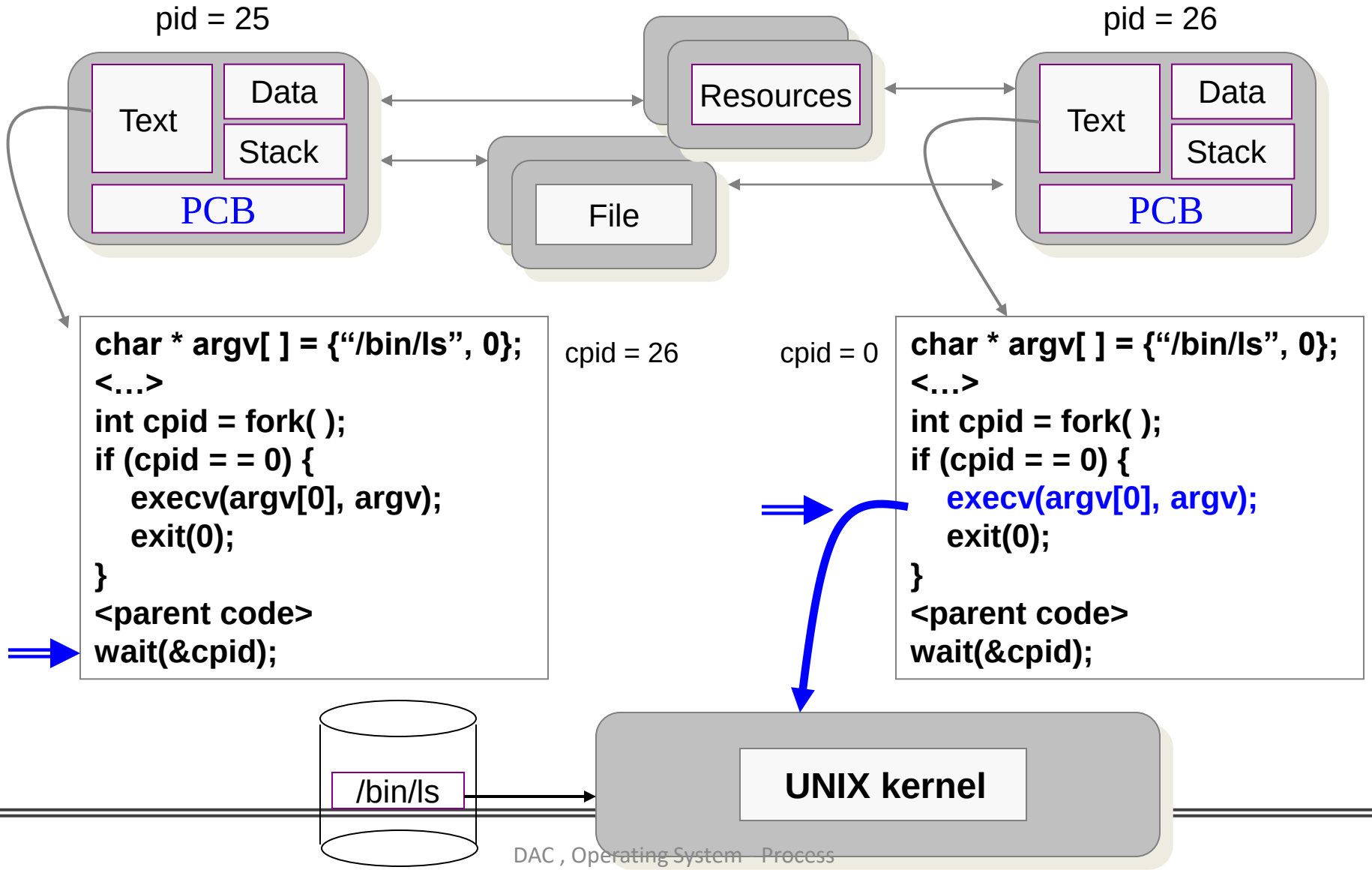
- path is the full path for the file to be executed
- argv is the array of arguments for the program to execute
  - each argument is a null-terminated string
  - the first argument is the name of the program
  - the last entry in argv is NULL
- `

execv Example

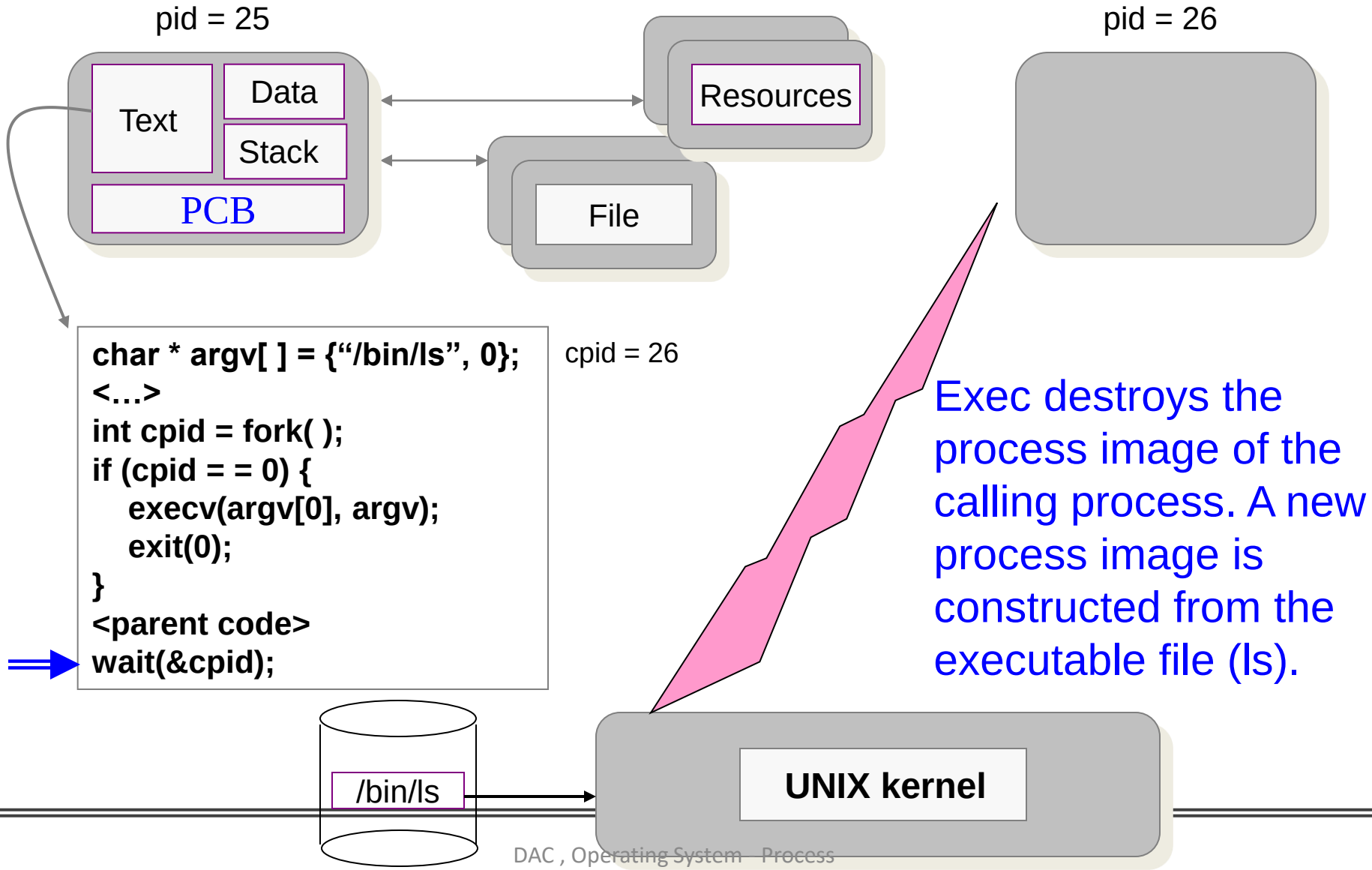
```
#include <stdio.h>
#include <unistd.h>

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

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

Note: the NULL string at the end

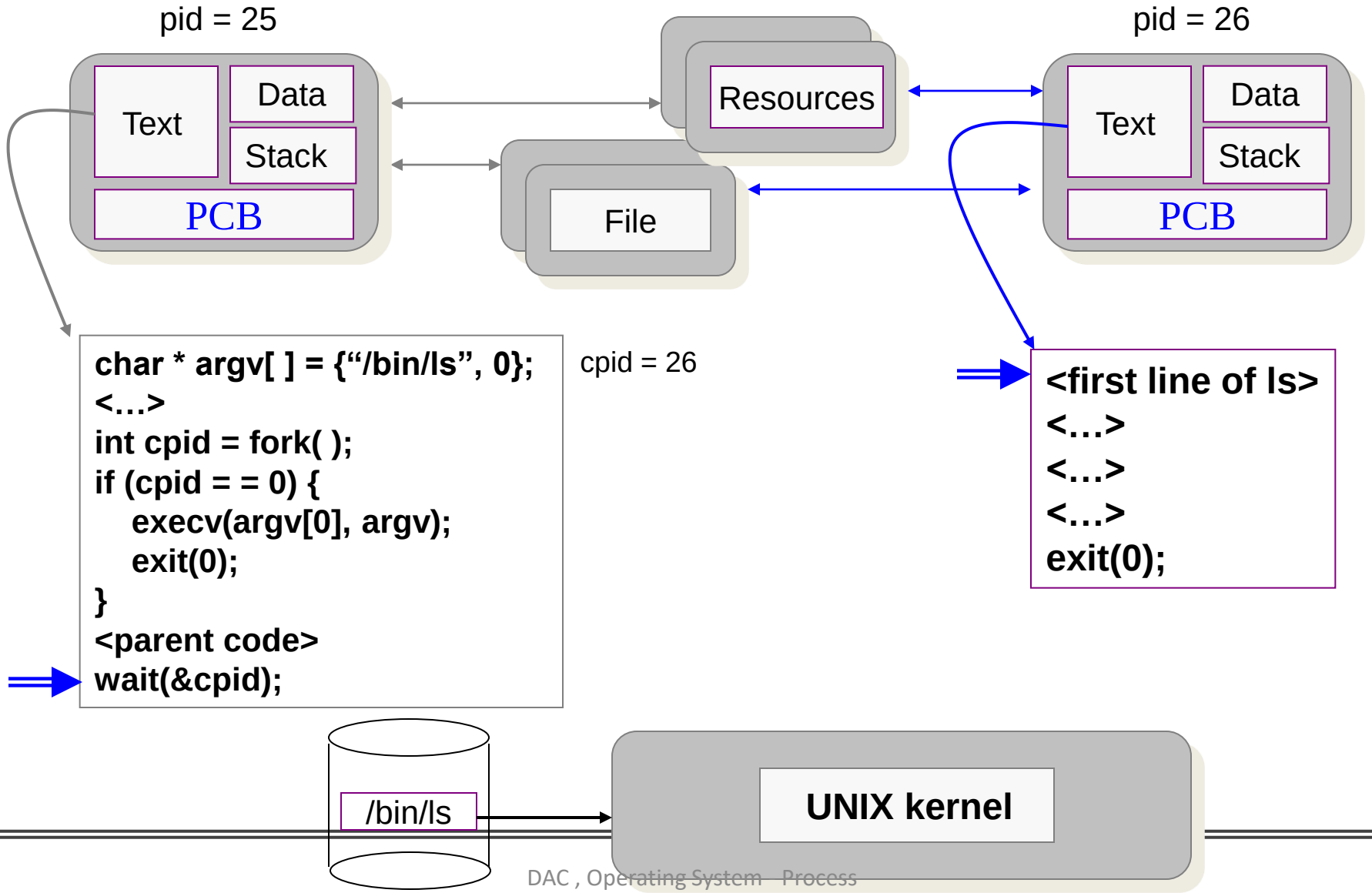
How execv Works (1)



How execv Works (2)



How execv Works (3)



execv Example

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

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

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

Note: the NULL string at the end

execl

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

- Example:

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

- Is equivalent to

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

Note the NULL string at the end

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

Variations of execv()

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

Example : execvp

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

exec variations

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

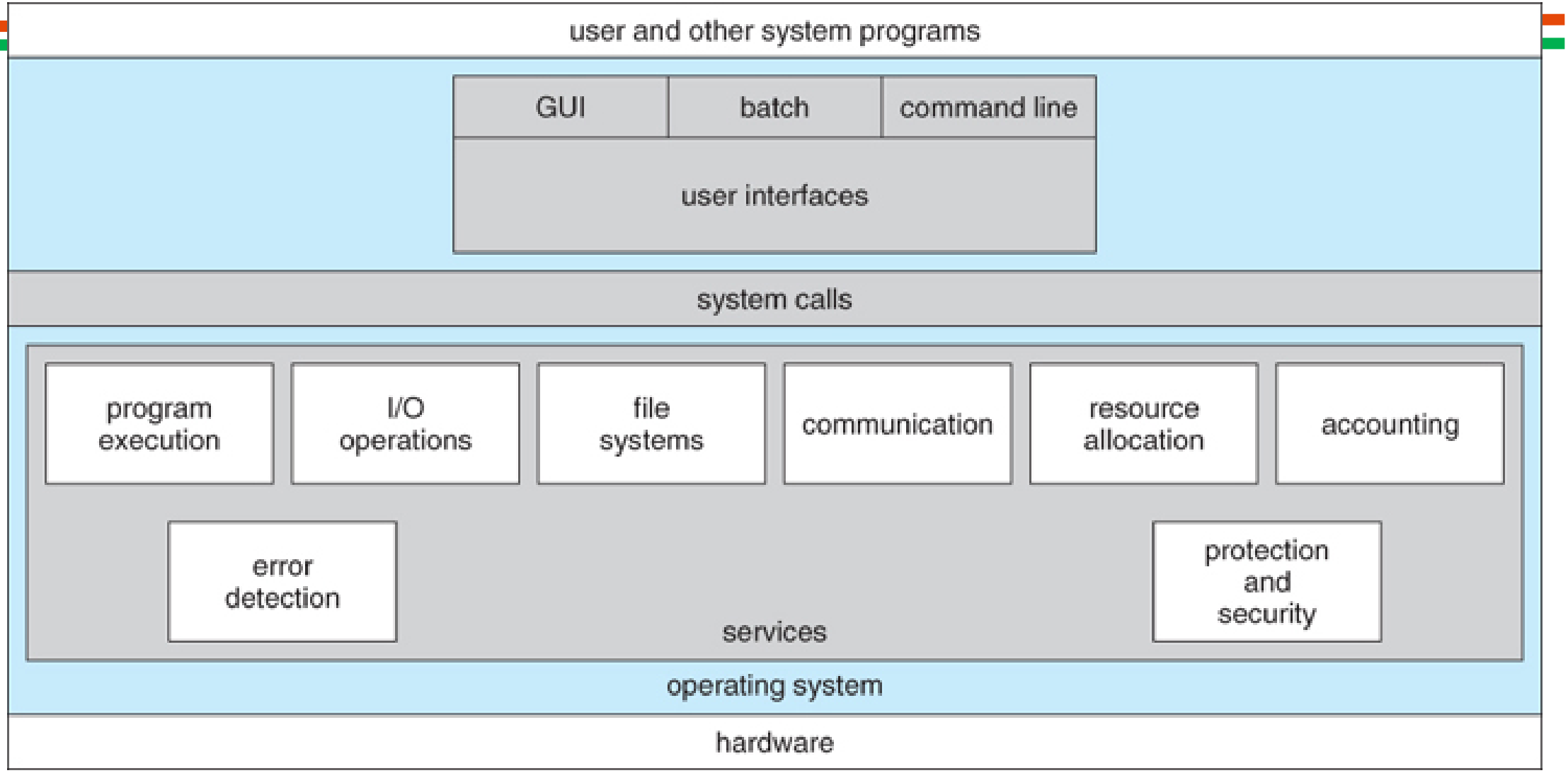
Combine fork,exec,wait

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

- **Example**

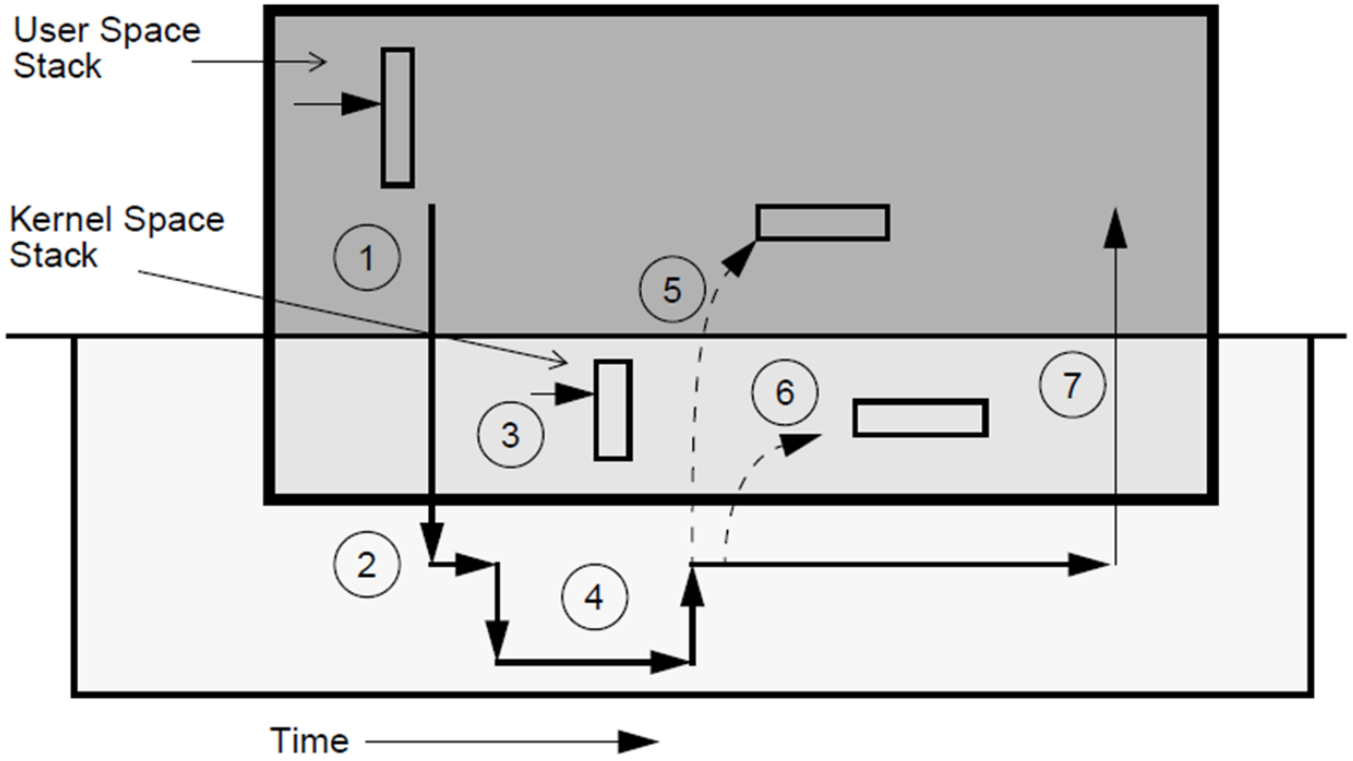
```
int main()
{
   system("echo Hello world");
}
```

Any Questions?


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

System Call

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



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

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

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

System API

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

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

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

Thank You