Operating system principle

Processes

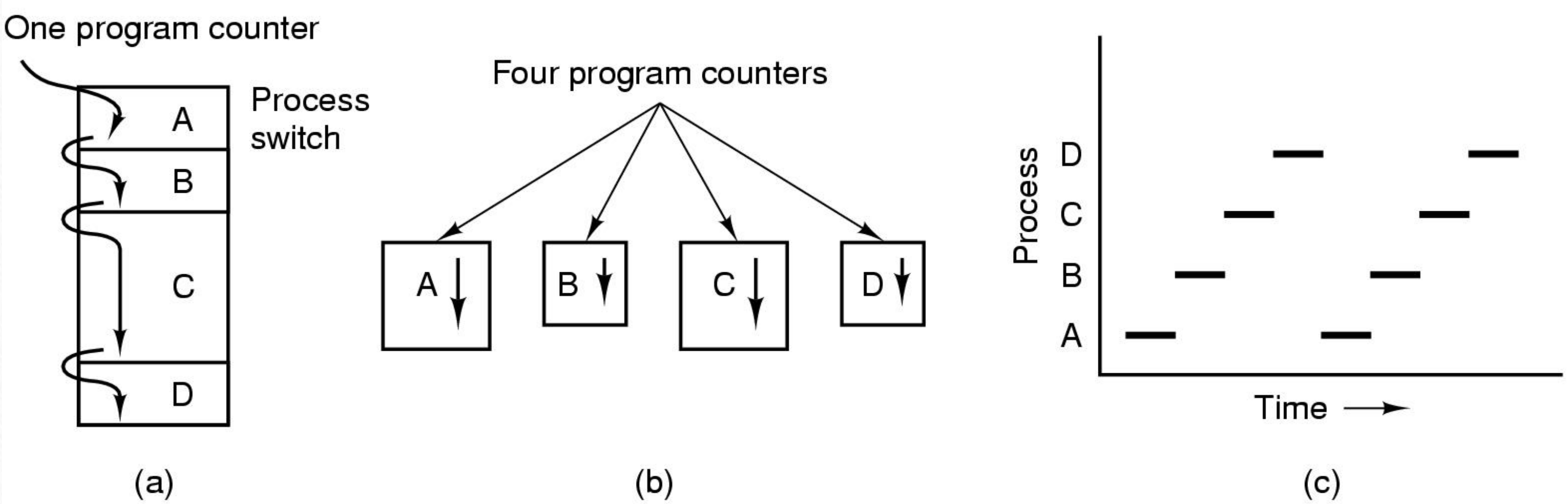
Unit Objectives

- After completing this unit, you should be able to:
  - Discuss what is process.
  - Discuss the difference between the program and the process.
  - Discuss the process hierarchies.
  - Discuss the process lifecycle.
  - Discuss the process states.
  - Use the system calls to control the process.

Process vs program

- What's program?
  - Program is just the static text (code).
  - Program can be loaded into memory and executed, called process.
  - A program can be executed many times at once.
- What's process?
  - A process is an instance of a computer program that is being executed.
  - It contains the program code and its current activity.

The Process Model



(a) Multiprogramming of four programs.

(b) Conceptual model of four independent, sequential processes.

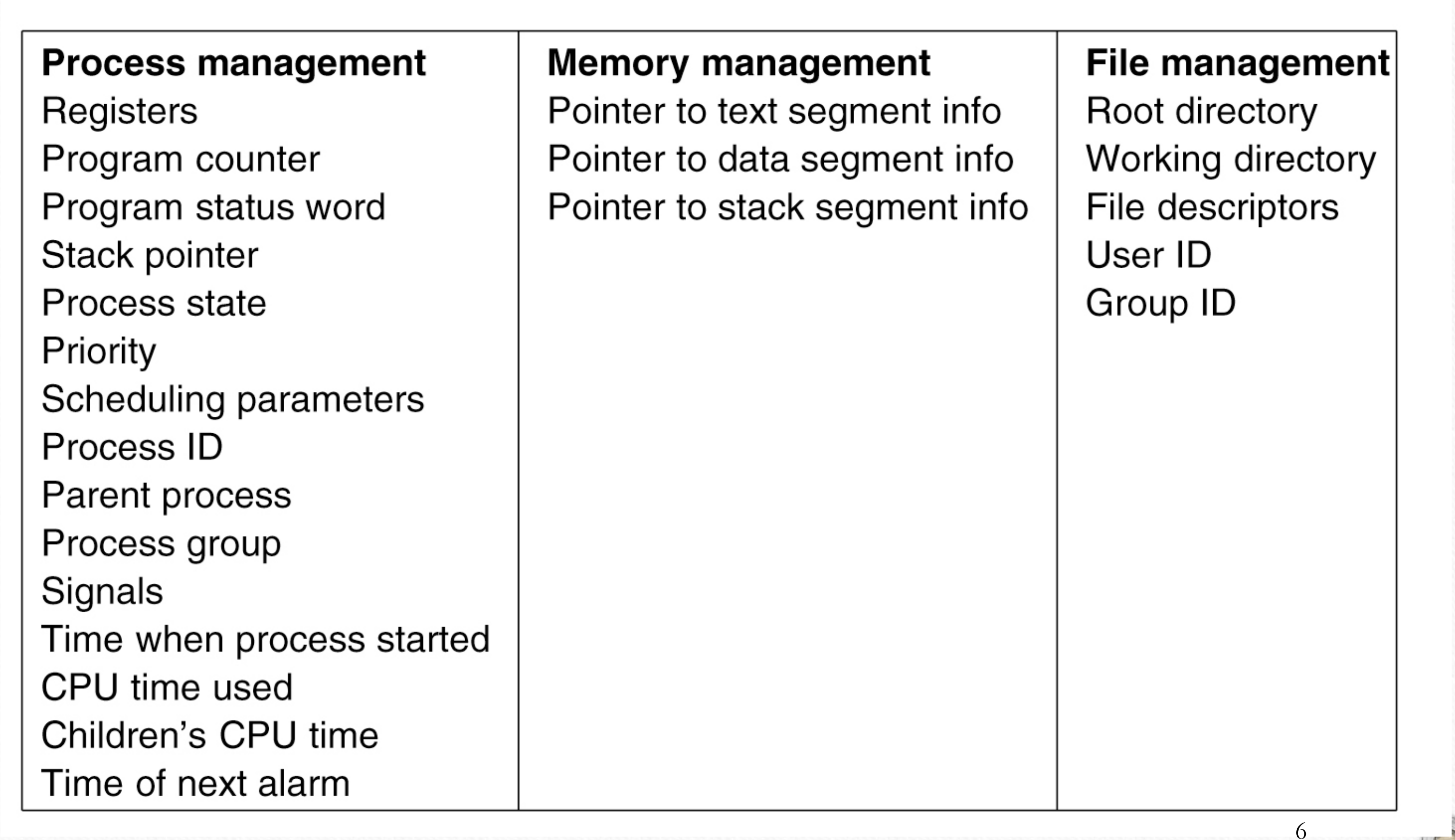
(c) Only one program is active at once.

Multiprogramming

- The modern operating systems have the ability to execute many processes concurrently.
- Concurrently?
- In case of one CPU with one core, processes can only be executed sequentially. Only one process is active at once.
- On a macroscale the processes can be considered to executed concurrently.
- At the microscopic level, they are executed sequentially.

A Process consists of three parts

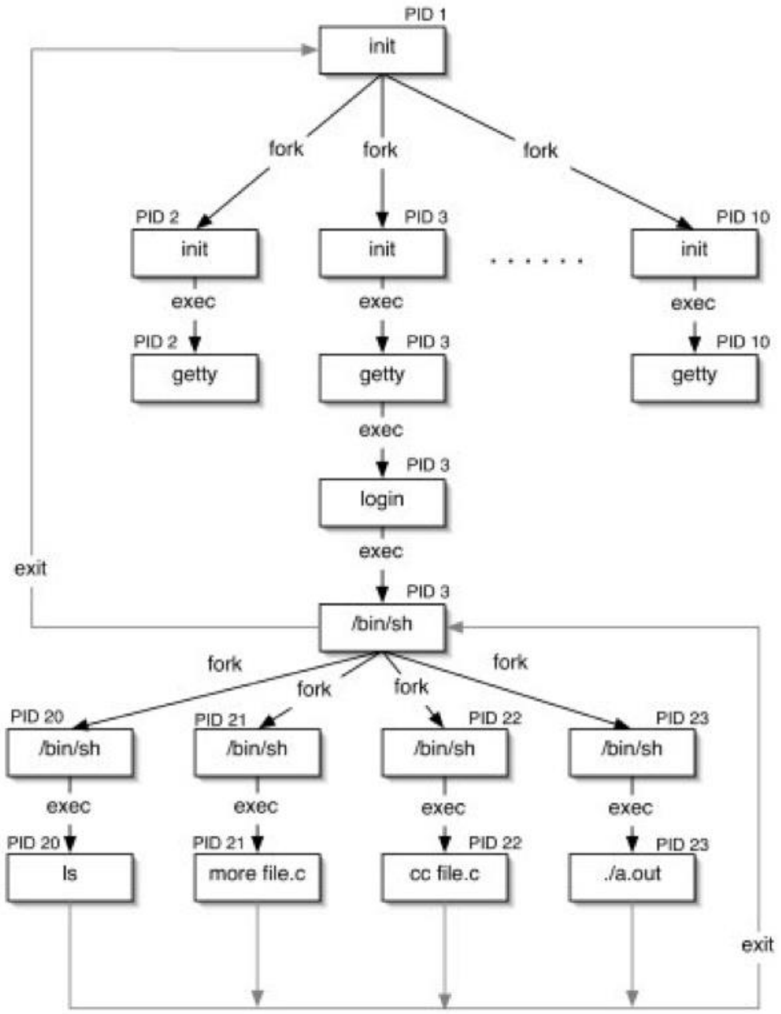
- Process Control Block (Process Table Entry)
- Program (Text)
- Data



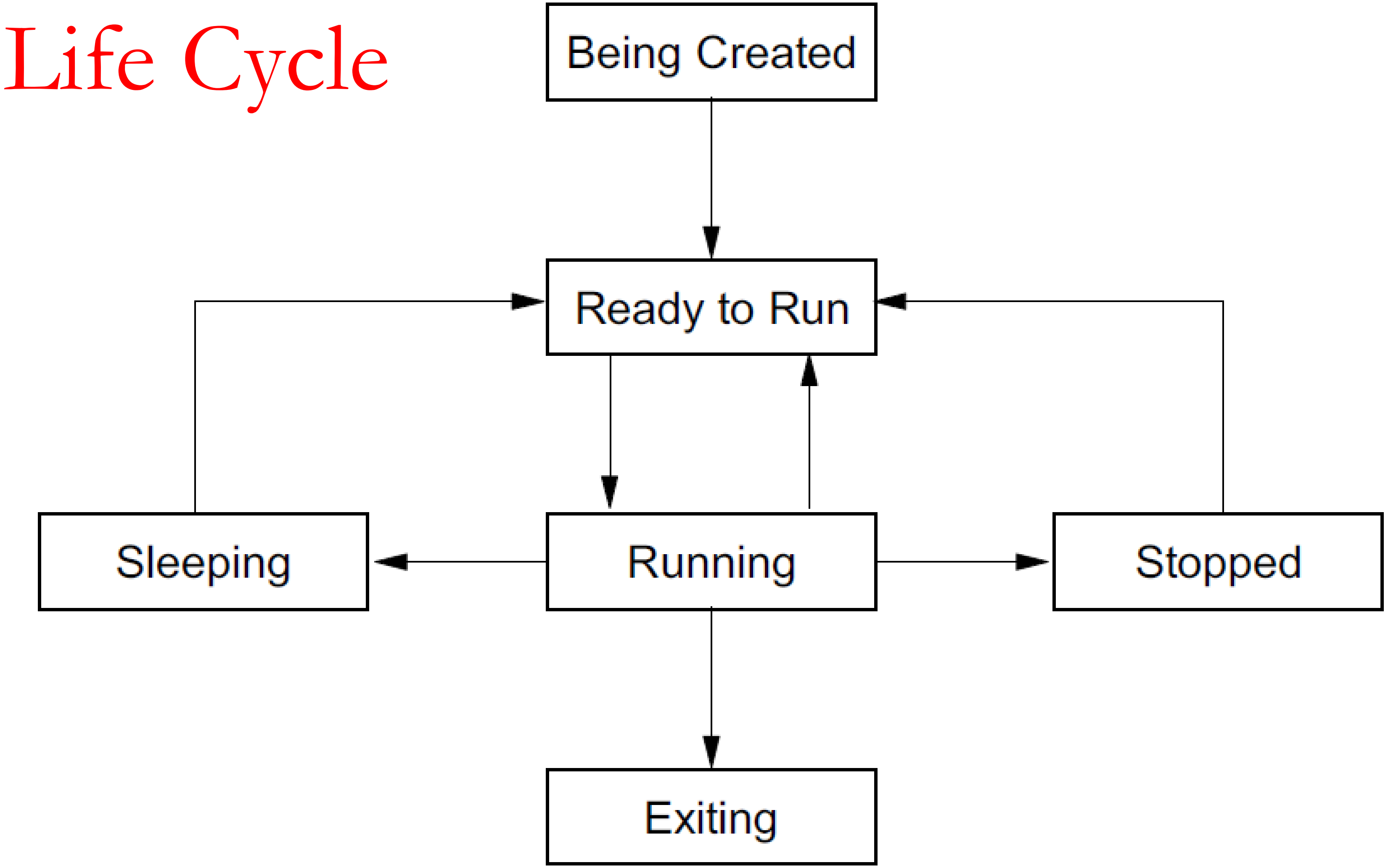
Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children’s CPU time		
Time of next alarm		

Process Hierarchy

- In UNIX, all processes are started by other processes.
- This is called a parent/child relationship.
- The first process, called "init", is the exception. It's created by the kernel when system boot.
- Processes forms a hierarchy, called "process tree".
- Windows has no concept of process hierarchy.



Process Life Cycle



Being Created

Ready to Run

Sleeping

Running

Stopped

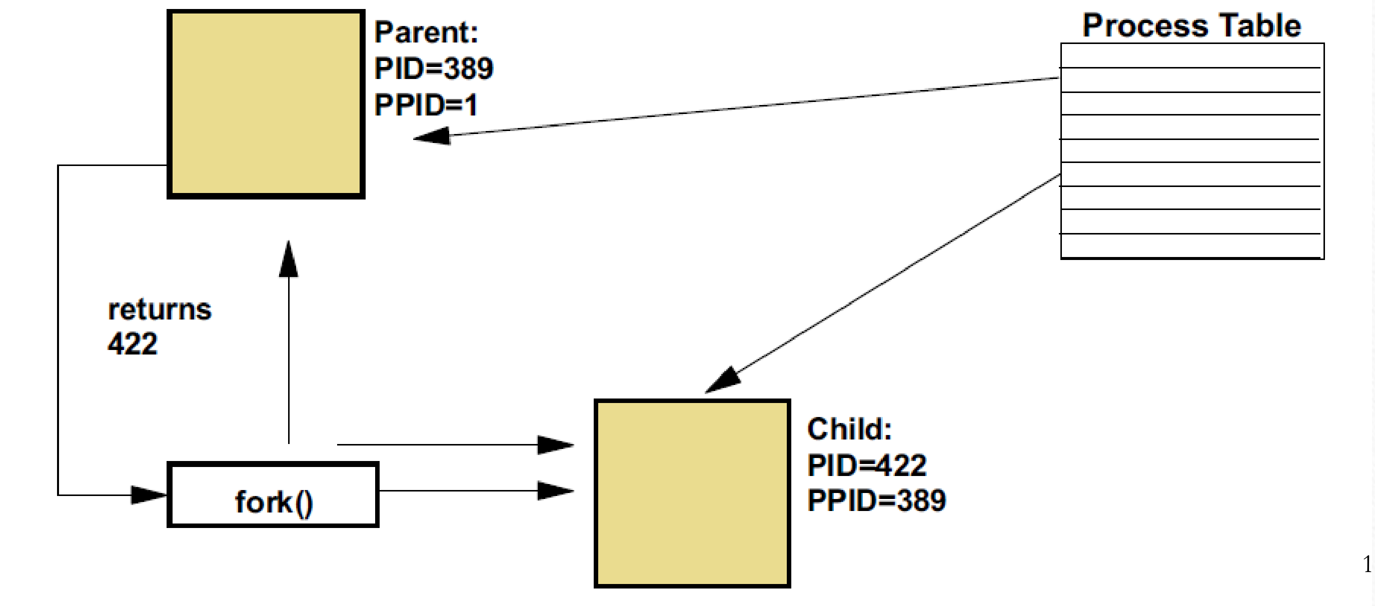
Exiting

Process Creation

- Events which cause process creation:
  - 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.

Creating a Process—fork()

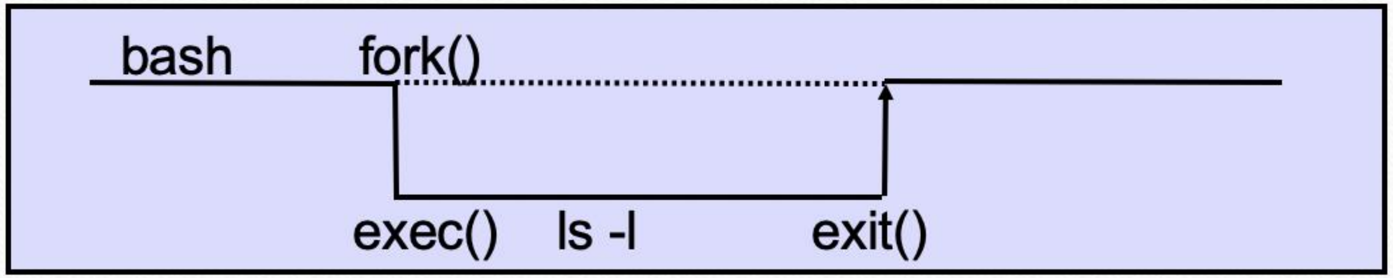
- Creates a new single-threaded process.
- The new process is a child to the calling process
- The child process is almost identical to its parent (calling process).
- The child inherits many properties from the parent.
- Include unistd.h



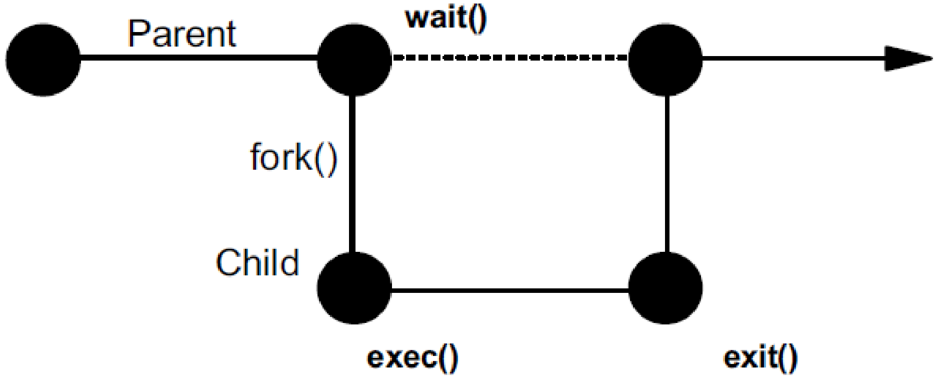
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    pid_t val;
    printf("PID before fork(): %d \n",(int)getpid());
    val=fork();
    if ( val > 0) {
            printf("Parent PID: %d\n",(int)getpid());
    } else if (val == 0) {
            printf("Child PID: %d\n",(int)getpid());
    } else {
            printf(“Fork failed!”);
            exit(1);
    }
}
```

fork()

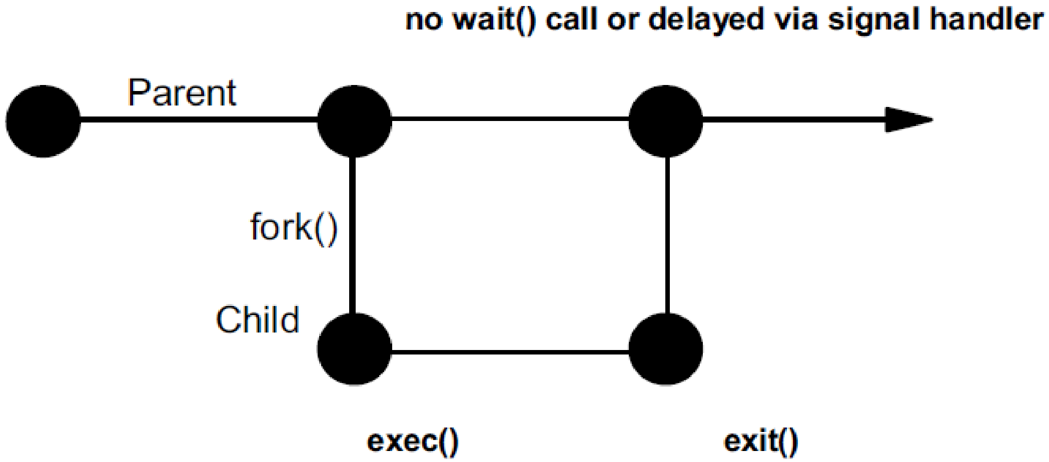
```
pid_t val;
int exit_code = 0;
val=fork();
if (val > 0) {
        int stat_val;
        pid_t child_pid;
        child_pid = waitpid(val,&stat_val,0);
        printf("Child has finished: PID = %d\n", child_pid);
        if (WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
        exit(exit_code);
} else if (val == 0)    {
        execlp("ls","ls","-l",NULL);
}
```



Example of foreground process:



Example of background process:



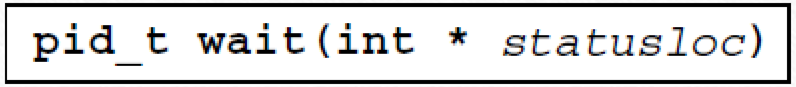
The exec System Calls

- Various P0SIX calls are used to replace the text of a calling process with text from another program - used with fork( ).
- The call used depends on requirements and preferences:

  execl() execle() execlp()

  execv() execve() execvp()
- The "**l**" family passes arguments to the new executable via a list, while the "**v**" family passes arguments via pointers.
- The "**e**" family (ending with "e") includes the ability to pass environmental variables.
- The "**p**" family allows the use of PATH in searching for the file and the program maybe a shell script instead of an executable.
  - **execl("/usr/bin/ls", "ls", "-l ", "/home/kelly/", NULL);**
  - **execlp("ls", "ls", "-l", "/home/kelly/", NULL);**

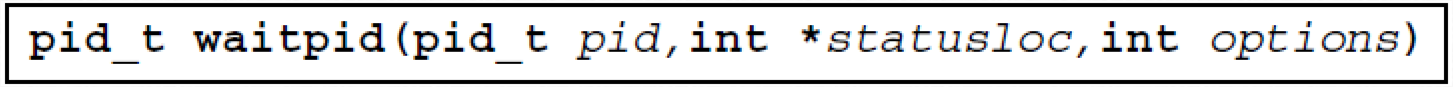
The wait() System Call



```
pid_t wait(int * statusloc)
```

- Called by the parent process to wait for the termination of the child process (responds to SIGCHLD).
- statusloc is a pointer to an int, where the exit status of child process can be stored and queried.
- Returns pid of a terminated child process.
- Clears the Process Table entry for the child process (removes defunct process). Included <sys/wait.h> for use of options described below.

The waitpid() System Call



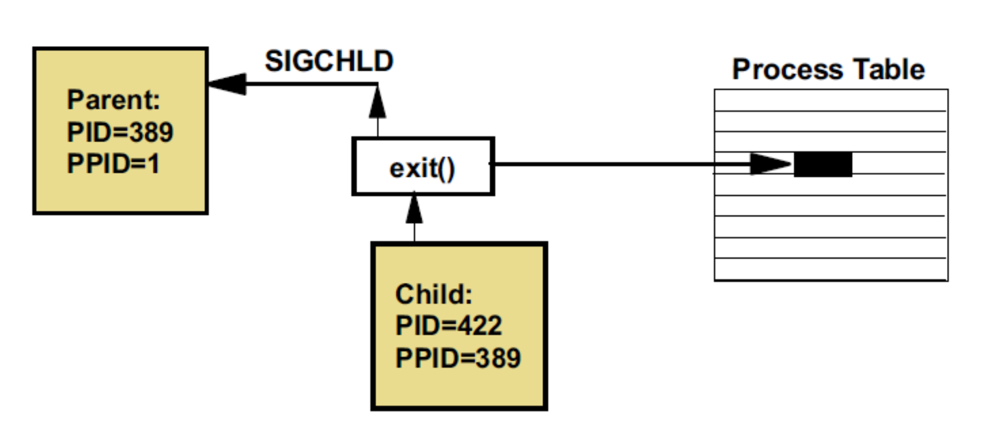
```
pid_t waitpid(pid_t pid,int *statusloc,int options)
```

- Allows specification of the pid for which to wait.
- Options for waitpid() call include:
  - WNOHANG - Calling process does not wait if there are no terminated child processes.
  - WUNTRACE - Returns information about a child process stopped by SIGTTIN, SIGTTOU, SIGSSTP, and SIGTSTOP signals.

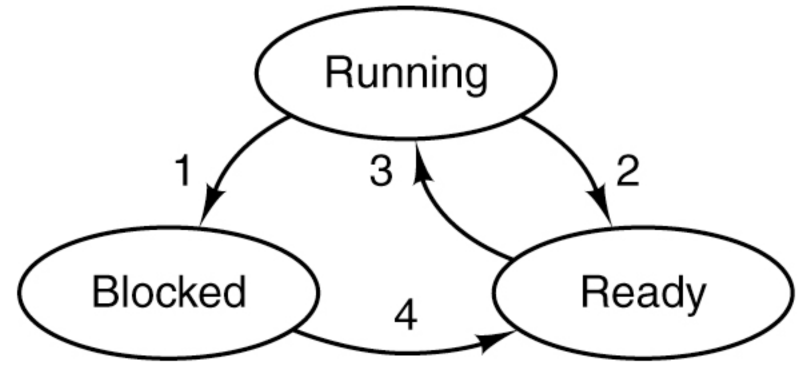
The exit() System Call

- Called by the terminating process:
  - Explicitly (that is, when error detected)
  - Implicitly, when closing brace of main() reached
  - Implicitly, when process terminated by signal
- Status is the exit status returned to the parent process. exit() sends a SIGCHLD SIGNAL (20) to the parent process.

exit(status)



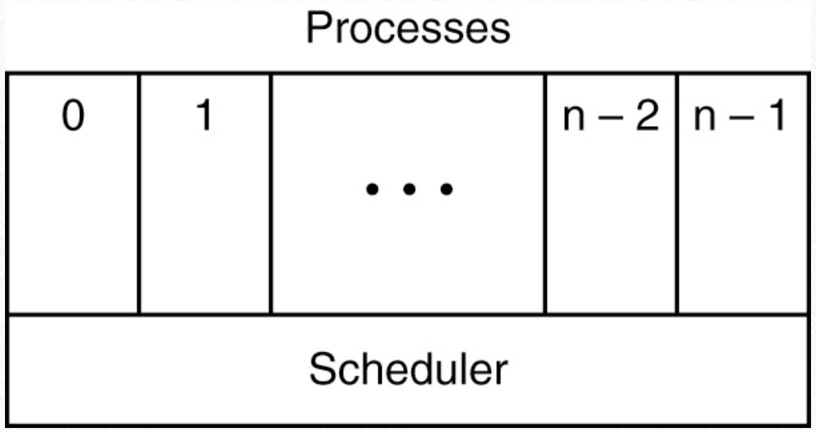
Process states



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Running: runnable , has processor
- Ready: runnable but no processor available
- Blocked: waiting for something, so has to sleep
- 1: The process blocks for resource or event.
- 2: Time's up and the scheduler picks another runnable process.
- 3: The scheduler picks a runnable process to run.
- 4: The blocked processes is waked up by the other process.

Scheduling



- Both running and ready are called runnable.
- All runnable processes are in a queue.
- The scheduler (in the kernel) uses the scheduling algorithm to pick up a runnable process to run and a special pointer to point to it.

Process termination

- A process can be terminated for the following reasons.
  - The process terminates itself when done.
  - The process terminates voluntarily with error exit.
  - The process terminates involuntarily with fatal exit.
  - The process is terminated by a signal from another process.
- When a process terminates itself, it falls into "zombie" state.

Deal with zombie

```
#include <unistd.h>
void handle_it();
int status;
main()
{
    signal(SIGCHLD,handle_it);
    if(fork()==0) {    /*We are the child... */
        execl("/usr/bin/ls", "ls",NULL);
        perror("exec failed");
        exit(1);
    }
    while(1);
}
```

```
void handle_it(){
    int rv=1;/* used to track return from waitpid */
    signal(SIGCHLD,handle_it);    /*   reregister SH */
    while (rv >0 )/* waitpid returns zero if no zombies */
        rv = waitpid(0,0,WNOHANG);
}
```

Unit summary

- Having completed this unit, you should be able to:
  - Discuss what is process.
  - Discuss the difference between the program and the process.
  - Discuss the process hierarchies.
  - Discuss the process lifecycle.
  - Discuss the process states.
  - Use the system calls to control the process.

References

- Chapter 2: Processes and threads, Modern Operating Systems . Forth Edition, Andrew S. Tanenbaum
- Unit 9.Working with processes, , Linux Basic and Installation , IBM, ERC 7.0
- Unit 8. AIX Process Management Systems Calls, AIX 5L Application Programming Environment , IBM, ERC 3.0
- Chapter 8: Controlling processes. Advanced Programming in the UNIX Environment, Third Edition. W. Richard Stevens , Stephen A. Rago