KERNEL MASTERS

Linux System Programming

Process Management

Authored and Compiled By: **Boddu Kishore Kumar**

Email: kishore@kernelmasters.org

Reach us online: www.kernelmasters.org

Contact: 9949062828

Important Notice

This courseware is both the product of the author and of freely available open source materials. Wherever external material has been shown, its source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of this courseware cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - source code and binaries (where applicable) - that form part of this courseware, and that are present on the participant CD, are released under the GNU GPL v2 license and can therefore be used subject to terms of the afore-mentioned license. If you do use any of them, in any manner, you will also be required to clearly attribute their original source (author of this courseware and/or other copyright/trademark holders).

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant CD are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

1.1. Introduction to Process Management:

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A word-processing program being run by an individual user on a PC is a process. A system task, such as sending output to a printer, can also be a process (or at least part of one). For now, you can consider a process to be a job or a time-shared program, but later you will learn that the concept is more general. It is possible to provide system calls that allow processes to create sub processes to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process whose function is to display the status of a file on the screen of a terminal. The process will be given as an input the name of the file and will execute the appropriate instructions and system calls to obtain and display on the terminal the desired information. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process; a program is a passive entity, such as the contents of a file stored on disk, whereas a process is an active entity. A single-threaded process has one program counter specifying the next instruction to execute. The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently— by multiplexing the CPU among them on a single CPU, for example.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization.
- Providing mechanisms for process communication.
- Providing mechanisms for deadlock handling

What is a Process?

- Process is a program in execution.
- Process is a instance of program.
- Program by itself is not a process.

Program	Process
In secondary memory. e.g.: Hard disk	In Main memory. e.g.: ROM,RAM
Without resources.	With resources like memory, I/O resource and processing resource.
Passive entity. (No dynamic behaviour)	Active Entity (Alive) – (Dynamic behaviour)

1.2. Process in Memory:

A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 1.1.

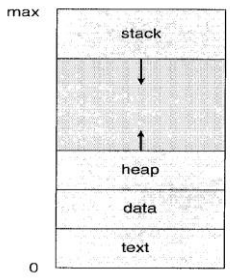


Fig 1.1: Process in Memory

A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)

Same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

1.3. Process State:

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

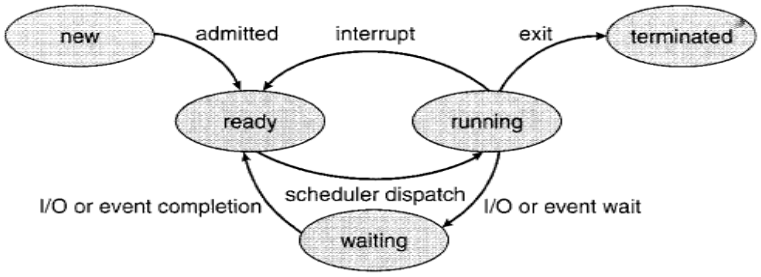


Fig 1.2: Diagram of process state.

1.4. Process control block:

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB is shown in Figure. It contains many pieces of information associated with a specific process, including these:

- **Process state.** The state may be new, ready, running, and waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information.** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information.** This information includes the amount of CPU and real time used, time limits, account members, job or process numbers, and so on.
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

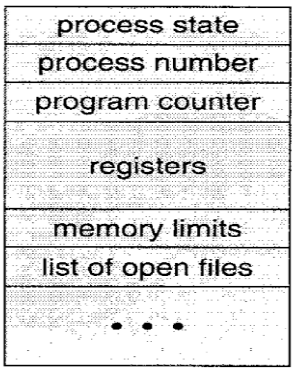
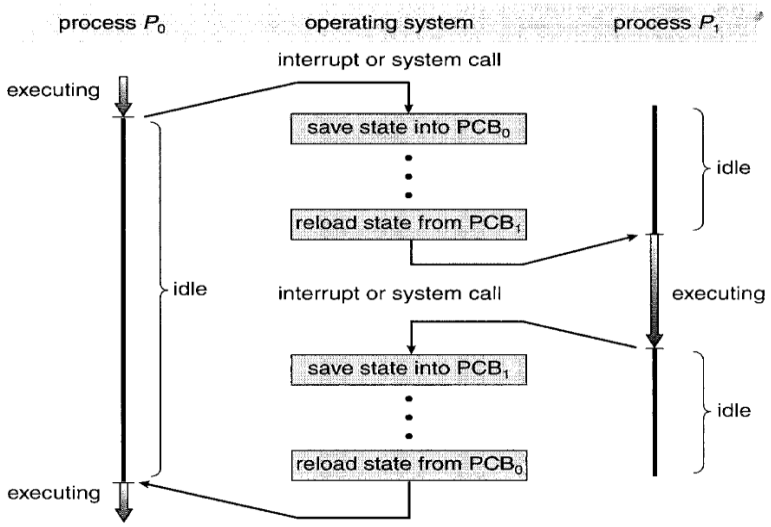


Fig 1.3: Process control Block



Process Management Libraries:

System() library to execute a command in shell.

Process Management System Calls:

getpid() , getppid() : shows current pid & parents pid numbers.

execve(): Replacing a process image.

fork(): Duplicating a process image.

exit(): process termination.

Wait(): wait for children process exit status.

Starting a new process:

Use system() library to execute a command in shell.

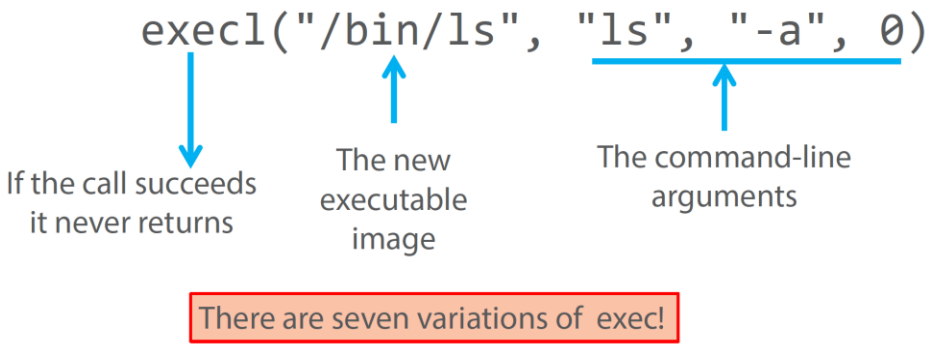
```
#include <stdlib.h>
int system(const char *command);
```

system() executes a command specified in command by calling **/bin/sh -c command**, and returns after the command has been completed.

Example: system.c

```
int main()
{
   printf("Running ps with system\n");
   system("ps -eaf");
   printf("Done.\n");
   exit(0);
}
```

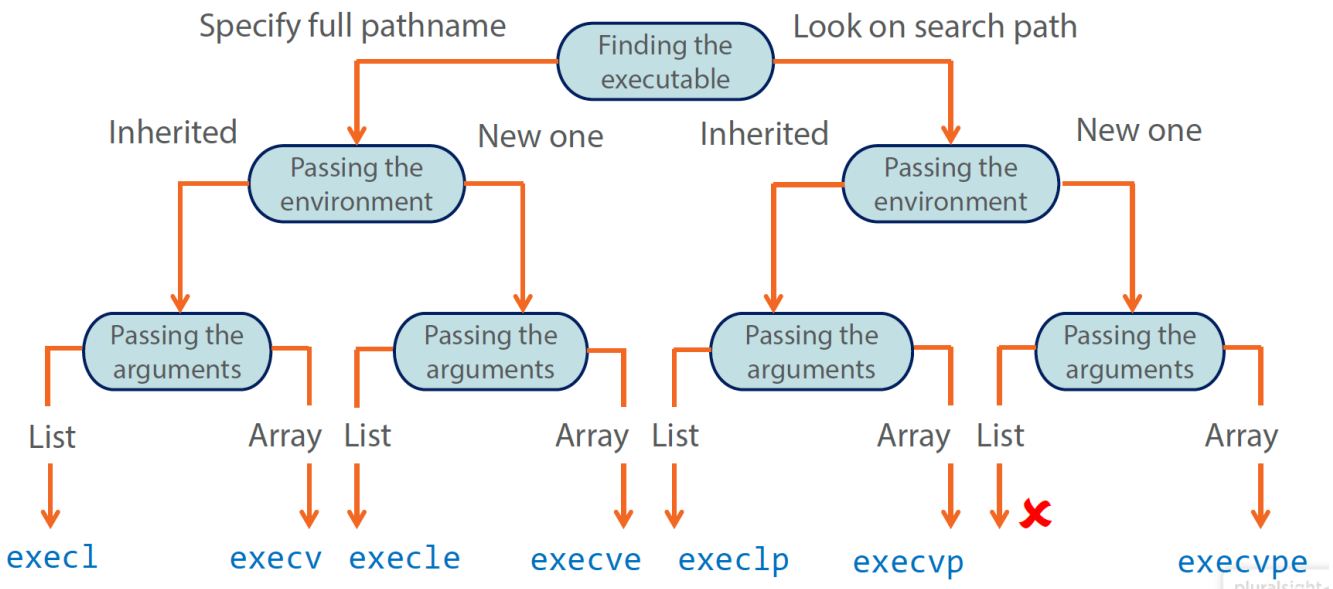
Replacing a process image: (using execve(2) system call)



- When the execl call executes, the **calling process's process image is overlayed by that of the executed (called) process. In effect, the caller's image is lost.**
- In addition, the process has the **same** process identification number as that of the calling program.
- There is no possibility of returning to the calling program, because the calling image no longer exists. **The successor overlays the predecessor.**
- All the execl system calls return no value on success because the calling image is lost; any return value (which on error is -1) signifies failure. The external integer errno contains information on the cause of error.

Note that of all the six functions, only execve() is a system call - the rest are library calls (wrapper functions) that transform their arguments and issue the execve() system call.

Choosing the Right Version of exec()



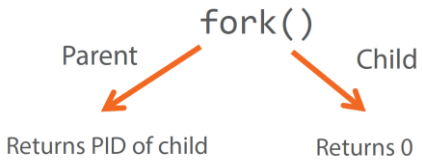
exec() Examples

```
char *argv[] = {"ls", "-a", 0};char *envp[] = {"EDITOR=vi", "TZ=:EST", 0};
execl("/bin/ls", "ls", "-a", 0};
execle("/bin/ls", "ls", "-a", 0, envp);
execv("/bin/ls", argv);
execve("/bin/ls", argv, envp);   -> System Call
execlp("ls", "ls", "-a", 0);
execvp("ls", argv);
execvpe("ls", argv, envp);
```

Example: pexec.c

```
char *const ps_argv[] = {"ps", "-eaf", 0};
int main()
{
   printf("Running ps with execlp\n");
   getchar();
  //execlp("ps", "ps", "-eaf");  // Library
   execve("/bin/ps",ps_argv, NULL); // system call
   printf("Done.\n");
   exit(0);
}
```

Duplicating a process image (Using fork() system call):



The only way for an application to create a new process (or task) on Linux is using the fork() system call. (Actually, on Linux, the __clone() system call is available as well and is used to create and implement threads).

ret = fork();

The system executes this call by creating a new process that is an identical copy of the calling image, including program text (code), data, and context (the content of the hardware registers at the time the call is executed).

After the call, there are two processes:
The calling process (known as the parent process)
The process created by the fork() call (known as the child process)

Execution of the program continues in both the parent and child processess at the next instruction after the call to fork().
Each process can access it's and it's parent's process ID by calling getpid() and getppid() respectively.
In order to distinguish between the parent and child process in code, the kernel arranges that the return value of fork() in the child process is always 0 and the return value in the parent process is always the PID of the child. The value -1 is returned on failure (errno should be checked for cause of failure).

Parent to Child Inheritance and Non-inheritance characteristics

When a process forks, the child process inherits :	**When a process forks, the child process differs from the parent in :**
• signal mask and registered dispositions • process group ID (PGID) • session ID • current working directory • root directory • umask setting • process credentials - the RUID, EUID, RGID and EGID • SUID and SGID flag (setting) • controlling terminal, if any • all open files (via the OFDT) • close-on-exec flag for open files. • environment variables • attached shared memory segments • resource limits	• PID and PPID • return values from fork() are different • utime - execution time and user CPU time for the child are set to zero • file locks held by the parent are not inherited by the child process • alarms and timers held by the parent are not inherited by the child process; also, pending alarms are cleared for the child • Pending signal mask is cleared for the child

Example: fork1.c Only Children Only Parent Both children & Parent

```
main()
{
      pid_t pid;
      getchar();
      pid = fork();

      if(pid == -1)
      perror("fork fails");

      if(pid == 0)
      {
            sleep(2);
            printf("children process pid:%d\n",getpid());
            printf("children process parent pid:%d\n",getppid());
      }
      else
      {
            sleep(2);
            printf("parent process pid:%d\n",getpid());
            printf("parent process parent pid:%d\n",getppid());
      }
}
```

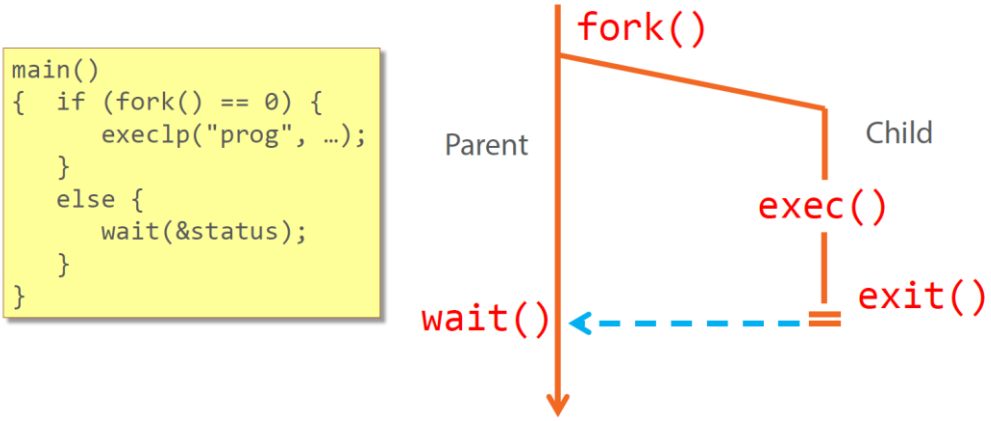
Example: fork2.c

```
int main()
{
   pid_t pid;
   char *message;
   int n;
   printf("fork program starting\n");
   pid = fork();
   switch(pid)
   {
   case -1:
      perror("fork failed");
      exit(1);
   case 0:
      message = "This is the child";
      printf ("child PID:%d Child PPID:%d\n",getpid(),getppid());
      n = 5;
      break;
   default:
      message = "This is the parent";
      printf ("Parent PID:%d Parent PPID:%d\n",getpid(),getppid());
      n = 3;
      break;
   }
   for(; n > 0; n--) {
      puts(message);
      sleep(1);
   }
   exit(0); }
```

Process Termination:

Parent process must wait for the child process to terminate.

```
main()
{  if (fork() == 0) {
      execlp("prog", …);
   }
   else {
      wait(&status);
   }
}
```



Process Termination

```
exit(n);
```

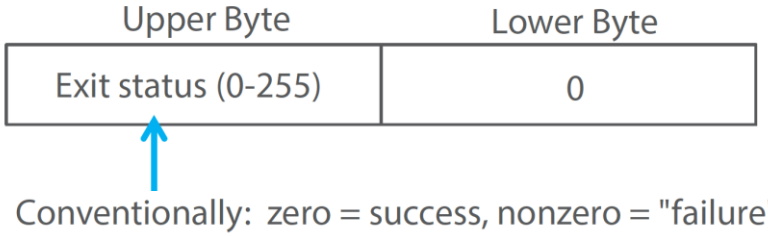
Exit status
Passed back to parent
0 means success
1-255 means failure

```
int status;
wait(&status);
```

Call waits until a child process terminates.
Returns PID of the child

The child's exit status is returned here.
Pass 0 (NULL) if not interested

Exit Status – Normal Termination



Conventionally: zero = success, nonzero = "failure"

MACRO	Meaning
WIFEXITED(status)	True if child exited normally
WEXITSTATUS(status)	The exit status

*The system primitive to suspend the parent process' execution until any of it's child processes die is **wait()**. Once the child dies, the parent continues execution normally. The wait() routine is a good case of a **blocking call** - one that blocks the caller until some event occurs.*

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

Return Value: wait returns the PID of the child process that died on success; if no child were present, it returns -1 immediately (without blocking).

***Status:** pointer to integer - if not null, values are stored into this integer denoting how exactly the child terminated; the wait.h header has several macros (conveniently grouped into three pairs) to determine status information:*

WIFEXITED(status)
is non-zero if the child exited normally.
WEXITSTATUS(status)
evaluates to the least significant eight bits of the return code of the child which terminated, which may have been set as the argument to a call to exit() or as the argument for a return statement in the main program. This macro can only be evaluated if WIFEXITED returned non-zero.
This is essentially the exit status of the child process (0 = success, non-zero implies failure).
WIFSIGNALED(status)
returns true if the child process exited because of a signal which was not caught.
WTERMSIG(status)
returns the number of the signal that caused the child process to terminate. This macro can only be evaluated if WIFSIGNALED returned non-zero.
WIFSTOPPED(status)
returns true if the child process which caused the return is currently stopped; this is only possible if the call was done using WUNTRACED.
WSTOPSIG(status)
returns the number of the signal which caused the child to stop. This macro can only be evaluated if WIFSTOPPED returned non-zero.

What is the output of the following program?

```
main()
{
        pid_t pid;
        pid = fork();
        pid = fork();
        pid = fork();
        printf("Hello World\n");
}
```

Orphan Process

When a process forks and the parent process exits before the child, an inconsistency results in the kernel's task structures (where all process information is stored). The inconsistency is with regard to the PPID of the child - since the parent has died, the PPID now in the child's task structure is invalid.

The kernel exit cleanup code corrects this inconsistency, setting the child's PPID to that of the init process (always having a PID of 1), in effect re-parenting the child. This is clever as the init process (now the parent) “cannot” die – init dies only when the system is shutting down. Hence the situation cannot recur.

Since the child lost it's immediate parent, it is termed an orphaned process.

Zombies

When a process forks and the child process terminates, the child's internal task structure information is maintained within the kernel (even though the child is technically "dead"). The reason that the kernel keeps the information is for the parent process to "fetch" it. The parent process should (is expected to) fetch the child's status information by doing a wait() on for the child process. Only then does the kernel release the child's context information and the child is truly dead.

Typically, the parent process would, after forking, wait for the child to terminate (directly or indirectly).

If, however, the parent process does not bother to wait for it's child (for whatever reason), then, when the child dies, it's task information is still maintained in the kernel. The state of the child process now is neither completely dead nor alive - thus, it is called a zombie.

Zombies are not desirable because:

a) PIDs can get used up

b) they lead to wastage of precious kernel memory.

Developers must avoid zombies (by waiting for their children).

Running the earlier-mentioned fork program below in the background, we can see that it has resulted in the generation of a zombie process.

The clone function

- New processes are created using the fork() system call − the call essentially works by creating a copy of the parent's areas − data, stack, file structures, signal information, etc to the child.
- To avoid wasteful copying and memory usage, the Copy-On-Write (COW) technique is used whereby the child shares the parent's data segments (initialized and uninitialized data).
- The text area is always truly shared in any case as it is has r-x permissions; the COW optimization applies to the data segment − if either process (parent or child) modifies a piece of data, first a copy of the containing page is made for the child, and then the modification takes place.
- The fork call creates what OS designers call a heavyweight process − the child is a full copy of the parent. Of course, this is more true of earlier Unices than Linux − Linux's fork call is noted for it's high efficiency.

- Still, there is the need for truly lightweight processes – the best example being threads; all threads of a process must truly share both the text and data segments.
- Linux's clone() system call (non-portable) can do this; in fact, clone creates a “custom-built” task: the task “type” and “weight” depend upon flags passed to the clone system call.
- In fact, both the vfork() system call as well as Linux's POSIX threads (pthreads) creation is based on the clone system call.