**LINUX SYSTEM PROGRAMMING**

**Process Management Subsystem**

1. **Explain the concept of process creation in operating systems.**

* A Process is a program in execution. a.out is program which is present in hard disk.
* When it is executed it is loaded into user space of RAM. The memory segments get loaded into RAM and PCB get created in Kernel space of RAM

1. **Differentiate between the fork() and exec() system calls.**

* **fork() :** fork() is used to create the another process i.e, child process. Fork() duplicate the parent process.
* **exec():** exec() family calls are replace the old process image with new process image. No new process is created instead the current process is replaced.

1. **What is the purpose of the wait() system call in process management?**

* Wait() system call used by the parent process wait to its child process termination.
* When a process calls wait(), the **parent is put to sleep** until one of its children finishes execution.
* By using wait(), the parent:

Collects the child’s exit status.

Frees the process table entry.

Prevents zombie processes.

1. **Describe the role of the exec() family of functions in process management.**

* The exec() family of functions is used to **replace the current process image** with a **new program image**.
* Unlike fork() (which creates a new process), exec() **does not create a new process**. Instead, it transforms the existing process into another program

1. **illustrate the use of the execvp() function**

execvp() is one of the exec() family functions used to replace the current process image with a new program.

**Syntax**:

int execvp(const char \*file, char \*const argv[]);

**Parameters**:

* **file** → name of the program to execute (can be searched in PATH).
* **argv[]** → NULL-terminated array of arguments (argv[0] = program name).

1. **How does the vfork() system call differ from fork()?**

* **fork():** Creates a new child process by duplicating the parent process. Both parent and child have separate address spaces.
* **vfork():** Creates a child process that shares the address space of the parent until it either calls exec() or exit().

1. **Discuss the significance of the getpid() and getppid() system calls.**

**getpid() →** Returns the Process ID (PID) of the calling process.

**getppid() →** Returns the Parent Process ID (PPID) of the calling process.

1. **Explain the concept of process termination in UNIX/LINUX-like operating systems.**

* Process termination is the completion of execution of a process and the release of its resources by the operating system.
* Once a process terminates, it moves from the running or waiting state to the terminated state.

1. **Write a program in C to create a child process using fork() and print its PID.**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

int pid;

pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1;

}

else if (pid == 0) {

printf("This is the Child Process.\n");

printf("Child PID: %d\n", getpid());

printf("Parent PID: %d\n", getppid());

}

else {

printf("This is the Parent Process.\n");

printf("Parent PID: %d\n", getpid());

printf("Child PID: %d\n", pid);

}

return 0;

}

1. **Describe the process hierarchy in UNIX-like operating systems.**

* A process hierarchy is the parent-child relationship among processes in UNIX.
* Every process (except the init process) has a parent process that created it using fork().
* Processes can also create multiple child processes, forming a tree-like structure.

1. **What is the purpose of the exit() function in C programming?**

The exit() function in C terminates a program immediately and returns a status code to the operating system or parent process.

1. **Explain how the execve() system call works and provide a code example.**

**execve()**

* System call in UNIX/Linux.
* **Syntax:**

**int execve(const char \*pathname, char \*const argv[], char \*const envp[]);**

* Requires full path to the executable (e.g., /bin/ls).
* Environment array (envp) must be provided.
* Does not search PATH; if you give ls instead of /bin/ls, it will fail.
* Does not return on success.

**execvp()**

* **Library function**, part of the exec() family.
* Syntax:

**int execvp(const char \*file, char \*const argv[]);**

* Searches the directories in PATH to find the executable.
* Only requires program name (e.g., ls) instead of full path.
* No environment array needed; uses current process environment automatically.
* Often easier to use in shell-like programs.

**Summary:**

* execve() → low-level, requires full path and env array.
* execvp() → higher-level, searches PATH automatically, easier for running commands.

**1.Example using execve()**

#include <stdio.h>

#include <unistd.h>

int main() {

char \*args[] = {"/bin/ls", "-l", NULL};

char \*env[] = {NULL}; // Use current environment

printf("Before execve()\n");

if (execve("/bin/ls", args, env) == -1) {

perror("execve failed");

}

printf("After execve()\n"); // Will not execute if execve succeeds

return 0;

}

**Explanation:**

* Requires **full path** /bin/ls.
* Must provide **environment array** env[].
* Replaces current process with ls -l.

**Sample Output:**

Before execve()

(total files and directories listed by 'ls -l')

**2. Example using execvp()**

#include <stdio.h>

#include <unistd.h>

int main() {

char \*args[] = {"ls", "-l", NULL};

printf("Before execvp()\n");

if (execvp("ls", args) == -1) {

perror("execvp failed");

}

printf("After execvp()\n"); // Will not execute if execvp succeeds

return 0;

}

**Explanation:**

* Only requires **program name** ls.
* Searches **directories in PATH** automatically.
* Uses current environment automatically.

**Sample Output:**

Before execvp()

(total files and directories listed by 'ls -l')

1. **Discuss the role of the fork() system call in implementing multitasking.**

fork() is a system call in UNIX/Linux used to create a new process by duplicating the calling process (parent).

The new process is called the child process, which has its own unique PID.

**Multitasking** = Running multiple processes seemingly simultaneously.

* **fork() enables multitasking** by creating multiple processes that can execute independently.
* Example:
  + Parent creates child processes for different tasks.
  + OS scheduler switches CPU between parent and child.

**Example Code:**

#include <stdio.h>

#include <unistd.h>

int main() {

printf("Parent process starting. PID: %d\n", getpid());

pid\_t pid1 = fork();

if (pid1 == 0) {

printf("First child running. PID: %d, Parent PID: %d\n", getpid(), getppid());

}

else {

pid\_t pid2 = fork();

if (pid2 == 0) {

printf("Second child running. PID: %d, Parent PID: %d\n", getpid(), getppid());

}

else {

printf("Parent still running. PID: %d, Children PIDs: %d, %d\n", getpid(), pid1, pid2);

}

}

return 0;

}

**OUTPUT** :

Parent process starting. PID: 1010

Parent still running. PID: 1010, Children PIDs: 1011, 1012

First child running. PID: 1011, Parent PID: 1010

Second child running. PID: 1012, Parent PID: 1010

1. **Write a C program to create multiple child processes using fork() and display their PIDs.**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

int n = 3;

pid\_t pid;

printf("Parent process PID: %d\n", getpid());

for (int i = 1; i <= n; i++) {

pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1;

}

else if (pid == 0) {

printf("Child %d running. PID: %d, Parent PID: %d\n", i, getpid(), getppid());

exit(0);

}

}

sleep(1);

printf("Parent process finished creating %d children.\n", n);

return 0;

}

**OUTPUT:**

Parent process PID: 1010

Child 1 running. PID: 1011, Parent PID: 1010

Child 2 running. PID: 1012, Parent PID: 1010

Child 3 running. PID: 1013, Parent PID: 1010

Parent process finished creating 3 children.

1. **How does the exec() system call replace the current process image with a new one?**

**Definition**

* The **exec() family of system calls** (like execl(), execv(), execvp(), execve()) is used to **replace the current process image** with a **new program**.
* Unlike fork(), it **does not create a new process**; the current process is **overwritten**.

**What “Process Image” Means**

A process image includes:

1. **Code segment** → the program instructions
2. **Data segment** → global/static variables
3. **Heap** → dynamically allocated memory
4. **Stack** → function calls, local variables
5. **Program counter** → instruction pointer

When exec() is called, all of the above (except PID) is replaced by the new program.

**Example Code:**

#include <stdio.h>

#include <unistd.h>

int main() {

char \*args[] = {"ls", "-l", NULL};

printf("Before execvp()\n");

**// Replace current process with 'ls -l'**

if (execvp("ls", args) == -1) {

perror("execvp failed");

}

printf("After execvp()\n"); **// This will not execute if execvp succeeds**

return 0;

}

**OUTPUT:**

Before execvp()

(total files and directories listed by 'ls -l')

1. **Explain the concept of process scheduling in operating systems.**

**Definition**

* Process scheduling is the method by which an operating system decides the order in which processes access the CPU.
* The goal is to maximize CPU utilization, throughput, and system responsiveness while minimizing waiting time and turnaround time.

1. **Describe the role of the clone() system call in process management.**

**Definition**

* **clone()** is a **Linux-specific system call** used to **create a new process or thread**.
* It is a **more flexible version of fork()**, allowing the child to **share certain resources with the parent**, such as memory, file descriptors, and signal handlers.

Declared in **sched.h**:

#include <sched.h>

int clone(int (\*fn)(void \*), void \*child\_stack, int flags, void \*arg, ...);

**2. Purpose**

* Unlike fork() which always creates a separate process, clone() allows the parent and child to share resources.
* Used for:
  + Implementing threads (lightweight processes)
  + Creating processes with customized sharing of memory, file descriptors, or signal handlers
  + Advanced process control in Linux

1. **Write a program in C to create a zombie process and explain how to avoid it.**

**Definition**

* A zombie process is a terminated child process whose exit status has not been collected by the parent process.
* It still occupies an entry in the process table but does not consume CPU or memory for execution.

**2. Why it Happens**

* When a child calls exit(), it becomes a zombie until the parent calls wait() or waitpid() to read its exit status.
* If the parent never waits, the zombie remains, wasting a slot in the process table.

**3. C Program to Create a Zombie Process**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1;

}

else if (pid == 0) {

**// Child process**

printf("Child process running. PID: %d\n", getpid());

exit(0); // Child terminates immediately

}

else {

**// Parent process**

printf("Parent process PID: %d\n", getpid());

printf("Child process created with PID: %d\n", pid);

**// Sleep to allow child to become zombie**

printf("Parent sleeping for 10 seconds to create zombie...\n");

sleep(10);

printf("Parent exiting now.\n");

}

return 0;

}

**How to Verify**

* Run the program in one terminal.
* In another terminal, use:

ps -l

* You will see the child process with status Z (Zombie) until the parent exits.

**4. How to Avoid Zombie Processes**

1. **Use wait() or waitpid() in the parent**
   * Collect the child’s exit status after it terminates.

#include <sys/wait.h>

...

**wait(NULL);** // Wait for child to finish

1. **Ignore SIGCHLD signal**
   * OS automatically cleans up terminated children.

**signal(SIGCHLD, SIG\_IGN);**

1. Parent exits before child
   * Init/systemd adopts the child, which is then cleaned automatically.

**Modified Program to Avoid Zombie**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1;

}

else if (pid == 0) {

**// Child process**

printf("Child process running. PID: %d\n", getpid());

exit(0);

}

else {

**// Parent process**

printf("Parent PID: %d, waiting for child to finish...\n", getpid());

wait(NULL); // Collect child exit status

printf("Child process reaped. No zombie created.\n");

}

return 0;

}

**Output:**

Child process running. PID: 1011

Parent PID: 1010, waiting for child to finish...

Child process reaped. No zombie created.

**Summary**

* **Zombie process =** terminated child not reaped by parent.
* **Avoid zombies by:**
  + Using wait() / waitpid()
  + Ignoring SIGCHLD
  + Allowing init to adopt orphaned child

**20. Discuss the significance of the setuid() and setgid() system calls in process management**

**Definition**

* **setuid()** and **setgid()** are UNIX/Linux system calls used to **change the effective user ID (UID) or group ID (GID) of a process**.
* They control **permissions and access rights** when a process executes.

**2. Purpose / Significance**

**a) setuid()**

* Changes the **effective user ID** of a process.
* Allows a process to **temporarily gain or drop privileges**.
* Commonly used in programs that need **privileged access**, e.g., passwd command that updates /etc/shadow.

**b) setgid()**

* Changes the **effective group ID** of a process.
* Enables access to files and resources that belong to a specific group.
* Useful in **group-based permission control**.

**Example Code:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

printf("Real UID: %d, Effective UID: %d\n", getuid(), geteuid());

printf("Real GID: %d, Effective GID: %d\n", getgid(), getegid());

// Change effective UID/GID to 1000

if (setuid(1000) == 0 && setgid(1000) == 0) {

printf("After setuid/setgid:\n");

printf("Effective UID: %d\n", geteuid());

printf("Effective GID: %d\n", getegid());

} else {

perror("setuid/setgid failed");

}

return 0;

}

**21. Explain the concept of process groups and their significance in UNIX-like operating systems.**