Process Operations in POSIX/UNIX

A **process** is an instance of a program in execution. In UNIX/POSIX systems, processes are managed by the kernel through a set of **system calls**. These system calls allow process **creation, execution, synchronization, communication, and termination**.

1. Process Lifecycle in UNIX

A process typically goes through these states:

1. **New** → created by fork()
2. **Ready** → waiting for CPU scheduling
3. **Running** → executing instructions
4. **Waiting (Blocked)** → waiting for I/O or event
5. **Terminated** → exited (via exit())

Parent-child relationships form a **process tree**. The **init/systemd process (PID 1)** is the ancestor of all processes.

2. Process Creation: fork()

- fork() is the fundamental system call to create a new process.
- The **child** is an almost identical copy of the parent:
  - Gets its own **PID**
  - Shares the same **code, heap, stack, file descriptors** (but as separate copies).
- Returns:
  - 0 → to the child process
  - >0 (child PID) → to the parent process
  - -1 → on error

Both processes continue execution from the next statement after fork().

Syntax: `pid_t fork(void);`

Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child process, PID = %d\n", getpid());
    } else if (pid > 0) {
        printf("Parent process, PID = %d\n", getpid());
    } else {
        perror("fork failed");
    }
    return 0;
}
```

3. Program Execution: exec() Family

Once a child process is created, we often want it to run **another program**. That's where exec() comes in.

- Replaces the process's **text (code), data, heap, stack** with a new program.

- The PID remains the same (process identity unchanged).
- On success, exec() does not return. If it returns, an error occurred.

Common Variants:

- execl(path, arg0, arg1, ..., NULL)
- execv(path, argv[])
- execlp(file, arg0, arg1, ..., NULL) → searches in PATH
- execvp(file, argv[]) → most commonly used

Syntax (execl): `int execl(const char *path, const char *arg0, ..., NULL);`

Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Before exec\n");
    execl("/bin/ls", "ls", "-l", NULL);
    perror("exec failed");
    return 0;
}
```

4. Process Termination

A process can end in two ways:

- **Voluntary Termination**
  - exit(status) → normal exit, cleans up resources
  - _exit(status) → immediate exit, skips cleanup
- **Involuntary Termination**
  - Signal received (e.g., SIGKILL, SIGSEGV)
  - Parent kills it using kill(pid, SIGKILL)

After termination, the process becomes a **zombie** until the parent collects its status using **wait()**.

Syntax:

```
void exit(int status);
void _exit(int status);
```

5. Process Synchronization (Parent-Child)

The parent can wait for child processes using:

- wait(int *status)
  - Blocks parent until *any* child exits
  - Status contains exit information
- waitpid(pid, &status, options)
  - Waits for a specific child (pid)
  - Non-blocking option (WNOHANG)

Prevents **zombie processes** by ensuring children are reaped.

Syntax:

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Example:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child process\n");
        _exit(0);
    } else {
        int status;
        wait(&status);
        printf("Parent waited, child exited\n");
    }
    return 0;
}
```

6. Process Identification

- getpid() → returns calling process's PID
- getppid() → returns parent's PID
- getuid() → real user ID
- geteuid() → effective user ID (used for permissions)

Syntax:

```
pid_t getpid(void);
pid_t getppid(void);
```

Example:

```
printf("PID = %d, Parent PID = %d\n", getpid(), getppid());
```

7. Process Control

- kill(pid, sig) → sends signal to process (e.g., kill(1234, SIGTERM))

  Syntax:

  ```
  int kill(pid_t pid, int sig);
  ```

  Example:

  ```
  kill(child_pid, SIGKILL);
  ```

- alarm(seconds) → schedules SIGALRM after given time

Syntax:

```c
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- Returns:
  - 0 if no previous alarm was set
  - Remaining seconds if a previous alarm was pending

Example:

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void handler(int sig) {
    printf("Caught SIGALRM after 5 seconds!\n");
}
int main() {
    signal(SIGALRM, handler);   // Register signal handler
    alarm(5);                   // Trigger SIGALRM in 5 seconds
    pause();                    // Wait until signal received
    return 0;
}
```

Output: Program waits 5 seconds, then prints message.

- pause() → process sleeps until signal received

Syntax:

```c
#include <unistd.h>
int pause(void);
```

- Returns -1 and sets errno when a signal is caught.

Example (combined with alarm):

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void handler(int sig) {
    printf("Pause ended! Signal received.\n");
}
int main() {
    signal(SIGALRM, handler);   // Handle SIGALRM
    alarm(3);                  // Schedule alarm after 3 seconds
    printf("Waiting for signal...\n");
    pause();                    // Block until signal arrives
    printf("Continuing execution.\n");
    return 0;}
```

8. Signals and Handlers

Signals are **asynchronous notifications** delivered to processes.

- `signal(int sig, void (*handler)(int))`: Registers a custom signal handler.

Syntax:

```
#include <signal.h>
void (*signal(int sig, void (*handler)(int)))(int);
```

sig → signal number (e.g., SIGINT, SIGTERM)

handler → function pointer to signal handler

Example:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void handler(int sig) {
    printf("Caught signal %d\n", sig);
}
int main() {
    signal(SIGINT, handler);  // Catch Ctrl+C
    while (1) {
        printf("Running... Press Ctrl+C\n");
        sleep(2);
    }
    return 0;
}
```

When you press Ctrl+C, instead of terminating, it prints "Caught signal 2".

Common signals:

- SIGINT → Ctrl+C
- SIGKILL → force kill (cannot be caught)
- SIGTERM → request to terminate
- SIGCHLD → sent to parent when child exits

9. Orphan and Zombie Processes

- **Zombie process** → Occurs when **child exits but parent does not call wait()**.
  - Entry remains in process table until collected.

```
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child process exiting...\n");
        _exit(0);  // Child terminates
    } else {
        printf("Parent sleeping... (not waiting for child)\n");
        sleep(20);  // Parent does not call wait()
    }
    return 0;
}
```

- **Orphan process** → Parent terminates before child.
  - Child is adopted by init/system (**PID 1**).

```
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        sleep(5);  // Child waits
        printf("Child (PID: %d), Parent (PID: %d)\n", getpid(),
         getppid());
    } else {
        printf("Parent exiting...\n");
        _exit(0);  // Parent exits immediately
    }
    return 0;
}
```

10. Process Priority and Scheduling

- UNIX scheduling uses **nice values** (-20 to 19):
  - Lower value = higher priority
  - Default = 0
- Functions:

```
#include <unistd.h>      // nice()
#include <sys/resource.h> // getpriority, setpriority

int nice(int inc);       //change priority
int getpriority(int which, id_t who);//directly manage priority
int setpriority(int which, id_t who, int prio);//directly manage
priority
```

Example: Changing Priority with `nice()`

```
#include <stdio.h>
#include <unistd.h>
int main() {
    int old = nice(0);  // get current priority
    printf("Current nice value = %d\n", old);

    int new = nice(5);  // increase by 5 (lower priority)
    printf("New nice value = %d\n", new);
    return 0;
}
```

Example: Using `getpriority()` and `setpriority()`

```
#include <stdio.h>
#include <sys/resource.h>
#include <unistd.h>
int main() {
    pid_t pid = getpid();
    printf("PID = %d\n", pid);

    int prio = getpriority(PRIO_PROCESS, pid);
    printf("Current priority = %d\n", prio);
```

```c
        setpriority(PRIO_PROCESS, pid, 10);
        prio = getpriority(PRIO_PROCESS, pid);
        printf("New priority = %d\n", prio);

        return 0;
    }
```

Important Headers

All process-related functions typically require:

```c
#include <unistd.h>      // fork, getpid, getppid, exec family
#include <sys/types.h>   // pid_t type
#include <sys/wait.h>    // wait, waitpid
#include <stdlib.h>      // exit, EXIT_SUCCESS, EXIT_FAILURE
#include <stdio.h>       // printf, perror
#include <signal.h>      // signal handling
```

Detailed Function Reference

Function	Header	Purpose
fork()	<unistd.h>	Create a child process
exec*()	<unistd.h>	Replace process image
exit()	<stdlib.h>	Terminate process gracefully
_exit()	<unistd.h>	Immediate termination
wait()	<sys/wait.h>	Wait for child process
waitpid()	<sys/wait.h>	Wait for specific child
getpid()	<unistd.h>	Get process ID
getppid()	<unistd.h>	Get parent process ID
kill()	<signal.h>	Send signal to process
signal()	<signal.h>	Define signal handler
nice()	<unistd.h>	Adjust priority

Process Management Commands

1. `ps` → Displays running processes.
   - `ps -ef` → Full list of all processes with details.
   - `ps aux` → BSD style, shows CPU and memory usage.
   - `ps -u <user>` → Show processes of a specific user.
2. `top` → Real-time dynamic view of processes.
   - Shows CPU, memory usage, process priority, etc.
   - Useful for monitoring and killing processes interactively.
3. `htop` (if installed) → Improved version of top with colors and scrolling.
4. `jobs` → Lists background and suspended jobs in the current shell.
5. `fg / bg` → Resume jobs in foreground/background.
6. `kill` → Sends a signal to a process.
   - `kill -9 <pid>` → Force kill (SIGKILL).
   - `kill -15 <pid>` → Graceful termination (SIGTERM).
7. `killall <name>` → Kill all processes with given name.
8. `pkill <pattern>` → Kill processes by name/regex.

Process Information Commands

1. `who` → Shows logged-in users.
2. `w` → Shows logged-in users + processes they are running.
3. `whoami` → Prints current logged-in user.
4. `id` → Shows UID, GID, and groups of the user.
5. `uname -a` → System and kernel info.

Scheduling and Priority

1. `nice -n <value> command` → Run a command with a given priority.
   - Example: `nice -n 10 ./myprog`
2. `renice <priority> -p <pid>` → Change priority of a running process.

Resource & System Monitoring

1. `uptime` → System load averages.
2. `free -m` → Memory usage.
3. `vmstat` → CPU, memory, I/O stats.
4. `iostat` → Disk I/O statistics.
5. `lsof` → List open files by processes.
6. `df -h` → Disk space usage.
7. `du -sh <dir>` → Disk usage of a directory.

Special Process/Signals

1. `strace <command>` → Trace system calls made by a process.
2. `time <command>` → Execution time of a process.
3. `nohup <command> &` → Run a command immune to hangups (useful for background jobs).
4. `at / cron` → Schedule processes for later execution.

Example:

```
ps -ef | grep firefox      # Check if firefox is running
kill -9 12345              # Kill process with PID 12345
nice -n 5 ./myapp          # Start app with lower priority
renice -10 -p 5678         # Increase priority of PID 5678
jobs                       # List jobs in this shell
fg %1                      # Bring job 1 to foreground
```

Process Management: `ps, top, htop, jobs, fg, bg, kill, pkill, killall`

Info: `who, w, whoami, id, uname -a`
Scheduling: `nice, renice`
Monitoring: `uptime, free -m, vmstat, iostat, lsof, df -h, du -sh`
Special: `strace, time, nohup, at, cron`

1. Write a program that calls `fork()` once and both parent and child print a message.
2. Write a program to print `PID` and `PPID` of both parent and child.
3. Write a program where the parent creates two children. Each child prints its PID and exits.
4. Write a program that creates a child process which terminates immediately, while the parent sleeps for 20 seconds. What will you see if you run `ps -l` during the sleep?
5. Write a program to show how to change process priority with `nice()`.