Lab 03: Processes in UNIX

3.1. UNIX Process Creation and `fork`

A process can create a new process by calling `fork`. The calling process becomes the **parent**, and the created process is called the **child**. The `fork` function copies the parent's memory image so that the new process receives a copy of the address space of the parent. Both processes continue at the instruction after the `fork` statement (executing in their respective memory images).

```
SYNOPSIS
   #include <unistd.h>
   pid_t fork(void);
```

Creation of two completely identical processes would not be very useful. The `fork` function **return value** is the critical characteristic that allows the parent and the child to distinguish themselves and to execute different code. The `fork` function returns 0 to the child and returns the child's process ID to the parent. When `fork` fails, it returns –1 and sets the `errno`. If the system does not have the necessary resources to create the child or if limits on the number of processes would be exceeded, `fork` sets `errno` to `EAGAIN`. In case of a failure, the `fork` does not create a child.

3.2. The wait Function

When a process creates a child, both parent and child proceed with execution from the point of the fork. The parent can execute `wait` or `waitpid` to **block** until the child finishes. The `wait` function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal. A process status most commonly becomes available after termination, but it can also be available after the process has been stopped.

The `waitpid` function allows a parent to wait for **a particular child**. The `waitpid` function takes three parameters: a `pid`, a pointer to a location for returning the status and a flag specifying options. If `pid` is –1, `waitpid` waits for any child. If `pid` is greater than 0, `waitpid` waits for the specific child whose process ID is `pid`. Two other possibilities are allowed for the `pid` parameter. If `pid` is 0, `waitpid` waits for any child in the same process group as the caller. Finally, if `pid` is less than –1, `waitpid` waits for any child in the process group specified by the absolute value of `pid`.

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

If `wait` or `waitpid` returns because the status of a child is reported, these functions return the process ID of that child. If an error occurs, these functions return –1 and set `errno`.

4.2.1. Status values

The `stat_loc` argument of `wait` or `waitpid` is a pointer to an integer variable. If it is not `NULL`, these functions store the return status of the child in this location. The child returns its status by calling `exit`, `_exit`, `_Exit` or `return` from main. A zero return value indicates `EXIT_SUCCESS;` any other value indicates `EXIT_FAILURE`. The parent can only access the 8 least significant bits of the child's return status.

POSIX specifies six macros for testing the child's return status. Each takes the status value returned by a child to `wait` or `waitpid` as a parameter.

```
SYNOPSIS
   #include <sys/wait.h>
   WIFEXITED(int stat_val)
   WEXITSTATUS(int stat_val)
   WIFSIGNALED(int stat_val)
   WTERMSIG(int stat_val)
   WIFSTOPPED(int stat_val)
   WSTOPSIG(int stat_val)
```

The six macros are designed to be used in pairs. The `WIFEXITED` evaluates to a nonzero value when the child terminates normally. If `WIFEXITED` evaluates to a nonzero value, then `WEXITSTATUS` evaluates to the low-order 8 bits returned by the child through `_exit()`, `exit()` or `return` from `main`.

The `WIFSIGNALED` evaluates to a nonzero value when the child terminates because of an uncaught signal. If `WIFSIGNALED` evaluates to a nonzero value, then `WTERMSIG` evaluates to the number of the signal that caused the termination.

The `WIFSTOPPED` evaluates to a nonzero value if a child is currently stopped. If `WIFSTOPPED` evaluates to a nonzero value, then `WSTOPSIG` evaluates to the number of the signal that caused the child process to stop.

The following function determines the exit status of a child.

```
void show_return_status(void) {
   pid_t childpid;   int status;
   childpid = wait(&status);
   if (WIFEXITED(status) && !WEXITSTATUS(status))
      printf("Child %ld terminated normally\n", (long)childpid);
   else if (WIFEXITED(status))
      printf("Child %ld terminated with return status %d\n",
             (long)childpid, WEXITSTATUS(status));
   else if (WIFSIGNALED(status))
      printf("Child %ld terminated due to uncaught signal %d\n",
             (long)childpid, WTERMSIG(status));
   else if (WIFSTOPPED(status))
      printf("Child %ld stopped due to signal %d\n",
            (long)childpid, WSTOPSIG(status));
}
```

Task 1: Create process chain as shown in figure 3.1(b) and fill the figure 3.1 (b) with actual IDs. The program shall take a single command-line argument that specifies the number of processes to be created. Before exiting, each process shall outputs its `i` value (loop variable), its process ID (using getpid()), its parent process ID (getppid()) and the process ID of its child (return value of fork). The parent does not execute `wait`.

Task 2: Create process fan as shown in figure 3.1 (a) and fill the figure 3.1 (a) with actual IDs.

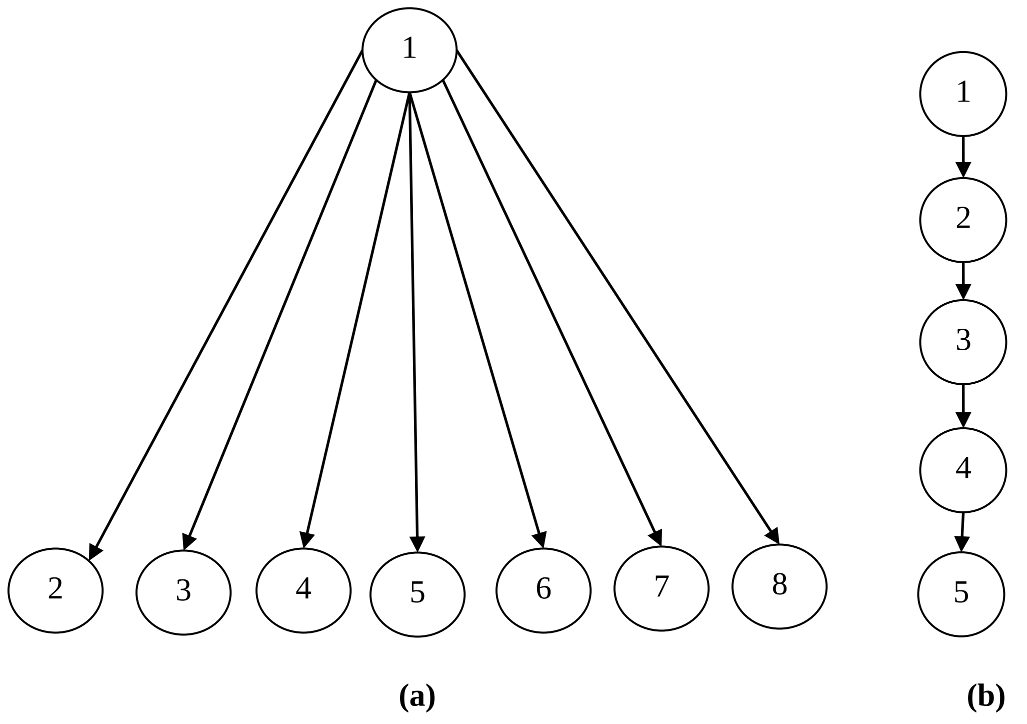


Figure 3.1 Multiple Processes (a) Process Fan (b) Process Chain

Task 3: Create process tree as shown in figure 3.2 and fill figure 3.2 with actual IDs.

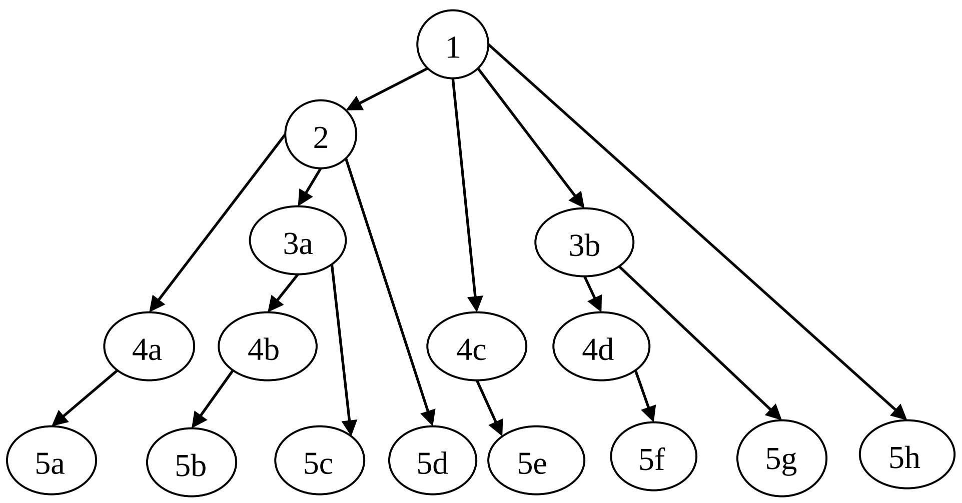


Figure 3.2 Multiple Processes: Process Tree

Task 4: This task expands on the process chain of Task 1. The chain is a vehicle for experimenting with `wait` and with sharing of devices. All of the processes in the chain created by Task 1 share standard input, standard output and standard error.

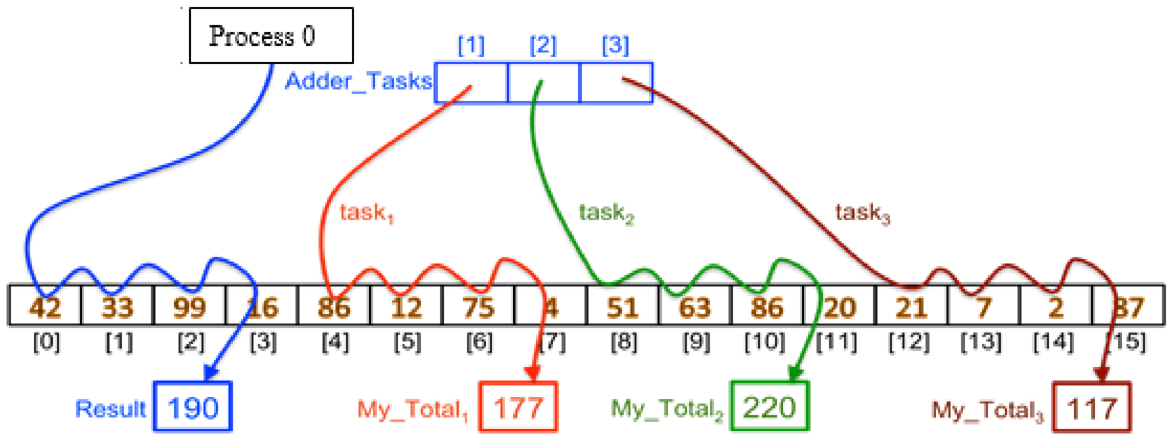
Task 3.1 creates a chain of processes. It takes a single command-line argument that specifies the number of processes to create. Before exiting, each process outputs its `i` value, its process ID, its parent process ID and the process ID of its child. The parent does not execute `wait`. If the parent exits before the child, the child becomes an **orphan**. In this case, the child process is adopted by a special system process (which traditionally is a process, `init`, with process ID of 1). As a result, some of the processes may indicate a parent process ID of 1.

a. Experiment with different values for the command-line argument to find out the largest number of processes that the program can generate. Observe the fraction that are adopted by `init`.

b. Place `sleep(10);` directly before the final `printf` statement. What is the maximum number of processes generated in this case?

Task 5: Write a program that takes N number of integers as argument and displays the factors of N integers. Create separate child process for each integer. Make sure no child is orphan/zombie.

Task 6: Write a program that creates an array of size 10,000. Initialize the array with random numbers. Create 10 child processes divide the array between them. Each child will add the portion and return their sum to parent process. Parent will add the results and display a final sum.



Process 0
[1]
[2]
[3]
Adder_Tasks
task₁
task₂
task₃
42 33 99 16 86 12 75 4 51 63 86 20 21 7 2 87
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15]
Result 190
My_Total₁ 177
My_Total₂ 220
My_Total₃ 117