Mohamed Boudiaf University – M'sila
Department of Computer Science
3rd year, Computer Science - SI

Solutions for Practical Session N°1

Process Handling in C under Linux

Objective: The aim of this practical session is to help students learn how to create and manipulate processes in Linux, understand parent–child relationships, and recognize the importance of process synchronization.

Basic Concepts:

fork() : Process Creation	**sleep() :** Sleep for a Specified Duration
getpid() – Get Process ID	**getppid()** – Get Parent Process ID
pause() : Sleep Until Signal	**kill() :** Send a Signal to a Process
wait() : Wait for Child Termination	**exit() :** Terminate a Process

- **fork()** may fail if there is insufficient memory or if the user has already created too many processes. In such cases, no child process is created, and fork() returns -1.
- To compile a C program on Linux, use the **gcc** compiler:

Compilation: gcc -o your_program your_program.c
Execution: ./your_program

Exercise 1:

Write a C program that creates 3 child processes using a **for loop**. For each iteration, the program should display the following information: **i = value of i, I am process: pid, my parent is: ppid**
The **parent process** should print the message "END"**,** after all child processes have finished**.**

Solution:

1. Try running the program without using ***exit*** and ***wait*** first, and observe the output.
2. Run it multiple times and notice that the processes run in parallel, so the output differs with each execution.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>  // for exit()

int main() {
    pid_t pid;
    int i;

    for (i = 0; i < 3; i++) {
        pid = fork();

        if (pid < 0) {
```

```
            // Fork failed
            perror("fork failed");
            exit(1);
        } else if (pid == 0) {
            // Child process
            printf("i = %d, I am process: %d, my parent is: %d\n", i, getpid(), getppid());
            exit(0);  // terminate child immediately
        }
        // Parent continues the loop
    }

    // Parent waits for all child processes
    for (i = 0; i < 3; i++) {
        wait(NULL);
    }

    printf("END\n");

    return 0;
}
```

Exercise 2:

Write a C program that creates two child processes; Child 1 prints the integers from 1 to 50, Child 2 prints the integers from 51 to 100. The parent process prints "End of processing".

- What do you notice about the output?
- Modify the program so that the numbers are printed in order from 1 to 100.

Note: Add **fflush(stdout)**; after each **printf** in your program to force immediate output and observe how the processes interleave.

Solution :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
    pid_t pid1, pid2;
    int i;

    // First child: prints 1 to 50
    pid1 = fork();
    if (pid1 < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid1 == 0) {
        for (i = 1; i <= 50; i++) {
            printf("%d ", i);
```

```
            fflush(stdout); // flush immediately to avoid buffering
        }
        printf("\n");
        exit(0);
    }

    // Second child: prints 51 to 100
    pid2 = fork();
    if (pid2 < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid2 == 0) {
        for (i = 51; i <= 100; i++) {
            printf("%d ", i);
            fflush(stdout); // flush immediately
        }
        printf("\n");
        exit(0);
    }

    // Parent waits for both children
    wait(NULL);
    wait(NULL);

    printf("End of processing\n");
    return 0;
}
```

Observation: The output is **not guaranteed to be in order** because Child 1 and Child 2 run concurrently (in parallel). So, the numbers from 1–50 and 51–100 may **interleave**, depending on the operating system scheduler (example : 1 2 51 3 52 4 53 … End of processing).

Without synchronization, the output order depends on the CPU scheduling, so it can look random or interleaved.

Modified solution (guaranteed order 1 → 100) :

We can ensure ordered output by **making the parent wait for the first child before starting the second child**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
    pid_t pid1, pid2;
    int i;
    // First child: prints 1 to 50
    pid1 = fork();
    if (pid1 < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid1 == 0) {
```

```
        for (int i = 1; i <= 50; i++) {
            printf("%d ", i);
        }
        printf("\n");
        exit(0);
    }

    // Parent waits for first child to finish
    waitpid(pid1, NULL, 0);

    // Second child: prints 51 to 100
    pid2 = fork();
    if (pid2 < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid2 == 0) {
        for (int i = 51; i <= 100; i++) {
            printf("%d ", i);
        }
        printf("\n");
        exit(0);
    }

    // Parent waits for second child
    waitpid(pid2, NULL, 0);

    printf("End of processing\n");
    return 0;
}
```

Exercise 3 :

Write a C program with a global counter starting at zero and create two child processes Child 1 and Child 2. Each child should increment the counter 5 times and print its value. Each increment should be printed along with the child's identifier. The parent process should wait for both children to finish and then print the final counter value.

- What do you notice about the output?
- Modify the program so that the counter increments in a fixed order, alternating between Child 1 and Child 2.

solution

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int counter = 0; // global counter

int main() {
    pid_t pid1, pid2;
    int i;
    // Create first child
    pid1 = fork();
    if (pid1 < 0) {
```

```
        perror("fork failed");
        exit(1);
    } else if (pid1 == 0) {
        for (i = 0; i < 5; i++) {
            counter++;
            printf("Child 1: counter = %d\n", counter);
        }
        exit(0);
    }

    // Create second child
    pid2 = fork();
    if (pid2 < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid2 == 0) {
        for (i = 0; i < 5; i++) {
            counter++;
            printf("Child 2: counter = %d\n", counter);
        }
        exit(0);
    }

    // Parent waits for both children
    wait(NULL);
    wait(NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Observation

- The output of Child 1 and Child 2 may interleave unpredictably.
- Each child increments its own copy of counter.
- Changes are not visible to the parent or to the other child.
- That's why the parent always prints Final counter value: 0.

Solution with synchronization (alternate increments)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int counter = 0; // global counter

int main() {
    pid_t pid1, pid2;
    int i;
    // First child
    pid1 = fork();
    if (pid1 < 0) {
```

```
        perror("fork failed");
        exit(1);
    } else if (pid1 == 0) {
        for (i = 0; i < 5; i++) {
            counter++;
            printf("Child 1: counter = %d\n", counter);
            fflush(stdout);
            sleep(1); // optional, for clarity
        }
        exit(0);
    }

    // Parent waits for first child to finish
    waitpid(pid1, NULL, 0);

    // Second child
    pid2 = fork();
    if (pid2 < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid2 == 0) {
        for (i = 0; i < 5; i++) {
            counter++;
            printf("Child 2: counter = %d\n", counter);
            fflush(stdout);
            sleep(1); // optional, for clarity
        }
        exit(0);
    }

    // Parent waits for second child
    waitpid(pid2, NULL, 0);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Global Observation

When using fork(), the parent and child do **not** share global variables. Each process gets its **own private copy** of memory, so changes made by one process are invisible to the others. This is why the parent's counter never increases.

To allow several processes to **really share and update the same variable**, we must use **synchronization mechanisms (example: Semaphore, Monitor…).**