National University
of computer and emerging sciences

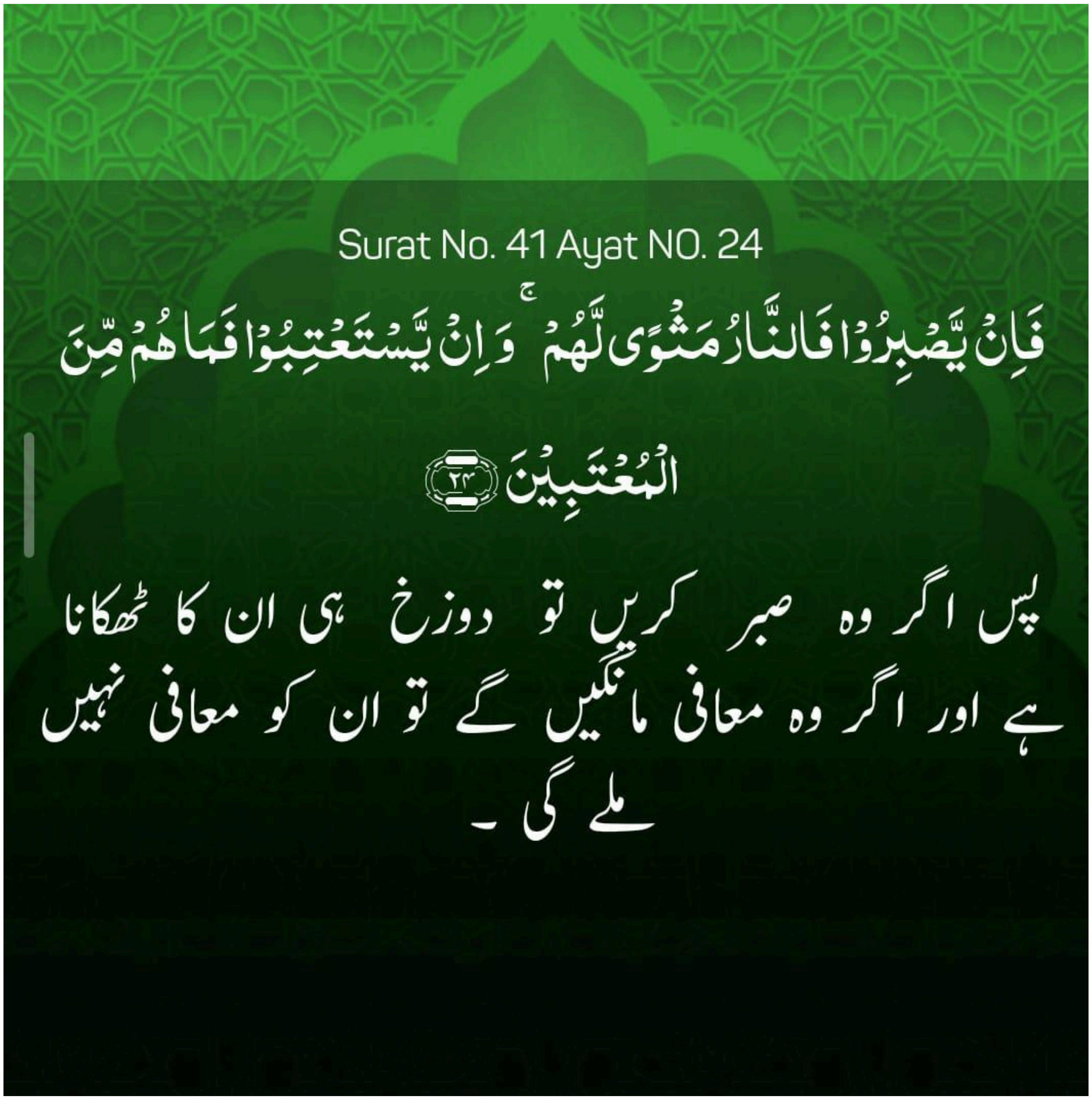
CL 2006 OPERATING SYSTEM

BS Software Engineering

Fall-2024

Course Instructors: Syed Daniyal Hussain Shah, Hifza Umar

Course Coordinator: Syed Daniyal Hussain Shah



Surat No. 41 Ayat NO. 24

فَاِنۡ یَّصۡبِرُوۡا فَالنَّارُ مَثۡوًی لَّهُمۡ ۚ وَاِنۡ یَّسۡتَعۡتِبُوۡا فَمَا هُمۡ مِّنَ الۡمُعۡتَبِیۡنَ ﴿۲۴﴾

پس اگر وہ صبر کریں تو دوزخ ہی ان کا ٹھکانا ہے اور اگر وہ معافی مانگیں گے تو ان کو معافی نہیں ملے گی ۔

Lab Policy and Rules:

1. **100% attendance is mandatory.** In case of an emergency, you can avail yourself of up to 3 absences. So, don't waste your absences , save them for emergencies. If you are debarred from the course due to low attendance, do not come to me to correct it.

2**. Disturbing the class environment during lab sessions.** such as by talking, using mobile phones, eating, etc. will result in penalties (e.g., deduction of marks).

3. **Lab tasks will not be accepted if you are absent.** If you miss a lab, you cannot submit that lab task.

4**. Lab demos for each lab task will be conducted at the end of each session.** If you miss the demo for any reason, no retake will be allowed, and that lab will be marked as 0.

5. **All quizzes will be unannounced**. Be prepared for surprise quizzes. Quizzes will cover content from previous labs as well as theory lectures from the corresponding theory course.

6. **You can take help from the internet for lab tasks,** but simply copying and pasting without understanding will be marked as 0. You should be able to explain the syntax or material used in your tasks.

7**. Do not ask for personal favors.** If you have concerns, such as short attendance, please speak with the relevant authority (e.g., academics).

8**. Students on warning:** Now is the time to study and earn a good grade. Do not ask for extra marks at the end if you are unable to clear your warning.

Steps to Install GCC:

Update Your Package Lists:

Open a terminal and run the following command to ensure that your package lists are up to date:

```
sudo apt update
```

Install GCC:

```
sudo apt install build-essential
```

This will install GCC, G++, and other necessary development tools.

Verify Installation:

After installation, verify that GCC was installed correctly by checking its version.

```
gcc --version
```

System call:

In computing, a system call is the programmatic way in which a computer requests a service from the kernel of the operating system. A system call is a request for the service that a program makes and that service is something that only the kernel has the privilege to do. It provides an interface between a process and the kernel to allow user level processes to request services of the kernel.

Fork ():

fork() is used to create a new process, called a child process, which runs concurrently with the parent process. The new process is an exact copy of the parent process except for the returned value.

Return Value:

0 in the child process.

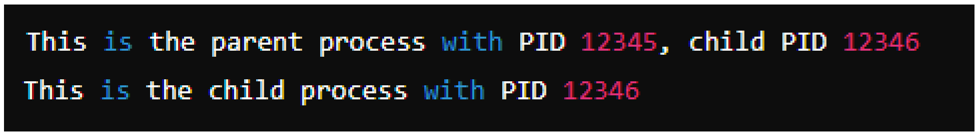
The child's process ID (PID) in the parent process.

-1 if the process creation fails.

Example 1:

```
#include <stdio.h>

#include <unistd.h>


int main() {

    pid_t pid = fork();


    if (pid == -1) {

        printf("Fork failed.\n");

        return 1;

    }


    if (pid == 0) {

        printf("This is the child process with PID %d\n", getpid());

    } else {

        printf("This is the parent process with PID %d, child PID %d\n", getpid(), pid);

    }


    return 0;

}
```

Output:



```
This is the parent process with PID 12345, child PID 12346
This is the child process with PID 12346
```

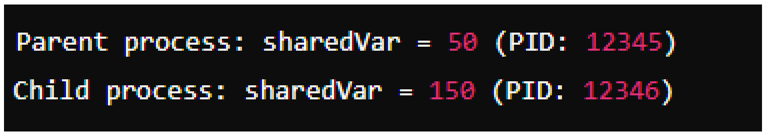
Example 2:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int sharedVar = 100;

    pid_t pid = fork();

    if (pid == -1) {
        printf("Fork failed.\n");
        return 1;
    }

    if (pid == 0) {
        sharedVar += 50;
        printf("Child process: sharedVar = %d (PID: %d)\n", sharedVar, getpid());
    } else {
        sharedVar -= 50;
        printf("Parent process: sharedVar = %d (PID: %d)\n", sharedVar, getpid());
    }

    return 0;
}
```

- The variable sharedVar is initialized to 100 before the fork() call. At this point, both the parent and the child process will have access to this variable in memory.

- The fork() system call creates a child process. After the fork, both processes have their own copies of the memory space. Thus, they have their own copies of sharedVar.
- In the child process (pid == 0), the value of sharedVar is modified by adding 50, so it becomes 150. This modification only affects the child's copy of the variable.
- In the parent process (pid != 0), the value of sharedVar is modified by subtracting 50, so it becomes 50. This modification only affects the parent's copy of the variable.
- The child process and parent process both print their versions of sharedVar, showing that the values differ after fork().

Output:

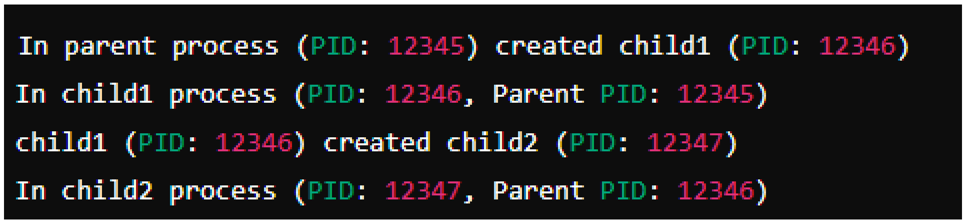


```
Parent process: sharedVar = 50 (PID: 12345)
Child process: sharedVar = 150 (PID: 12346)
```

Example 3:

```
#include <stdio.h>

#include <unistd.h>


int main() {

    pid_t pid1 = fork();


    if (pid1 == -1) {

        printf("Fork failed.\n");

        return 1;

    }


    if (pid1 == 0) {

        printf("In child1 process (PID: %d, Parent PID: %d)\n", getpid(), getppid());


        pid_t pid2 = fork();


        if (pid2 == -1) {
            printf("Fork failed.\n");
            return 1;
        }

        if (pid2 == 0) {
            printf("In child2 process (PID: %d, Parent PID: %d)\n", getpid(), getppid());
        } else {
            printf("child1 (PID: %d) created child2 (PID: %d)\n", getpid(), pid2);
        }

    } else {
        printf("In parent process (PID: %d) created child1 (PID: %d)\n", getpid(), pid1);
    }

    return 0;
}
```

Output:



```
In parent process (PID: 12345) created child1 (PID: 12346)
In child1 process (PID: 12346, Parent PID: 12345)
child1 (PID: 12346) created child2 (PID: 12347)
In child2 process (PID: 12347, Parent PID: 12346)
```

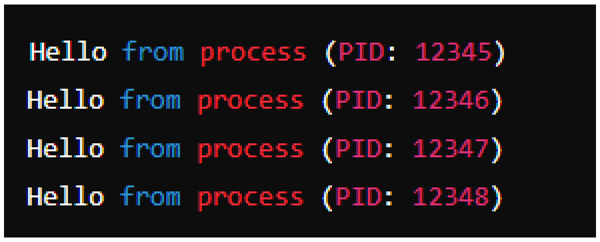
Example 4:

```
#include <stdio.h>
#include <unistd.h>
```

```
int main() {
   fork();  // First fork
   fork();  // Second fork

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

   return 0;
}
```

- When the first fork() is called, a new child process is created. Now, there are two processes: the parent and the child.
- When the second `fork()` is called, both the parent and child processes from the first fork will execute the `fork()` again. This will result in a total of 4 processes.

Output:



```
Hello from process (PID: 12345)
Hello from process (PID: 12346)
Hello from process (PID: 12347)
Hello from process (PID: 12348)
```

getpid():

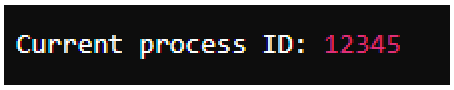
getpid() returns the process ID (PID) of the calling process.

Example:

```
#include <stdio.h>
```

```
#include <unistd.h>


int main() {
   printf("Current process ID: %d\n", getpid());
   return 0;
}
```

Output:



```
Current process ID: 12345
```

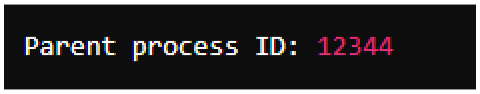
getppid():

`getppid()` returns the parent process ID (PPID) of the calling process.

Example:

```
#include <stdio.h>

#include <unistd.h>


int main() {
   printf("Parent process ID: %d\n", getppid());
   return 0;
}
```

Output:



```
Parent process ID: 12344
```

wait() and sleep():

wait() makes the parent process wait until all of its child processes have terminated. It returns the PID of the terminated child and allows the parent process to retrieve the child's exit status.

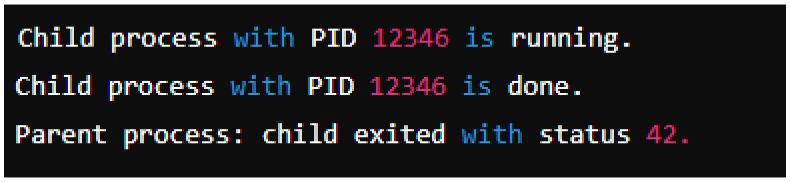
It returns PID of the terminated child. If no child process is available to be waited on, it returns -1.

sleep() causes the process to suspend its execution temporarily for a period of time in seconds specified as argument.

Example:

```
#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>


int main() {

  pid_t pid = fork();


  if (pid == 0) {

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

    sleep(2);  // Simulate some work with sleep

    printf("Child process with PID %d is done.\n", getpid());

    return 42;  // Child exits with status 42

  } else {

    int status;

    wait(&status);  // Parent waits for child to finish

    printf("Parent process: child exited with status %d.\n", WEXITSTATUS(status));

  }


  return 0;

}
```

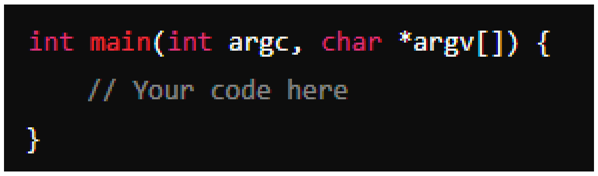
Output:



```
Child process with PID 12346 is running.
Child process with PID 12346 is done.
Parent process: child exited with status 42.
```

Command line arguments:

Command-line arguments are passed to the main() function when the program is executed. These arguments allow the user to pass data into the program from the command line.



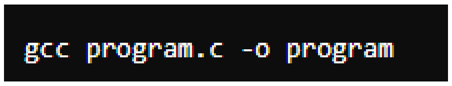
```
int main(int argc, char *argv[]) {
    // Your code here
}
```

- **argc:** This is an integer that represents the number of command-line arguments passed to the program. It includes the program's name, so argc is always at least 1.
- **argv:** This is an array of strings (character pointers) representing the command-line arguments. argv[0] is always the name of the program (or the path used to execute the program), and argv[1] to argv[argc-1] are the additional arguments passed to the program.

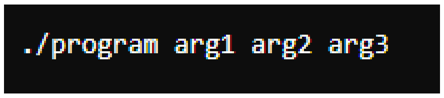
Example:

```
#include <stdio.h>


int main(int argc, char *argv[]) {
  printf("Number of arguments: %d\n", argc);


  for (int i = 0; i < argc; i++) {
    printf("Argument %d: %s\n", i, argv[i]);
  }


  return 0;
}
```

How to Compile and Run the Program:

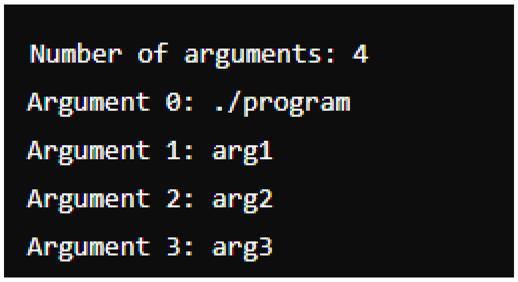


```
gcc program.c -o program
```



```
./program arg1 arg2 arg3
```

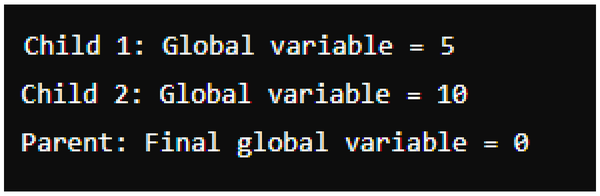
Output:



```
Number of arguments: 4
Argument 0: ./program
Argument 1: arg1
Argument 2: arg2
Argument 3: arg3
```

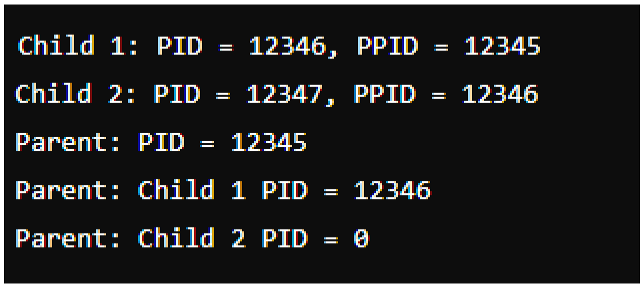
Tasks:

I. Write a program that creates two child processes using fork(). The program should use a global variable to demonstrate how changes in the global variable are reflected in the child processes. Each child process should increment the global variable by a specific amount and print its result. The parent process should print the final value of the global variable after both child processes have completed. You are not allowed to use wait(). Output is given below: **[Estimated Time: 20 mins]**



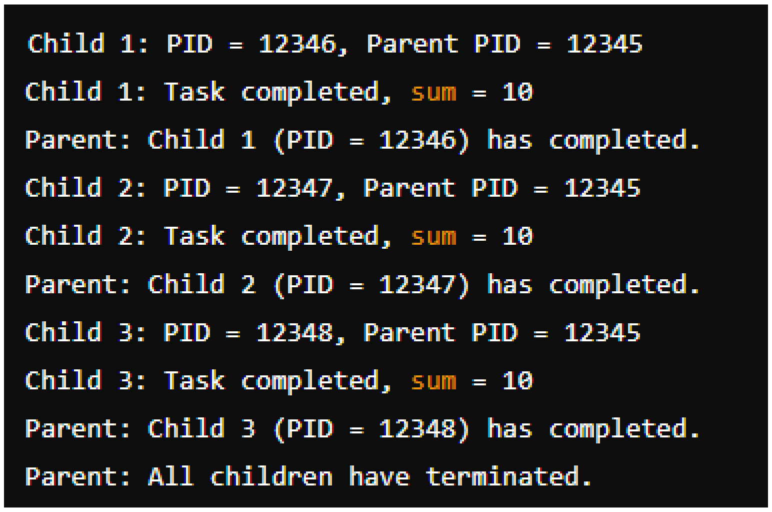
```
Child 1: Global variable = 5
Child 2: Global variable = 10
Parent: Final global variable = 0
```

II. Write a program that creates a single child process and then creates another child process from the first child. Each child process should print its PID and PPID. The parent process should print the PIDs of all active processes (including itself and its children) after both children have been created. You are not allowed to use wait(). Parent process details should be printed after child process runs. Output is given below: **[Estimated Time: 25 min]**



```
Child 1: PID = 12346, PPID = 12345
Child 2: PID = 12347, PPID = 12346
Parent: PID = 12345
Parent: Child 1 PID = 12346
Parent: Child 2 PID = 0
```

III. Write a program that creates three child processes sequentially. Each child process should print its PID, perform a simple task (like a loop or arithmetic operation), and then exit. The parent process should wait for each child to complete before creating the next one. After all child processes have finished, the parent should print a final message indicating that all children have terminated. You are allowed to use fork() statement only once. It means in whole program there should be only one fork() system call but have three childs. Output is given below: **[Estimated Time: 30 mins]**



```
Child 1: PID = 12346, Parent PID = 12345
Child 1: Task completed, sum = 10
Parent: Child 1 (PID = 12346) has completed.
Child 2: PID = 12347, Parent PID = 12345
Child 2: Task completed, sum = 10
Parent: Child 2 (PID = 12347) has completed.
Child 3: PID = 12348, Parent PID = 12345
Child 3: Task completed, sum = 10
Parent: Child 3 (PID = 12348) has completed.
Parent: All children have terminated.
```

Task Submission Guidelines:

1. **Make a word document and paste the code with output's screenshot there and save it as pdf or odt.**
2. **Include your name and roll no. at the front page.**
3. **Files other than pdf or odt will not be accepted and will be marked as 0.**