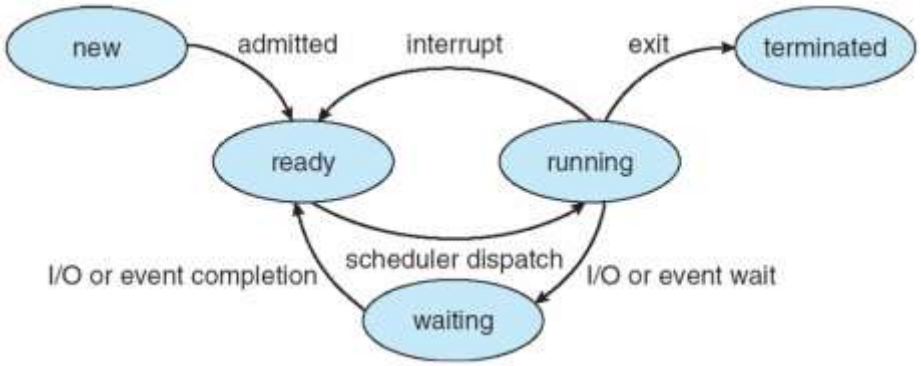
Xi'an Jiaotong-Liverpool University 西交利物浦大学

CPT104 - Operating Systems Concepts

Lab 6

System Calls for Process Management

fork(), wait() and exit() System Calls



Each process has a unique positive (nonzero) process ID (**PID**).

From a programmer's perspective, we can think of a process as being in one of three states:

Running. The process is either executing on the CPU or waiting to be executed and will eventually be scheduled by the kernel.

Stopped. The execution of the process is suspended and will not be scheduled.

Terminated. The process is stopped permanently.

A process becomes terminated for one of three reasons: (1) receiving a signal whose default action is to terminate the process, (2) returning from the main routine, or (3) calling the **exit()** function

- Follow **POSIX** standard.

Process Creation – fork() System Call

#include <unistd.h>
pid_t fork(void); Returns: 0 in child, process ID of child in parent, negative on error

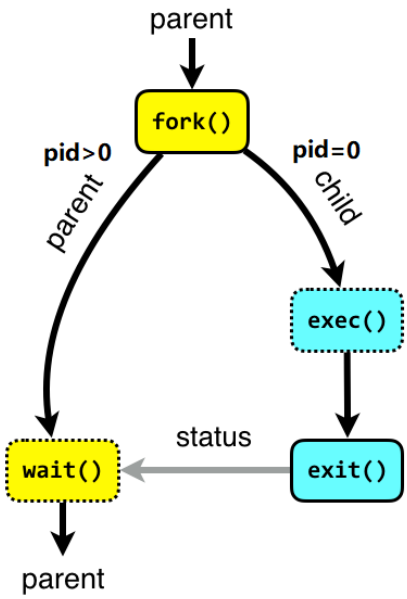
Creating your own process within a program is done with a **fork()** system call.

A newly created process is called a **child process**, and the process that is initiated to create the new process is considered a parent process.

fork() call is called once, but **returns twice**. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

A parent process can have many child processes, but a child process has only one parent process.

Mechanism of process creation



The process created to perform particular operations does a specific job in its life cycle.

Before the creation of the process done, it undergoes four steps.

1. Programmer requests the process be created by the program
2. System initialization
3. Batch job initialization
4. Execution of the fork() system call by the running process

The built-in **fork()** system call creates its own process. **The return type of this system call is an integer**.

It returns the three types of values.

If the child process is created successfully, it returns 0.

If the parent process is successfully created, it returns a positive value.

If the process is unable to create it, a negative value is returned.

exec() is used to replace the program executed by a process. The child may use exec after a fork to replace the process' memory space with a new program executable making the child execute a different program than the parent.

Wait() System Call

In some situations, a process needs to wait for resources or for other processes to complete execution. A common situation that occurs during the creation of a child process is that the parent process needs to wait or suspend until the child process execution is completed. After the child process execution completes, the parent process resumes execution.

The work of the wait system call is to suspend the parent system call until its child process terminates.

The following shows the syntax.

pid_t wait(int *status)

This system call takes the child status as an argument and returns the terminated child process ID.

If you don't want to give the child status, you can use the **NULL** value.

Exit() System Call

An **exit()** system call exits the calling process without executing the rest of the code that is present in the program.

The return of this system call is void. It doesn't return anything on execution.

The following shows the syntax.

void exit(int status)

status takes the value that is returned to the parent process.

Header files:

<stdio.h> library uses what are called streams to operate with physical devices such as keyboards, printers, terminals or with any other type of files supported by the system.

<unistd.h> defines system calls including **fork(), getpid(), getppid()** .

<sys/types.h> defines **pid_t** definition. The **pid_t** data type is a signed integer type which is capable of representing a **process ID**.

<sys/wait.h> defines system call **wait()**

<stdlib.h> defines system call **exit()**

The **getpid()** function **returns the PID of the calling/current process**.

The **getppid()** function **returns the PID of its parent** (i.e., the process that created the calling process).

Whenever, we want to declare a variable that is going to be deal with the process ids, we can use **pid_t** data type.

The type of **pid_t** data is a signed integer type (**signed int** or we can say **int**).

For practice, use **LMO**

Example 1. C program using fork() and pid_t data type.

pid_t data type stands for process identification and it is used to represent process ids.

The header file which is required to include in the program to use **pid_t**

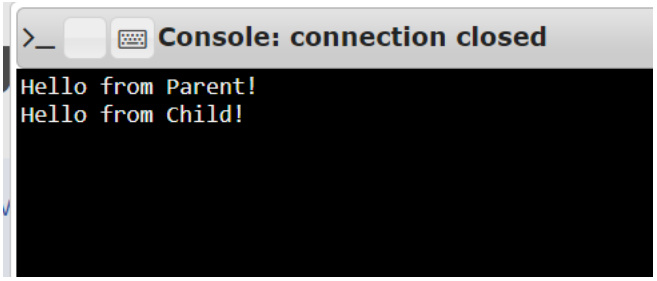
fork() creates a new process by duplicating the calling process

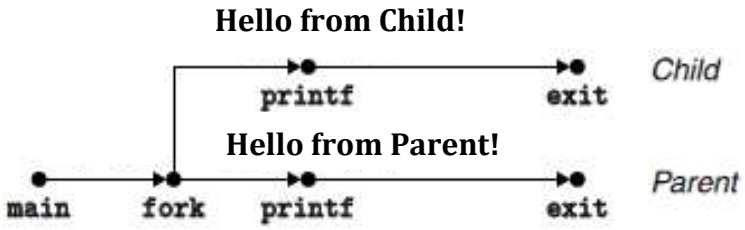
parent process created. the fork() has executed successfully

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

int main(){
    pid_t pid;
    pid=fork();
    if(pid!=0){
        printf("Hello from Parent!\n");
    }else{
        printf("Hello from Child!\n");
}
return 0;
}
```

Output





- what if put printf() before return 0 ?

Example 2. C program using fork() and **NO** data type **pid_t**

then No need the library <sys/types.h>

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

int main() {
    int pid;
    pid = fork();
    if(pid > 0){
        printf("Parent Process is created\n");
    }else if(pid == 0){
        printf("Child Process is created\n");
    }
    return 0;
}
```

Output

```
~ $
~ $ ./program
Parent Process is created
Child Process is created
~ $
```

Example 3. C program to get parent process ID and child process ID.

There is **exit()** system call.

This code prints the parent and child process ID and exits the program without executing the last *printf* statement.
This is because the **exit()** system call has exited the parent and child processes, and there is no process left to execute the last *printf* statement, so it doesn't print to the console screen.

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

int main(){
    pid_t process;
    process = fork();
    if(process == 0){
        // process == 0 means child process created
        // getpid() returns process id of calling process
        // here it will return process id of child process
        printf("Child Process ID: %d\n", getpid());
        exit(0);
    }else{
        // Prints the Parent Process ID.
        printf("Parent Process Id: %d\n", getppid());
        exit(0);
    }
    printf("Processes are exited and this line will not print\n");
    return 0;
}
```

Output

```
~ $
~ $ ./program
Parent Process Id: 43
Child Process ID: 547
~ $
```

Example 4. C program using wait() System Call

In this program, the parent will enter **wait state** until child completes.

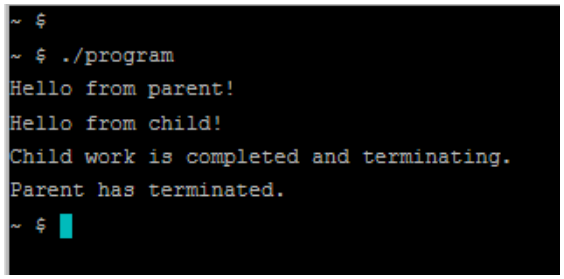
When the parent process enters a wait state, the child process enters the action to execute its assigned task. Once the child task is completed and terminated, the parent completes the remaining tasks that are assigned to it.

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

int main() {
    pid_t process;
    process= fork();
    if (process  > 0) {
       // process > 0 means parent process created
       printf("Hello from parent!\n");
       wait(NULL);
       printf("Parent has terminated.\n");

    }else if(process == 0){
       // process == 0 means child process created
       printf("Hello from child!\n");
       printf("Child work is Completed and terminating.!\n");
    }
    return 0;
}
```

Output



Reference book

Advanced Programming in the UNIX Environment, 3rd Edition
by W. Stevens (Author), Stephen Rago (Author)

For more information: refer to book 8.3, 8.5, 8.6, 7.3

We now continue with the **Lab Exercise**

- Just like the Lab Example, you will use the

  https://remisharrock.github.io/sysbuild/#/VM

to write and test your code;

- You will also need to submit your code using the LMO VPL.