CpSc 360: Distributed and Network Programming

Concurrent Programming

James Wang

Textbook Chapter 4, 5

Fork()

- System call fork() is used to create processes. It takes no arguments and returns a process ID.
- The purpose of fork() is to create a *new* process, which becomes the *child* process of the caller.
- After a new child process is created, *both* processes will execute the next instruction following the *fork()* system call.
- Therefore, we have to distinguish the parent from the child.

2

Parent or Child?

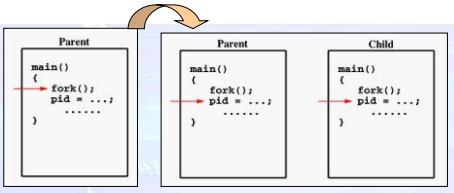
- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero to the newly created child process.
- fork() returns a positive value, the *process ID* of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer.
- The child process can use function getpid() to retrieve the process ID assigned to this process.

3

Process Duplication

If the call to fork() is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call. In this case, both processes will start their execution at the assignment statement as shown below:



4

After Duplication

- Both processes start their execution right after the system call fork().
- Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces.
- Since every process has its own address space, any modifications will be independent of the others.
- In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

5

Example

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
    pid_t  pid;
    int    i;
    char   buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

6

Output

```
.....
This line is from pid 3456, value 13
This line is from pid 3456, value 14
.....
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
.....
This line is from pid 3456, value 21
This line is from pid 3456, value 22
.....
```

- **Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way.**
- **The order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.**

7

Who Am I?

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

#define   MAX_COUNT  200

void  ChildProcess(void);
void  ParentProcess(void);

void  main(void)
{
    pid_t  pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}
```

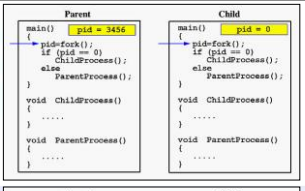
```
void  ChildProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("Child, value = %d\n", i);
    printf("   *** Child is done ***\n");
}

void  ParentProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("Parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}
```

8

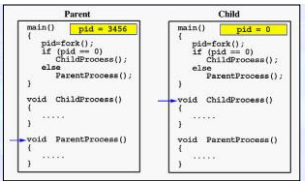
What is happening in processes?

After fork()

before if



After if



9

wait() a moment!!

- **Purpose:** Wait for child process to complete.
- **This function blocks the calling process until one of its *child* processes exits or a signal is received.**
- **wait() takes the address of an integer variable and returns the process ID of the completed process.**
- **Some flags that indicate the completion status of the child process are passed back with the integer pointer.**
  - If there are at least one child processes running when the call to **wait()** is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
  - If there is no child process running when the call to **wait()** is made, then this **wait()** has no effect at all. That is, it is as if no **wait()** is there.

http://www.cs.clemson.edu/~jzwang/0608360/waitsample.c

10

Zombie Process

- **When a child exits, its process becomes a "*zombie*" until its parent process either dies or calls wait() for it.**
- **By a "zombie", we mean that it takes up no resources, and doesn't run, but it is just being maintained by the operating system so that when the parent calls wait(), it will get the proper information.**
- **wait() returns whenever a child exits. If a process has more than one child, then you can't force wait() to wait for a specific child. You simply get whichever child exits first.**

11

waitpid()

*pid_t waitpid(pid_t pid, int *statloc, int options);*

- **waitpid() gives us more control over which process to wait for and whether or not to block.**
- **the pid argument lets us specify the process ID. -1 means wait for first child to terminate.**
- **the options argument lets us specify additional options.**
- **The most common option is WNOHANG. This option tells the kernel not to block if there is no terminated children.**

12

exec()

- The created child process does not have to run the same program as the parent process does.
- The exec type system calls allow a process to run any program files, which include a binary executable or a shell script.

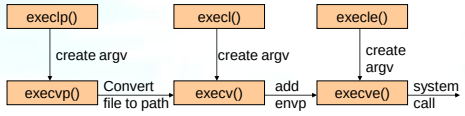
13

exec() variants

- *execl(char *path, char *arg0, char *arg1, ..., char *argn, NULL):* **This uses your current envp, and lets you specify the argv as parameters, rather than building an array of pointers. Path must specify the path name exactly.**
- *execv(char *path, char **argv):* **This is just like execve, but uses your current envp.**
- *execle(char *path, char *arg0, char *arg1, ..., char *argn, NULL, char **envp):* **This is just like execl, but you must specify envp.**
- *execve(char *path, char **argv, char **envp).*
- *execlp(char *path, char *arg0, char *arg1, ..., char *argn, NULL):* **This is just like execl, except that if path is a relative filename, then the directories in your PATH variable will be searched to find path.**
- *execvp(char *path, char **argv):* **This is just like execv, except that the PATH variable will be searched to find path.**

14

Relationship Among exec Functions



- These functions return to caller only if an error occurs. Otherwise control passes to the start of the new program.
- Only execve() is a system call within the kernel and the other five are library functions that call execve().

15

execve() example

```
#include <stdio.h>

main(int argc, char **argv, char **envp)
{
  char *newargv[3];
  int i;

  newargv[0] = "cat";
  newargv[1] = "exec2.c";
  newargv[2] = NULL;

  i = execve("/bin/cat", newargv, envp);
  perror("exec2: execve failed");
  exit(1);
}
```

16

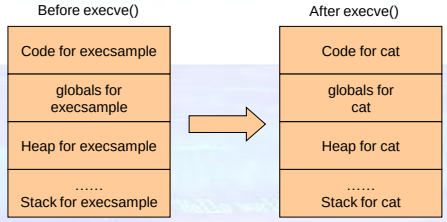
What happened?

- Execve() assumes that path is the name of an executable file. Argv is an array of null-terminated strings, such that the last element is NULL, and envp is another null-terminated array of null-terminated strings.
- Execve() overwrites the current process so that it executes the file in path with the arguments in argv, and the environment variables in envp.
- Execve() does not return unless it encounters an error, such as the file in "path" not existing, or not being an executable file.

17

Memory State

- **Suppose we compile the sample program to executable "execsample". Then we execute it with no arguments. The memory state changes after execve() is called:**



18

Effect of execve()

- You'll notice that everything concerning execsample.c is gone.
- This is because the state of memory has been overwritten to run cat. There is no trace of execsample left.
- This is why execve() cannot return if it is sucessful -- the state to which it might have returned has been overwritten. It is gone.
- When cat exits, the operating system simply destroys the process.
- So how come when you execute cat in the shell it looks like it returns to the shell?
  - This is because the shell calls fork - exec - wait.

Program Reading

Please read sample programs:

- tcpcliserv/tcpcli04.c
- tcpcliserv/tcpserv04.c
- tcpcliserv/tcpchldwaitpid.c