System Calls fork(), getpid() and getppid() example fork_example.c

```
#include <stdio.h>
main()
{  int pid;
   printf("I'm the original process with PID %d and PPID %d. \n", getpid(), getppid() );
   pid = fork(); /* Duplicate. Child and parent continue from here */
   if ( pid!= 0 ) /* pid is non-zero, so I must be the parent */ ← Parent and Child execute from this point
   {
      printf("I'm the parent process with PID %d and PPID %d. \n", getpid(), getppid() );
      printf("My child's PID is %d \n", pid );
   }
```

System Calls fork(), getpid() and getppid() example

```
 else /* pid is zero, so I must be the child */
    {
        printf("I'm the child process with PID %d and PPID %d. \n", getpid(), getppid() );
    }
    printf("PID %d terminates. \n", getpid() ); /* Both processes execute this */
}
```

System Calls fork(), getpid() and getppid() example

$ fork_example

I’m the original process with PID 13292 and PPID 13273.

I’m the parent process with PID 13292 and PPID 13273.

My child’s PID is 13293.

I’m the child process with PID 13293 and PPID 13292.

PID 13293 terminates. ---> child terminates.

PID 13292 terminates. ---> parent terminates.

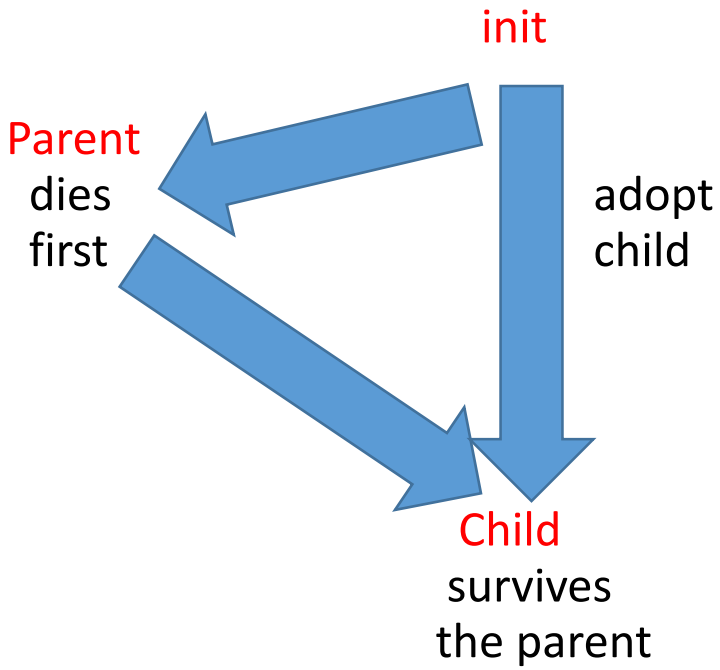
WARNING:

it is dangerous for a parent process to terminate without waiting for the death of its child.

The only reason our program doesn’t wait for its child to terminate is because we haven’t yet used the “wait()” system call!.

Orphan Process

- If parent process does not wait for child and it first terminates leaving child process orphan
  - Orphan processes are adopted by init process which started the parent (i.e. parent of parent)



Orphan Process Example orphan_child.c

```
   else /* pid is zero, so I must be the child */
   {
      sleep(10); // add sleep so child process will terminate later then parent
      printf("I'm the child process with • PID 13364 terminates. PID %d and PPID %d. \n", getpid(), getppid() );
   }
   printf("PID %d terminates. \n", getpid() ); /* Both processes execute this */
}
```

$ orphan ---> run the program.

I'm the original process with PID 13364 and PPID 13346.

I'm the parent process with PID 13364 and PPID 13346.

PID 13364 terminates.

I'm the child process with PID 13365 and PPID 1. ---> orphaned!

PID 13365 terminates.

$

Note the change in PPID for child processes

System Call wait() to avoid orphans

wait_example.c

```
#include <stdio.h>
main()
{   int pid, status;
    printf("I'm the original process with PID %d and PPID %d. \n", getpid(), getppid() );
    pid = fork(); /* Duplicate. Child and parent continue from here */
    if ( pid!= 0 ) /* pid is non-zero, so I must be the parent */
    {
        printf("I'm the parent process with PID %d and PPID %d. \n", getpid(), getppid() );
        printf("My child's PID is %d \n", pid );
        childPid = wait( &status ); // add wait in parent process to wait for child process, child will not become orphan
    }
    else.....
```

Zombie Process zambi_example.c

```
#include <stdio.h>
main()
{
   int pid;
   pid = fork(); /* Duplicate */
   if ( pid!= 0 ) /* Branch based on return value from fork() */
   {
      while (1) /* Never terminate, and never execute a wait() */
      sleep(1000);
   }
   else
   {
      exit(42); /* Exit with a silly number */
   }
}
```

```
$ zombie & ---> execute the program in the background using &
[1] 13545
$ ps ---> obtain process status.
PID   TT STAT TIME COMMAND
13535 p2 S    0:00 -ksh (ksh) ---> the shell.
13545 p2 S    0:00 zombie ---> the parent process.
13546 p2 Z    0:00 <defunct> ---> the zombile child.
13547 p2 R    0:00 ps

$ kill 13545 ---> kill the parent process.
[1] Terminated zombie

$ ps ---> notice that the zombie is gone now.
PID   TT STAT TIME COMMAND
13535 p2 S    0:00 -ksh (ksh)
13547 p2 R    0:00 ps
```

```
jayprakash@jayprakash-System-AsusPG500:~$ gcc zombie.c -o zombie
jayprakash@jayprakash-System-AsusPG500:~$ ./zombie &
[1] 2825
jayprakash@jayprakash-System-AsusPG500:~$ ps
  PID TTY          TIME CMD
 2806 pts/10   00:00:00 bash
 2825 pts/10   00:00:00 zombie
 2826 pts/10   00:00:00 zombie <defunct>
 2827 pts/10   00:00:00 ps
jayprakash@jayprakash-System-AsusPG500:~$ gcc zombie.c -o zombie-1
jayprakash@jayprakash-System-AsusPG500:~$ ./zombie-1 &
[2] 2833
jayprakash@jayprakash-System-AsusPG500:~$ ps
  PID TTY          TIME CMD
 2806 pts/10   00:00:00 bash
 2825 pts/10   00:00:00 zombie
 2826 pts/10   00:00:00 zombie <defunct>
 2833 pts/10   00:00:00 zombie-1
 2834 pts/10   00:00:00 zombie-1 <defunct>
 2835 pts/10   00:00:00 ps
jayprakash@jayprakash-System-AsusPG500:~$
```

Race Condition racecondition_example.c

You can call this as critical section

```
static void charatatime(char *str)
{
char *ptr;
int c;
setbuf(stdout, NULL);
for (ptr=str;c=*ptr++;) putc(c,stdout);
}
main()
{
pid_t pid;
if ((pid = fork())<0) printf("fork error!\n");
else if (pid ==0) charatatime("12345678901234567890\n");
else charatatime("abcdefghijklmnopqrstuvwxyz\n");
}
```

```
$ test_fork
12345678901234567890
abcdefghijklmnopqrstuvwxyz
$ test_fork
12a3bc4d5e6f78901g23hi4567jk890
Lmnopqrstuvwxyz
```

Need to have parent wait for child or child wait for parent to complete the critical section code. This can be done using signals which will study in next chapter

Additional Status Info from wait() System Call

childpid = wait(&wstatus); → returns the exit status from child which can further be inspected using these macros

WIFEXITED(wstatus) → returns true if child terminated normally

WEXITSTATUS(wstatus) → returns exit status (least significant 8 bits)

WIFSIGNALED(wstatus) → returns true if child process was terminated by a signal

WTERMSIG(wstatus) → returns the number of signal

WCOREDUMP(wstatus) → returns true if child produced a core dump

WIFSTOPPED(wstatus) → returns true if child was stopped by a signal

WSTOPSIG(wstatus) → returns the signal number which caused child to stop

WIFCONTINUED(wstatus) → returns true if child was resumed with SIGCONT signal

exec() family of System Calls

When fork() creates a child process with a copy of same code, data etc as parent process but if you need to run another process as child process then →

A process may replace its current code, data, and stack with those of another executable by using one of the "exec()" family of system calls

When a process executes an "exec()" system call, its PID and PPID numbers stay the same - only the code that the process is executing changes.

System Call:

int execl(const char* path, const char* arg0, const char* arg1,…, const char* argn, NULL)

int execv(const char* path, const char* argv[])

int execlp(const char* path, const char* arg0, const char* arg1, …, const char* argn, NULL)

int execvp(const char* path, const char* argv[])

The "exec()" family of system calls replaces the calling process' code, data, and stack with those of the executable whose pathname is stored in path.

Difference in exec() System Calls

- “execlp()” and “execvp()” use the $ PATH environment variable to find path.
  - If the executable is not found, the system call returns a value of -1; otherwise, the calling process replaces its code, data, and stack with those of the executable and starts to execute the new code.
- “execl()” and “execlp()” invoke the executable with the string arguments pointed to by arg1 through argn.
  - arg0 must be the name of the executable file itself, and the list of arguments must be null terminated.
- “execv()” and “execvp()” invoke the executable with the string arguments pointed to by argv[1] to argv[n], where argv[n+1] is NULL.
  - argv[0] must be the name of the executable file itself.

System Call exec() example exec_example.c

the program displays a small message and then replaces its code with that of the “ls”.

```
#include <stdio.h>
main()
{
   printf(“I’m process %d and I’m about to exec an ls -l \n”, getpid() );
   execl( “/bin/ls”, “ls”, “-l”, NULL ); /* Execute ls */
   printf(“This line should never be executed \n”);
}
```

$ myexec ---> run the program.

```
I’m process 13623 and I’m about to exec an ls -l
total 125
-rw-r--r-- 1 glass   277 Feb 15 00:47 myexec.c
-rwxr-xr-x 1 glass 24576 Feb 15 00:48 myexec
```