고려대학교
KOREA UNIVERSITY

KU-The Future

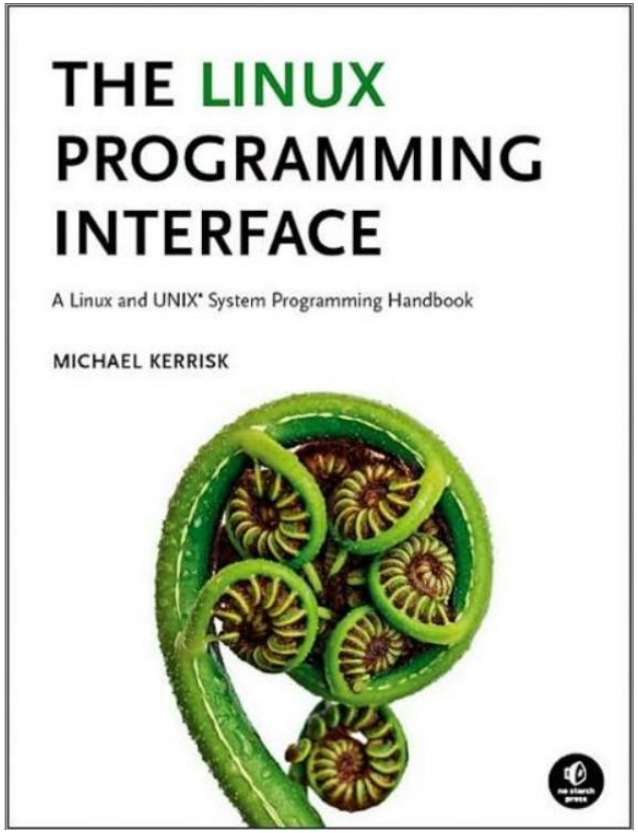
Process

Jinwon Jeong

jin4812@korea.ac.kr

Reference

- The Linux Programming Interface
  - Published in October 2010
  - No Starch Press
  - ISBN 978-1-59327-220-3



Process

Overview of system calls

- fork(), exec(), wait(), and exit().
- Each of these system calls has variants.

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

https://www.guru99.com/c-gcc-install.html

Overview of system calls

- **Fork()**
  - Allows one process, the parent, to **create a new process**, the **child**.
  - This is done by **making the new child process** an (almost) exact duplicate of the parent.
  - The child obtains copies of the **parent's stack, data, heap, and text segments**.

Overview of system calls

- **Exit()**
  - **Terminates** a process.
  - Making all resources (memory, open file descriptors, and so on) used by the process available for subsequent reallocation by the kernel.
  - The ***status*** **argument is an integer** that determines the termination status for the process.
  - Using the *wait()* system call, the parent can **retrieve this status**.

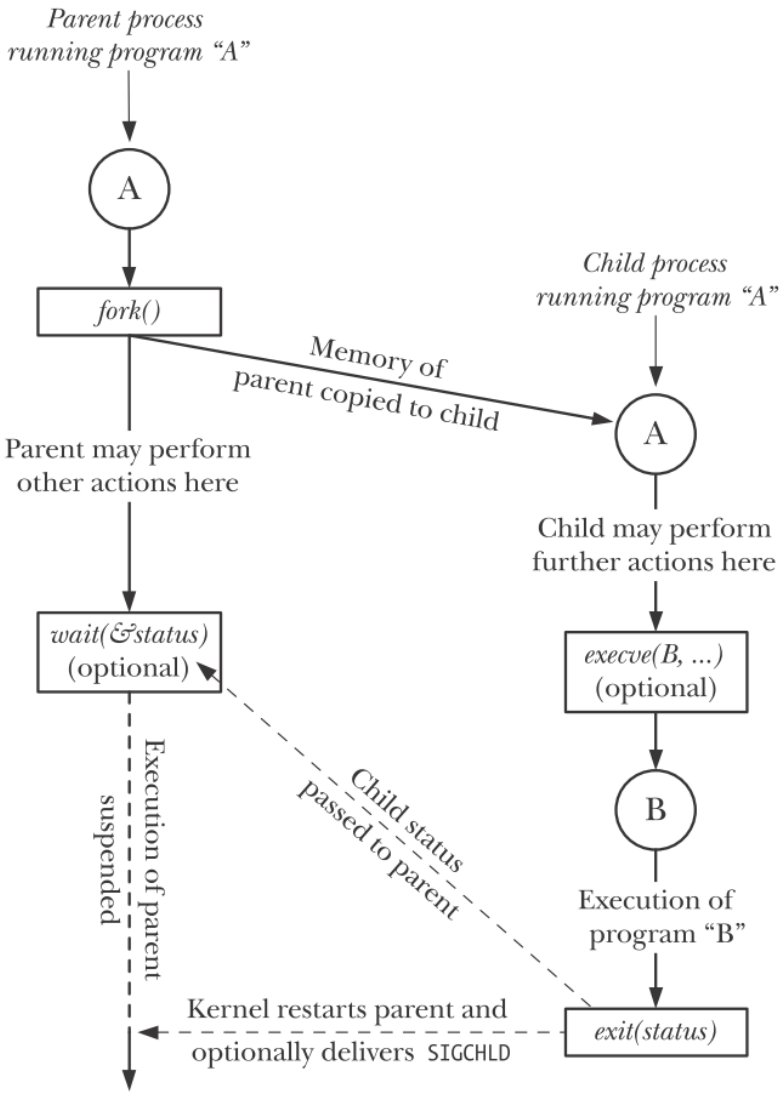
Overview of system calls

- **Wait()**
  - (purpose) if a **child** of this process **has not yet terminated** by calling *exit()*, then *wait()* **suspends execution** of the process until on of its children has terminated.
  - (purpose) the termination status of the child is returned in the status argument of *wait().*
  - #include <wait.h>

Overview of system calls

- **Exec()**
  - **Loads a new program** (*pathname*, with argument list *argv*, and environment list *envp*) into a process's memory.
  - The **existing program text is discarded**, and the stack, data, and heap segments are freshly created for the new program.
  - When Exec() invocation success, it doesn't return, but when fail, it returns -1.

Overview of system calls



Creating a New Process: *fork()*

- The fork() system call **creates a new process**, the ***child***, which is an almost exact duplicate of the calling process, the *parent.*

```
#include <unistd.h>

pid_t fork(void);

                    In parent: returns process ID of child on success, or -1 on error;
                                in successfully created child: always returns 0
```

Creating a New Process: *fork()*

- The key point to understanding *fork()* is to realize that after it has completed its work, **two processes exist**, and, in each process, **execution continues** from the point where *fork()* returns.

- The two processes are **executing the same program text**, but they have **separate copies** of the stack, data, and heap segments.

Creating a New Process: *fork()*

- The child's stack, data, and heap segments are initially exact duplicates of the corresponding parts **the parent's memory**.
- We can **distinguish** the two processes via the value returned from *fork().*
- For the parent, *fork()* **returns the process ID** of the newly created child.

- For the child, ***fork()* returns 0**
  - If necessary, the child can obtain its own process ID using ***getpid()***, and the process ID of its parent using ***getppid()***.

Creating a New Process: *fork()*

- The following idiom is sometimes employed when calling ***fork()***.

```
pid_t childPid;             /* Used in parent after successful fork()
                               to record PID of child */
switch (childPid = fork()) {
case -1:                    /* fork() failed */
    /* Handle error */

case 0:                     /* Child of successful fork() comes here */
    /* Perform actions specific to child */

default:                    /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```

Creating a New Process: *fork()*

- It is important to realize that after a ***fork()***,

  it is **indeterminate** which of the two processes is next

  scheduled to use the CPU.

  – In poorly written programs, this indeterminacy can lead to

  errors known as **race conditions**.

Creating a New Process: *fork()*

- **Copy on write(COW)**
  - When parents want a make child, copying all of parent's pages in memory is makes the overhead.
  - Just give a pointer that indicates the parent's pages to child.
  - When child wants a modify the value inside the pages, COW gives a copy of that page to child for integrity.

Creating a New Process: *fork()*

fork1.c

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

static int idata = 111; /*Allocated in data segment*/

int main(int argc, char *argv[])
{
        int istack=222; /*Allocated in stack segment*/
        pid_t childPid;

        switch(childPid = fork())
        {
                case -1:
                        exit(childPid);
                case 0:
                        idata*=3;
                        istack*=3;
                        break;
                default:
                        sleep(3);
                        break;
        }
        printf("PID=%ld %s idata=%d istack=%d₩n", (long)getpid(), (childPid==0) ?
        "(child)" : "(parent)", idata, istack);

        exit(childPid);
        return 0;
}
```

Creating a New Process: *fork()*

- The use of *sleep()* (in the code executed by the parent) in this program permits the child to be scheduled for the CPU before the parent.

```
root@kali:~# vi fork1.c
root@kali:~# gcc -o fork1 fork1.c
root@kali:~# ./fork1
PID=3876 (child) idata=333 istack=666
PID=3875 (parent) idata=111 istack=222
```

- The child process gets its own copy of the stack and data segments at the time of the *fork()*, and it is able to modify variables in these segments without affecting the parent.

Creating a New Process: *fork()*

fork2.c

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

int main(int argc, char *argv[])
{
        pid_t pid;
        int i=0;

        i++;
        printf("before(%d)₩n", i);
        pid = fork();

        if(pid == 0)
                printf("child process(%d)₩n", ++i);
        else if(pid>0)
                printf("parent(%d)₩n", --i);
        else
                printf("fail");

        return 0;
}
```

Creating a New Process: *fork()*

fork3.c

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

#define max_pid 5
int i=5;
int main(){
        int j;
        //int status;
        pid_t pid[max_pid];

        for(j=0; j< max_pid; j++){
                pid[j] = fork();
                switch(pid[j]){
                        case -1:
                                perror("fork failed!\n");
                                break;
                        case 0 :
                                i--;
                                printf("child : %d, i = %d \n", getpid(), i);
                                exit(2);
                                break;
                        default :
                                i++;
                                //wait(&status);
                                printf("parent : %d, i = %d\n", getpid(), i);
                }
        }
        printf("last value i on parent's code : %d, i = %d \n",getpid(), i);
        return 0;
}
```

Executing a New Program: exec()

- The exec() system call **loads a new program** into a process's memory.

- During this operation, the **old program is discarded**, and the process's stack, data, and heap are **replaced** by those of the new program.

Executing a New Program: execl()

```
int execl(const char *pathname, const char *arg, ...
                /* , (char *) NULL */);
```

None of the above returns on success; all return −1 on error

- The **pathname** argument contains the pathname of the new program to be loaded into the process's memory.
- The **arg** argument specifies the command-line arguments to be passed to the new program.
- After an execl(), the **process ID** of the process remains the **same**, because the same process continues to exist.

Executing a New Program: exec()

- **The exec() Library Functions**

```
#include <unistd.h>

int execle(const char *pathname, const char *arg, ...
                /* , (char *) NULL, char *const envp[] */ );
int execlp(const char *filename, const char *arg, ...
                /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
                /* , (char *) NULL */);
```

None of the above returns on success; all return −1 on error

Executing a New Program: execl()

execl1.c

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

int main(){
        printf("before executing ls -l\n");
        execl("/bin/ls", "ls", "-l", (char*)0);
        printf( "where is my printf() result???\n");
        return 0;
}
```

Fork() + execl()

execl2.c

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

int main(){
        pid_t pid;
        printf("start\n");
        pid = fork();

        if(pid > 0){
                printf("parent\n");
                sleep(1);
        }else if(pid == 0){
                printf("child");
                execl("/bin/ls","ls","-l",(char*)0);
                printf("fail to execute ls -l \n");
        }else{
                printf("paraent fail to fork \n");
        }
        printf("bye\n");
        return 0;

}
```

Terminating a Process: *exit()*

- The *status* argument given to *exit()* defines the ***termination status*** of the process, which is available to the parent of this process when it calls ***wait().***

```
#include <unistd.h>

void exit(int status);
```

- By convention, a termination status of **0** indicates that a process completed **successfully**, and a **nonzero** status value indicates that the process terminated **unsuccessfully**.

Waiting on a Child Process: *wait ()*

- waits for **one of the children** of the calling process **to terminate** and

- returns the **termination status** of that child in the buffer pointed to by status.

```
#include <sys/wait.h>

pid_t wait(int *status);
```

Returns process ID of terminated child, or −1 on error

Waiting on a Child Process: *wait ()*

wait1.c

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

int main(int argc, char *argv[])
{
        pid_t pid;
        int status;

        pid = fork();

        if(pid>0)
        {
                /* parent process */
                printf("parent waiting..₩n");
                wait(&status); // status = return value of child * 256
        }
        else if(pid==0)
        {
                /* child process */
                sleep(1);
                printf("child: bye!₩n");
                exit(2);
        }
        else
                printf("parent : fail to fork₩n");

        printf("bye!₩n");
        return 0;
}
```

Waiting on a Child Process: *wait ()*

fork3.c

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

#define max_pid 5
int i=5;
int main(){
        int j;
        //int status;
        pid_t pid[max_pid];

        for(j=0; j< max_pid; j++){
                pid[j] = fork();
                switch(pid[j]){
                        case -1:
                                perror("fork failed!\n");
                                break;
                        case 0 :
                                i--;
                                printf("child : %d, i = %d \n", getpid(), i);
                                exit(2);
                                break;
                        default :
                                i++;
                                //wait(&status);
                                printf("parent : %d, i = %d\n", getpid(), i);
                }
        }
        printf("last value i on parent's code : %d, i = %d \n",getpid(), i);
        return 0;
}
```

Erase the annotation on fork3.c

Waiting on a Child Process: *waitpid ()*

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status_ptr, int options);
```

- Waits for specific child process to be end
- First : specific child process
- Second : child process's status
- Third : waitpid()'s option. 0, WNOHANG …

Waiting on a Child Process: *waitpid ()*

- WIFEXITED(status) : returns **true** if the child terminated normally
- WEXITSTATUS(status) : returns the **exit status** of the child.
- WTERMSIG(status) : returns the **number of the signal** that caused the child process to terminate.

Waiting on a Child Process: *waitpid ()*

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

int main()
{
   int   counter  = 1;
   int   status;
   pid_t pid1,pid2;
   pid_t pid_child;
   printf( "parent : Hello, my ID is %d\n",getpid());
   printf( "parent creates child 1.\n");
   pid1   = fork();

   switch(pid1){
      case -1  :
                printf( "fail, child 1\n");
                return -1;
      case 0   :
                 printf( "child 1(%d) starts counting...!\n",getpid());
                while( 10 > counter ){
                   printf( "child 1: %d\n", counter++);
                   sleep(1);
                 }
                 return 1;
      default  :

                 printf( "parent creates child 2.\n");
                 pid2   = fork();
                 switch(pid2){
                   case -1  :
                             printf( "fail, child 2\n");
                             return -1;
                   case 0   :

                             printf( "child 2(%d) starts counting...!\n",getpid());
                             while( 100 > counter ){
                             printf( "child 2: %d\n", counter++);
                             sleep(1);
                             }
                             return 1;

                   default  :

                             printf("parent : Im out of swith() now!! \n");
                 }
    }
```

```c
printf( "parent : I will start counting when child 1's counter value is 9...!\n");

pid_child   = waitpid(pid1, &status, 0);

printf("parent : im start counting now... \n");
while(1){
    printf("parent : %d\n",counter++);
    sleep(1);
    if(counter == 8){
            break;
    }
}

printf( "terminated process is %d\n", pid_child);
if(WIFEXITED(status)){
       printf( "normal termination, status : %d\n", WEXITSTATUS(status));

}else{
       printf( "abnormal termination, status : %d\n", WIFSIGNALED(status));

}
kill(pid1, SIGKILL);
kill(pid2, SIGKILL);
return 0;
}
```

Waiting on a Child Process: *waitpid ()*

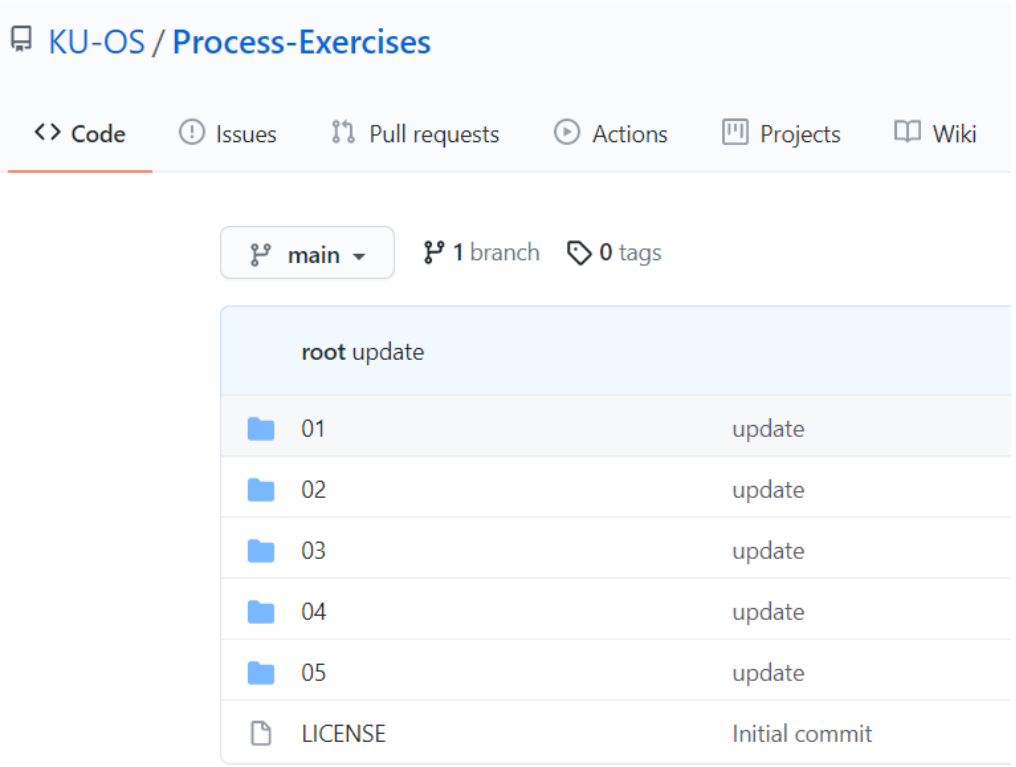
```c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>

int main()
{
   int   counter  = 1;
   int   status;
   pid_t pid1,pid2;
   pid_t pid_child;
   printf( "parent : Hello, my ID is %d\n",getpid());
   printf( "parent creates child 1.\n");
   pid1   = fork();

   switch(pid1){
      case -1  :
                printf( "fail, child 1\n");
                return -1;
      case 0   :
                printf( "child 1(%d) starts counting...!\n",getpid());
                while( 10 > counter ){
                   printf( "child 1: %d\n", counter++);
                   sleep(1);
                }
                return 1;
      default  :

                printf( "parent creates child 2.\n");
                pid2   = fork();
                switch(pid2){
                  case -1  :
                            printf( "fail, child 2\n");
                            return -1;
                  case 0   :

                            printf( "child 2(%d) starts counting...!\n",getpid());
                            while( 100 > counter ){
                            printf( "child 2: %d\n", counter++);
                            sleep(1);
                            }
                            return 1;

                  default  :

                            printf("parent : Im out of swith() now!! \n");
                  }
    }
```

```c
   printf( "parent : I will start counting now without block...!\n");

   pid_child   = waitpid(pid1, &status, WNOHANG);

   printf("parent : now im start counting!! \n");
   while(1){
       printf("parent : %d\n",counter++);
       sleep(1);
       if(counter == 8){
               break;
       }
   }

   printf( "terminated process is %d\n", pid_child);
   if(WIFEXITED(status)){
          printf( "normal termination, status : %d\n", WEXITSTATUS(status));

   }else{
          printf( "abnormal termination, status : %d\n", WIFSIGNALED(status));

   }
   kill(pid1, SIGKILL);
   kill(pid2, SIGKILL);
   return 0;
   }
```

Assignment

Visit → https://github.com/KU-OS/Process-Exercises

Do : 01 ~ 05



Assignment

리눅스 환경(가상머신) 또는 macOS에서 실습 진행

1) 각 문제의 빈칸에 들어갈 코드 작성(전체 코드 작성할 필요 없음)

<?1/> xxx
<?2/> xxx
…

2) 각 문제마다 출력 결과 스크린샷(계정 보이게)

```
root@kali:~# gcc -o fork1 fork1.c
root@kali:~# ./fork1
PID=10878 (child) idata=333 istack=666
PID=10877 (parent) idata=111 istack=222
```

PDF파일로 제출