Process 2

Jo, Heeseung

목차

프로세스 생성[1]

프로그램 실행 : system(3)

```
#include <stdlib.h>
int system(const char *string);
```

- 새로운 프로그램을 실행하는 가장 간단한 방법
- 실행할 프로그램명을 인자로 지정

```
01  #include <stdlib.h>
02  #include <stdio.h>
03
04  int main(void) {
05      int a;
06      a = system("ps -ef | grep bash > result.txt");
07      printf("Return Value : %d\n", a);
08
09      return 0;
10  }
```

```
# ex6_1.out

Return Value : 0
```

프로세스 생성[2]

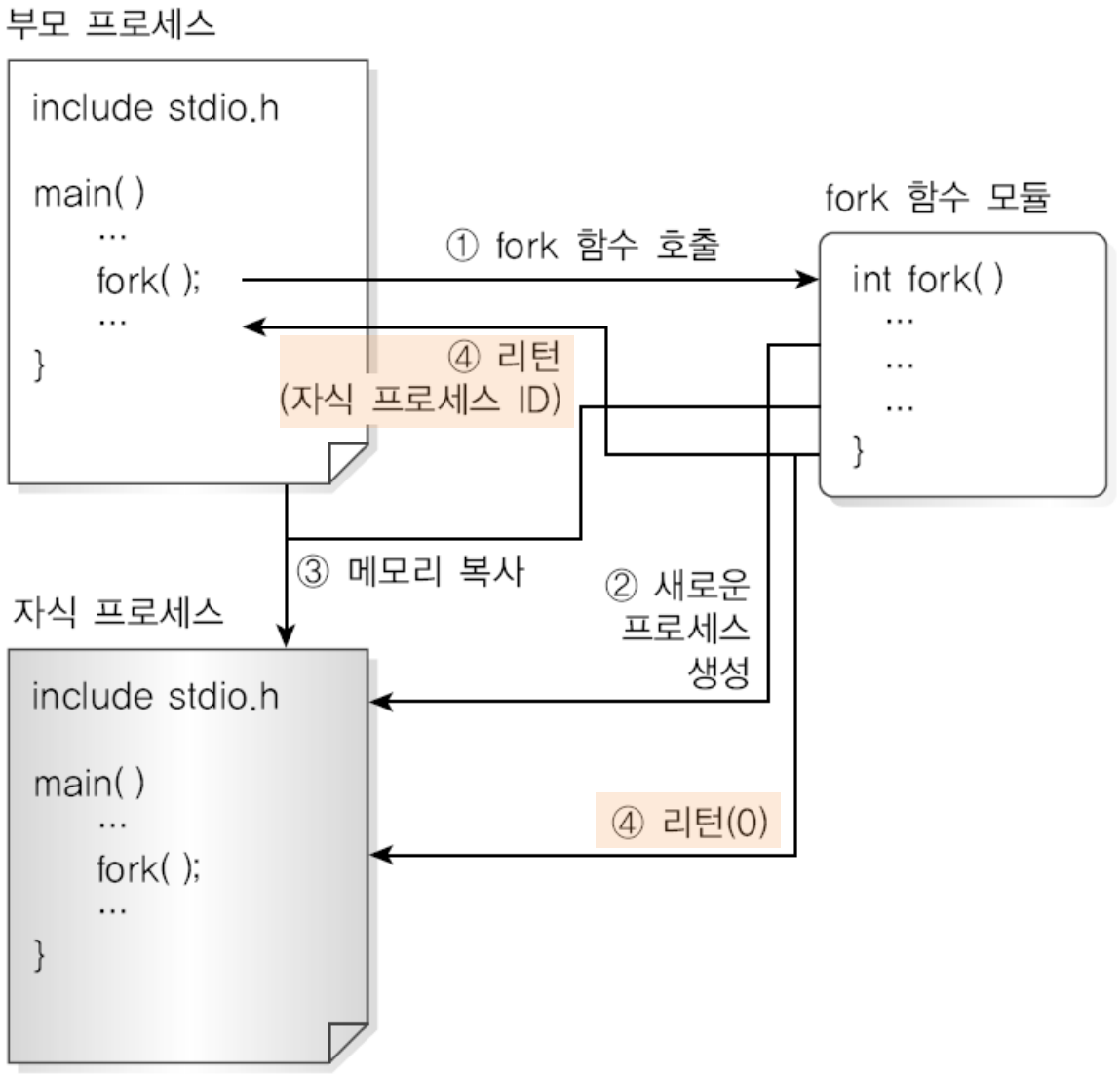
프로세스 생성: fork(2)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- 새로운 프로세스를 생성 : 자식 프로세스 (return value 0)
- fork 함수를 호출한 프로세스 : 부모 프로세스 (rv CPID)
- 자식 프로세스는 부모 프로세스의 메모리를 복사
  - RUID, EUID, RGID, EGID, 환경변수
  - 열린 파일기술자, 시그널 처리, setuid, setgid
  - 현재 작업 디렉토리, umask, 사용가능자원 제한
- 부모 프로세스와 자식 프로세스는 열린 파일을 공유
  - Read/Write에 주의

프로세스 생성[2]

프로세스 생성



[그림 6-1] fork 함수를 이용한 새로운 프로세스 생성

fork 함수 사용하기

```
...
06  int main(void) {
07      pid_t pid;
08
09      switch (pid = fork()) {
10          case -1 :  /* fork failed */
11              perror("fork");
12              exit(1);
13              break;
14          case 0 :   /* child process */
15              printf("Child Process - My PID:%d, My Parent's PID:%d\n",
16                  (int)getpid(), (int)getppid());
17              break;
18          default : /* parent process */
19              printf("Parent process - My PID:%d, My Parent's PID:%d,
                    My Child's PID:%d\n", (int)getpid(), (int)getppid(),
                    (int)pid);
21              break;
22      }
23
24      printf("End of fork\n");
26      return 0;
27  }
```

```
# ex6_2.out
Child Process - My PID:796, My Parent's PID:795
End of fork
Parent process - My PID:795, My Parent's PID:695,
My Child's PID:796
End of fork
```

fork함수의 리턴값 0은
자식 프로세스가 실행

Exercise

Ex.

- Parent process는 2-5단, child process는 6-9단을 출력하는 프로그램을 작성

```
./a.out

Parent process - My PID:15695, My Parent's PID:15602, My
Child's PID:15696
2 x 1 = 2
2 x 2 = 4
Child Process - My PID:15696, My Parent's PID:15695
6 x 1 = 6
2 x 3 = 6
2 x 4 = 8
6 x 2 = 12                (순서가 섞여서 출력됨)
6 x 3 = 18
6 x 4 = 24
...
```

Exercise

Ex.

- 100,000까지의 모든 prime number를 출력하는 psingle 프로그램을 작성
  - time 명령을 이용하여 몇 초가 걸리는지 확인할 것

- 위의 프로그램과 동일하지만 4개의 프로세스를 이용하는 pmulti 프로그램을 작성
  - 4개의 프로세스는 구간을 나누어서 병렬처리
  - time 명령을 이용하여 몇 초가 걸리는지 확인할 것

프로세스 종료 함수[1]

프로그램 종료: exit(2)

```
#include <stdlib.h>
void exit(int status);
```

- status : 종료 상태값

프로그램 종료시 수행할 작업 예약: atexit(2)

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

- func : 종료시 수행할 작업을 지정한 함수명

exit, atexit 함수 사용하기

```
01  #include <stdlib.h>
02  #include <stdio.h>
03
04  void cleanup1(void) {
05      printf("Cleanup 1 is called.\n");
06  }
07
08  void cleanup2(void) {
09      printf("Cleanup 2 is called.\n");
10  }
11
12  int main(void) {
13      atexit(cleanup1);
14      atexit(cleanup2);
15
16      exit(0);
17  }
```

```
# ex6_3.out
Cleanup 2 is called.
Cleanup 1 is called.
```

종료시 수행할 함수 지정
지정한 순서의 역순으로 실행
(실행결과 확인)

exec 함수군 활용

exec 함수군

- exec로 시작하는 함수들로, 명령이나 실행 파일을 실행
- exec 함수가 실행되면 프로세스의 메모리 이미지는 실행파일로 교체

exec 함수군의 형태 6가지

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ..., const char *argn, (char *)0,  char
*const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0);
int execvp(const char *file, char *const argv[]);
```

- path : 명령의 경로 지정
- file : 실행 파일명 지정
- arg#, argv : main 함수에 전달할 인자 지정
- envp : main 함수에 전달할 환경변수 지정
- 함수의 형태에 따라 NULL 값 지정에 주의

execlp 함수 사용하기

```
01  #include <unistd.h>
02  #include <stdlib.h>
03  #include <stdio.h>
04
05  int main(void) {
06      printf("--> Before exec function\n");
07
08      if (execlp("ls", "ls", "-a", (char *)NULL) == -1) {
09          perror("execlp");
10          exit(1);
11      }
12
13      printf("--> After exec function\n");
14
15      return 0;
16  }
```

```
# ex6_4.out
--> Before exec function
.       ex6_1.c      ex6_3.c      ex6_4.out
..      ex6_2.c      ex6_4.c      han.txt
```

인자의 끝을 표시하는 NULL 포인터

첫 인자는 관례적으로 실행파일명 지정

메모리 이미지가 'ls' 명령으로 바뀌어 13행은 실행안됨

execv 함수 사용하기

```
01  #include <unistd.h>
02  #include <stdlib.h>
03  #include <stdio.h>
04
05  int main(void) {
06      char *argv[3];
07
08      printf("Before exec function\n");
09
10      argv[0] = "ls";
11      argv[1] = "-a";
12      argv[2] = NULL;
13      if (execv("/usr/bin/ls", argv) == -1) {
14          perror("execv");
15          exit(1);
16      }
17
18      printf("After exec function\n");
19
20      return 0;
21  }
```

```
# ex6_5.out
--> Before exec function
.     ex6_1.c   ex6_3.c   ex6_5.c     han.txt
..    ex6_2.c   ex6_4.c   ex6_5.out
```

첫 인자는 관례적으로 실행파일명 지정

인자의 끝을 표시하는 NULL 포인터

경로로 명령 지정

역시 실행 안됨

exec 함수군과 fork 함수

fork로 생성한 자식 프로세스에서 exec 함수군을 호출

- exec 함수군
  - 자식 프로세스의 메모리 이미지로 대체
  - 부모 프로세스 이미지는 사라짐
- 부모 프로세스와 다른 프로그램 실행 가능
- 부모 프로세스와 자식 프로세스가 각기 다른 작업을 수행
  - fork와 exec 함수를 함께 사용

fork와 exec 함수 사용하기

```
...
06  int main(void) {
07      pid_t pid;
08
09      switch (pid = fork()) {
10          case -1 :  /* fork failed */
11              perror("fork");
12              exit(1);
13              break;
14          case 0 :   /* child process */
15              printf("--> Child Process\n");
16              if (execlp("ls", "ls", "-a", (char *)NULL) == -1) {
17                  perror("execlp");
18                  exit(1);
19                }
20              exit(0);
21              break;
22          default :   /* parent process */
23              printf("--> Parent process - My PID:%d\n“,(int)getpid());
24              break;
25       }
27      return 0;
28  }
```

```
# ex6_7.out
--> Child Process
ex6_1.c  ex6_3.c  ex6_5.c  ex6_6_arg.c  ex6_7.out
ex6_2.c  ex6_4.c  ex6_6.c  ex6_7.c      han.txt
--> Parent process - My PID:10535
```

자식프로세스에서 execlp 함수 실행

부모프로세스는 이 부분 실행

프로세스 동기화

부모 프로세스와 자식 프로세스의 종료절차

- 부모 프로세스와 자식 프로세스는 순서와 상관없이 실행하고 먼저 실행을 마친 프로세스는 종료
- 종료절차가 제대로 진행되지 않으면 좀비 프로세스 발생

좀비프로세스 (zombie process)

- 실행을 종료하고 자원을 반납한 자식 프로세스의 종료 상태를 부모 프로세스가 가져가지 않으면 좀비 프로세스 발생
- 좀비 프로세스는 프로세스 테이블에만 존재
- 좀비 프로세스를 방지하기 위해 부모 프로세스와 자식 프로세스를 동기화 해야함

고아프로세스 (orphan process)

- 자식 프로세스보다 부모 프로세스가 먼저 종료할 경우 자식 프로세스들은 고아 프로세스가 됨
- 고아 프로세스는 1번 프로세스(init)의 자식 프로세스로 등록

프로세스 동기화 함수[1]

프로세스 동기화: wait(3)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

- stat_loc : 상태정보를 저장할 주소
- wait 함수는 자식 프로세스가 종료할 때까지 부모 프로세스를 기다리게 함

- 부모 프로세스가 wait 함수를 호출하기 전에 자식 프로세스가 종료하면 wait 함수는 즉시 리턴
- wait 함수의 리턴값은 자식 프로세스의 PID
- 리턴값이 -1이면 살아있는 자식 프로세스가 하나도 없다는 의미

wait 함수 사용하기

```
int main(void) {
    int status;
    pid_t pid;
    switch (pid = fork()) {
        case -1 :  /* fork failed */
            perror("fork");
            exit(1);
            break;
        case 0 :   /* child process */
            printf("--> Child Process\n");
            sleep(2);
            exit(2);
            break;
        default :   /* parent process */
            wait(&status);

            printf("--> Parent process\n");
            printf("Status: %d, %x\n", status, status);
            printf("Child process Exit Status:%d\n" ,status >> 8);
            break;
    }
    return 0;
}
```

```
# ex6_8.out
--> Child Process
--> Parent process
Status: 512, 200
Child process Exit Status:2
```

자식 프로세스의 종료를 기다림

오른쪽으로 8비트 이동해야 종료 상태 값을 알 수 있음

Exercise

Ex.

- Child process와 기존의 gugudan 프로그램을 이용하여 구구단을 출력하는 프로그램을 작성
- Child process는 gugudan 프로그램을 실행 (exec 함수 사용)
- Child process가 출력을 완료할 때까지 parent process는 대기

```
./a.out

Child Process - My PID:15696, My Parent's PID:15695
2 x 0 = 0
2 x 1 = 2
...

Parent process - My PID:15695, My Parent's PID:15602, My Child's
PID:15696
```

Exercise

Ex.

- 지정된 구구단 한 단을 출력하는 guguone 프로그램 작성
  - guguone은 command line argument로 출력 단을 받음
  - ./guguone 3

- guguone을 child process로 이용하여 구구단을 출력하는 프로그램 작성
- Child process는 guguone 프로그램을 실행 (exec 함수 사용)
- Child process가 출력을 완료할 때까지 parent process는 대기

```
./a.out

2 x 0 = 0
2 x 1 = 2
...
```