**강의명: 유닉스 시스템**

**실습 번호: 7**

**실습 제목: 프로세스 제어(Process Control)**

**학생 이름: 황귀훈**

**학번: 201710885**

**1. 프로세스 번호**

**1.1**

****

**1.2**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

프로세스 번호(Process Identifier)는 non-negative integer 값을 가지며 unique하나 reuse될 수 있다. 위에서 “./pid”를 수행한 결과와 “ps” 명령어를 이용해서 pid를 출력 한 결과를 비교해 보면 “./pid”에서 parent pid(=ppid)는 15882이고, “ps” 명령어를 이용해서 출력한 bash의 pid는 15882로 같은 값이 출력이 된 것을 확인할 수 있었다. 그 이유는 내 tera term bash에서 pid프로그램을 실행시켰기 때문에 pid프로그램이 bash의 child process(./pid)가 되는 것이다. 따라서 pid프로그램의 parent process는 bash가 되기 때문에 출력물의 ppid와 bash의 pid가 같은 것이다

**2. 프로세스 생성 1**

**2.1**

**텍스트, 화이트보드이(가) 표시된 사진

자동 생성된 설명**

번호 2 프로세스 생성 1은 시스템 호출 함수 fork를 사용하여 child 프로세스를 생성하는 프로그램이다. 이 프로그램은 fork 후 child 프로세스에서는 자신의 pid를 형식 "child pid=%d\n"으로 출력하고 종료하며, parent 프로세스에서는 자신의 pid를 형식 "parent pid=%d\n"으로 출력하고 또 fork 후 받은 return 값을 형식 "parent return pid=%d\n"으로 출력 해야한다.

따라서 먼저 fork() 시스템 호출을 사용하기 위해 <unistd.h>를 include해주고 나머지 perror()와 기본 c함수들을 사용하기 위해서 <stdlib.h>, <stdio.h>를 include해주었다.

그 다음 fork의 return value를 저장하기 위해서 int 타입의 pid라는 변수를 정의해주었다. fork의 return값은 child는 0, parent는 child pid이고 error 시 -1을 반환한다.

그리고 child 프로세스를 생성하기 위하여 fork 시스템 호출을 사용했다. fork()를 부르는 순간 parent 프로세스의 address space를 복사하여 child 프로세스를 생성한다. fork()는 한번 call 되지만 두 곳에서 return이된다. 생성 도중 error가 발생한다면 -1이 return이 되기 때문에 if문을 사용하여 pid가 0보다 작을 경우 perror(“fork”)를 호출하여 에러 메시지를 출력하고 exit(1)함수를 호출하여 비정상적인 종료했다.

만약 pid가 0인 경우는 child process가 실행하는 부분이므로 printf()함수를 이용하여 child process의 pid를 문제에 주어진 형식에 맞게 출력했다. 이 떄 pid는 getpid()함수를 호출하여 현재 프로세스(child process)의 번호를 얻을 수 있다.

마지막으로 pid가 0이 아니라면 parents process가 실행하는 부분이기 때문에 printf()함수를 이용하여 parent process의 pid를 문제에 주어진 형식에 맞게 출력을 하고 fork 시스템 호출 후 받은 return 값인 child process의 pid를 주어진 형식에 맞게 출력했다. 이 때 parent process의 pid는 getpid()함수를 호출하여 현재 프로세스(parent process)의 번호를 얻을 수 있고, child process의 pid는 fork()함수를 호출할 때 반환된 pid변수에 저장 되어있다. 모든 수행을 정상적으로 마치면 return 0를 해주어 프로그램을 정상 종료했다.

**2.2**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

명령 줄 "./fork1"를 수행한 결과 출력된 pid 값 중 parent return pid의 값과 child pid의 값이 같다. 그 이유는 fork() 시스템 호출은 return이 되는 곳이 두 곳인데 parent 프로세스와 child 프로세스이다. Parent process에서 return할 때 return값은 child의 pid이고, child에서의 return할 때 return 값은 0이다. 따라서 위의 프로그램 작성 시 fork() 시스템 함수 호출 시 반환 값을 저장할 변수로 pid를 사용했고, 그 pid가 0인경우, 0보다작은 경우, 0보다 큰 경우로 나누어서 프로그램을 작성하였다. pid가 0인 경우는 child process가 동작하는 경우이므로 그때의 child process pid값을 getpid()함수를 통해 얻은 것이 16803이다. pid가 0보다 큰 경우는 parent process에서 반환한 값이기 때문에 그 pid가 child process의 pid가 되고 그때 출력한 parent return pid가 16803인 것이다. 따라서 명령 줄 "./fork1"를 수행한 결과 출력된 pid 값 중 parent return pid의 값과 child pid의 값이 같은 것이다.

**3. 프로세스 생성 2**

**3.1**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**3.2**

프로그램 fork2.c를 보면 먼저 프로그램에 사용하는 함수들을 위해 헤더파일들을 include했고, 전역변수 glob는 6으로 초기화 되어있고 main 함수 안에 지역변수 loc이 88로 초기화 되어있음을 알 수 있다. 위 프로그램은 fork()함수를 호출하기 전에 printf()함수를 사용하여 “before fork”를 출력한다.

그리고 fork()함수를 호출하여 child process를 생성하는데 이 때 child process는 parent의 address space를 복사하기 때문에 glob(=6)과 loc(=88)도 복사하게 된다. 그리고 fork()호출 시 반환 값을 pid에 저장하고 pid가 0보다 작다면 perror(“fork”)를 호출하여 에러 메시지를 출력하고 exit(1)을 호출하여 비정상 종료한다. 만약 pid가 0이라면 이는 child process에서 반환한 것이므로 child process에서 작동함을 알 수 있는데, 이때에는 glob 변수와 loc변수를 각각 1씩 증가시킨다. 하지만 이때 parent process와 memory space가 다르기 때문에 parent process의 glob와 loc에는 변화가 없다. 마지막으로 만약 pid가 0보다 크다면 parent process에서 반환한 child process의 pid이므로 parent process에서 작동하는 부분인데 이 때에는 sleep(2)함수를 호출하여 2ms 동안 멈춘다.

마지막으로 if문이 끝나면 printf()함수를 사용하여 그 프로세스의 pid, glob, loc를 출력하고 return 0를 사용하여 프로그램을 정상 종료한다.

따라서 처음 fork2를 실행시키면 “before fork”가 한번 출력된 뒤 child process의 pid, child process의 glob, loc 변수의 값이 출력되고, parent process의 pid, parent process의 glob, loc이 출력된다. 이 때 parent process의 glob, loc변수의 값은 변하지 않음을 확인할 수 있다.

하지만 fork2의 출력을 stdout하여 screen에 출력할 때와 명령 “fork2 > output”를 사용하여 fork2의 출력을 file에 출력할 때 “before fork”가 한번 더 출력되는 차이가 있었는데 이유는 다음과 같다. Stdout은 line buffered되고, 일반 file은 full buffered되는 차이다. buffer가 차면 그 때 read/write를 호출하는데 line buffer는 buffer 크기가 작아서 printf를 하면 바로 화면에 출력하기 때문에 fork2의 출력을 stdout하여 screen에 출력할 경우 fork()를 불러서 child process를 생성할 때 buffer에 “before fork”가 없는 상태로 memory space를 복사하여 “before fork”가 한번 출력 되는 것이고, full buffer는 buffer가 크기 때문에 “fork2 > output”를 사용하여 file로 redirection하면 child process를 생성할 때 buffer에 “before fork”가 있는 상태로 memory space를 복사하여 “before fork”가 두 번 출력되는 것이다.

**4. 프로세스 기다림 1**

**4.1**

**텍스트, 영수증이(가) 표시된 사진

자동 생성된 설명**

**4.2**

Parent process가 fork하여 child process를 생성하면 두 process는 서로 독립적으로 수행된다. Parent는 child가 종료할 때까지 수행하지 않고(=block 되어서) 기다릴 수 있다. 함수 wait/waitpid는 parent가 child의 종료를 기다리는 함수이다. 만약 parent가 함수 wait을 부르기 전에 child가 종료되었으면 함수 wait은 즉시 return한다. 만약 child가 종료하였는데 parent가 wait하지 않으면 child는 “zombie”상태가 된다. 위에서 명령 줄 "./wait1 &"을 수행한 후 쉘로 나온 후 명령 줄 "ps" 를 수행시켜 보면 parent process(pid = 16860)는 동작 중이고, child process(pid = 16861)은 <defunct>상태임을 알 수 있는데, 위의 프로그램 wait1는 child가 종료하였는데 parent process가 wait하지 않아 child process가 “zombie”상태가 된 것이다.

“defunct” process는 wait1프로세스 즉, parent process가 끝나면 사라진다.

**5. 프로세스 기다림 2**

**5.1**

**텍스트, 영수증이(가) 표시된 사진

자동 생성된 설명**

**5.2**

앞 실습에서는 child process가 종료하였는데 parent process가 wait하지 않아 child process가 “zombie”상태<defunct>가 되었다. 이번 실습에서는 Parent 프로세스에서 wait함수를 호출하여 child process가 끝나는 것을 기다렸는데, 그래서 ps명령어로 프로세스들을 확인한 결과 자식 프로세스가 좀비 프로세스 <defunct>로 변하지 않고 끝내는 것을 확인할 수 있었다.

**6. 프로그램 수행**

**6.1**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**6.2**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

위에 작성된 프로그램을 보며 6.1에서 명령 줄 "./exe ls /"을 수행한 결과가 어떻게 나온 것인지 설명하겠다. 먼저 fork() 시스템 호출을 사용하기 위해 <unistd.h>를, wait()함수를 사용하기 위해 <wait.h>를 include하고 나머지 perror()와 기본 c함수들을 사용하기 위해서 <stdlib.h>, <stdio.h>를 include해주었다.

그리고 main()함수에서는 fork()함수의 return값을 저장할 pid변수를 선언하고 printf()함수로“parent: fork…”를 출력한 뒤 fork()함수를 호출하여 child process를 생성한다. fork()함수의 호출로 pid에 값이 저장되는데 이 값이 0보다 작다면 오류가 발생한 것이므로 perror(“fork”)를 호출하여 에러 메시지를 호출하고 exit(1)함수를 호출하여 프로그램을 비정상 종료했다. 만약 pid가 0이라면 child process가 동작하는 부분으로 printf()함수로 “child: exec…”을 출력한 뒤 execvp()함수를 사용하여 인수에 주어지는 수행파일로 process를 수행시킨 뒤 exit(0)을 호출하여 정상 종료한다.

만약 pid가 0보다 크다면 parent process가 동작하는 부분으로 먼저 printf()를 사용하여 “parent: wait…”을 출력하고 wait()함수를 호출하여 child process가 종료될 때까지 기다린다. 그리고 child process가 종료되면 printf()를 사용하여 “parent: exit…”을 출력하고 exit(0)을 호출해 정상 종료한다.

따라서 6.1에서 명령 줄 "./exe ls /"을 수행하면 가장 먼저 **“parent: fork…”**가 출력되고 parent process에서 **“parent: wait…”**을 출력한 뒤 child process를 기다린다. Child process는 **“child: exec…”**를 출력하고 execvp()함수에 의해 **“ls /”**을 수행한다. 그 후 child process가 종료되면 parent process의 wait이 끝나고 **“parent: exit…”**를 출력한 뒤 프로그램이 종료된다. 6.1의 출력물과 같은 결과임을 알 수 있다.

**7. 프로세스 시간**

**7.1**

**$ ./mytime ls -lR /etc**

**…**

**parent: exit...**

**real: 2.560 sec**

**usr: 0.030 sec**

**sys: 0.040 sec**

**7.2**

**텍스트, 영수증, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

앞 실습의 프로그램 exe.c를 위와 같이 수정하였다. 먼저 프로세스가 실행한 시간을 파악하기 위해서 <sys/times.h>을 추가로 include했고 그 안에 선언되어 있는 times()함수를 이용해서 시간을 출력했다. 함수 times는 현재 프로세스 및 종료한 child 프로세스가 소비한 user CPU time과 system CPU time을 알려준다. 그리고 현재 프로세스의 wall clock time은 return값으로 주어진다. 다음은 time()함수에 넘겨주는 tms구조체이다. time()함수를 호출하면 아래 구조체에 시간이 저장된다. tms\_cutime, tms\_cstime에 종료된 child process의 user, sys time이 저장된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

구조체 tms 타입의 변수 buf와 시작시간, 종료시간을 저장할 변수 s\_time, e\_time을 main함수 시작부분에 선언했다. 그리고 초(sec) 단위의 시간을 얻기 위해 sysconf(\_SC\_CLK\_TCK)로 나누어 주어야 하기 때문에 tck변수를 따로 sysconf(\_SC\_CLK\_TCK)로 초기화했다.

Times()함수의 return 값이 wall clock time이기 때문에 이번 실습에서 수행한 프로세스의 wall time을 얻기 위해 parent 프로세스 시작 전에 time()함수를 호출해 s\_time에 저장하고, parent 프로세스 수행 후 time()함수를 호출해 e\_time에 저장하여 그 차이를 tck변수로 나누어 프로세스의 wall time을 얻었다.

그 다음 수행한 프로세스의 시간 (user\_cpu\_time = tms\_cutime, sys\_cpu\_time = tms\_cstime)을 구해야 한다. Child process는 “child: exec…”를 출력한 뒤 execvp()함수를 사용하여 인수에 주어지는 수행파일로 process를 수행시킨 뒤 exit(0)을 호출하여 정상 종료한다. 위 실습에서는 **ls -lR /etc**를 수행한다. Child process가 종료되면 parent process에서 times()함수를 호출하여 child 프로세스가 소비한 시간을 알 수 있다.

따라서 parent process에서 wait()함수 뒤에 times()를 호출하면 그때 구조체 tms안에는 child의 시간이 저장된다. wait()함수 뒤에 times()를 호출하는 이유는 parent process는 wait()함수를 호출하여 child process가 종료될 때까지 기다리기 때문이다.

따라서 buf의 tms\_cutime 그리고 tms\_cstime을 tck변수로 나누어 user\_CPU\_time과 sys\_CPU\_time

을 얻었다.

프로그램 종료 전에 printf()함수를 사용하여 주어진 형식에 맞게 수행한 프로세스의 시간(wall time, user CPU time, system CPU time)을 출력하고 exit(0)으로 프로그램을 정상 종료했다.

따라서 7.1에서 “./mytime ls -lR /etc”을 수행하면 실습 6에서 설명한 것과 같은 순서로 출력 되고 마지막에 real, user, sys time이 출력되는 것을 볼 수 있다.