8

막간: 프로세스 API

여담: 막간

이번 절에서는 운영체제 API와 그의 사용법을 포함한 시스템의 실제적인 측면을 다룰 것이다. 실제적인 측면에 관심이 없다면 이 막간 절은 넘어갈 수 있다. 실생활에서는 이런 실제적인 측면이 유용하기 때문에 관심을 가지고 좋아해야 한다. 회사는 사용할 수 없는 지식을 가진 사람을 채용하지 않는다.

이번 절에서는 UNIX 시스템의 프로세스 생성에 관해 논의한다. UNIX는 프로세스를 생성하기 위하여 `fork()`와 `exec()` 시스템 콜을 사용한다. `wait()`는 프로세스가 자신이 생성한 프로세스가 종료되기를 기다리기 원할 때 사용된다. 이제부터 이 인터페이스에 대해 간단한 예제를 이용하여 더 자세하게 설명한다. 우리의 문제는 다음과 같다.

핵심 질문: 프로세스를 생성하고 제어하는 방법

프로세스를 생성하고 제어하려면 운영체제가 어떤 인터페이스를 제공해야 하는가? 유용하고 편하게 사용하기 위해서 이 인터페이스는 어떻게 설계되어야 하는가?

8.1 `fork()` 시스템 콜

프로세스 생성에 `fork()` 시스템 콜이 사용된다 [Con63]. 미리 경고하자면, 이 시스템 콜이 당신이 사용할 시스템 콜 중에서 가장 이해하기 힘든 시스템 콜이다[1].

그림 8.1에 실행 가능한 프로그램이 나와 있다. 코드를 분석해 보라. 아니면 타이핑해서 직접 실행시켜 보기 바란다!

이 프로그램을 실행하면(`p1.c`) 다음과 같은 출력을 보게 될 것이다.

1) 물론 가장 이해하기 힘들다는 것을 보장할 수는 없다. 어쨌거나 `fork()` 시스템 콜은 매우 특이하다.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {           // fork 실패; 종료
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) {   // 자식 (새 프로세스)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {                // 부모 프로세스는 이 경로를 따라 실행한다 (main)
16         printf("hello, I am parent of %d (pid:%d)\n",
17                 rc, (int) getpid());
18     }
19     return 0;
20 }
```

〈그림 8.1〉 `fork()` 호출 (`p1.c`)

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

`p1.c` 프로그램에서 어떤 일들이 벌어지는지 더 자세히 보도록 하자. 실행이 시작될 때 프로세스는 "hello world ..." 메시지를 출력한다. 이 메시지에는 **PID**로도 알려진 **프로세스 식별자**(**process identifier**)가 포함된다. 이 프로세스는 29146이라는 PID를 가진다. UNIX 시스템에서 PID는 프로세스의 실행이나 중단과 같이 특정 프로세스를 대상으로 작업을 해야 할 경우 프로세스를 지칭하기 위해 사용된다. 지금까지는 오케이다.

재미있는 부분은 지금부터 시작이다. 프로세스는 `fork()` 시스템 콜을 호출한다. 운영체제는 프로세스 생성을 위해 이 시스템 콜을 제공한다. 이상한 부분이 있다: 생성된 프로세스가 *호출한 프로세스의 복사본이라는* 것이다. `fork()` 호출 직후를 살펴보자. 운영체제 입장에서 보면, 프로그램 `p1`이 2개가 존재한다. 두 프로세스가 모두 `fork()`에서 리턴하기 직전이다. 새로 생성된 프로세스는(일반적으로 **자식** 프로세스, 생성한 프로세스는 **부모** 프로세스라 불린다) `main()`함수 첫 부분부터 시작하지 않았다는 것을 알 수 있다 ("hello, world" 메시지가 한 번만 출력되었다는 것에 유의하기 바란다). 자식 프로세스는 `fork()`를 호출하면서부터 시작되었다.

자식 프로세스는 부모 프로세스와 완전히 동일하지는 않다. 자식 프로세스는 자신의 주소 공간, 자신의 레지스터, 자신의 PC 값을 갖는다. 매우 중요한 차이점이 있다. 아무리 강조해도 지나치지 않다. `fork()` 시스템 콜의 반환 값이 서로 다르다. `fork()`로 부터 부모 프로세스는 생성된 자식 프로세스의 PID를 반환받고, 자식 프로세스는 0을 반환받는다. 이 반환 값의 차이로 인해, 그림 8.1과 같이 부모와 자식 프로세스가 서로 다른 코드를 실행하는 프로그램을 쉽게 작성할 수 있다.

이 프로그램의 출력 결과가 항상 동일하지는 않다. 단일 CPU 시스템에서 이 프로그램을 실행하면, 프로세스가 생성되는 시점에는 2개(부모와 자식) 프로세스 중 하나가

실행된다. 위의 출력 예에서는 부모 프로세스 실행 후에 자식 프로세스가 실행되었다. 예에서 볼 수 있는 바와 같이, 그 반대 경우도 발생할 수 있다.

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

CPU **스케줄러**(**scheduler**)는 실행할 프로세스를 선택한다. 스케줄러에 관해서는 곧 자세하게 논의한다. 스케줄러의 동작은 일반적으로 상당히 복잡하고 상황에 따라 다른 선택이 이루어지기 때문에, 어느 프로세스가 먼저 실행된다라고 단정하는 것은 매우 어렵다. 이 **비결정성**(**nondeterminism**)으로 인해 **멀티 쓰레드 프로그램** 실행 시 다양한 문제가 발생한다. 이 책의 제2편 **병행성** 부분에서 비결정성에 대해 자세히 다룰 것이다.

8.2 wait() 시스템 콜

아직 배울 것이 많이 남았다. 이제까지 메시지를 출력하는 자식 프로세스를 만들어 보았다. 부모 프로세스가 자식 프로세스의 종료를 대기해야 하는 경우도 발생할 수 있다. 이러한 작업을 위해 `wait()` 시스템 콜이 (혹은 더 많은 기능을 가진 `waitpid()`) 있다. 자세한 사항은 그림 8.2를 보라.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {           // fork 실패; 종료
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) {   // 자식 (새 프로세스)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {                // 부모 프로세스는 이 경로를 따라 실행한다 (main)
16         int wc = wait(NULL);
17         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
18                 rc, wc, (int) getpid());
19     }
20     return 0;
21 }
```

〈그림 8.2〉 `fork()`와 `wait()` 호출 (`p2.c`)

이 예제에서 (`p2.c`) 부모 프로세스는 `wait()` 시스템 콜을 호출하여 자식 프로세스 종료 시점까지 자신의 실행을 잠시 중지시킨다. 자식 프로세스가 종료되면 `wait()`는 리턴한다.

`wait()` 호출을 위와 같이 코드에 추가하면 프로그램은 항상 동일한 결과를 출력한다. 왜 그런지 알 수 있겠지? 좋아, 생각해 보도록.

(생각하는 동안 기다림 .... 끝)

자 이제 생각해 보았으니, 출력 결과를 보자.

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

이 코드에서는 항상 자식 프로세스가 먼저 출력을 수행한다는 것을 알 수 있다. 왜 그럴까? 자식 프로세스가 부모 프로세스보다 먼저 실행되면 당연히 자식 프로세스가 먼저 출력된다. 부모 프로세스가 먼저 실행되면 곧바로 `wait()`를 호출한다. 이 시스템 콜은 자식 프로세스가 종료될 때까지 리턴하지 않는다[2]. 부모가 먼저 실행되더라도 자식 종료 후 `wait()`가 리턴한다. 그런 후 부모 프로세스가 출력한다.

8.3 드디어, `exec()` 시스템 콜

프로세스 생성 관련 API 중에서 마지막으로 중요한 시스템 콜은 `exec()` 시스템 콜이다[3]. 이 시스템 콜은 자기 자신이 아닌 다른 프로그램을 실행해야 할 때 사용한다. `p2.c`의 `fork()` 시스템 콜은 자신의 복사본을 생성하여 실행한다. 자신의 복사본이 아닌 다른 프로그램을 실행해야 할 경우에는 바로 `exec()` 시스템 콜이 그 일을 한다 (그림 8.3).

이 예에서 자식 프로세스는 `wc` 프로그램을 실행하기 위해 `execvp()` 시스템 콜을 호출한다. `wc` 프로그램은 단어의 개수를 세는 프로그램이다. 사실 예제 프로그램은 자신의 소스 파일인 `p3.c`를 인자로 하여 `wc`를 실행하고 소스 코드의 행 개수, 단어의 개수, 바이트의 개수를 알려 준다.

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107     1030    p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

`fork()` 시스템 콜만큼 `exec()` 시스템 콜도 이해하기 어렵다. `exec()` 시스템 콜은 다음과 같은 과정으로 수행된다. 실행 파일의 이름과(예, `wc`) 약간의 인자가(예, `p3.c`) 주어지면 해당 실행 파일의 코드와 정적 데이터를 읽어 들여 현재 실행 중인 프로세스의 코드 세그멘트와 정적 데이터 부분을 덮어 쓴다. 힙과 스택 및 프로그램 다른

2) `wait()`가 자식 프로세스가 종료하기 전에 리턴하는 몇 가지 경우가 있다. 자세한 사항은 `man` 페이지를 참조하기 바란다. 이 책의 주장에(예를 들어, "자식이 항상 먼저 프린트 한다" 혹은, "UNIX가 최고다, 아이스크림보다 더 좋다" 등) 대해서는 주의해야 한다.

3) 사실 `exec()` 시스템 콜은 6가지 변형이 존재한다. `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()`, `execve()`가 그것이다. 더 자세한 사항은 `man` 페이지를 읽어보기 바란다.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6  int
7  main(int argc, char *argv[])
8  {
9      printf("hello world (pid:%d)\n", (int) getpid());
10     int rc = fork();
11     if (rc < 0) { // fork 실패함; exit
12         fprintf(stderr, "fork failed\n");
13         exit(1);
14     } else if (rc == 0) {            // 자식 (새 프로세스)
15     printf("hello, I am child (pid:%d)\n", (int) getpid());
16     char *myargs[3];
17     myargs[0] = strdup("wc");      // 프로그램: "wc"(단어 세기)
18     myargs[1] = strdup("p3.c");    // 인자: 단어 셀 파일
19     myargs[2] = NULL;          // 배열의 끝 표시
20     execvp(myargs[0], myargs);  // "wc" 실행
21     printf("this shouldn't print out");
22     } else {           // 부모 프로세스는 이 경로를 따라 실행한다 (main)
23         int wc = wait(NULL);
24         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
25             rc, wc, (int) getpid());
26     }
27     return 0;
28 }
```

〈그림 8.3〉 fork(), wait(), 및 exec() 호출하기 (p3.c)

주소 공간들로 새로운 프로그램의 실행을 위해 다시 초기화된다. 그런 다음 운영체제는 프로세스의 argv와 같은 인자를 전달하여 프로그램을 실행시킨다. 새로운 프로세스를 생성하지는 않는다. 현재 실행 중인 프로그램을(p3) 다른 실행 중인 프로그램으로(wc) 대체하는 것이다. 자식 프로세스가 exec()을 호출한 후에는 p3.c는 전혀 실행되지 않은 것처럼 보인다. exec() 시스템 콜이 성공하게 되면 p3.c는 절대로 리턴하지 않는다.

8.4 왜, 이런 API를?

여기서 이런 의문이 생길 수 있다. 새로운 프로세스를 생성하는 간단한 작업 같은데, 왜 이런 이상한 인터페이스를 사용할까? UNIX의 쉘을 구현하기 위해서는 fork()와 exec()을 분리해야 한다. 그래야만 쉘이 fork()를 호출하고 exec()를 호출하기 전에 코드를 실행할 수 있다. 이때 실행하는 코드에서 프로그램의 환경을 설정하고, 다양한 기능을 준비한다.

쉘은 단순한 사용자 프로그램이다[4]. 쉘은 **프롬프트**를 표시하고 사용자가 무언가 입력하기를 기다린다. 그리고 명령어를 입력한다(예, 실행 프로그램의 이름과 필요한 인자). 대부분의 경우 쉘은 파일 시스템에서 실행 파일의 위치를 찾고 명령어를 실행하기

4) 굉장히 많은 종류의 쉘이 존재한다. 몇 가지 꼽아 보면 tcsh, bash, 그리고 zsh 등이 있다. 그 중 하나를 골라 man 페이지를 읽어 보라. 모든 UNIX 전문가들이 그렇게 한다.

팁: 올바른 선택하기 (Lampson's Law)

그 유명한 "Hints for Computer Systems Design" [Lam83]에서 Lampson이 말한바 같이 "**올바르게 선택하라**. 추상화도 단순함도 올바른 선택을 대체할 수 없다." 올바른 선택을 해야만 한다. 올바른 선택은 다른 어떤 대안보다 나아야 한다. 프로세스 생성을 위한 API를 설계하는 방법은 많다. 그러나 `fork()`와 `exec()`의 조합은 단순하면서 매우 강력하다. UNIX 설계자들은 단순하고 올바른 방법으로 구현하였다. Lampson은 매우 자주 올바른 선택을 했기 때문에 그를 기념하여 이 법칙에 그의 이름을 붙였다.

위하여 `fork()`를 호출하여 새로운 자식 프로세스를 만든다. 그런 후 `exec()`의 변형 중 하나를 호출하여 프로그램을 실행시킨 후 `wait()`를 호출하여 명령어가 끝나기를 기다린다. 자식 프로세스가 종료되면 쉘은 `wait()`로부터 리턴하고 다시 프롬프트를 출력하고 다음 명령어를 기다린다.

`fork()`와 `exec()`을 분리함으로써 쉘은 많은 유용한 일을 조금 쉽게 할 수 있다. 다음 예를 생각해 보자.

```
prompt> wc p3.c > newfile.txt
```

위의 예제에서 `wc` 프로그램의 출력은 `newfile.txt`라는 출력 파일로 방향이 **재지정**된다 (〉 표시가 재지정을 나타낸다). 이러한 작업을 수행하는 방법은 간단하다. 자식이 생성되고 `exec()`이 호출되기 전에 **표준 출력**(**standard output**) 파일을 닫고 `newfile.txt` 파일을 연다. 이런 작업을 해 놓으면 곧 실행될 프로그램인 `wc`의 출력은 모두 화면이 아니라 파일로 보내진다.

그림 8.4에 작업을 수행하는 프로그램이 나와 있다. UNIX 시스템은 미사용 중인 파일 디스크립터를 0번부터 찾아 나간다. 이 경우, `STDOUT_FILENO`가 첫 번째 사용 가능 파일 디스크립터로 탐색되어, `open()`이 호출될 때 할당된다. 표준 출력 파일을 닫았기 때문이다. 이후 자식 프로세스가 표준 출력 파일 디스크립터를 대상으로 하는 모든 쓰기, 예를 들어 `printf()`에 의한 쓰기는 화면이 아니라 새로 열린 파일로 향하게 된다.

`p4.c`의 실행 결과는 다음과 같다.

```
prompt> ./p4
prompt> cat p4.output
      32      109     846    p4.c
prompt>
```

이 출력 결과에는 두 가지의 흥미로운 점이 있다. 첫째, `p4`를 실행하면, 화면에 아무 일도 일어나지 않는다. 그러나 실제로는 다음과 같은 일이 발생하였다. 프로그램 `p4`는 `fork()`를 호출하여 새로운 자식 프로세스를 생성하고 `execvp()`를 호출하여 `wc` 프로그램을 실행시킨다. 출력이 `p4.output` 파일로 재지정되었기 때문에 화면에는 아무 것도 출력되지 않는다. 출력 파일을 `cat`해 보면 `wc`를 실행시켰을 때 얻을 수 있는 모든 출력이 파일에 저장되어 있는 것을 발견하게 될 것이다. 멋지지 않나?

UNIX 파이프가 이와 유사한 방식으로 구현되지만 `pipe()` 시스템 콜을 통하여 생성된다. 이 경우, 한 프로세스의 출력과 다른 프로세스의 입력이 동일한 **파이프**에

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <fcntl.h>
6  #include <sys/wait.h>
7  int
8  main(int argc, char *argv[])
9  {
10     int rc = fork();
11     if (rc < 0) { // fork 실패함; exit
12         fprintf(stderr, "fork failed\n");
13         exit(1);
14     } else if (rc == 0) {          // 자식: 표준 출력 파일로 재지정
15         close(STDOUT_FILENO);
16         open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
17
18         // 자 exec "wc"...
19
20
21         char *myargs[3];
22         myargs[0] = strdup("wc");      // 프로그램: "wc"(단어 세기)
23         myargs[1] = strdup("p4.c");    // 인자: 단어 셀 파일
24         myargs[2] = NULL;          // 배열의 끝 표시
25         execvp(myargs[0], myargs);  // wc 실행
26     } else {     // 부모 프로세스는 이 경로를 따라 실행한다 (main)
27         int wc = wait(NULL);
28     }
29     return 0;
30 }
```

〈그림 8.4〉 **입출력 재지정의 모든 것 (p4.c)**

연결된다. 한 프로세스의 출력은 자연스럽게 다음 프로세스의 입력으로 사용되고, 명령어 체인이 형성된다. 파일에서 특정 단어가 몇 번 나오는지 세는 예제를 생각해 보자. 파이프와 `grep`과 `wc`를 사용하면 쉽게 할 수 있다. `grep foo file | wc -l`을 명령어 프롬프트에 입력하고 결과에 놀라기만 하면 된다.

프로세스 API를 간략하게 설명하였지만, 아직 실제로 배우고 이해해야 할 것들이 많이 남아있다. 예를 들면, 파일 디스크립터에 관해서는 이 책의 세 번째 영역인 파일 시스템에 관해서 논의할 때 다루게 될 것이다. 지금 당장은 `fork()/exec()` 조합이 프로세스를 생성하고 조작하는 강력한 방법이라는 정도만 이해하면 충분하다.

8.5 여타 API들

Unix 시스템에는 `fork()`, `exec()`, 및 `wait()` 외에 많은 프로세스 관련 인터페이스가 있다. 예를 들면, `kill()` 시스템 콜은 프로세스에게 **시그널**(**signal**)을 보내는 데 사용된다. 시그널은 프로세스를 중단시키고(block), 삭제하는 등의 작업에 사용된다. 시그널이라는 운영체제의 메터니즘은 외부 사건을 프로세스에게 전달하는 토대이다. 이 기반 구조는 시그널을 보내거나 전달받는 방법을 모두 포함한다.

유용한 명령어들이 많이 있다. 예를 들면, `ps` 명령어는 어떤 프로세스가 실행 중인지 알아보기 위하여 사용된다. `ps`의 유용한 플래그를 알기 위해서는 **man** 페이지를 읽어 보라. `top` 역시 시스템에 존재하는 프로세스와 그 프로세스가 CPU 및 다른 자원들을

여담: RTFM — man 페이지를 읽어라

특정 시스템 콜이나 라이브러리 콜을 언급할 때 **매뉴얼 페이지** 또는 간략하게 **man** 페이지를 읽으라는 말을 여러 번 들었을 것이다. `man` 페이지는 UNIX 시스템에 존재하는 문서의 원형이다. 이른바 **web**이라고 불리는 것이 존재하기 전에 만들어졌다는 것을 기억하라.

man 페이지를 읽는 것은 시스템 프로그래머로서 성장하는 데 매우 중요한 단계이다. 이 페이지들 속에는 매우 유용한 정보가 셀 수 없이 숨어 있다. 특히, 유용한 페이지는 사용 중인 쉘(예, **tcsh** 또는 **bash**)의 페이지와 프로그램에서 사용한 시스템 콜에(반환값이 무엇이고 어떤 에러 조건이 존재하는지 보기 위하여) 관한 페이지다.

마지막으로 `man` 페이지를 읽으면 당황하는 경우를 줄일 수 있다. `fork()`의 복잡함에 대하여 동료에게 물어보면 "RTFM"이라는 아주 간단한 대답이 돌아올 것이다. `man` 페이지를 읽어야 한다는 것을 강조하는 동료들의 점잖은 표현이다. RTFM의 F는 표현을 더 맛깔나게 한다.

얼마나 사용하고 있는지를 보여 주기 때문에 매우 유용하다. `top` 명령어를 여러 번 실행할 경우 자기 스스로가 가장 많은 자원을 사용하고 있다고 지적하는 것이 재밌는 부분이다. 매우 자기중심적인 명령어라고 할 수 있다. 마지막으로 다양한 CPU 측정기가 제공되어 시스템의 부하 정도를 알 수 있다. 예를 들어, Raging Menace software 사의 **MenuMeter**를 Macintosh의 toolbar에 실행시키면 어느 때건 CPU의 이용률을 점검할 수 있다. 일반적으로 현재 벌어지고 있는 일에 대해 더 많은 정보를 가질수록 더 유리하다.

8.6 요약

UNIX 프로세스를 다루는 API 중 몇 개를 소개하였다. `fork()`, `exec()` 및 `wait()`이 바로 그것이다. 매우 기본적인 것만 살펴보았다. 더 자세한 사항에 관해서는 Stevens and Rago [SR05]를 읽기 바란다. 특히, 프로세스 제어, 프로세스 관계 및 시그널 부분을 읽어 보기 바란다. 그 책에 유용한 내용들이 많이 있다.

참고 문헌

[Con63] **"A Multiprocessor System Design"**
Melvin E. Conway
AFIPS, '63 Fall Joint Computer Conference
New York, USA 1963
멀티 프로세스 시스템의 설계 방법에 관한 초기 논문. 아마도 새 프로세스 생성에 관한 논의 중 `fork()`라는 용어가 처음으로 사용된 논문일 것이다.

[DH66] **"Programming Semantics for Multiprogrammed Computations"**
Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
멀티 프로그램 컴퓨터 시스템의 기본에 관한 개략을 보여주는 고전 논문. MAC, Multics 및 궁극적으로 Unix 시스템에 영향을 주었다.

[Lam83] **"Hints for Computer Systems Design"**
Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
컴퓨터 시스템의 설계에 관한 Lampson의 그 유명한 조언. 언젠가 한 번은 읽어야 할 논문이자 아마 여러 번 읽어야 할 논문일 것이다.

[SR05] **"Advanced Programming in the Unix Environment"**
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
Unix API 사용의 미묘한 차이와 절묘함을 발견할 수 있는 책. 책을 사서 읽어라. 그리고 더 중요한 건 항상 곁에 두어야 한다.

숙제 (코드)

이 숙제에서는 방금 읽은 프로세스 관리 API와 친숙해 질 것이다. 걱정하지 마라. 들리는 것 보다는 훨씬 재미있으니까! 일반적으로 코드를 작성하기 위해 더 많은 시간을 투자하면 잘 하게 될 것이다[5]. 자 당장 시작하자.

여담: 코딩 숙제

코딩 숙제는 코드를 작성하여 실제 컴퓨터에서 실행시켜 봄으로써 현대 운영체제가 제공해야 하는 기본 API의 일부를 경험할 수 있는 작은 연습이다. 어쨌든 당신은 (아마도) 컴퓨터 과학자이니까 코딩을 좋아하겠지? 물론 진짜 전문가가 되려면 컴퓨터를 분해하는 시간보다는 더 많은 시간을 투자해야 한다. 사실 코드를 작성하여 실행시켜 동작을 이해할 수 있는 모든 구실을 찾아보아라. 시간을 투자해라. 그리고 될 수 있는 한 현명한 마스터가 되라.

문제

1. `fork()`를 호출하는 프로그램을 작성하라. `fork()`를 호출하기 전에 메인 프로세스는 변수에 접근하고 (예, x) 변수에 값을 지정하라 (예, 100). 자식 프로세스에서 그 변수의 값은 무엇인가? 부모와 자식이 변수 x를 변경한 후에 변수는 어떻게 변했는가?
2. `open()` 시스템 콜을 사용하여 파일을 여는 프로그램을 작성하고 새 프로세스를 생성하기 위하여 `fork()`를 호출하라. 자식과 부모가 `open()`에 의해 반환된 파일 디스크립터에 접근할 수 있는가? 부모와 자식 프로세스가 동시에 파일에 쓰기 작업을 할 수 있는가?
3. `fork()`를 사용하는 다른 프로그램을 작성하라. 자식 프로세스는 "hello"를 출력하고 부모 프로세스는 "goodbye"를 출력해야 한다. 항상 자식 프로세스가 먼저 출력하게 하라. 부모가 `wait()`를 호출하지 않고 할 수 있는가?
4. `fork()`를 호출하고 `/bin/ls`를 실행하기 위하여 `exec()` 계열의 함수를 호출하는 프로그램을 작성하라. `exec()`의 변형 `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()`, `execve()` 모두를 사용할 수 있는지 시도해 보라. 기본적으로는 동일한 기능을 수행하는 시스템 콜에 여러 변형이 있는 이유를 생각해 보라.
5. `wait()`를 사용하여 자식 프로세스가 종료되기를 기다리는 프로그램을 작성하라. `wait()`가 반환하는 것은 무엇인가? 자식 프로세스가 `wait()`를 호출하면 어떤 결과가 발생하는가?

5) 코딩이 싫지만 컴퓨터 과학자가 되기 원한다면 (a) 컴퓨터 과학 이론 분야에서 잘 하거나 (b) 지금까지 들어 왔던 "컴퓨터 과학"이라는 것에 대해 다시 생각해 보아야 한다.

6. 위 문제에서 작성한 프로그램을 수정하여 `wait()` 대신에 `waitpid()`를 사용하라. 어떤 경우에 `waitpid()`를 사용하는 것이 좋은가?

7. 자식 프로세스를 생성하고 자식 프로세스가 표준 출력(`STDOUT_FILENO`)을 닫는 프로그램을 작성하라. 자식이 설명자를 닫은 후에 아무거나 출력하기 위하여 `printf()`를 호출하면 무슨 일이 생기는가?

8. 두 개의 자식 프로세스를 생성하고 `pipe()` 시스템 콜을 사용하여 한 자식의 표준 출력을 다른 자식의 입력으로 연결하는 프로그램을 작성하라.