7

프로세스의 개념

이 장에는 운영체제가 제공하는 핵심 개념 중 하나인 **프로세스**(**process**)에 대해 논의한다. 일반적으로 프로세스는 **실행 중인 프로그램**으로 정의한다 [VCG65; Han70]. 프로그램 자체는 생명이 없는 존재다. 프로그램은 디스크 상에 존재하며 실행을 위한 명령어와 정적 데이터의 묶음이다. 이 명령어와 데이터 묶음을 읽고 실행하여 프로그램에 생명을 불어넣는 것이 운영체제이다.

사용자는 하나 이상의 프로그램을 동시에 실행시키기를 원한다. 예를 들어, 웹 브라우저, 메일 프로그램, 게임, 음악 플레이어 등을 실행하는 데스크톱 또는 랩톱 컴퓨터를 생각해 보라. 컴퓨터들은 동시에 수십 혹은 수백 개의 프로세스를 실행하는 것처럼 보인다. 여러 프로그램을 동시에 실행할 수 있으면, 시스템을 쉽게 사용할 수 있다. 사용자는 사용 가능한 CPU가 있는지 신경쓰지 않고 그저 프로그램만 실행시키면 된다. 우리의 도전 과제는 다음과 같다.

핵심 질문: CPU가 여러 개 존재한다는 환상을 어떻게 제공하는가
적은 개수의 CPU 밖에 없더라도, 운영체제는 어떻게 거의 무한개에 가까운 CPU가 있는 듯한 환상을 만들 수 있을까?

운영체제는 CPU를 **가상화**하여 이러한 환상을 만들어 낸다. 하나의 프로세스를 실행하고, 얼마 후 중단시키고 다른 프로세스를 실행하는 작업을 반복하면서 실제 하나 또는 소수의 CPU로 여러 개의 가상 CPU가 존재하는 듯한 환상을 만들어 낸다. **시분할**(**time sharing**)이라 불리는 이 기법은 원하는 수 만큼의 프로세스를 동시에 실행할 수 있게 한다. 시분할 기법은 CPU를 공유하기 때문에, 각 프로세스의 성능은 낮아진다.

운영체제에서 CPU 가상화를 잘 구현하기 위해, 저수준의 도구와 고차원적인 "지능"이 필요하다. 저수준 도구를 **메커니즘**(**mechanism**)이라 한다. 메커니즘은 필요한 기능을 구현하는 방법이나 규칙을 의미한다. 예를 들어, 나중에 **문맥 교환**(**context switch**)의 구현에 대해 배우게 될텐데, CPU에서 프로그램 실행을 잠시 중단하고 다른 프로그램을 실행하는 것을 문맥 교환이라고 한다. 이 **시분할** 기법은 모든 현대 운영체제들이 채택하고 있다.

팁: 시분할과 공간 분할 이용

시분할은 자원 공유를 운영체제가 사용하는 가장 기본 기법 중 하나이다. 한 개체가 잠깐 자원을 사용한 후, 다른 개체가 또 잠깐 자원을 사용하고, 그 다음 개체가 사용하면서 이 자원(CPU 또는 네트워크 링크 등)을 많은 개체들이 공유한다. 시분할과 자연스럽게 대응되는 개념은 **공간 분할**(**space sharing**)이 될 것이다. 공간 분할은 개체에게 공간을 분할해 준다. 공간 분할의 예로 디스크가 있다. 디스크는 자연스럽게 공간 분할할 수 있는 자원으로, 블럭이 하나의 파일에 할당되면 파일을 삭제하기 전에는 다른 파일이 할당될 가능성이 낮다.

운영체제의 지능은 **정책**(**policy**)의 형태로 표현된다. 정책은 운영체제 내에서 어떤 결정을 내리기 위한 알고리즘이다. 예를 들어, 실행 가능한 여러 프로그램들이 있을 때, 운영체제는 어느 프로그램을 실행시켜야 하는가? 운영체제의 **스케줄링 정책**(**scheduling policy**)이 이러한 결정을 내린다. 이러한 결정을 내리기 위하여 스케줄링 정책은 과거 정보(예, 직전 1분 동안 어떤 프로그램이 자주 실행되었는지), 워크로드에 관한 지식(예, 어떤 유형의 프로그램들이 실행되었는지), 및 성능 측정 결과(예, 시스템이 대화 성능 혹은 처리량을 높이려 하는지)를 이용한다.

7.1 프로세스의 개념

운영체제는 **실행 중인 프로그램**의 개념을 제공하는데, 이를 **프로세스**(**process**)라 한다. 전술한 바와 같이 프로세스는 실행 중인 프로그램이다. (특정 순간의) 프로세스를 간단하게 표현하려면, 실행되는 동안 접근했거나 영향을 받은 자원의 목록을 작성하면 된다.

프로세스의 구성 요소를 이해하기 위해서 **하드웨어 상태**(**machine state**)를 이해해야 한다. 프로그램이 실행되는 동안 하드웨어 상태를 읽거나 갱신할 수 있다. 이때 가장 중요한 하드웨어 구성 요소는 무엇일까?

프로세스의 하드웨어 상태 중 가장 중요한 구성 요소는 *메모리이다*. 명령어는 메모리에 저장된다. 실행 프로그램이 읽고 쓰는 데이터 역시 메모리에 저장된다. 프로세스가 접근할 수 있는 메모리(**주소 공간**(**address space**)이라 불림)는 프로세스를 구성하는 요소이다.

레지스터도 프로세스의 하드웨어 상태를 구성하는 요소 중 하나이다. 많은 명령어들이 레지스터를 직접 읽거나 갱신한다. 프로세스를 실행하는 데 레지스터도 빠질 수 없다.

프로세스의 하드웨어 상태를 구성하는 레지스터 중에 특별한 레지스터들이 존재한다. **프로그램 카운터**(**program counter**, **PC**)는 프로그램의 어느 명령어가 실행 중인지를 알려준다. 프로그램 카운터는 **명령어 포인터**(**instruction pointer**, **IP**)라고도 불린다. **스택 포인터**(**stack pointer**)와 **프레임 포인터**(**frame pointer**)는 함수의 변수와 리턴 주소를 저장하는 스택을 관리할 때 사용하는 레지스터이다.

프로그램은 영구 저장장치(persistent storage)에 접근하기도 한다. 이 입출력 *정보*는 프로세스가 현재 열어 놓은 파일 목록을 가지고 있다.

팁: 정책과 구현의 분리

많은 운영체제에서 공통된 설계 패러다임은 고수준 정책을 저수준 기법으로부터 분리하는 것이다 [Lev+75]. 기법은 시스템에 관한 "어떻게"라는 질문에 답을 제공하는 것이라고 생각할 수 있다. 예를 들어, 운영체제는 어떻게 문맥 교환을 하는가? 등이 해당된다. 정책은 "어느 것"이라는 질문에 답한다. 예를 들어, 운영체제는 지금 당장 어느 프로세스를 실행시켜야 하는가? 등이 해당된다. 둘을 분리하면 정책을 변경할 때 기법의 변경을 고민하지 않아도 된다. 따라서 일반적인 소프트웨어 설계 원칙인 **모듈성(modularity)**의 한 형태이다.

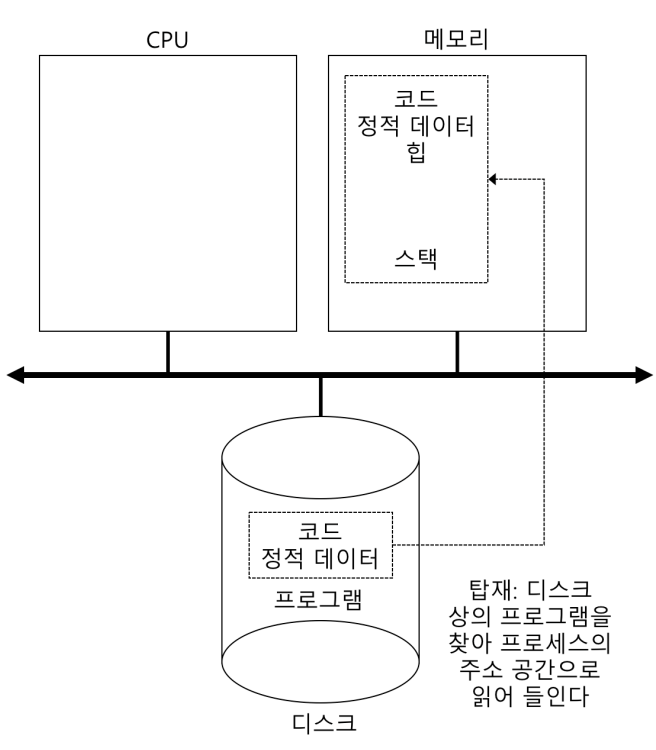
7.2 프로세스 API

실제 프로세스 API에 대한 논의는 다음의 장으로 미루겠지만, 운영체제가 반드시 API로 제공해야 하는 몇몇 기본 기능에 대해 간단히 살펴본다. 이 API들은 형태는 다르지만 모든 현대 운영체제에서 제공된다.

- 생성(Create): 운영체제는 새로운 프로세스를 생성할 수 있는 방법을 제공해야 한다. 쉘에 명령어를 입력하거나, 응용 프로그램의 아이콘을 더블-클릭하여 프로그램을 실행시키면, 운영체제는 새로운 프로세스를 생성한다.
- 제거(Destroy): 프로세스 생성 인터페이스를 제공하는 것처럼 운영체제는 프로세스를 강제로 제거할 수 있는 인터페이스를 제공해야 한다. 물론, 많은 프로세스는 실행되고 할 일을 다하면 스스로 종료한다. 그러나 프로세스가 스스로 종료하지 않으면 사용자는 그 프로세스를 제거하길 원할 것이고, 필요없는 프로세스를 중단시키는 API는 매우 유용하다.
- 대기(Wait): 때론 어떤 프로세스의 실행 중지를 기다릴 필요가 있기 때문에 여러 종류의 대기 인터페이스가 제공된다.
- 각종 제어(Miscellaneous Control): 프로세스의 제거, 대기 이외에, 여러 가지 제어 기능들이 제공된다. 예를 들어, 대부분의 운영체제는 프로세스를 일시정지하거나 재개(일시정지되었던 프로세스의 실행을 다시 시작)하는 기능을 제공한다.
- 상태(Status): 프로세스 상태 정보를 얻어내는 인터페이스도 제공된다. 상태 정보에는 얼마 동안 실행되었는지 또는 프로세스가 어떤 상태에 있는지 등이 포함된다.

7.3 프로세스 생성: 좀 더 자세하게

우리가 밝혀내야 할 한 가지 미스테리는 프로그램이 어떻게 프로세스로 변형되는가이다. 운영체제는 어떻게 프로그램을 준비하고 실행시키는가? 실제로 어떻게 프로세스를 생성하는가?



〈그림 7.1〉 **탑재 : 프로그램에서 프로세스로**

프로그램 실행을 위하여 운영체제가 하는 첫 번째 작업은 프로그램 코드와 정적 데이터(static data, 예를 들어, 초기값을 가지는 변수)를 메모리, 프로세스의 주소 공간에 **탑재**(**load**)하는 것이다. 프로그램은 **디스크** 또는 요즘 시스템에서는 **플래시-기반 SSD**에 특정 **실행 파일 형식**으로 존재한다. 코드와 정적 데이터를 메모리에 탑재하기 위해서 운영체제는 디스크의 해당 바이트를 읽어서 메모리의 어딘가에 저장해야 한다 (그림 7.1을 보시오).

초기 운영체제들은 프로그램 실행 전에 코드와 데이터를 모두 메모리에 탑재하였다. 현대의 운영체제들은 이 작업을 늦추었다. 즉, 프로그램을 실행하면서 코드나 데이터가 필요할 때 필요한 부분만 메모리에 탑재한다. 코드와 데이터의 늦은 탑재의 동작을 정확하게 이해하기 위해서는 **페이징**(**paging**)과 **스와핑**(**swapping**) 동작의 이해가 필요하다. 이에 대해서는 나중에 메모리 가상화를 논의할 때 자세히 다루게 될 것이다. 지금은 어떤 프로그램이든 실행시키기 전에 운영체제는 프로그램의 중요 부분을 디스크에서 메모리로 탑재해야 한다는 것만 기억하자.

코드와 정적 데이터가 메모리로 탑재된 후, 프로세스를 실행시키기 전에 운영체제가 해야 할 일이 몇 가지 있다. 일정량의 메모리가 프로그램의 **실행시간 스택**(run-time stack, 혹은 그냥 **스택**) 용도로 할당되어야 한다. 이미 알고 있겠지만 C 프로그램은 지역 변수, 함수 인자, 리턴 주소 등을 저장하기 위해 스택을 사용한다. 운영체제는 스택을 주어진 인자로 초기화한다. 특히, `main()`함수의 인자인 `argc`와 `argv` 벡터를

사용하여 스택을 초기화한다.

운영체제는 프로그램의 **힙**(**heap**)을 위한 메모리 영역을 할당한다. C 프로그램에서 힙은 동적으로 할당된 데이터를 저장하기 위해 사용된다. 프로그램은 `malloc()`을 호출하여 필요한 공간을 요청하고 `free()`를 호출하여 사용했던 공간을 반환하여 다른 프로그램이 사용할 수 있도록 한다. 힙은 연결 리스트, 해시 테이블, 트리 등 크기가 가변적인 자료 구조를 위해 사용된다. 프로그램이 실행되면 `malloc()` 라이브러리 API를 호출하여 메모리를 요청하고, 운영체제가 이를 충족하도록 메모리를 할당한다.

운영체제는 또 입출력과 관계된 초기화 작업을 수행한다. 예를 들어, Unix 시스템에서 각 프로세스는 기본적으로 표준 입력(`STDIN`), 표준 출력(`STDOUT`), 표준 에러(`STDERR`) 장치에 해당하는 세 개의 **파일 디스크립터**(**file descriptor**)를 갖는다. 이 디스크립터들을 사용하여 프로그램은 터미널로부터 입력을 읽고 화면에 출력을 프린트하는 작업을 쉽게 할 수 있다. 입출력, 파일 디스크립터 등에 관해서는 이 책의 세 번째 부분인 **영속성**에서 다룬다.

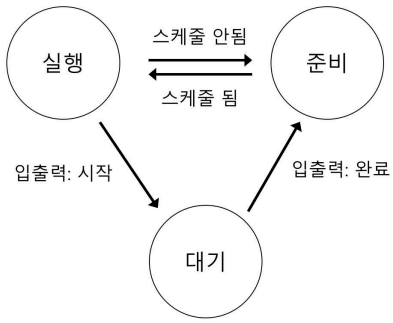
코드와 정적 데이터를 메모리에 탑재하고, 스택과 힙을 생성하고 초기화하고, 입출력 셋업과 관계된 다른 작업을 마치게 되면, 운영체제는 프로그램 실행을 위한 준비를 마치게 된다. 프로그램의 시작 지점(entry point), 즉 `main()`에서부터 프로그램 실행을 시작하는 마지막 작업만이 남는다. `main()` 루틴으로 분기함으로써 (다음 장에서 이를 가능하게 하는 특수 기법을 설명할 것이다), 운영체제는 CPU를 새로 생성된 프로세스에게 넘기게 되고 프로그램 실행이 시작된다.

7.4 프로세스 상태

프로세스가 무엇인지(이 개념에 대해 명확히 정의하겠지만), 어떻게 생성되는지(대강이라도) 알게 되었으므로, 프로세스의 **상태**(**state**)에 대해 논의해 보자. 프로세스 상태의 개념은 초기 컴퓨터 시스템에서 등장하였다 [DH66; VCG65]. 프로세스 상태를 단순화하면 다음 세 상태 중 하나에 존재할 수 있다.

- 실행(Running): 실행 상태에서 프로세스는 프로세서에서 실행 중이다. 즉, 프로세스는 명령어를 실행하고 있다.
- 준비(Ready): 준비 상태에서 프로세스는 실행할 준비가 되어 있지만 운영체제가 다른 프로세스를 실행하고 있는 등의 이유로 대기 중이다.
- 대기(Blocked): 프로세스가 다른 사건을 기다리는 동안 프로세스의 수행을 중단시키는 연산이다. 흔한 예: 프로세스가 디스크에 대한 입출력 요청을 하였을 때 프로세스는 입출력이 완료될 때까지 대기 상태가 되고, 다른 프로세스가 실행 상태로 될 수 있다.

이러한 상태를 그림으로 표현하면 그림 7.2와 같이 될 것이다. 그림에서 보듯이 프로세스는 준비 상태와 실행 상태를 운영체제의 정책에 따라 이동한다. 프로세스는 운영체제의 스케줄링 정책에 따라 **스케줄**이 되면 준비 상태에서 실행 상태로 전이한다.



〈그림 7.2〉 **프로세스 : 상태 전이**

실행 상태에서 준비 상태로의 전이는 프로세스가 나중에 다시 스케줄 될 수 있는 상태가 되었다는 것을 의미한다. 프로세스가 입출력 요청 등의 이유로 대기 상태가 되면 요청 완료 등의 이벤트가 발생할 때까지 대기 상태로 유지된다. 이벤트가 발생하면 프로세스는 다시 준비 상태로 전이되고 운영체제의 결정에 따라 바로 다시 실행될 수도 있다.

두 개의 프로세스가 어떻게 전이될 수 있는지를 보여주는 예를 보자. 실행 중인 두 프로세스가 있다고 하자. 각 프로세스는 오직 CPU만 사용하고 입출력을 행하지 않는다. 이 경우, 각 프로세스의 상태 전이는 그림 7.3과 같은 모양이 될 수 있다.

시간	Process0	Process1	비고
1	실행	준비	
2	실행	준비	
3	실행	준비	
4	실행	준비	Process0 종료
5		실행	
6		실행	
7		실행	
8		실행	Process1 종료

〈그림 7.3〉 **프로세스 상태 추이 : CPU만 이용할 때**

다음 예에서는 첫 번째 프로세스가 어느 정도 실행한 후에 입출력을 요청한다. 그 순간 프로세스는 대기 상태가 되고 다른 프로세스에게 실행 기회를 준다. 그림 7.4가 시나리오의 진행 과정을 보인다.

자세히 살펴보면, Process0은 입출력을 요청하고 요청한 작업이 완료되기를 기다린다. 프로세스는 디스크를 읽거나 네트워크로부터 패킷을 기다릴 때 대기 상태로 전이한다. 운영체제는 Process0이 CPU를 사용하지 않는다는 것을 감지하고 Process1을 실행시킨다. Process1이 실행되는 동안 입출력은 완료되고 Process0은 준비 상태로 다시 전이된다. 결국 Process1은 종료되고 Process0이 실행되어 종료된다.

위와 같은 간단한 예에서조차 운영체제가 내려야 할 결정이 매우 많다는 사실에

시간	Process0	Process1	비고
1	실행	준비	
2	실행	준비	
3	실행	준비	Process0이 입출력을 시작한다.
4	대기	실행	Process0은 대기 상태가 되고,
5	대기	실행	Process1이 실행된다.
6	대기	실행	
7	준비	실행	Process0 입출력 종료
8	준비	실행	Process1 종료
9	실행	-	
10	실행	-	Process0 종료

〈그림 7.4〉 **프로세스 상태 추이 : CPU 이용과 입출력 작업을 할 때**

주목해야 한다. 우선 시스템은 Process0이 입출력을 요청할 때 Process1의 실행여부를 결정해야 한다. Process1을 실행키로한 결정은 CPU를 계속 동작시키므로 자원 이용률을 높인다. 또한, 시스템은 Process0이 요청한 입출력이 완료되었을 때, Process0을 바로 실행하지 않고 실행 중이던 Process1을 계속 실행하였다. 이러한 결정이 좋은 결정이었는지는 확실하지 않다. 당신은 어떻게 생각하는가? 운영체제는 스케줄러를 통해 이러한 결정을 내린다. 운영체제의 스케줄러는 차후에 자세히 다룰 것이다.

여담 : 자료 구조—프로세스 리스트

운영체제에는 다양한 중요 **자료 구조**들이 많이 존재한다. 프로세스 리스트가 그 중 첫 번째이다. 이 자료 구조는 단순하다. 다수의 프로그램을 동시에 실행할 수 있는 모든 운영체제는 이와 유사한 자료 구조를 가지고 있고, 이 자료 구조를 이용하여 시스템에서 실행 중인 프로그램을 관리한다. 프로세스의 관리를 위한 정보를 저장하는 자료 구조를 **프로세스 제어 블럭**(**Process Control Block**, **PCB**)이라 부른다. 각 프로세스에 관한 정보를 저장하는 C 자료 구조를 이야기 할 때 부르는 멋진 이름이다.

7.5 자료 구조

운영체제도 일종의 프로그램이다. 다른 프로그램들과 같이 다양한 정보를 유지하기 위한 자료 구조를 가지고 있다. 예를 들어, 프로세스 상태를 파악하기 위해 준비 상태의 프로세스들을 위한 **프로세스 리스트**(**process list**)와 같은 자료 구조를 유지한다. 또한, 어느 프로세스가 실행 중인지를 파악하기 위한 부가적인 자료 구조도 유지한다. 운영체제는 또 대기 상태인 프로세스도 파악해야 한다. 입출력 요청이 완료되면 운영체제는 적절한 프로세스를 깨워 준비 상태로 다시 전이시킬 수 있어야 한다.

그림 7.5는 xv6 커널에서 각 프로세스를 추적하기 위해 운영체제가 필요로 하는 정보를 보이고 있다 [Cox+08]. Linux, Mac OS X, 또는 Windows 같은 운영체제들도 이와 비슷한 프로세스 구조를 가지고 있다. 찾아 보면 훨씬 복잡하다는 것을 알 수 있을 것이다.

그림을 통해서 운영체제가 관리하는 있는 프로세스 정보를 알 수 있다. **레지스터 문맥**(**register context**) 자료 구조는 프로세스가 중단되었을 때 해당 프로세스의 레지스터값들을 저장한다. 이 레지스터값들을 복원하여(예, 해당 값을 실제 물리 레지스터에 다시 저장함으로써) 운영체제는 프로세스 실행을 재개한다. **문맥 교환**(**context switch**)이라고 알려진 이 기법에 관해서 나중에 좀 더 자세히 다루기로하자.

```
// 프로세스를 중단하고 이후에 재개하기 위해
// xv6가 저장하고 복원하는 레지스터
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// 가능한 프로세스 상태
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
// 레지스터 문맥과 상태를 포함하여
// 각 프로세스에 대하여 xv6가 추적하는 정보
struct proc {
    char *mem;                // 프로세스 메모리 시작 주소
    uint sz;                  // 프로세스 메모리의 크기
    char *kstack;             // 이 프로세스의 커널 스택의 바닥 주소
    enum proc_state state;    // 프로세스 상태
    int pid;                  // 프로세스 ID
    struct proc *parent;      // 부모 프로세스
    void *chan;               // 0이 아니면, chan에서 수면
    int killed;               // 0이 아니면 종료됨
    struct file *ofile[NOFILE];   // 열린 파일
    struct inode *cwd;        // 현재 디렉터리
    struct context context;   // 프로세스를 실행시키려면 여기로 교환
    struct trapframe *tf;     // 현재 인터럽트에 해당하는 트랩 프레임
};
```

〈그림 7.5〉 **xv6 Proc 구조**

그림에서 실행, 준비, 대기 외에 다른 상태들이 존재하는 것을 볼 수 있다. **초기**(**initial**) 상태를 가지는 시스템도 있다. 프로세스가 생성되는 동안에는 초기 상태에 머무른다. 프로세스는 종료되었지만 메모리에 남아있는 상태인 **최종**(**final**)상태도 있다 (Unix-기반 시스템에서 이 상태는 **좀비**(**zombie**) 상태[1]라고 불린다). 이 상태는 프로세스가 성공적으로 실행했는지를 다른 프로세스(보통은 **부모**(**parent**) 프로세스)가 검사하는 데 유용하다. 이를 위하여 최종 상태를 활용한다(Unix-기반 시스템에서는 프로세스가 성공적으로 종료되었으면 0을, 그렇지 않으면 0이 아닌 값을 반환한다). 부모

1) 이 좀비들은 제거 명령으로 쉽게 제거할 수 있으나 일반적으로 다른 방법을 사용하여 제거한다.

프로세스는 자식 프로세스의 종료를 대기하는 시스템 콜을 호출(예, `wait()`) 한다. 이 호출은 종료된 프로세스와 관련된 자원들을 정리할 수 있다고 운영체제에 알리는 역할도 한다.

7.6 요약

운영체제의 기본 개념인 프로세스를 소개하였다. 프로세스는 간단히 말하면 실행 중인 프로그램이다. 이 개념을 염두에 두고 이제 좀 더 구체적인 핵심 사항을 살펴볼 것이다. 여기서 핵심 사항은 프로세스 구현에 필요한 기법, 구현한 프로세스를 스케줄링하는 정책 등을 뜻한다. 이러한 여러 방법들을 조합하여 운영체제가 CPU를 가상화하는 방식을 이해하자.

여담: 시뮬레이션 숙제

시뮬레이션 숙제는 논의된 주제를 이해하였는지 확인할 수 있도록 실행 가능한 시뮬레이터 형태로 제공된다. 시뮬레이터는 통상 python 프로그램 형태로 제공된다. 이 프로그램을 사용하면 랜덤 시드를 이용하여 다른 문제를 출제할 수 있고 `-c` 플래그를 지정하면 주어진 문제에 대한 해답을 만들 수 있다. 이렇게 생성된 해결책과 비교하여 당신의 해결책을 점검할 수 있다. `-h` 또는 `--help` 옵션을 지정하고 시뮬레이터를 실행시키면 시뮬레이터가 제공하는 모든 옵션에 대한 정보를 얻을 수 있다.

각 시뮬레이터와 함께 제공되는 README 파일은 시뮬레이터를 실행시키는 방법에 대해 상세히 설명한다. 각 플래그에 대한 상세한 설명을 확인할 수 있다.

참고 문헌

[Cox+08] **"The xv6 Operating System"**
Russ Cox, Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
URL: `http://pdos.csail.mit.edu/6.828/2008/index.html`
세상에서 가장 멋지고 작은 실제 운영체제. 운영체제의 상세한 동작을 배우려면 다운로드하여 가지고 놀아 보아라.

[DH66] **"Programming Semantics for Multiprogrammed Computations"**
Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
이 논문은 멀티프로그래밍 시스템과 관련된 많은 초기 용어와 개념을 정의한다.

[Han70] **"The Nucleus of a Multiprogramming System"**
Per Brinch Hansen
Communications of the ACM, Volume 13, Number 4, April 1970
이 논문은 운영체제 역사상 첫 번째 마이크로커널 중의 하나인 Nucleus를 소개한다. 크기가 작고 꼭 필요한 기능만 가진 시스템을 구현하려는 아이디어는 운영체제 역사에서 반복해서 등장하는 주제이다. 그 모든 것이 이 논문에서 설명된 Brinch Hansen의 연구와 함께 시작되었다.

[Lev+75] **"Policy/mechanism separation in Hydra"**
R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf
SOSP 1975
Hydra로 알려진 연구 운영체제의 구조에 관한 초기 논문. Hydra는 주류 운영체제가 된 적은 없지만 몇몇 아이디어는 운영체제 설계자에게 많은 영향을 주었다.

[VCG65] **"Structure of the Multics Supervisor"**
V.A. Vyssotsky, F.J. Corbato, and R.M. Graham
Fall Joint Computer Conference, 1965
Multics에 관한 초기 논문. 이 논문은 현대 시스템에서 발견되는 많은 기본 아이디어와 용어를 설명한다. 유틸리티 컴퓨팅의 배경이 되는 몇몇 선견지명은 마침내 현대 클라우드 시스템에서 구현되었다.

숙제

process-run.py 프로그램은 프로세스가 실행되면서 변하는 프로세스의 상태를 추적할 수 있고, 프로세스가 CPU를 사용하는지(예, add 명령어 실행) 입출력을 하는지(예, 디스크에 요청을 보내고 완료되기를 기다린다)를 알아볼 수 있다. 상세한 사항은 README 파일을 확인하기 바란다.

문제

1. 다음과 같이 플래그를 지정하고 프로그램을 실행시키시오: ./process-run.py -l 5:100, 5:100. CPU 이용률은 얼마가 되어야 하는가(예, CPU가 사용 중인 시간의 퍼센트?) 그러한 이용률을 예측한 이유는 무엇인가? -c 플래그를 지정하여 예측이 맞는지 확인하시오.
2. 이제 다음과 같이 플래그를 지정하고 실행시키시오: ./process-run.py -l 4:100, 1:0. 이 플래그는 4개의 명령어를 실행하고 모두 CPU만 사용하는 하나의 프로세스와 오직 입출력을 요청하고 완료되기를 기다리는 하나의 프로세스를 명시한다. 두 프로세스가 모두 종료되는 데 얼마의 시간이 걸리는가? -c 플래그를 사용하여 예측한 것이 맞는지 확인하시오.
3. 옵션으로 지정된 프로세스의 순서를 바꾸시오: ./process-run.py -l 1:0, 4:100. 이제 어떤 결과가 나오는가? 실행 순서를 교환하는 것은 중요한가? 이유는 무엇인가? (언제나처럼 -c 플래그를 사용하여 예측이 맞는지 확인하시오.)
4. 자, 다른 플래그에 대해서도 알아보자. 중요한 플래그 중 하나는 -S로서 프로세스가 입출력을 요청했을 때 시스템이 어떻게 반응하는지를 결정한다. 이 플래그가 SWITCH_ON_END로 지정되면 시스템은 요청 프로세스가 입출력을 하는 동안 다른 프로세스로 전환하지 않고 대신 요청 프로세스가 종료될 때까지 기다린다. 입출력만 수행하는 프로세스와 CPU 작업만 하는 프로세스 두 개를 실행시키면 어떤 결과가 발생하는가?(-l 1:0,4:200 -c -S SWITCH_ON_END)
5. 이번에는 프로세스가 입출력을 기다릴 때마다 다른 프로세스로 전환하도록 플래그를 지정하여 같은 프로세스를 실행시켜 보자 (-l 1:0,4:100 -c -S SWITCH_ON_IO). 이제 어떤 결과가 발생하는가? -c를 사용하여 예측이 맞는지 확인하시오.
6. 또 다른 중요한 행동은 입출력이 완료되었을 때 무엇을 하느냐이다. -I IO_RUN_LATER가 지정되면 입출력이 완료되었을 때 입출력을 요청한 프로세스가 바로 실행될 필요가 없다. 완료 시점에 실행 중이던 프로세스가 계속 실행된다. 다음과 같은 조합의 프로세스를 실행시키면 무슨 결과가 나오는가? (./process-run.py -l 3:0, 5:100, 5:100, 5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p) 시스템 자원은 효과적으로 활용되는가?
7. 같은 프로세스 조합을 실행시킬 때 -I IO_RUN_IMMEDIATE를 지정하고 실행시키시오. 이 플래그는 입출력이 완료되었을 때 요청 프로세스가 곧바로 실행되는 동작을

의미한다. 이 동작은 어떤 결과를 만들어 내는가? 방금 입출력을 완료한 프로세스를 다시 실행시키는 것이 좋은 생각일 수 있는 이유는 무엇인가?

8. 이제 다음과 같이 무작위로 생성된 프로세스를 실행시켜 보자. 예를 들면, `-s 1 -l 3:50,3:50`, `-s 2 -l 3:50,3:50`, `-s 3 -l 3:50,3:50`. 어떤 양상을 보일지 예측할 수 있는지 생각해 보시오. `-I IO_RUN_IMMEDIATE`를 지정했을 때와 `-I IO_RUN_LATER`를 지정했을 때 어떤 결과가 나오는가? `-S SWITCH_ON_IO` 대 `-S SWITCH_ON_END`의 경우에는 어떤 결과가 나오는가?