10

스케줄링 : 개요

이제 프로세스를 실행시키기 위한 문맥 교환 등의 저수준 **기법**에 대해서는 분명하게 이해하고 있어야 한다. 그렇지 않다면 한 두 장 이전으로 돌아가서 이러한 기법들이 어떻게 동작하는지에 대한 설명을 다시 읽기 바란다. 운영체제 스케줄러의 고수준 **정책**에 관해서는 이해가 필요하다. 이제부터 다양한 **스케줄링 정책**(**scheduling policy**)을 소개하고 그에 관한 이해를 높일 것이다. 이러한 정책은 **원칙**(**discipline**)이라고도 불리며 많은 똑똑하고 부지런한 사람들이 오랫동안 개발한 정책들이다.

사실 스케줄링의 기원은 컴퓨터 시스템 개발 이전으로 거슬러 올라간다. 초기의 방법들은 생산 관리 분야에서 개발되었으며 컴퓨터에 적용되었다. 이 사실은 그렇게 놀라운 일은 아니다. 생산 공정과 기타 많은 인간의 행동 역시 스케줄링이 필요하고, 효율성에 대한 요구 등, 공통된 해결 사항이 존재하기 때문이다. 우리의 문제는 다음과 같다.

핵심 질문 : 스케줄링 정책은 어떻게 개발하는가
스케줄링 정책을 생각하기 위한 기본적인 프레임워크를 어떻게 만들어야 하는가? 핵심 가정은 무엇인가? 어떤 평가 기준이 중요한가? 컴퓨터 시스템의 초창기에 사용되었던 기본 접근법은 무엇인가?

10.1 워크로드에 대한 가정

가장 먼저 프로세스에 대하여 몇 가지 가정을 할 것이다. 일련의 프로세스들이 실행하는 상황을 **워크로드**(**workload**)라고 부르기로 한다. 워크로드를 결정하는 것은 정책 개발에 매우 중요한 부분이다. 워크로드에 관해 더 잘 알수록 그에 맞게 정책을 정교하게 손질할 수 있다.

우리가 지금 설명에서 사용할 워크로드에 대한 가정은 비현실적이다. 하지만, 당장 논의하는 데 아무 문제 없다. 앞으로 논의가 진행됨에 따라 가정을 완화시킬 것이고 최종적으로 **제대로 동작하는 스케줄링 정책**을 만들게 될 것이다.

우리는 시스템에서 실행 중인 프로세스 혹은 작업(job)에 대해 다음과 같은 가정을 한다.

1. 모든 작업은 같은 시간 동안 실행된다.

2. 모든 작업은 동시에 도착한다.

3. 각 작업은 시작되면 완료될 때까지 실행된다.

4. 모든 작업은 CPU만 사용한다(즉, 입출력을 수행하지 않는다).

5. 각 작업의 실행 시간은 사전에 알려져 있다.

이러한 가정들이 비현실적이라고 이미 언급하였다. Animal Farm [Orw45]에서 어떤 동물은 다른 동물보다 더 평등하다고 한 것처럼 몇몇 가정은 다른 가정에 비해 더 비현실적이다. 특히, 각 작업의 실행 시간이 미리 알려져 있다는 가정은 더욱 그렇다. 이 가정은 스케줄러가 모든 것을 다 알 수 있다는 것을 의미한다. 좋은 일이기는 하지만 가까운 미래에 나올 것 같지는 않다.

10.2 스케줄링 평가 항목

워크로드에 대한 가정 이외에 스케줄링 정책의 비교를 위해 **스케줄링 평가 항목**(**scheduling metric**)을 결정해야 한다. 스케줄링의 경우 다양한 평가 기준이 존재한다.

우선, 문제를 간단하게 하기 위해 **반환 시간**(**turnaround time**)이라는 하나의 평가 기준을 사용한다. 작업 반환 시간은 작업이 완료된 시각에서 작업이 시스템에 도착한 시각을 뺀 시간으로 정의된다. 반환 시간 $T_{turnaround}$는

$$T_{turnaround} = T_{completion} - T_{arrival} \tag{10.1}$$

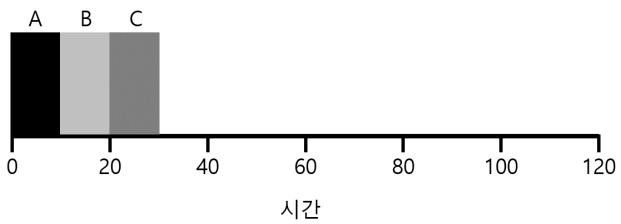
모든 작업은 동시에 도착한다고 가정했으므로 $T_{arrival}$=0으로 생각할 수 있다. $T_{turnaround} = T_{completion}$이다. 이 가정은 완화해 나갈 것이다.

반환 시간은 **성능** 측면에서의 평가 기준이라는 것을 주목해야 한다. 다른 평가 기준으로 **공정성**(**fairness**)이다. 예를 들면 **Jain's Fairness Index** [Jai91]에 따라 측정된다. 성능과 공정성은 스케줄링에서는 서로 상충되는 목표이다. 예를 들어, 스케줄러는 성능을 극대화하기 위해 몇몇 작업의 실행을 중지하며, 결과적으로 공정성이 악화된다. 인생은 대부분 완벽하지 않다.

10.3 선입선출

가장 기초적인 알고리즘은 **선입선출**(**First In First Out**, **FIFO**) 또는 **선도착선처리**(**First Come First Served**, **FCFS**) 스케줄링이라고 알려져 있다. FIFO는 많은 장점을 가진다. 단순하고 구현하기 쉽다. 우리의 가정 하에서는 매우 잘 동작한다.

간단한 예를 스케줄 해 보자. 시스템에 3개의 작업 A, B, C가 거의 동시에 도착했다고 가정하자 ($T_{arrival} = 0$). 간발의 차이로 A, B, C 순서대로 도착했다고 가정하자. 또한, 각 작업은 10초 동안 실행된다고 가정하자. 이 작업들의 평균 반환 시간은 얼마인가?



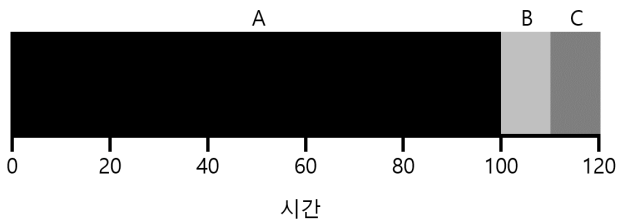
〈그림 10.1〉 **FIFO 간단한 예**

그림 10.1에서 A는 10, B는 20, C는 30에 종료했다는 것을 알 수 있다. 세 작업의 평균 반환 시간은 $\frac{(10+20+30)}{3} = 20$이다. 반환 시간의 계산은 쉽다.

자 이제 가정 중 한 가지를 완화하자. 구체적으로 1번 가정을 완화하여 작업을 실행 시간이 모두 같지 않다고 가정하자. FIFO는 어떻게 동작하는가? FIFO는 어떤 워크로드에서 좋지 않은 성능을 보일까?

(다음으로 넘어가기 전에 생각해 보자... 생각에 생각... 알았지?!)

FIFO 스케줄링이 어떤 문제를 야기하는지 같이 생각해 보자. 구체적으로 다시 A, B, C 세 개의 작업을 가정하자. 그러나 이번에는 A는 100초, B와 C는 10초 동안 실행된다.



〈그림 10.2〉 **FIFO가 그렇게 좋은 스케줄링이 아닌 이유**

그림 10.2에서 볼 수 있듯이, 작업 A가 B와 C 보다 먼저 100초 동안 실행된다. 따라서 시스템의 평균 반환 시간은 110초로 늘어난다. $\frac{(100+110+120)}{3} = 110$.

이 문제점은 **convoy effect** [Bla+79]라고 칭해지며 짧은 시간 동안 자원을 사용할 프로세스들이 자원을 오랫동안 사용하는 프로세스의 종료를 기다리는 현상을 말한다. 이 스케줄링 시나리오는 슈퍼마켓에서 줄 서서 계산을 기다릴 때, 앞사람이 카트 세 개에 물건을 가득 싣고 있는 경우와 느낌이 비슷하다. 한참 걸리겠지?[1]

그러면 어떻게 해야 할까? 작업 실행 시간이 다른 경우 더 좋은 알고리즘은 무엇인가? 먼저 생각해 보아라. 그런 후 다음 내용을 읽기 바란다.

1) 이 경우 빨리 다른 줄에 가서 서거나 길고 깊게 심호흡을 하라고 권하고 싶다. 오케이, 들이쉬고 내쉬고.

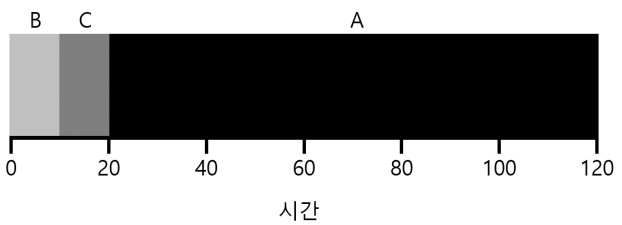
팁 : SJF의 원칙

최단 작업 우선 원칙은 평균 반환 시간이 (혹은 우리의 경우, 작업당) 강조되는 모든 시스템에 적용될 수 있는 일반적인 스케줄링 원리이다. 당신이 기다린 경험이 있는 모든 줄을 생각해 보라. 문제의 장소가 고객 만족을 중요시한다면 SJF의 사용을 고려했을 가능성이 높다. 예를 들어, 식료품 가게들은 적은 개수의 물건을 구매한 사람들이 다가올 겨울을 준비하는 가족 뒤에 서서 마냥 기다리지 않도록 "소량 구매 계산대"를 갖추고 있다.

10.4 최단 작업 우선

이 문제는 간단히 해결할 수 있다. 사실 이 아이디어는 오퍼레이션 리서치 분야에서 차용하여 [Cob54; JV56] 컴퓨터 시스템의 작업 스케줄링에 적용한 것이다. 이 새로운 스케줄링 개념용 **최단 작업 우선**(**Shortest Job First**, **SJF**)으로 이름 자체가 정책을 설명하고 있기 때문에 기억하기 쉽다. 이 원칙은 가장 짧은 실행 시간을 가진 작업을 먼저 실행시킨다.

위의 예를 이번에는 SJF로 스케줄 해 보자. 그림 10.3이 A, B, C를 실행시킨 결과를 보이고 있다. SJF가 평균 반환 시간 기준으로 더 나은 성능을 보이는지 보인다. B, C, A 순서로 실행시킴으로써 SJF는 평균 반환 시간을 110초에서 50초($\frac{(10+20+120)}{3} = 50$)로 2배 이상 향상시켰다.



〈그림 10.3〉 **SJF 간단한 예**

모든 작업이 동시에 도착한다면 SJF가 **최적**(**optimal**)의 스케줄링 알고리즘임을 증명할 수 있다. 지금은 이론이나 오퍼레이션 리서치 수업이 아니라 시스템 수업을 듣고 있기 때문에 이에 대한 증명은 하지 않을 것이다.

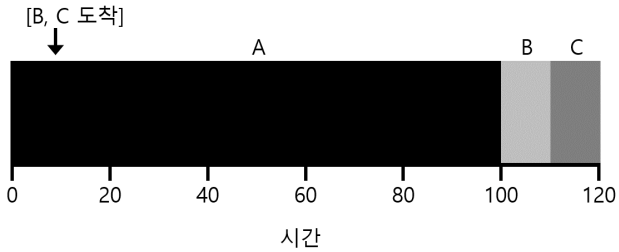
SJF라는 좋은 스케줄링 기법을 개발하였지만 가정이 아직 비현실적이다. 이제 가정을 완화해 보자. 특히, 가정 2를 완화하여 이제 모든 작업은 동시에 도착하는 것이 아니라 임의의 시간에 도착할 수 있다고 가정한다. 이러면 어떤 문제가 발생하는가?

(또 생각할 시간... 생각하고 있는 거지?... 자, 할 수 있어!)

자 문제가 무엇인지 설명하기 위하여 예를 들어 보자. 이번에는 A는 t = 0에 도착하고 100초의 실행 시간을 가지고 있고, 반면에 B와 C는 t = 10에 도착하고 각각 10초의 실행

별 문제 아니다, 걱정하지말자.

시간을 가진다고 가정하자. 순수한 SJF로 스케줄 했을 경우의 스케줄이 그림 10.4에 나와 있다.



〈그림 10.4〉 **B와 C가 늦게 도착한 경우의 SJF 스케줄**

그림에서 알 수 있듯이 B와 C가 A 바로 뒤에 도착한다고 하더라도 A가 끝날 때까지 기다릴 수밖에 없어서 이전의 convoy 문제가 다시 발생한다. 이 경우 평균 반환 시간은 103.33($\frac{(100+(110-10)+(120-10))}{3}$)이다. 스케줄러는 어떻게 스케줄링 해야 할까?

여담: 선점형 스케줄러

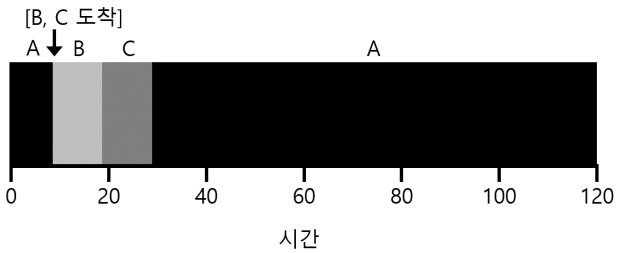
예전 일괄처리 시절에 많은 **비선점**(**non-preemptive**) 스케줄러가 개발되었다. 이 시스템은 각 작업이 종료될 때까지 계속 실행한다. 이론적으로 모든 현대 스케줄러는 **선점**형이고, 다른 프로세스를 실행시키기 위하여 필요하면 현재 프로세스의 실행을 중단한다. 이 사실은 스케줄러가 앞에서 배웠던 기법을 채용하고 있다는 것을 의미한다. 구체적으로 스케줄러는 **문맥 교환**을 수행할 수 있고, 실행 중인 프로세스를 잠시 중단시킬 수 있으며 다른 프로세스를 재개 또는 시작시킬 수 있다.

10.5 최소 잔여시간 우선

이 문제를 해결하기 위해서는 가정 3 (작업은 끝날 때까지 계속 실행된다)을 완화해야 한다. 이제 그 작업을 해 보자. 타이머 인터럽트와 문맥 교환을 고려하면 B와 C가 도착했을 때 스케줄러는 다른 일을 할 수 있다. 작업 A를 **중지**하고 지금 도착한 B나 C를 실행하기로 결정할 수도 있다. 물론, 선점된 A는 나중에 다시 실행된다. SJF는 **비선점**형 스케줄러이기 때문에 이와 같은 동작을 하지 못한다.

다행히도 이렇게 동작할 수 있는 스케줄러는 이미 존재한다. SJF에 선점 기능을 추가한 **최단 잔여시간 우선**(**Shortest Time-to-Completion First**, **STCF**) 또는 선점형 최단 작업 우선 (PSJF)으로 알려진 스케줄러이다 [CK68]. 언제든 새로운 작업이 시스템에 들어오면, 이 스케줄러는 남아 있는 작업과 새로운 작업의 잔여 실행 시간을 계산하고 그 중 가장 적은 잔여 실행 시간을 가진 작업을 스케줄한다. 우리의 예에서는

STCF는 A를 선점하고 B와 C를 끝날 때까지 실행시킨다. 두 작업이 모두 끝난 후에 A가 스케줄되어 남은 실행 시간 만큼 실행된다. 그림 10.5의 스케줄 예를 보라.



〈그림 10.5〉 **STCF의 간단한 예**

그 결과 평균 반환 시간이 단축되어 50초 ($\frac{((120-0)+(20-10)+(30-10))}{3}$)가 된다. 먼저와 같이, 새로운 가정 하에서 STCF가 최적의 스케줄러이다. 작업들이 동시에 도착할 경우, SFJ가 최적의 결과를 낸다는 것을 고려하면, STCF가 최적의 스케줄링이 되는 이유를 쉽게 알 수 있을것이다.

10.6 새로운 평가 기준 : 응답 시간

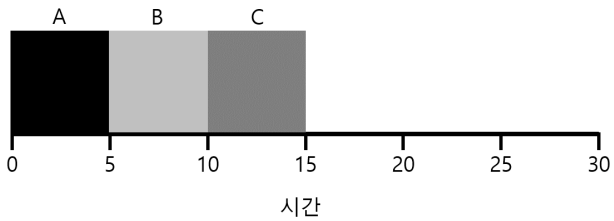
작업의 길이를 미리 알고 있고, 작업이 오직 CPU만 사용하며, 평가 기준이 반환 시간 하나라면, STCF는 매우 훌륭한 정책이다. 초기 일괄처리 컴퓨터 시스템에서는 이러한 스케줄링 알고리즘이 의미가 있었다. 그러나 시분할 컴퓨터의 등장이 모든 것을 바꾸었다. 이제 사용자는 터미널에서 작업하게 되어 시스템에게 상호작용을 원활히 하기 위한 성능을 요구하게 되었다. **응답 시간**(**response time**)이라는 새로운 평가 기준이 태어나게 된다.

응답 시간은 작업이 도착할 때부터 처음 스케줄 될 때까지의 시간으로 정의된다. 다음과 같다.

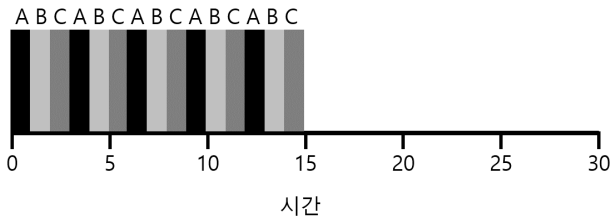
$$T_{response} = T_{firstrun} - T_{arrival} \tag{10.2}$$

예를 들어, 위와 같은 스케줄의 (A는 시간 0, B 와 C는 시간 10에 도착) 경우, 각 작업의 응답 시간은 A는 0, B는 0, C는 10, 평균 3.33이 된다.

이미 생각하고 있는지 모르지만, STCF를 비롯하여 비슷한 류의 정책들은 응답 시간이 짧다고 할 수 없다. 예를 들어, 3개의 작업이 동시에 도착한 경우, 세 번째 작업은 딱 한 번 스케줄 되기 위해 먼저 실행된 두 작업이 *완전히 끝날 때까지* 기다린다. 반환 시간 기준으로는 매우 훌륭한 반면, 응답 시간과 상호작용 측면에서는 매우 나쁜 방법이다. 터미널 앞에 앉아 입력한 후, 단지 다른 작업이 당신의 작업보다 먼저 스케줄 되었다는 이유로 시스템의 응답이 올 때까지 10초 기다리는 것을 상상해 보자. 별로 즐거운 일은 아니다.



〈그림 10.6〉 **다시 SJF 스케줄링(응답 시간은 좋지 않음)**



〈그림 10.7〉 **라운드 로빈(응답 시간이 좋음)**

우리는 다른 문제를 해결해야 한다. 응답 시간에 민감한 스케줄러를 어떻게 만들 수 있는가?

10.7 라운드 로빈

응답 시간 문제를 해결하기 위하여 **라운드 로빈**(**Round-Robin**, **RR**) 스케줄링 [Kle64]이라 불리는 스케줄링 알고리즘을 도입한다. 기본 발상은 간단하다. RR은 작업이 끝날 때까지 기다리지 않는다. 대신 일정 시간 동안 실행한 후 실행 큐의 다음 작업으로 전환한다. 이때 작업이 실행되는 일정 시간을 **타임 슬라이스**(**time slice**) 또는 **스케줄링 퀀텀**(**scheduling quantum**)이라 부른다. 작업이 완료될 때까지 이런 식으로 계속 진행된다. 이러한 이유로 RR은 때때로 **타임 슬라이싱**이라고 불린다. 타임 슬라이스의 길이는 타이머 인터럽트 주기의 배수여야 한다. 타이머가 10 msec 마다 인터럽트를 발생시킨다면 타임 슬라이스는 10, 20 등 10 msec의 배수가 될 수 있다.

RR을 더 자세하게 이해하기 위해 하나의 예를 살펴보자. 3개의 작업 A, B, C가 시스템에 동시에 도착하고, 각각 5초간 실행된다고 가정한다. SJF 스케줄러는 다른 작업을 실행하기 전에 각 작업을 종료할 때까지 실행한다(그림 10.6). 대조적으로, 1초의 타임 슬라이스를 가지는 RR은 작업을 빠르게 순환하게 된다(그림 10.7). RR의 평균 응답 시간은 $\frac{(0+1+2)}{3} = 1$이고, SJF의 평균 응답 시간은 $\frac{(0+5+10)}{3} = 5$이다.

쉽게 알 수 있듯이, 타임 슬라이스의 길이는 RR에게 매우 중요하다. 타임 슬라이스가 짧을수록, 응답 시간 기준으로 RR의 성능은 더 좋아진다. 그러나 타임 슬라이스를 너무 짧게 지정하면 문제가 생긴다. 짧게 지정하면 문맥 교환 비용이 전체 성능에 큰 영향을 미치게 된다. 시스템 설계자는 시스템이 최적의 상태로 동작할 수 있도록 타임 슬라이스의

팁 : 비용의 상쇄

일반적인 **상쇄**(**amortization**) 기술은 어떤 연산에 고정 비용이 존재하는 시스템에서 흔히 사용된다. 그 비용이 적은 횟수로 발생하게 하여, 즉 해당 연산을 적게 실행함으로써 시스템의 전체 비용이 감소된다. 예를 들어, 타임 슬라이스는 10 msec로 설정되고 문맥 교환 비용이 1 msec라면 대강 10%의 시간이 문맥 교환에 사용되고 따라서 낭비된다. 이 비용을 상쇄하고 싶은 경우, 타임 슬라이스를 예로 들어 100 msec로 늘릴 수 있다. 이러면 문맥 교환에 1% 미만의 시간이 소모되고 타임 슬라이싱의 비용이 상쇄된다.

길이를 결정해야 한다. 문맥 교환 비용을 **상쇄**할 수 있을 만큼 길어야 하지만 그렇다고 응답 시간이 너무 길어지면 안 된다.

문맥 교환 비용에는 레지스터를 저장/복원하는 작업만 있는 것이 아니다. CPU 캐시, TLB, 분기 예측 등을 비롯하여 다른 하드웨어에도 프로그램과 관련된 다양한 작업 정보들이 저장되어 있다. 작업이 전환되면 이 정보들은 모두 갱신되어야 한다. 갱신 작업이 매우 큰 성능 비용을 유발한다 [MB91].

적당한 길이의 응답 시간이 유일한 평가 기준인 경우 타임 슬라이스를 가진 RR은 매우 훌륭한 스케줄러이다. 반환 시간 기준으로는 어떤가? 앞의 예를 다시 한 번 살펴보자. A, B, C 각 작업은 5초간 실행되고, 동시에 도착하며, RR은 1초의 타임 슬라이스를 가진다. A는 13, B는 14, C는 15에 종료하고 평균 14의 반환 시간을 보인다는 것을 알 수 있다. 매우 안 좋다!

반환 시간이 유일한 측정 기준인 경우 RR이 확실히 최악의 정책이라는 것은 놀라운 일이 아니다. 직관적으로, 충분히 이해 가능하다. RR은 각 작업을 잠깐 실행하고 다음 작업으로 넘어가고 하면서 가능한 한 각 작업을 늘리는 것이 목표다. 반환 시간은 작업 완료 시간만을 고려하기 때문에, RR은 거의 최악이며, 많은 경우 단순한 FIFO보다도 성능이 좋지 않게 된다.

일반적으로 RR과 같은 **공정**한 정책, 즉 작은 시간 단위로 모든 프로세스에게 CPU를 분배하는 정책은 반환 시간과 같은 평가 기준에서는 성능이 나쁘다. 이건 당연하다. 불공정하게 한다면 하나의 작업을 끝까지 실행하고 종료할 수 있지만 나머지 작업들에 대한 응답 시간은 포기해야 한다. 반대로 공정성을 더 중히 여긴다면 응답 시간은 줄어들지만 반환 시간은 나빠지게 된다. 이러한 유형의 절충은 시스템에서는 흔히 있는 일이다. 당신은 케익을 먹으면서 동시에 케익을 보관할 수는 없다[2].

우리는 2종류의 스케줄러를 개발하였다. 첫째 유형은 (SJF, STCF) 반환 시간을 최적화하지만 응답 시간은 나쁘다. 두 번째 유형은 (RR) 응답 시간을 최적화하지만 반환 시간이 나쁘다. 아직 완화해야 하는 가정 두 개가 남아 있다. 가정 4 (작업은 입출력을 하지 않는다)와 가정 5 (각 작업의 실행 시간은 알려져 있다)이다. 이제 이 가정들을 정복해 보자.

2) 사람을 혼란시키는 속담. 원래는 "당신은 케익을 먹으면서 보존할 수 없다"여야 하기 때문이다. (분명하지 않은가, 아닌가?). 놀랍게도 이 속담에 관한 Wikipedia 페이지가 있다. 더욱 놀라운 것은 재미있는 읽을거리라는 것이다. `http://en.wikipedia.org/wiki/You_can't_have_your_cake_and_eat_it`

팁: 중첩은 높은 이용률을 가능하게 한다

가능하면 시스템의 활용도를 극대화하기 위해 연산을 **중첩**되게 실행한다. 중첩은 디스크 입출력을 할 때나 원격 컴퓨터에 메시지를 보낼 때 등 다양한 영역에서 유용하다. 어느 경우라도 이 작업들을 시작한 후에 다른 작업으로 전환하는 것은 좋은 생각이다. 시스템의 전반적인 사용률과 효율성이 향상된다.

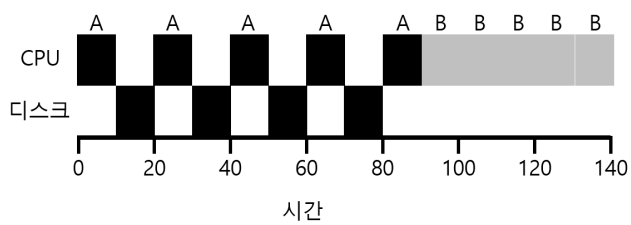
10.8 입출력 연산의 고려

우선 가정 4를 완화하자. 모든 프로그램은 입출력 작업을 수행한다. 아무런 입력도 받아들이지 않는 프로그램을 상상해 보자. 매번 같은 출력을 만들어 낼 것이다. 출력하지 않는 프로그램을 상상해 보자. 그건 속담에도 나올 법한 아무도 보는 사람이 없이 숲에서 잎이 떨어지는 것과 같다. 실행되었다는 사실이 아무 의미가 없다.

입출력 작업을 요청한 경우 스케줄러는 다음에 어떤 작업을 실행할지를 결정해야 한다. 현재 실행 중인 작업은 입출력이 완료될 때까지 CPU를 사용하지 않을 것이기 때문이다. 입출력 요청을 발생시킨 작업은 입출력 완료를 기다리며 대기 상태가 된다. 입출력이 하드 디스크 드라이브에 보내진 경우, 프로세스는 드라이브의 현재 입출력 워크로드에 따라 몇 초 또는 좀 더 긴 시간 동안 블록될 것이다. 스케줄러는 그 시간 동안 실행될 다른 작업을 스케줄 해야 한다.

마찬가지로 스케줄러는 입출력 완료 시에도 의사 결정을 해야 한다. 입출력이 완료되면 인터럽트가 발생하고 운영체제가 실행되어 입출력을 요청한 프로세스를 대기 상태에서 준비 상태로 다시 이동시킨다. 물론, 인터럽트가 발생했을 때 요청 프로세스를 즉시 실행시키기로 결정할 수도 있다. 운영체제는 각 작업을 어떻게 처리해야 하는가?

이 쟁점을 더 잘 이해하기 위해서 두 개의 작업 A와 B가 있다고 하자. 각 작업은 50 msec의 CPU 시간을 필요로 한다. 그러나 둘은 한 가지 큰 차이를 가진다. A는 10 msec 동안 실행된 후, 입출력 요청을 한다(여기서 입출력 시간은 10 msec라고 가정한다). 반면에 B는 입출력을 수행하지 않는다. 스케줄러는 A를 먼저 실행시키고 B를 다음에 실행시킨다(그림 10.8).

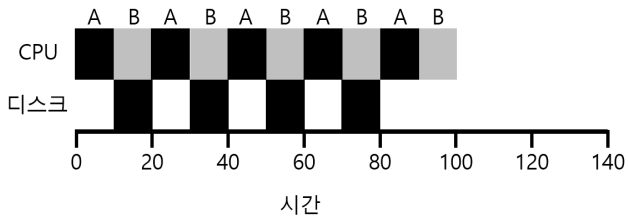


〈그림 10.8〉 **자원의 비효율적인 활용**

STCF 스케줄러를 구축하려고 한다고 가정하자. A가 5개의 10 msec 작업으로 분할되는 반면에 B는 하나의 50 msec의 CPU를 요청한다는 사실을 어떻게 활용할까?

분명히 입출력을 고려하지 않고 작업을 하나씩 실행시키는 것은 의미가 없다.

일반적인 접근 방식은 A의 각 10 msec 하위 작업을 독립적인 작업으로 다루는 것이다. 시스템이 시작할 때 10 msec 작업들과 50 msec를 스케줄하는 것이다. STCF의 경우 선택은 자명하다. 가장 짧은 작업을 선택하라. 이 경우에는 A가 된다. 그러면 A의 첫 번째 소 작업이 완료되면 B만 남게 되어 실행을 시작한다. A의 다음 작업이 제출되고 B를 선점하여 10 msec 동안 실행한다. 이렇게 하면 프로세스의 입출력이 끝나기를 기다리는 동안 CPU는 다른 프로세스에 의해 사용되어 연산의 중첩이 가능해진다. 시스템은 좀 더 잘 활용된다 (그림 10.9).



〈그림 10.9〉 **중첩을 통해 효율적 자원 활용이 가능하다**

입출력을 고려한 스케줄러 방법에 대해서 알아보았다. 각 CPU 버스트를 하나의 작업으로 간주함으로써 스케줄러는 대화형 프로세스가 더 자주, 즉 유리하게 실행되는 것을 보장한다. 이러한 대화형 작업이 입출력을 실행하는 동안 다른 CPU-집중 작업들이 실행되고 CPU의 이용률은 더 높아진다.

10.9 만병통치약은 없다(No More Oracle)

입출력을 고려한 기본적인 접근 방식에 대해 논의하였으므로 마지막 가정에 대해 생각해 보자. 스케줄러가 각 작업의 실행 시간을 알고 있다는 가정이다. 전에 얘기했듯이, 이 가정은 아마 우리가 할 수 있는 최악의 가정이다. 사실 범용 운영체제에서(우리가 현재 고려 중인 것과 같은) 작업의 길이에 대해서 알 수 있는 길은 없다. 따라서 아무런 사전 지식 없이 SJF/STCF 처럼 행동하는 알고리즘을 구축할 수 있을까? 게다가, 응답 시간도 좋게 하기 위하여 RR 스케줄러의 경우에 보았던 아이디어를 어떻게 하면 포함시킬 수 있을까?

10.10 요약

우리는 스케줄링의 기본적인 개념과 두 가지 부류의 접근법을 살펴보았다. 첫 번째 부류는 남아 있는 작업 중 실행 시간이 제일 짧은 작업을 수행하고, 반환 시간을 최소화한다. 두 번째 부류는 모든 작업을 번갈아 실행시키고 응답 시간을 최소화한다. 안타깝게도 반환 시간과 응답 시간 중 하나를 최적화하면 나머지 하나는 좋지 않은 특성을 나타낸다. 이는 시스템에서 흔히 보이는 절충을 요구하는 상황이다. 전체적인 그림에 입출력을

어떻게 통합해야 하는지도 보았다. 그렇지만 미래를 예측할 수 없는 운영체제의 근본적인 문제는 해결할 수 없었다. 가까운 과거를 이용하여 미래를 예측하는 스케줄러를 구현하여 이 문제를 어떻게 해결하는지 곧 보게 될 것이다. 이 스케줄러는 **멀티 레벨 피드백 큐**(**multi-level feedback queue**)라고 불리며 다음 장의 주제이다.

참고 문헌

[Bla+79] **"The Convoy Phenomenon"**
M. Blasgen, J. Gray, M. Mitoma, and T. Price
ACM Operating Systems Review, 13:2, April 1979
아마도 운영체제뿐 아니라 데이터베이스에서 발생한 호위함 효과에 대한 첫 번째 참고 문헌.

[Cob54] **"Priority Assignment in Waiting Line Problems"**
A. Cobham
Journal of Operations Research, 2:70, pages 70-76, 1954
기계 수리 일정을 정하는 데 SJF 방식을 사용한 선구적인 논문.

[CK68] **"Computer Scheduling Methods and their Countermeasures"**
Edward G. Coffman and Leonard Kleinrock
AFIPS '68 (Spring), April 1968
다양한 기본 스케줄링 알고리즘에 대한 우수한 초기 소개 및 분석.

[Jai91] **"The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling"**
R. Jain
Interscience, New York, April 1991
컴퓨터 시스템 측정에 관한 표준 교과서. 당신의 라이브러리를 빛낼 훌륭한 참고 문헌.

[JV56] **"Machine Repair as a Priority Waiting-Line Problem"**
Thomas E. Phipps Jr. and W.R. Van Voorhis
Operations Research, 4:1, pages 76-86, February 1956
기계 수리 스케줄링에 사용된 Cobham의 SJF를 일반화한 후속 연구. 또한 이러한 환경에서 STCF 방식의 유용성을 생각한 논문. 구체적으로 "수리 작업에는 특정 유형이 존재한다 ... 많은 해체와 마루를 너트와 볼트로 뒤덮음을 동반하기 때문에 한 번 시작하면 중단되서는 안 된다. 다른 경우, 하나 이상의 짧은 작업들이 존재할 때 긴 작업을 계속하는 것은 권하고 싶지 않을 것이다 (p.81)"

[Kle64] **"Analysis of a Time-Shared Processor"**
Leonard Kleinrock
Naval Research Logistics Quarterly, 11:1, pages 59-73, March 1964
라운드 로빈 스케줄링 알고리즘에 대한 첫 번째 참고 문헌. 시분할 시스템을 스케줄링 할 때 라운드 로빈 스케줄링 알고리즘 사용에 대한 분석을 제시한 논문 중 하나.

[Lav78] **"The Manchester Mark I and Atlas: A Historical Perspective"**
S.H. Lavington
Communications of the ACM, 21:1, January 1978
초창기 컴퓨터 개발과 Atlas의 개척적인 노력에 관한 역사.

[MB91] **"The effect of context switches on cache performance"**
Jeffrey C. Mogul and Anita Borg
ASPLOS, 1991
문맥 교환이 캐시의 성능에 미치는 영향에 관한 좋은 연구. 프로세서는 초당 수십억 명령어를 실행하지만 문맥 교환은 msec 시간 단위로 발생하기 때문에 현재 시스템에서는 크게 문제가 되지 않는다.

[Orw45] **"Animal Farm"**
George Orwell

Secker and Warburg (London), 1945
권력과 그의 변질에 관한 슬프지만 우울한 우화책. 어떤 이는 제2차 세계대전 전 스탈린 시대의 소련에 대한 비판이라고 말한기도 한다. 우리는 그저 돼지에 대한 비판이라고 말한다.

숙제

`scheduler.py` 프로그램은 응답 시간, 반환 시간, 총 대기 시간의 관점에서 여러 스케줄러가 어떻게 동작하는지를 볼 수 있게 한다. 자세한 내용은 README를 참조하시오.

문제

1. 길이가 200인 세 개의 작업을 SJF와 FIFO 스케줄링 방식으로 실행할 경우 응답 시간과 반환 시간을 계산하시오.
2. 같은 조건이지만 작업의 길이가 각각 100, 200 및 300일 경우에 대해 계산하시오.
3. 2번과 같은 조건으로 타임 슬라이스가 1인 RR 스케줄러에 대해서도 계산하시오.
4. SJF와 FIFO가 같은 반환 시간을 보이는 워크로드의 유형은 무엇인가?
5. SJF가 RR과 같은 응답 시간을 보이기 위한 워크로드와 타임 퀀텀의 길이는 무엇인가?
6. 작업의 길이가 증가하면 SJF의 응답 시간은 어떻게 되는가? 변화의 추이를 보이기 위해서 시뮬레이터를 사용할 수 있는가?
7. 타임 퀀텀의 길이가 증가하면 RR의 응답 시간은 어떻게 되는가? N 개의 작업이 주어졌을 때, 최악의 응답 시간을 계산하는 식을 만들 수 있는가?