11

스케줄링 : 멀티 레벨 피드백 큐

이 장에서는 **멀티 레벨 피드백 큐**(**Multi-level Feedback Queue**, **MLFQ**)로 알려진 가장 유명한 스케줄링 기법에 대해 논의한다. 멀티 레벨 피드백 큐(MLFQ) 스케줄러는 Compatible Time-Sharing System(CTSS)에 사용되며 Corbato 등에 의해 1962년에 최초로 소개되었다 [CDD62] . 이 연구와 Multics에 대한 후속 연구로 Corbato는 최고 영예인 **Turing Award**를 안았다. 이 스케줄러는 수년 동안 다듬어져 일부 현대 시스템에까지 발전되었다.

MLFQ가 해결하려고 하는 기본적인 문제는 두 가지이다. 첫째, 짧은 작업을 먼저 실행시켜 *반환 시간을* 최적화하고자 한다. 이에 대해서는 전 장에서 논의한 바 있다. SJF나 STCF 같은 알고리즘은 작업의 실행 시간 정보를 필요로 하지만, 불행히도 운영체제는 이 실행 시간을 미리 알 수 없다. 둘째, MLFQ는 대화형 사용자(즉, 화면 앞에 앉아 바라보면서 프로세스의 종료를 기다리는 사용자)에게 응답이 빠른 시스템이라는 느낌을 주고 싶었기 때문에 응답 시간을 최적화한다. 불행히도 RR과 같은 알고리즘은 응답 시간을 단축시키지만 반환 시간은 거의 최악이다. 우리의 문제는 다음과 같다. 우리가 프로세스에 대한 정보가 없다면 이러한 스케줄러를 어떻게 만들 수 있을까? 실행 중인 작업의 특성을 알아내고 이를 이용하여 더 나은 스케줄링 결정을 하기 위한 방법은 무엇인가?

핵심 질문 : 정보 없이 스케줄 하는 방법은 무엇인가
작업의 실행 시간에 대한 선행 정보 없이 대화형 작업의 응답 시간을 최소화하고 동시에 반환 시간을 최소화하는 스케줄러를 어떻게 설계할 수 있는가?

11.1 MLFQ: 기본 규칙

멀티 레벨 피드백 큐의 기본 알고리즘을 설명한다. 현재 구현되어 있는 여러 MLFQ들은 자세하게 살펴보면 차이가 있지만 [Epe95], 기본적으로 비슷한 방법을 사용하고 있다.

MLFQ는 여러 개의 **큐**로 구성되며, 각각 다른 **우선순위**(**priority level**)가 배정된다. 실행 준비가 된 프로세스는 이 중 하나의 큐에 존재한다. MLFQ는 실행할 프로세스를

팁: 역사로부터 배우다

멀티 레벨 피드백 큐는 미래를 예측하기 위해 과거의 경험을 활용하는 훌륭한 예이다. 이러한 접근 방식은 운영체제 (하드웨어 분기 예측기와 캐시 알고리즘을 포함한 컴퓨터 과학의 다른 많은 분야에서도) 분야에서 흔히 사용된다. 이러한 방식은 작업이 단계별로 진행되어 예측 가능할 때 잘 동작한다. 물론, 신중하게 사용해야 한다. 잘못 동작하기 쉬우며, 이러한 정보가 없을 때 보다 더 나쁜 결정을 하게 만들기 때문이다.

결정하기 위하여 우선순위를 사용한다. 높은 우선순위를 가진 작업이, 즉 높은 우선순위 큐에 존재하는 작업이 선택된다.

큐에 둘 이상의 작업이 존재할 수 있다. 이들은 모두 같은 우선순위를 가진다. 이 작업들 사이에서는 라운드 로빈 (Round-Robin, RR) 스케줄링 알고리즘이 사용된다.

MLFQ 스케줄링의 핵심은 우선순위를 정하는 방식이다. MLFQ는 각 작업에 고정된 우선순위를 부여하는 것이 아니라 각 작업의 *특성에 따라* 동적으로 우선순위를 부여한다. 예를 들어, 어떤 작업이 키보드 입력을 기다리며 반복적으로 CPU를 양보하면 MLFQ는 해당 작업의 우선순위를 높게 유지한다. 이러한 패턴은 대화형 프로세스가 나타내는 패턴이다. 대신에 한 작업이 긴 시간 동안 CPU를 집중적으로 사용하면 MLFQ는 해당 작업의 우선순위를 낮춘다. 이렇게 MLFQ는 작업이 진행되는 동안 해당 작업의 정보를 얻고, 이 *정보를* 이용하여 *미래* 행동을 예측한다.

MLFQ의 두 가지 기본 규칙은 다음과 같다.

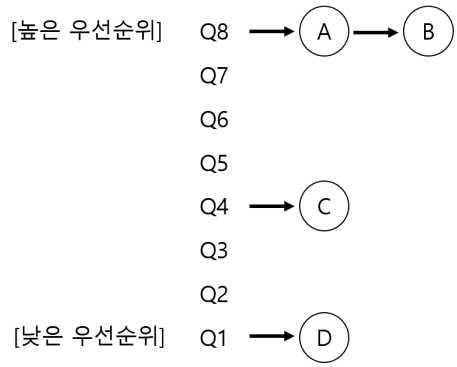
- **규칙 1:** Priority(A) > Priority(B) 이면, A가 실행된다 (B는 실행되지 않는다).
- **규칙 2:** Priority(A) = Priority(B) 이면, A와 B는 RR 방식으로 실행된다.

임의의 시간에 큐의 모양은 그림 11.1과 같다. 그림에서 두 작업(A와 B)이 가장 높은 우선순위에 존재하고, C는 중간, D는 가장 낮은 우선순위 큐에 존재한다. 우리가 알고 있는 MLFQ의 동작을 고려하면 스케줄러는 가장 높은 우선순위의 큐의 A와 B를 번갈아 실행할 것이다. 불쌍한 작업 C와 D는 실행되지도 않는다. 어떻게 이런 일이!!!

정적인 스냅 사진만으로는 MLFQ가 어떻게 동작하는지 알 수 없다. 작업 우선순위가 시간에 따라 어떻게 변화하는지 알아보자.

11.2 시도 1: 우선순위의 변경

MLFQ가 작업의 우선순위를 어떻게 바꿀 것인지 결정해야 한다. 작업의 우선순위를 변경하는 것은 작업이 존재할 큐를 결정하는 것과 마찬가지다. 이를 위해서 우리는 워크로드의 특성을 반영해야 한다. 짧은 실행 시간을 갖는 CPU를 자주 양보하는 대화형 작업과 많은 CPU 시간을 요구하지만 응답 시간은 중요하지 않은 긴 실행 시간의 CPU 위주 작업이 혼재되어 있다. 우선순위 조정 알고리즘을 위한 첫 번째 시도는 다음과 같다.

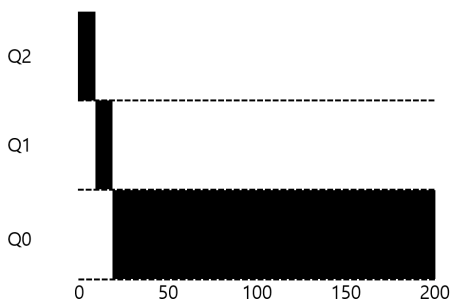


〈그림 11.1〉 **MLFQ 예**

- **규칙 3:** 작업이 시스템에 진입하면, 가장 높은 우선순위, 즉 맨 위의 큐에 놓여진다.
- **규칙 4a:** 주어진 타임 슬라이스를 모두 사용하면 우선순위는 *낮아진다*. 즉, 한 단계 아래 큐로 이동한다.
- **규칙 4b:** 타임 슬라이스를 소진하기 전에 CPU를 양도하면 *같은* 우선순위를 유지한다.

예 1: 한 개의 긴 실행 시간을 가진 작업

몇 가지 예를 살펴보자. 우선 긴 실행 시간을 가진 작업이 도착했을 때 어떤 일이 일어나는지 알아보자. 그림 11.2는 세 개의 큐로 이루어진 스케줄러에서 시간이 지남에 따라 작업의 우선순위가 어떻게 변하는지 보인다.

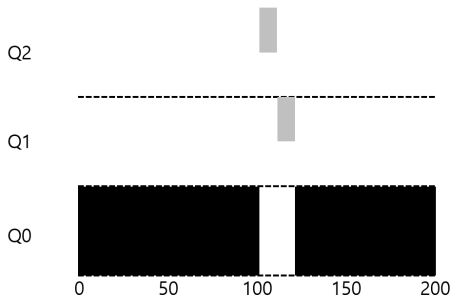


〈그림 11.2〉 **긴 실행 시간을 가진 작업의 우선순위 변화**

예에서 보는 것처럼 작업은 최고 우선순위로 진입한다(Q2). 10 msec 타임 슬라이스가 하나 지나면 스케줄러는 작업의 우선순위를 한 단계 낮추어 해당 작업을 Q1으로 이동시킨다. 다시 하나의 타임 슬라이스 동안 Q1에서 실행한 후 작업은 마침내 가장 낮은 우선순위를 가지게 되고 Q0로 이동된다. 이후에는 Q0에 계속 머무르게 된다. 간단하다. 그렇지 않은가?

예 2: 짧은 작업과 함께

좀 더 복잡한 예를 살펴보자. MLFQ가 어떻게 SJF에 근접할 수 있는지 이해하기 바란다. 이 예에서는 2개의 작업이 존재한다. A는 오래 실행되는 CPU 위주 작업이고 B는 짧은 대화형 작업이다. A는 얼마 동안 이미 실행해 온 상태이고 B는 이제 도착했다고 가정하자. 어떤 일이 벌어질까? MLFQ는 SJF와 근사하게 동작해서 B를 선호할 것인가?



〈그림 11.3〉 **대화형 작업이 들어온 경우**

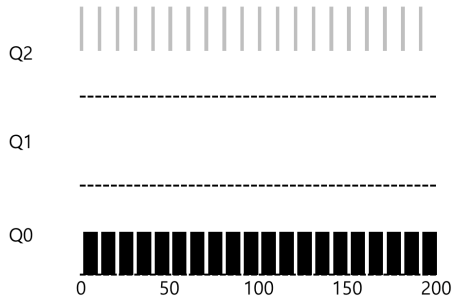
그림 11.3은 이 시나리오의 결과를 보인다. 다른 오래 실행되는 CPU 위주 작업들처럼 A(검은색)는 가장 낮은 우선순위 큐에서 실행되고 있다. B(회색)는 $T = 100$에 시스템에 도착하고 가장 높은 우선순위 큐에 놓여진다. 실행 시간이 짧기 때문에 (단지 20 ms), 두 번의 타임 슬라이스를 소모하면 B는 바닥의 큐에 도착하기 전에 종료한다. 그런 후에 A는 낮은 우선순위에서 실행을 재개한다.

이 예에서 이 알고리즘의 주요 목표를 알 수 있다. 스케줄러는 작업이 짧은 작업인지 긴 작업인지 알 수 없기 때문에 일단 짧은 작업이라고 *가정하여* 높은 우선순위를 부여한다. 진짜 짧은 작업이면 빨리 실행되고 바로 종료할 것이다. 짧은 작업이 아니라면 천천히 아래 큐로 이동하게 되고 스스로 긴 배치형 작업이라는 것을 증명하게 된다. 이러한 방식으로 MLFQ는 SJF를 근사할 수 있다.

예 3: 입출력 작업에 대해서는 어떻게?

다음으로 입출력 작업을 수행하는 예를 살펴보자. 규칙 4b가 말하는 것처럼, 프로세스가 타임 슬라이스를 소진하기 전에 프로세서를 양도하면 같은 우선순위를 유지하게 한다. 이 규칙의 의도는 간단하다. 대화형 작업이 키보드나 마우스로부터 사용자 입력을 대기하며 자주 입출력을 수행하면 타임 슬라이스가 종료되기 전에 CPU를 양도하게 될 것이다. 그런 경우 동일한 우선순위를 유지하게 하는 것이다.

그림 11.4는 이 규칙이 동작하는 예를 보이고 있다. B(회색)는 대화형 작업으로서 입출력을 수행하기 전에 1 msec 동안만 실행된다. A(검정색)는 긴 배치형 작업으로 B와 CPU를 사용하기 위하여 경쟁한다. B는 CPU를 계속해서 양도하기 때문에 MLFQ 방식은 B를 가장 높은 우선순위로 유지한다. B가 대화형 작업이라면 MLFQ는 대화형 작업을 빨리 실행시킨다는 목표에 더 근접하게 된다.



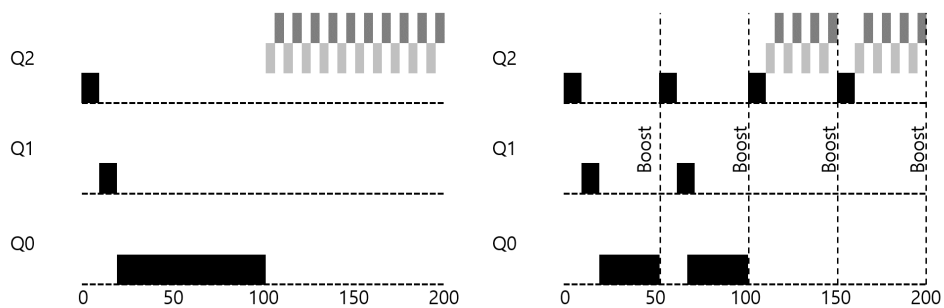
〈그림 11.4〉 **입출력-집중 작업과 CPU-집중 작업이 혼합된 워크로드**

현재 MLFQ의 문제점

현재 MLFQ는 단순하다. CPU를 긴 작업들과 짧은 작업들 사이에 잘 공유하고, 입출력-중점 대화형 작업을 빨리 실행시키기 때문에 잘 동작하는 것처럼 보인다. 이 방식은 심각한 결점을 가진다. 하나라도 생각해 낼 수 있겠는가?

(여기서 잠깐 중지하고 할 수 있는 한 모든 생각을 해 보도록)

첫째, **기아 상태(starvation)**가 발생할 수 있다. 시스템에 너무 많은 대화형 작업이 존재하면 그들이 모든 CPU 시간을 소모하게 될 것이고 따라서 긴 실행 시간 작업은 CPU 시간을 할당받지 못할 것이다(굶어 죽는다). 이러한 시나리오에서도 긴 실행 시간 작업들도 진행이 되도록 만들고 싶다.



〈그림 11.5〉 **우선순위 상향이 없는 경우(좌측)와 있는 경우(우측)**

둘째, 똑똑한 사용자라면 스케줄러를 자신에게 유리하게 동작하도록 프로그램을 다시 작성할 수 있다. 스케줄러를 자신에게 유리하게 동작시킨다는 것은 일반적으로 스케줄러를 속여서 지정된 몫보다 더 많은 시간을 할당하도록 하게 만드는 것을 가리킨다. 지금까지 논의한 알고리즘은 다음과 같은 공격에 취약하다. 타임 슬라이스가 끝나기 전에 아무 파일을 대상으로 입출력 요청을 내려 CPU를 양도한다. 그렇게 하면 같은 큐에 머무를 수 있고 따라서 더 높은 퍼센트의 CPU 시간을 얻게 된다. 제대로 된다면, 예를 들어 타임 슬라이스의 99%를 실행하고 CPU를 양도하게 되면 CPU를 거의 독점할 수 있다.

마지막으로 프로그램은 시간 흐름에 따라 *특성이 변할* 수 있다. CPU 위주 작업이 대화형 작업으로 바뀔 수 있다. 현재 구현 방식으로는 그런 작업은 운이 없게도 다른 대화형 작업들과 같은 대우를 받을 수 없다.

11.3 시도 2: 우선순위의 상향 조정

규칙을 보완하여 기아 문제를 방지할 수 있는지 살펴보자. CPU 위주 작업이 조금이라도 진행하는 것을 보장하기 위해서 무엇을 할 수 있는가?

간단한 아이디어는 주기적으로 모든 작업의 우선순위를 **상향 조정**(**boost**)하는 것이다. 목적을 달성하기 위해 여러 방법이 존재하지만 간단한 방법을 사용하기로 하자. 모두 최상위 큐로 보내는 것이다. 새로운 규칙은 다음과 같다.

- **규칙 5:** 일정 기간 S가 지나면, 시스템의 모든 작업을 최상위 큐로 이동시킨다.

새 규칙은 두 가지 문제를 한 번에 해결한다. 첫째, 프로세스는 굶지 않는다는 것을 보장받는다. 최상위 큐에 존재하는 동안 작업은 다른 높은 우선순위 작업들과 라운드 로빈 방식으로 CPU를 공유하게 되고 서비스를 받게 된다. 둘째, CPU 위주의 작업이 대화형 작업으로 특성이 변할 경우 우선순위 상향을 통해 스케줄러가 변경된 특성에 적합한 스케줄링 방법을 적용한다.

예를 들어 보자. 긴 실행 시간을 가진 작업이 두 개의 대화형 작업과 CPU를 두고 경쟁할 때의 행동을 보여주는 시나리오다. 이 두 개의 그래프가 그림 11.5에 나와 있다. 왼쪽 그래프는 우선순위 상향이 없는 경우를 보이고 있다. 긴 실행 시간 작업은 두 개의 짧은 작업이 도착한 이후에는 굶게 된다. 오른쪽 그래프는 50 msec 마다 우선순위 상향이 일어난다. 물론, 너무 짧은 시간이지만 예를 위해 사용되었다. 긴 실행 시간 작업도 꾸준히 진행된다는 것을 보장할 수 있으며, 50 msec 마다 상향되고 따라서 주기적으로 실행된다.

물론 S 값의 결정이 필요하다. S를 얼마로 해야 하는가? 존경받는 시스템 연구자인 John Ousterhout [Ous11]는 이러한 종류의 값을 **부두 상수**(**voo-doo constants**)라고 불렀다. 이러한 값을 정확하게 결정하기 위해서는 흑마술이 필요한 것처럼 보이기 때문이다. 불행하게도 S는 그러한 종류의 변수이다. 너무 크면 긴 실행 시간을 가진 작업은 굶을 수 있으며 너무 작으면 대화형 작업이 적절한 양의 CPU 시간을 사용할 수 없게 된다.

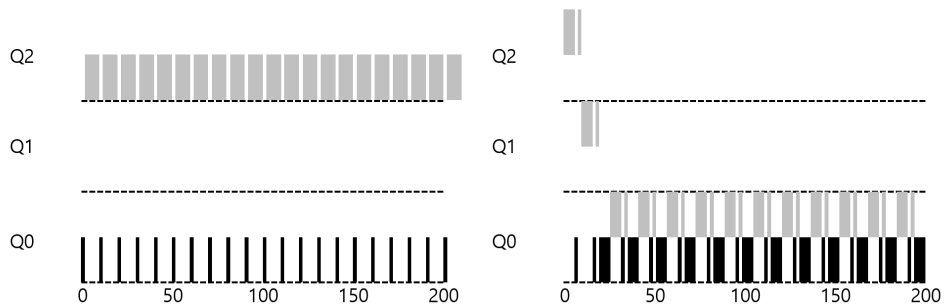
11.4 시도 3: 더 나은 시간 측정

해결해야 할 문제가 하나 더 있다. 스케줄러를 자신에게 유리하게 동작시키는 것을 어떻게 막을 수 있는가? 이러한 일을 가능하게 만든 주범은 규칙 4a와 4b이다. 두 규칙은 작업이 타임 슬라이스가 끝나기 전에 CPU를 양보하여 우선 순위를 유지가 가능하게 한다. 우리는 무엇을 해야 하는가?

여기서의 해결책은 MLFQ의 각 단계에서 CPU 총 사용 시간을 **측정**하는 것이다. 스케줄러는 현재 단계에서 프로세스가 소진한 CPU 사용 시간을 저장한다. 프로세스가 타임 슬라이스에 해당하는 시간을 모두 소진하면 다음 우선순위 큐로 강등된다. 타임 슬라이스를 한 번에 소진하든 짧게 여러 번 소진하든 상관 없다. 규칙 4a와 4b를 합쳐 하나의 규칙으로 재정의한다.

- **규칙 4:** 주어진 단계에서 시간 할당량을 소진하면 (CPU를 몇 번 양도하였는지 상관없이), 우선순위는 낮아진다 (즉, 아래 단계의 큐로 이동한다).

예를 살펴보도록 하자. 그림 11.6은 워크로드가 스케줄러를 자신에게 유리하게 동작시키려고 할 때 예전 규칙 4a와 4b일 때의 행동과(왼쪽) 새로운 조작 방지 규칙 4일 때의 행동 양식을 보이고 있다. 이런 자신에게 유리하도록 조작하는 데 대한 방지책이 없다면 프로세스는 타임 슬라이스가 끝나기 직전에 입출력 명령어을 내릴 수 있어서 CPU 시간을 독점할 수 있다. 방지책이 마련되면 프로세스의 입출력 행동과 무관하게 아래 단계 큐로 천천히 이동하게 되어 CPU를 자기 몫 이상으로 사용할 수 없게 된다.



〈그림 11.6〉 **조작에 대한 내성이 없는 경우(좌측)와 있는 경우(우측)**

11.5 MLFQ 조정과 다른 쟁점들

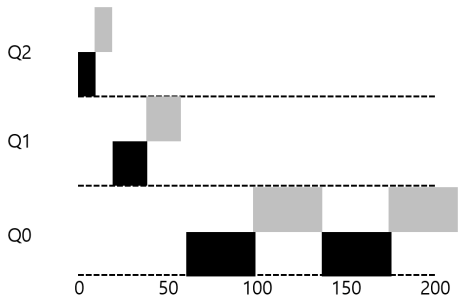
MLFQ 스케줄링에는 여러 다른 쟁점들이 남아 있다. 필요한 변수들을 스케줄러가 어떻게 설정해야 하는지도 중요한 문제다. 예를 들어, 몇 개의 큐가 존재해야 하는가? 큐당 타임 슬라이스의 크기는 얼마로 해야 하는가? 기아를 피하고 변화된 행동을 반영하기 위하여 얼마나 자주 우선순위가 상향 조정되어야 하는가? 이러한 질문들에 대해 쉽게 대답할 수는 없다. 워크로드에 대해 충분히 경험하고 계속 조정해 나가면서 균형점을 찾아야 한다.

예를 들어, 대부분의 MLFQ 기법들은 큐 별로 타임 슬라이스를 변경할 수 있다. 우선순위가 높은 큐는 보통 짧은 타임 슬라이스가 주어진다. 이 큐는 대화형 작업으로 구성되고, 결국 이 작업들을 빠르게 교체하는 것은 의미가 있다(예, 10 msec 이하). 낮은 우선순위는 반대로 CPU-중심의 오래 실행되는 작업들을 포함한다. 긴 타임 슬라이스

팁: 부두 상수 피하기 (Ousterhout's Law)

가능하다면 부두 상수를 피하는 것이 좋은 생각이다. 불행하게도, 앞의 예처럼, 피할 수 없는 경우가 더 자주 일어난다. 더 좋은 값을 찾고 그 방법도 쉽지는 않다. 흔한 결과: 디폴트 값으로 가득 찬 설정 파일. 설정 파일의 값들은 무언가 정확히 동작하지 않을 때 풍부한 경험의 관리자가 조정 가능하다. 상상할 수 있듯이, 이 값은 대부분 있는 그대로 사용되며 디폴트 값이 현장에서 잘 작동하기를 바랄 뿐이다. 이 팁은 오래된 운영체제 교수 John Ousterhout에 의해 알려졌기 때문에 **Ousterhout's Law**라고 부른다.

(예, 수백 msec)가 적합하다. 그림 11.7은 가장 높은 우선순위 큐는 10 ms, 중간 큐는 20 ms, 가장 낮은 큐는 40 ms의 스케줄러에서 두 개의 작업이 실행되는 모양을 보인다.



〈그림 11.7〉 **낮은 우선순위, 더 긴 할당 시간**

Solaris의 MLFQ 구현, 시분할 스케줄링 클래스 또는 TS는 설정이 특히 쉽다. Solaris는 프로세스의 우선순위가 일생 동안 어떻게 변하는지, 타임 슬라이스의 길이는 얼마인지, 작업의 우선순위는 얼마나 자주 상향되는지를 결정하는 테이블을 제공한다 [Arp00]. 관리자는 이 테이블을 수정하여 스케줄러의 동작 방식을 바꿀 수 있다. 테이블의 기본 값은 큐의 개수는 60, 각 큐의 타임 슬라이스 크기는 가장 높은 우선순위 큐가 20 msec에서 가장 낮은 우선순위 큐가 수백 msec까지 천천히 증가하고, 우선순위 상향 조정은 1초 정도마다 일어난다.

다른 MLFQ 스케줄러는 테이블이나 이 장에서 설명한 정확한 규칙 같은 것은 사용하지 않는다. 수학 공식을 사용하여 우선순위를 조정한다. 예를 들어, FreeBSD의 스케줄러(버전 4.3)는 작업의 현재 우선순위를 계산하기 위하여 프로세스가 사용한 CPU 시간을 기초로 한 공식을 사용한다 [Lef+89]. CPU 사용은 시간이 지남에 따라 감쇠되어 이 장에서 설명한 방식과는 다른 방식으로 우선순위 상향을 제공한다. 이와 같은 **감쇠-사용**(**decay-usage**) 알고리즘과 특성에 대한 훌륭한 개요는 Epema의 논문을 참조하기 바란다 [Epe95].

마지막으로, 스케줄러들은 다른 여러 기능을 제공한다. 예를 들어, 일부 스케줄러의 경우 가장 높은 우선순위를 운영체제 작업을 위해 예약해 둔다. 일반적인 사용자 작업은 시스템 내에서 가장 높은 우선순위를 얻을 수 없다. 일부 시스템은 사용자가 우선순위를

팁 : 가능하면 조언을 이용하시오

운영체제 시스템이 모든 프로세스에게 어떤 것이 최선인지 알 수는 없다. 사용자 또는 관리자가 힌트를 전달할 수 있는 인터페이스를 제공하는 것이 도움이 된다. 우리는 그러한 **힌트**(**hint**)를 **조언**(**advice**)이라고 부른다. 운영체제가 힌트를 반드시 고려할 필요는 없지만 더 나은 결정을 내리는 데 힌트가 도움이 될 수는 있기 때문이다. 이러한 힌트는 스케줄러(예, `nice`), 메모리 관리자(예, `madvise`), 파일 시스템(예, 제공 정보에 기초한 선반입과 캐싱 [Pat+95]) 등을 포함한 운영체제의 여러 부분에서 유용하다.

정하는 데 도움을 줄 수 있도록 허용한다. 예를 들어, 명령어 라인 도구인 nice를 사용하여 작업의 우선순위를 높이거나 낮출 수 있다. 작업의 실행 순서를 바꿀 수 있다. 자세한 사항은 `man` 페이지를 보시오.

11.6 MLFQ: 요약

우리는 멀티 레벨 피드백 큐로 알려진 스케줄링 방법을 기술하였다. 이제 왜 그런 이름으로 불리는지 이해했을 것이다. 알고리즘은 *멀티 레벨* 큐를 가지고 있으며, 지정된 작업의 우선순위를 정하기 위하여 *피드백을* 사용한다. 과거에 보여준 행동이 우선순위 지정의 지침이 된다. 작업이 시간이 지남에 따라 어떻게 행동하고 그에 맞게 어떻게 처리하는지에 대해 주의를 기울여라.

이 장 전체에 산재해 있는 정교한 MLFQ 규칙의 집합을 보기 쉽게 여기에 다시 적어 보았다.

- 규칙 1 : 우선순위(A)> 우선순위(B) 일 경우, A가 실행, B는 실행되지 않는다.
- 규칙 2 : 우선순위(A) = 우선순위(B), A와 B는 RR 방식으로 실행된다.
- 규칙 3 : 작업이 시스템에 들어가면 최상위 큐에 배치된다.
- 규칙 4 : 작업이 지정된 단계에서 배정받은 시간을 소진하면(CPU를 포기한 횟수와 상관없이), 작업의 우선순위는 감소한다(즉, 한 단계 아래 큐로 이동한다).
- 규칙 5 : 일정 주기 S가 지난 후, 시스템의 모든 작업을 최상위 큐로 이동시킨다.

MLFQ는 다음과 같은 측면에서 매우 흥미로운 스케줄러이다. 작업의 특성에 대한 *정보 없이*, 작업의 실행을 관찰한 후 그에 따라 우선순위를 지정한다. MLFQ는 반환 시간과 응답 시간을 모두 최적화한다. 짧게 실행되는 대화형 작업에 대해서는 우수한 전반적인 성능을 제공한다(SJF/ STCF와 유사). 오래 실행되는 CPU-집중 워크로드에 대해서는 공정하게 실행하고 조금이라도 진행되도록 한다. 이런 이유로 BSD Unix와 여기서 파생된 다양한 운영체제 [Lef+89; Bac86], Solaris [McD06], Windows NT 및 이후 Windows 운영체제 [CS97]를 포함한 많은 시스템이 기본 스케줄러로 MLFQ를 사용한다.

참고 문헌

[Arp00] **"Multilevel Feedback Queue Scheduling in Solaris"**
Andrea Arpaci-Dusseau
URL: `http://www.cs.wisc.edu/%CB%9Cremzi/solaris-notes.pdf`
Solaris 스케줄러에 대한 저자 중 하나의 위대한 짧은 노트 모음. 오케이, 아마 저자의 설명에는 편향된 의견이 들어 있지만 노트들은 정말 훌륭하다.

[Bac86] **"The Design of the Unix Operating System"**
M.J. Bach
Prentice-Hall, 1986
실제 Unix 운영체제가 어떻게 구축되었는지에 관한 고전적인 오래된 책들 중 하나. 커널 해커가 반드시 읽어야 할 필독서.

[CDD62] **"An Experimental Time-Sharing System"**
F.J. Corbato, M.M. Daggett, and R.C. Daley
IFIPS 1962
읽기는 약간 어렵지만 멀티 레벨 피드백 스케줄링의 초창기 아이디어의 원천. 이 중 많은 발상이 Multics에 포함되었으며, 논쟁이 있겠지만 Multics는 지금까지 가장 영향력 있는 운영체제이다.

[CS97] **"Inside Windows NT"**
Helen Custer and David A. Solomon
Microsoft Press, 1997
Unix 이외의 무언가를 배우고 싶다면 이 NT 책을 봐야 한다. 물론 왜 그래야 할까? OK, 농담. 당신이 Microsoft를 위해 언젠가 일할 수도 있으니까.

[Epe95] **"An Analysis of Decay-Usage Scheduling in Multiprocessors"**
D.H.J. Epema
SIGMETRICS '95
감쇠-사용 스케줄러의 배경이 되는 기본적인 방법에 관한 훌륭한 개요를 포함하여 1990년대 중반 당시 스케줄링 기술의 현황에 관한 멋진 논문.

[Lef+89] **"The Design and Implementation of the 4.3BSD Unix Operating System"**
S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman
Addison-Wesley, 1989
BSD를 만든 주요 사람들 중 4인에 의해 쓰여진 또 다른 운영체제의 고전. 이후 버전들은 최신 내용을 담고 있기는 하지만 이 버전의 아름다움에는 미치지 못한다.

[McD06] **"Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture"**
Richard McDougall
Prentice-Hall, 2006
Solaris와 그의 동작에 관한 좋은 책.

[Ous11] **"John Ousterhout's Home Page"**
John Ousterhout
URL: `http://www.stanford.edu/%CB%9Couster/`
유명한 Ousterhout 교수의 홈페이지. 이 책의 공동저자 모두는 대학원에 있을 때 Ousterhout 교수의 운영체제 강의를 수강했던 즐거운 기억을 가지고 있다. 사실 거기서 서로를 알게 되었고 마침내 결혼, 자녀, 심지어 이 책까지 오게 되었다. 따라서, 당신이 경험하는 이 모든 혼란은 Ousterhout에게 항의할 수도 있겠다.

[Pat+95] **"Informed Prefetching and Caching"**
R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka
SOSP '95
파일 시스템의 매우 근사한 아이디어에 관한 흥미로운 논문. 어떤 파일을 접근할 것인지와 어떻게 접근할 것인지에 관해 응용 프로그램이 운영체제에게 조언을 줄 수 있는 방법을 설명하고 있다.

숙제

프로그램 `mlfq.py`는 지금의 이 장에서 논의한 MLFQ 스케줄러가 어떻게 동작하는지를 살펴볼 수 있게 한다. 자세한 사항은 README를 참조하시오.

문제

1. 두 개의 작업과 두 개의 큐를 무작위로 구성하여 실행시켜 보시오. 각 문제에 대한 MLFQ 실행 추적을 계산하시오. 문제를 쉽게 하기 위해 각 작업의 길이를 제한하고 입출력은 하지 않는다고 가정하시오.
2. 이 장의 예제를 재현하려면 스케줄러를 어떻게 실행해야 하는가?
3. 라운드 로빈 스케줄러처럼 동작시키려면 스케줄러의 매개변수를 어떻게 설정해야 하는가?
4. 두 개의 작업과 스케줄러 매개변수를 가진 워크로드를 고안하시오. 두 작업 중 하나의 작업은 옛날 규칙 4a 및 4b를 이용하여(`-S` 플래그를 켠다) 스케줄러를 자신에게 유리하게 동작하도록 만들어 특정 구간에서 99%의 CPU를 차지하도록 고안해야 한다.
5. 가장 높은 우선순위 큐의 타임 퀀텀의 길이가 10 ms인 시스템이 있다고 하자. 하나의 장기 실행(및 잠재적인 기아 위험) 작업이 적어도 5%의 CPU를 사용할 수 있도록 보장하려면 얼마나 자주 가장 높은 우선순위로 이동시켜야 하는가(`-B` 플래그 켠다)?
6. 스케줄링에서 제기되는 질문 중 하나는, 입출력이 방금 종료된 작업은 큐의 어느 쪽에 추가해야 하는가이다. 플래그 `-I`가 시뮬레이터의 이 행동 양식을 변경한다. 몇 개의 워크로드를 가지고 실험하여 이 플래그의 영향을 확인할 수 있는지 보라.