05. CPU 스케줄링

version 2018.37

기본개념

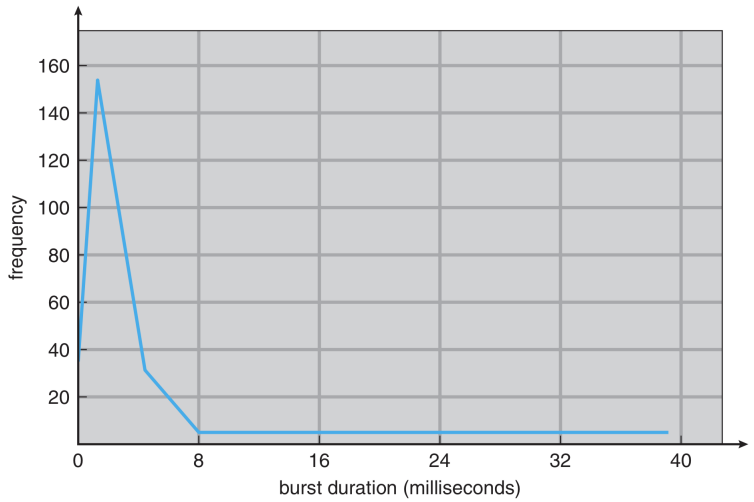
다중 프로그래밍의 목적은 CPU 이용률을 최대화하기 위해 CPU가 항상 프로세스를 실행하게 하는데 있습니다. 프로세스는 보통 어떤 입출력을 요청할 때 CPU를 사용하지 않고 대기합니다. 이러한 모든 대기 시간은 낭비이기 때문에 운영체제는 CPU를 대기 상태인 프로세스로부터 회수해 다른 프로세스에 할당합니다.

이러한 종류의 스케쥴링은 운영체제의 기본적인 기능입니다. CPU는 중요한 컴퓨터 자원 중 하나이므로 스케줄링은 운영체제 설계의 핵심이 됩니다.

CPU-입출력 버스트 사이클(CPU-I/O Burst Cycle)

프로세스는 **CPU 실행(CPU 버스트)**과 **입출력 대기(입출력 버스트)**을 순환하며 실행 됩니다.

CPU 버스트들의 지속 시간을 측정하였을 때 아래의 그림처럼 빈도수 곡선을 갖는 경향이 있습니다.



짧은 CPU 버스트가 많이 있으며 , 긴 CPU 버스트는 적습니다. 입출력 중심의 프로그램은 전형적으로 짧은 CPU 버스트를 많이 가질 것입니다. 이러한 분포는 적절한 CPU 스케줄링 알고리즘을 선택하는 데 매우 중요할 수 있습니다.

CPU 스케줄러(CPU Scheduler)

CPU가 유휴 상태가 될 때마다, 운영체제는 **준비 완료 큐**에 있는 프로세스들 중에서 하나를 선택해 실행해야 합니다. 선택 절차는 **단기 스케줄러(또는, CPU 스케줄러)**에 의해 수행됩니다.

준비 완료 큐는 반드시 선입 선출(FIFO) 방식의 큐가 아니어도 됩니다. 준비 완료 큐는 선입 선출 큐, 우선순위 큐, 트리 또는 단순히 순서가 없는 연결 리스트로 구현할 수 있습니다. 준비 완료 큐에 있는 레코드들은 일반적으로 프로세들의 프로세스 제어 블록(PCB)입니다.

선점 스케줄링(Preemptive Scheduling)

CPU 스케줄링 결정은 다음의 네 가지 상황 하에서 발생할 수 있습니다.

- 실행 상태 -> 대기 상태
    - 예를 들어, 입출력 요청이나 자식 프로세스가 종료되기를 기다리기 위해 wait를 호출할 때
- 실행 상태 -> 준비 완료 상태
    - 예를 들어, 인터럽트가 발생할 때
- 대기 상태 -> 준비 완료 상태
    - 예를 들어, 입출력의 종료 시
- 프로세스 종료

디스패처(Dispatcher)

디스패처(dispatcher)는 CPU 스케줄링 기능에 포함된 요소 중 하나 입니다. 디스패처는 단기 스케줄러가 선택한 프로세스에게 CPU를 할당 해주는 모듈이며 다음과 같은 작업을 포함합니다.

- 문맥 교환
- 사용자 모드로 전환
- 프로그램을 다시 시작하기 위해 사용자 프로그램의 적절한 위치로 이동(jump)

> **사용자 모드:** 사용자가 접근할 수 있는 영역을 제한적으로 두고, 프로그램의 자원에 함부로 침범하지 못하는 모드

디스패처는 모든 프로세스의 문맥 교환 시 호출 되므로, 가능한 빨리 수행되어야 합니다. 디스패처가 하나의 프로세스를 정지하고 다른 프로세스의 수행을 시작하는 데까지 소요되는 시간을 **디스패치 지연(dispatch latency)**이라고 합니다.

스케줄링 기준

CPU 스케줄링 알고리즘들은 서로 다른 특성을 가지고 있습니다. 특정 상황에 필요한 알고리즘을 선택하려면, 우리는 알고리즘들의 특성들을 반드시 이해하고 있어야 합니다.

아래의 항목들은 CPU 스케줄링 알고리즘을 비교하기 위한 기준들입니다.

- **CPU 이용률(utilization):** 우리는 가능한 CPU를 최대한 바쁘게 유지하기를 원합니다. 실제 시스템에서는 40%에서 90%까지의 범위를 가져야 합니다.
- **처리량(throughput):** 단위 시간당 완료된 프로세스의 개수로 작업량을 측정 할 수 있습니다. 이것을 **처리량**이라고 합니다.

- **총 처리 시간(turnaround time):** 프로세스의 입장에서 보면, 프로세스를 실행하는 데 소요된 시간이 중요한 기준될 수 있습니다. 프로세스의 제출 시간과 완료 시간의 간격을 총처리 시간이라고 합니다. 총처리 시간은 메모리에 들어가기 위해 기다리는 시간, 준비 완료 큐에서 대기한 시간, CPU에서 실행하는 시간, 그리고 입출력 시간을 합한 시간입니다.
- **대기 시간(waiting time):** 스케줄링 알고리즘은 단지 프로세스가 준비 완료 큐에서 대기하는 시간의 양에만 영향을 줍니다. **대기 시간은 준비 완료 큐에서 대기하면서 보낸 시간의 총 합**입니다.
- **응답 시간(response time):** 응답시간은 요구를 제출한 후 첫 번째 응답이 나올때까지의 시간입니다. 여기서 주의할 점은, 응답시간은 응답이 시작되는 데까지 걸리는 시간이지 그 응답을 출력하는 데 걸리는 시간이 아니라는 것입니다.

CPU 이용률과 처리량을 최대화하고 총처리 시간, 대기 시간, 응답 시간을 최소화하는 것이 바람직합니다. 대부분의 경우, 평균 측정 시간을 최적화 하려고 하지만 최소값 또는 최대값을 최적하는 것이 바람직한 경우도 있습니다.

스케줄링 알고리즘

CPU 스케줄링은 준비 완료 큐에 있는 프로세스 중에서 어떤 프로세스에게 CPU를 할당할 것인지에 대한 문제를 다룰것입니다.

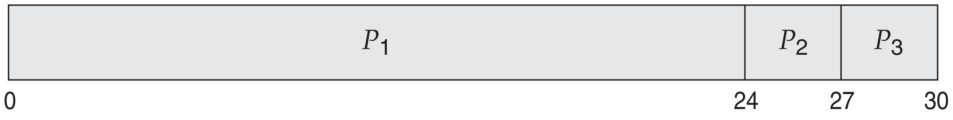
선입 선처리 스케줄링(First-Come, First-Served Scheduling)

선입 선처리(FCFS) 스케줄링 알고리즘에서는 CPU를 먼저 요청하는 프로세스가 CPU를 먼저 할당 받습니다. FCFS는 선입선출(FIFO) 큐로 쉽게 구현될 수 있습니다.

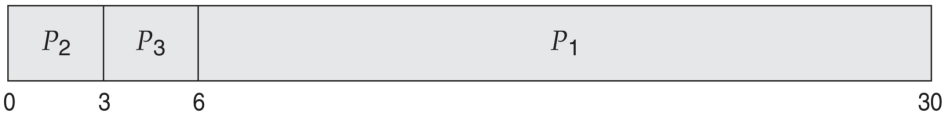
부정적인 측면으로 FCFS에서 평균 대시 시간은 굉장히 길어 질 수 있습니다. 시간 0에 도착한 다음의 프로세스 집합을 생각해봅시다.

Process	Burst Time
P_1	24
P_2	3
P_3	3

프로세스들이 P1, P2, P3 순으로 도착하고, 선입 선처리 순으로 서비스를 받는다면, 다음의 **Gantt 차트**에 보인 결과를 얻습니다.



이에 대해 평균 대기 시간은 (0 + 24 + 27)/3 = 17입니다. 그러나 프로세스들이 P2, P3, P1 순으로 도착하면, 결과는 다음 Gantt 차트와 같습니다.



평균 대기 시간은 이제 (6 + 0 + 3)/3 = 3입니다. 우리는 평균 대기 시간이 크게 감소한 것을 확인할 수 있습니다. 이처럼 FCFS 스케줄링은 평균 대기 시간이 매우 길어질 수 있다는 단점이 있습니다. 특히 시분할 시스템에서 문제가 되는데, 시분할 시스템에서는 각 사용자가 규칙적인 간격으로 CPU의 몫을 얻는 것이 매우 중요하기 때문입니다.

P1, P2, P3 순으로 도착한 경우, CPU 버스트 시간이 짧은 프로세스(P2, P3)들이 긴 프로세스(P1)가 CPU를 양도할 때 까지 기다리는 것을 **호위효과**라고 하며 이 효과는 짧은 프로세스들이 먼저 처리 되도록 허용 할 때보다 CPU와 장치 이용률을 저하되는 결과를 낳습니다.

최단 작업 우선 스케줄링(Shortest-Job-First Scheduling)

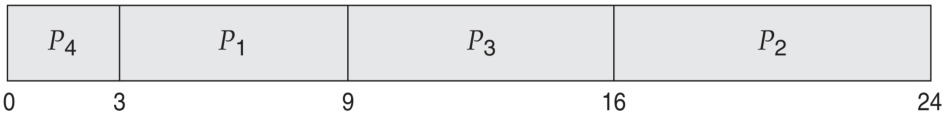
최단 작업 우선(SJF) 스케줄링에서는 각 프로세스의 다음 CPU 버스트 길이가 짧은 순서로 CPU를 할당 시킵니다. 두 프로세스가 동일한 CPU 버스트 길이를 가지면 FCFS를 적용합니다.

SJF는 프로세스의 전체 길이가 아니라 다음 CPU 버스트 길이에 의해 스케줄링 하므로 사실 **최단 다음 CPU 버스트(shortest-next-CPU-burst)** 라는 용어가 더 적합합니다. 하지만 일반적으로는 SJF라고 불려집니다.

다음의 프로세스 집합을 고려해 봅시다.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF 스케줄링을 적용하면, 결과는 다음과 같습니다.



평균 대기 시간은 (3 + 16 + 9 + 0)/4 = 7입니다. 앞서 FCFS에서 확인 했듯이 짧은 프로세스가 먼저 실행 될 때 평균 대기 시간이 줄어드는 것을 확인 할 수 있었습니다. SJF는 이러한 특징으로 인해 다른 스케줄링 알고리즘과 비교해서 최소의 평균 대기 시간을 가집니다.

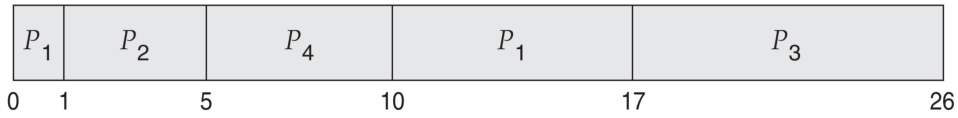
하지만 단기 스케줄링에서는 다음 CPU 버스트 길이를 알 수 있는 방법이 없기 때문에 SJF 알고리즘을 사용할 수 없습니다. 이러한 문제를 해결하기 위해 이전의 버스트 길이를 통하여 다음 버스트 길이를 유추한 근사값을 이용하기도 합니다.

SJF 알고리즘은 **선점형**이거나 또는 **비선점형** 일 수 있습니다. 새로운 프로세스가 준비 완료 큐에 도착하면 현재 실행되고 있는 프로세스의 남은 시간보다더 더 짧은 CPU 버스트를 가질 수도 있습니다. 선점형 SJF 알고리즘에서는 현재 실행하고 있는 프로세스를 선점할 것입니다. 반면에 비선점형 SJF 알고리즘에서는 선점하지 않고 대기합니다. 선점형 SJF 알고리즘은 **최소 잔여 시간 우선(shortest remaining time first)** 스케줄링이라고 불리기도 합니다.

다음의 프로세스들을 고려해봅시다.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

결과는 다음과 같습니다.



이 시나리오에서 평균 대기 시간은 (9 + 0 + 15 + 2)/4 = 6.5입니다. 비선점형 SJF 스케줄링의 평균 대기 시간은 7.75 입니다.

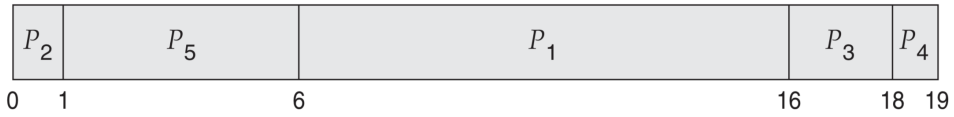
우선순위 스케줄링(Priority Scheduling)

우선순위 스케줄링에서는 가장 높은 우선순위를 가지는 프로세스에게 CPU를 먼저 할당합니다. 우선 순위가 같은 프로세스들은 FCFS 순서로 스케줄링 됩니다.

다음의 프로세스 집합을 고려해봅시다. 낮은 값이 높은 우선순위를 나타냅니다.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

결과는 다음과 같습니다.



이 시나리오에서 평균 대기 시간은 8.2입니다.

우선순위 스케줄링은 선점형이거나 비선점형이 될 수 있습니다. 선점형 우선순위 알고리즘에서 새로운 프로세스가 준비 완료 큐에 도착했을 때 실행 중인 프로세스 보다 우선순위가 높다면 실행 중인 프로세스를 선점합니다. 비선점형에서는 새로운 프로세스는 선점 하지 않고 준비 완료 큐의 머리 부분에 위치하게 됩니다.

우선순위 스케줄링에서는 **무한 봉쇄(indefinite blocking)** 또는 **기아 상태(starvation)**이 발생 할 수 있습니다. 낮은 우선순위를 가지는 프로세스가 CPU를 무한히 대기하는 경우가 발생할 수 있기 때문입니다. 이러한 문제를 해결하는 방법 중 하나는 **노화(aging)**입니다. 노화는 시스템에서 오랫동안 대기하는 프로세스들의 우선순위를 점진적으로 증가 시키므로써 문제를 해결합니다.

라운드 로빈 스케줄링(Round-Robin Scheduling)

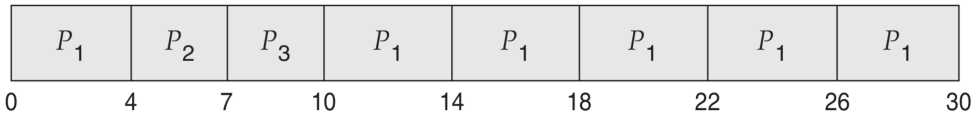
라운드 로빈(RR) 스케줄링 알고리즘은 시분할 시스템을 위해 설계되었습니다. 이 스케줄링 알고리즘은 FCFS 스케줄링과 유사하지만 **시간 할당량(time quantum)** 또는 **시간 조각(time slice)** 라고 하는 작은 단위의 시간 동안 CPU를 할당합니다.

프로세스의 CPU 버스트가 시간 할당량 보다 작은 경우, 프로세스는 CPU 버스트를 완료하고 자발적으로 CPU를 방출합니다. 반대의 경우에서 프로세스는 시간 할달량 만큼 CPU를 실행 한 후 준비 완료 큐의 꼬리 부분으로 이동 됩니다.

다음의 프로세스 집합을 고려해봅시다.

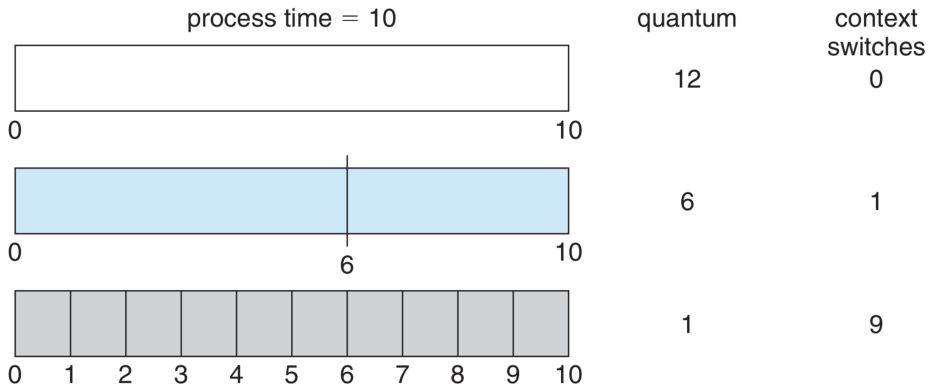
Process	Burst Time
P_1	24
P_2	3
P_3	3

결과는 다음과 같습니다.



이 시나리오에서 평균 대기 시간은 5.66입니다.

RR 알고리즘의 성능은 시간 할당량의 크기에 매운 많은 영향을 받습니다. 극단적인 경우, 시간 할당량이 매우 크면 FCFS 알고리즘과 같습니다. 반대로 시간 할당량이 매우 적다면 매우 많은 문맥교환을 야기하게 됩니다. 아래의 그림은 이러한 RR 알고리즘의 특징을 잘 설명해줍니다.



이러한 특징으로 시간 할당량이 문맥 교환 시간에 비해 커야 하지만 너무 커서는 안됩니다. 저자는 CPU 버스트의 80%가 시간 할당량 보다 짧아야 한다고 얘기합니다.

다단계 큐 스케줄링(Multilevel Queue Scheduling)

다단계 큐 스케줄링 알고리즘은 준비 완료 큐를 다수의 큐로 분류합니다. 프로세스들은 메모리 크기, 프로세스의 우선순위 혹은 프로세스 유형과 같은 프로세스 특성에 때라 한 개의 큐에 영구적으로 할당됩니다. 각 큐는 자신의 스케줄링 알고리즘을 갖습니다.

- 포그라운드 큐
    - 높은 우선순위
    - RR 알고리즘
    - 대화형 프로세스
- 백그라운드 큐
    - 낮은 우선순위
    - FCFS 알고리즘

    - 연산 작업을 주로 수행하는 일괄처리 프로세스

포그라운드 큐가 백그라운드 큐보다 절대적으로 높은 우선순위를 가집니다. 따라서 포그라운드 큐가 비어있지 않다면 백그라운드 큐에 있는 프로세스는 실행될 수 없습니다. 또 백그라운드에 있는 프로세스가 실행되는 중이라 하더라도 포그라운드 큐에 프로세스가 새로 들어간다면 백그라운드는 선점됩니다.

큐들 사이에 시간을 나누어 사용할 수 있습니다. 각 큐는 CPU 시간의 일정부분을 받아서 자기 큐에 있는 프로세스들을 스케줄링합니다. 예를 들어 아래와 같이 스케줄링할 수 있습니다.

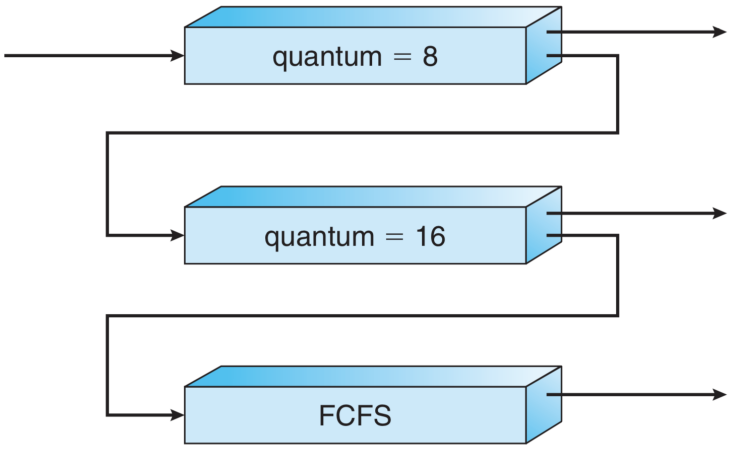
- 포그라운드 큐
    - 80% 할당받음
    - Round Robin
- 백그라운드 큐
    - 20% 할당받음
    - FCFS

다단계 피드백 큐 스케줄링(Multilevel Feedback Queue Scheduling)

기존의 다단계 큐 스케줄링에서는 프로세스가 큐 사이를 이동하는 것을 허용하지 않습니다. 스케줄링 오버헤드는 적다는 장점이 있지만 융통성이 부족하다는 단점이 있었습니다.

이 문제를 극복하기 위해 나온 것이 **다단계 피드백 큐** 스케줄링입니다. 이 스케줄링 기법에서는 프로세스가 큐 사이를 이동하는 것이 허용됩니다. 예를 들어, 특정 프로세스가 자신이 속한 큐의 시간 할당량 안에 끝내지 못하면 우선순위가 낮은 큐로 강등됩니다.

우선 순위 알고리즘을 기반으로 하기 때문에 낮은 우선순위의 큐에서 너무 오래 대기하는 프로세스들은 높은 곳으로 이동(노화)하여 기아 상태를 예방합니다. 다음의 큐 3개를 가정합시다.



CPU 버스트가 8ms 이하인 프로세스는 최고의 우선순위로 실행될 것입니다. 8ms~24ms 인 프로세스는 그 다음의 우선순위를 받게 될 것입니다. 그리고 CPU 버스트가 너무 긴 프로세스는 큐 2로 가게 될 것입니다.

다단계 큐와 비교하였을 때 단점은, 프로세스를 다른 큐로 올려주거나 내리는 시기를 결정하는 등 큐 간 이동가능성 때문에 고려해야할 것이 많아 설계하기 복잡하다는 것입니다.

스레드 스케줄링

사용자 스레드와 **커널 스레드**를 지원하는 운영체제에서 스케줄링 되는 대상은 프로세스가 아니라 커널 스레드입니다. 사용자 스레드는 스레드 라이브러리에 의해 관리되고 커널은 사용자 스레드의 존재를 알 지 못합니다. 또한 CPU 상에서 실행되기 위해서는 사용자 스레드는 커널 스레드에 사상되어야합니다.

경쟁 범위(Contention Scope)

사용자 스레드와 커널 스레드는 다른 경쟁 범위를 가집니다. 다대일과 다대다 모델을 구현하는 시스템에서, 스레드 라이브러리는 사용자 스레드를 사용 가능한 LWP 상에서 스케줄링 합니다. 사용자 스레드는 동일한 프로세스에 속한 스레드들 끼리 CPU를 경쟁하기 때문에 **프로세스-경쟁-범위(process-contention scope, PCS)**를 가집니다

스레드 라이브러리가 사용자 스레드를 사용 가능한 LWP에 스케줄링 하였다 하더라도, 이는 사용자 스레드가 실제로 CPU 상에서 실행 중인 것은 아닙니다. 실제로 CPU 상에서 실행되기 위해서는 운영체제가 커널 스레드를 물리적인 CPU로 스케줄 해야합니다. 사용자 스레드와 달리, 커널 스레드는 CPU를 할당 받기 위해서 시스템의 모든 스레드와 경쟁합니다. 때문에 이를 **시스템-경쟁-범위(system-contention scope, SCS)**라고 합니다.

일대일 모델을 사용하는 시스템은 오직 SCS만을 사용하여 스케줄합니다.

다중 처리기 스케줄링

저희는 지금까지 단일 처리기를 가진 시스템에서의 스케줄링만 다뤘습니다. 만약 여러 개의 CPU를 사용 할 수 있다면, **부하공유(load sharing)**가 가능해집니다. 그러나 그에 따라 스케줄링 문제는 더욱 복잡해집니다. 이 절에서는 다중 처리기 스케줄링에 관련된 주제들을 논의하겠습니다.

다중 처리기 스케줄링에 대한 접근 방법(Approaches to Multiple-Processor Scheduling)

주 서버(master server)라는 하나의 처리기가 모든 스케줄링 결정을 담당하고 다른 처리기들은 사용자 코드만 수행하는 구조로 동작할 수 있습니다. 이러한 구조를 **비대치 다중 처리(asymmetric multiprocessing)**라고 합니다.

이와 대조적인 **대칭 다중 처리(sysmetric multiprocessing, SMP)** 방식이 있습니다. 이 방식에서는 각 처리리가 독자적으로 스케줄링 합니다. 모든 프로세스는 공동의 준비 완료 큐에 있거나 각 처리기 마다 가지고 있는 사유의 준비 완료 큐에 있게 됩니다. 이 때 공동의 준비 완료 큐를 사용하는 경우, 서로 다른 2개의 처리기가 같은 프로세스를 선택하지 않도록 스케줄링 해야 합니다.

처리기 친화성(Processor Affinity)

프로세스가 CPU_A에서만 실행 되고 있었다고 가정합시다. 그런데 만약 CPU_B로 이주하게 된다면 CPU_A의 캐시에 있던 프로세스에 대한 내용은 의미가 없어지고 CPU_B의 캐시에 다시 프로세스의 내용으로 채워야합니다. 캐시를 무효화 하고 다시 채우는 작업은 비용이 많이 들기 때문에 대부분의 SMP 시스템은 한 처리리에서 다른 처리기의 이주를 피합니다. 이러한 현상을 **처리기 친화성(processor affinity)**라고 합니다.

처리기 친화성은 여러 형태를 가지는데, 운영체제가 동일한 처리기에서 프로세스를 실행 시키려고 노력은 하지만 보장 하지 않을 때, 우리는 이를 **연성 친화성(soft affinity)**를 가진다고 합니다. 반면에 프로세스가 실행될 CPU를 명시 하므로써 프로세스가 이주 되지 않는 것을 보장하는 방식을 **강성 친화성(hard affinity)**라고 합니다.

부하 균등화(Load Balancing)

SMP 시스템에서 처리기가 하나 이상이는 것을 최대한 활용하려면, 부하를 모든 처리기에 균등하게 배분하는 것이 매우 중요합니다. **부하 균등화(load balancing)**은 SMP 시스템의 모든 처리기 사이에 부하가 고르게 배분되도록 합니다.

> 부하 균등화는 각 처리기가 자기 자신만의 큐를 가지고 있는 시스템에만 필요한 기능입니다.

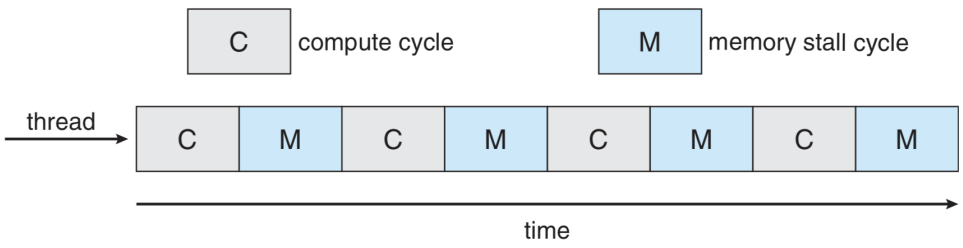
부하 균들화를 위해서는 **push 이주(migration)**과 **pull 이주** 방식을 이용합니다. push는 특정 처리기의 태스크 량이 너무 많아지면 여유가 있는 처리기로 프로세스를 이주 시키는 것을 말합니다. 반대로 pull은 여유로운 처리기가 바쁜 처리기의 프로세스를 가져오는 것을 말합니다.

하지만 부하 균등화는 처리기 친화성과 상충하게됩니다. 프로세스를 이주 시키므로써 기존 처리기의 캐시를 이용 못하는 문제를 갖기 때문입니다. 때문에 이에 대한 고민과 함께 시스템을 구성해야합니다.

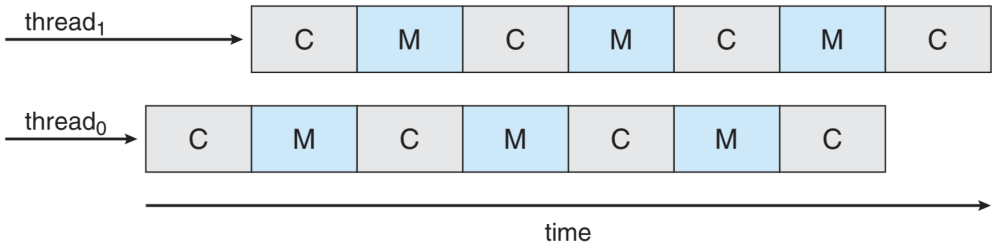
다중코어 프로세서(Multicore Processors)

최근 CPU를 만들때 하나의 물리적인 칩 안에 여러 개의 처리기 코어를 장착합니다. 우리는 이런 구조를 **다중코어 프로세서(multicore processor)**라고 합니다. 이 때 운영체제는 각 코어를 별개의 물리 처리기처럼 취급합니다. 때문에 다중코어 프로세서는 스케줄링 문제를 복잡하게 만듭니다.

연구자들에 의해 프로세서가 메모리를 접근할 때 데이터가 사용가능 해지기를 기다리면서 많은 시간을 허비한다는 것이 발견되었습니다. 이러한 문제를 **메모리 멈춤(memory stall)**이라고 합니다.



메모리 멈춤 현상을 해결하기 위해 두 개 이상의 하드웨어 스레드가 각 코어에 할당 될 수 있게 하므로써 메모리를 기다리는 동안 코어는 다른 스레드를 실행합니다. 이러한 방식을 **다중스레드 프로세서 코어**라고 합니다.



이 구조에서 운영체제는 각 하드웨어 스레드를 소프트웨어 스레드를 실행할 수 있는 논리 프로세서로 봅니다. 따라서 이중스레드, 이중코어 시스템에서는 4개의 논리 프로세서가 운영체제에게 제공됩니다.

다중스레드 다중코어 프로세스는 두 단계의 스케줄링이 필요합니다. 소프트웨어 스레드가 각 하드웨어 스레드에서 실행되어야 하는 지를 운영체제가 결정하는 것이 한 단계이고 각 코어가 어떤 하드위어 스레드를 실행시킬지 지정하는 것이 한 단계입니다.