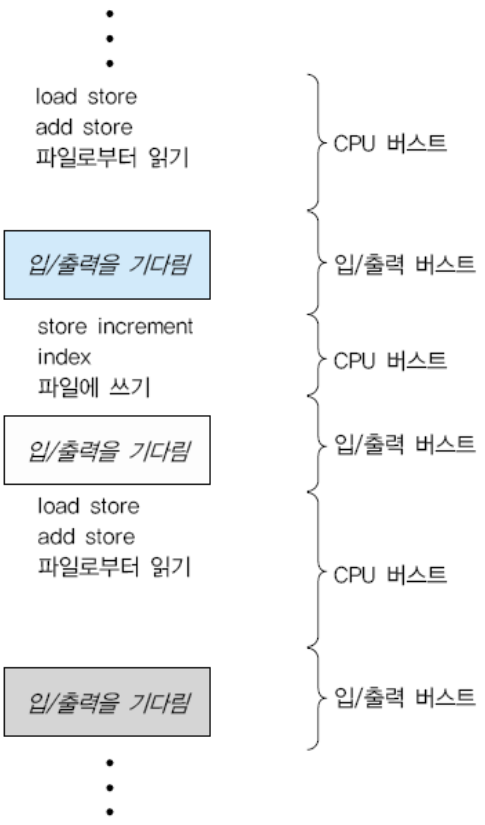
5장. CPU 스케줄링 (CPU Scheduling)

순천향대학교 컴퓨터공학과
이 상 정

강의 목표 및 내용

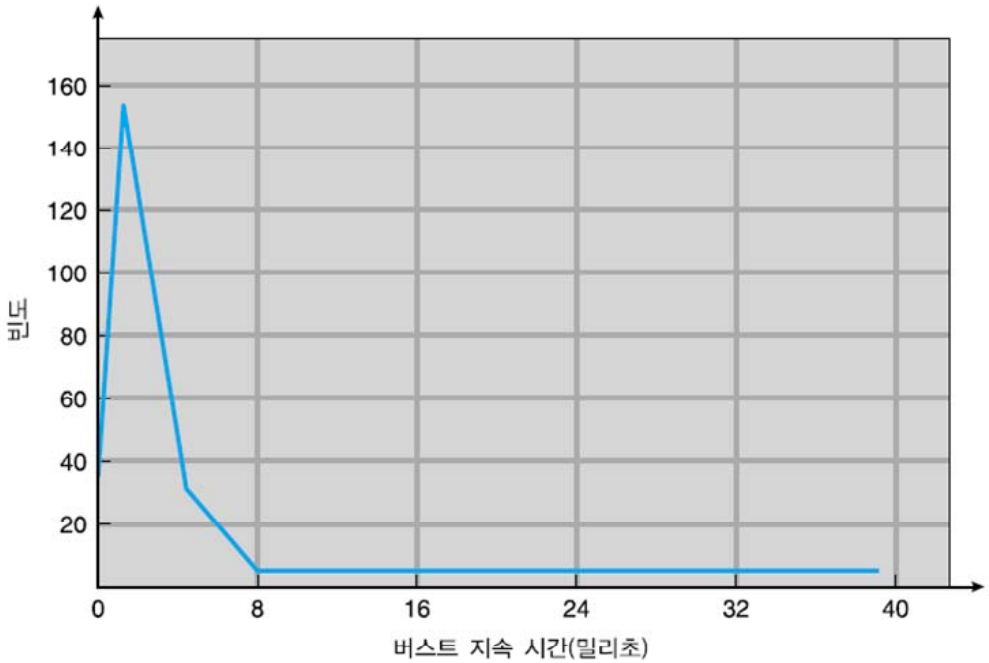
- 목표
  - 다중 프로그램 운영체제의 기반인 CPU 스케줄링을 소개
  - 다양한 CPU 스케줄링 알고리즘을 설명
  - CPU 스케줄링 알고리즘을 선택하는 평가 기준을 논의
- 내용
  - 기본 개념
  - 스케줄링 기준
  - 스케줄링 알고리즘
  - 다중 처리기 스케줄링
  - 실시간 CPU 스케줄링
  - Linux 스케줄링
  - 알고리즘의 평가

기본 개념(Basic Concept)



- 다중 프로그래밍(multiprogramming)은 CPU 이용률을 최대화하기 위해 항상 실행중인 프로세스를 시도
- CPU-입/출력 버스트 사이클(CPU-I/O burst cycle)
  - 프로세스 실행은 CPU 실행과 입/출력 대기의 사이클로 구성

CPU 버스트의 지속 시간



CPU 스케줄러 (CPU Scheduler) (1)

- 단기 스케줄러는 실행 준비가 되어 있는 메모리 내의 프로세스들 중에서 선택하여, 이들 중 하나에게 CPU를 할당

- CPU 스케줄링 결정은 다음의 네 가지 상황 하에서 발생
  1. 한 프로세스가 실행 상태에서 대기 상태로 전환될 때
     - 예를 들어, 입/출력 요청이나 자식 프로세스들 중의 하나가 종료되기를 기다리기 위해 wait를 호출할 때
  2. 프로세스가 실행 상태에서 준비 완료 상태로 전환될 때
     - 예를 들어, 인터럽트가 발생할 때
  3. 프로세스가 대기 상태에서 준비 완료 상태로 전환될 때
     - 예를 들어, 입/출력의 종료 시
  4. 프로세스가 실행 상태에서 종료할 때

CPU 스케줄러 (2)

- 상황 1과 4에서 스케줄링인 경우 비선점(nonpreemptive) 또는 협조적(cooperative)인 스케줄링
  - 스케줄링 면에서는 선택의 여지가 없음
  - 실행을 위해 새로운 프로세스(준비 완료 큐에 하나라도 존재할 경우)가 반드시 선택
- 상황 1과 4 이외의 경우 선점(preemptive) 스케줄링
  - 선택의 여지 있음
  - 선점 스케줄링은 공유 자료에 대한 접근을 조정하는 데 필요한 비용을 유발
  - 선점은 또한 운영체제 커널 설계에 영향
    - 시스템 호출로 커널 프로세스 선점으로 커널 자료구조 변경
    - 인터럽트에 의해서 영향을 받는 코드 부분은 반드시 동시 사용으로부터 보호

디스패처 (Dispatcher)

- 디스패처는 CPU의 제어를 단기 스케줄러가 선택한 프로세스에게 주는 모듈이며 다음과 같은 작업을 수행
  - 문맥(context)을 교환하는 일
  - 사용자 모드로 전환하는 일
  - 프로그램을 다시 시작하기 위해 사용자 프로그램의 적절한 위치로 이동(jump)하는 일
- 디스패치 지연 (dispatch latency)
  - 하나의 프로세스를 정지하고 다른 프로세스의 수행을 시작하는 데까지 소요되는 시간

스케줄링 기준 (Scheduling Criteria)

- CPU 스케줄링 시 프로세스 선택 기준
  - CPU 이용률 (CPU utilization)
    - 가능한 한 CPU를 최대한 바쁘게 유지
  - 처리량 (throughput)
    - 단위 시간당 완료된 프로세스의 개수
  - 총 처리 시간 (turnaround time)
    - 프로세스를 실행하는 데 소요된 시간
  - 대기 시간 (waiting time)
    - 프로세스가 준비 완료 큐에서 대기하는 시간
  - 응답 시간 (response time)
    - 대화식 시스템(interactive system)에서 하나의 요구를 제출한 후 첫 번째 응답이 나올 때까지의 시간
    - 응답이 시작되는 데까지 걸리는 시간이지, 그 응답을 출력하는 데 걸리는 시간은 아님

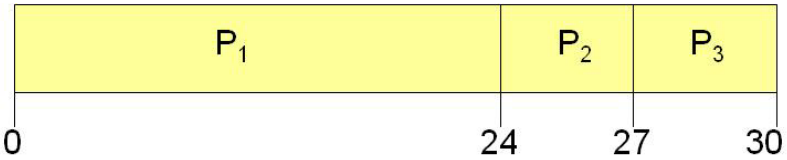
최적화 기준 (Optimization Criteria)

- CPU 이용률 최대화 (Max CPU utilization)
- 처리량 최대화 (Max throughput)
- 총 처리 시간 최소화 (Min turnaround time)
- 대기 시간 최소화 (Min waiting time)
- 응답 시간 최소화 (Min response time)

선입 선처리 스케줄링 (1) (First-Come, First-Served Scheduling)

프로세스	버스트 시간
P_1	24
P_2	3
P_3	3

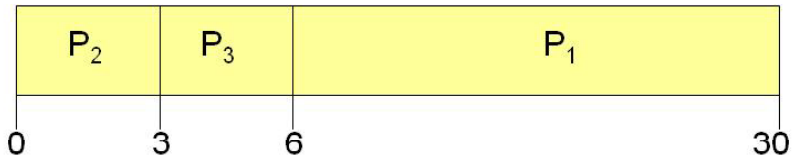
- 프로세스들이 P_1, P_2, P_3 순으로 도착하고, 선입 선처리 순으로 서비스 받는다면, 다음의 Gantt 차트에 보인 결과를 보임



- 대기시간: P_1 = 0; P_2 = 24; P_3 = 27
- 평균 대기 시간: (0 + 24 + 27)/3 = 17

선입 선처리 (FCFS) 스케줄링 (2)

- 프로세스들이 P_2, P_3, P_1 순으로 도착하면, 결과는 다음 Gantt 차트와 같음



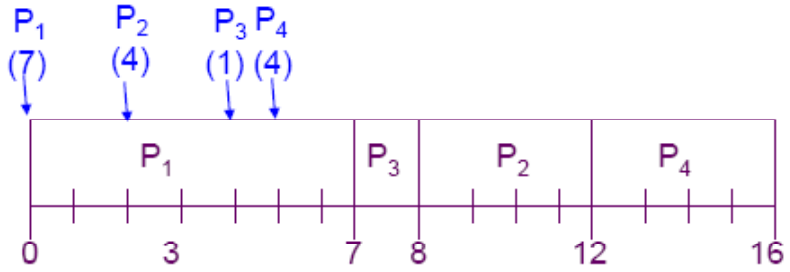
- 대기 시간: $P_1 = 6; P_2 = 0; P_3 = 3$
- 평균 대기 시간: $(6 + 0 + 3)/3 = 3$
- 앞의 경우 보다 평균 대기 시간이 크게 감소
- 호위 효과(convoy effect)
  - 짧은 프로세스들이 긴 프로세스가 CPU를 양도하기를 기다림
  - 짧은 프로세스들이 먼저 처리될 때보다 CPU와 장치 이용률이 저하

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

- 최단 작업 우선 스케줄링 알고리즘은 각 프로세스에 다음 CPU 버스트 길이를 연관
- 두 가지 방식
  - 비선점형 (nonpreemptive)
    - 현재 CPU에서 실행되고 있는 프로세스는 자신의 CPU 버스트를 끝낼 때까지는 선점되지 않음
  - 선점형 (preemptive)
    - 새로운 프로세스가 현재 실행되고 있는 프로세스의 남은 시간보다도 더 짧은 CPU 버스트를 가지면 현재 실행 중인 프로세스를 선점
    - 최소 잔여 시간 우선(Shortest-Remaining-Time-First , SRTF) 스케줄링
- SJF 스케줄링 알고리즘은 주어진 프로세스 집합에 대해 최소의 평균 대기 시간을 가진다는 점에서 최적(optimal)

비선점형 SJF 예

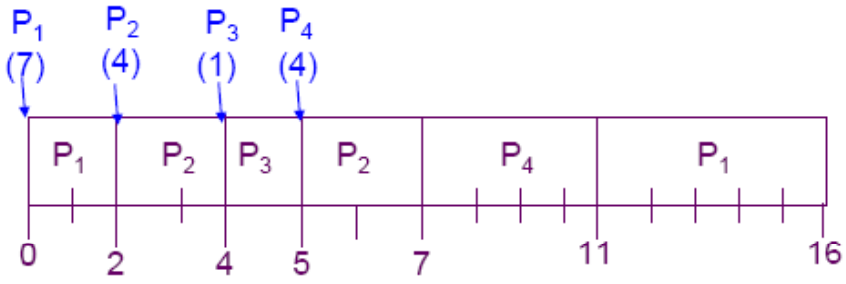
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



- 평균 대기 시간 = (0 + 6 + 3 + 7)/4 = 4

선점형 SJF 예

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



- 평균 대기 시간 = (9 + 1 + 0 +2)/4 = 3

다음 CPU 버스트의 길이 추정 (1)

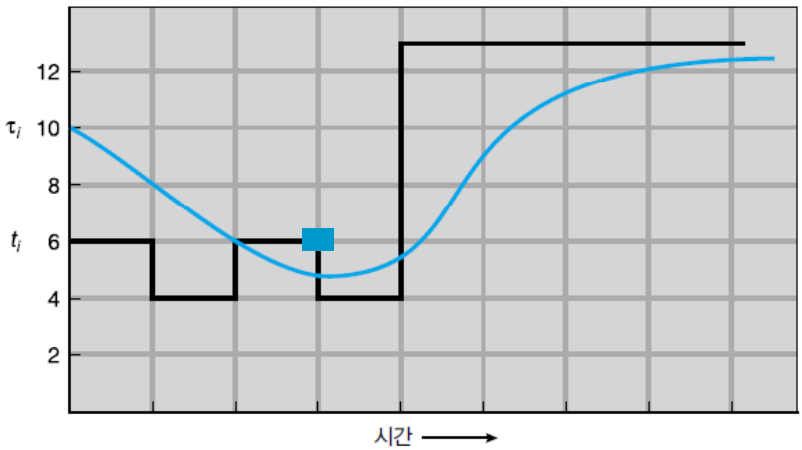
- 다음 CPU 버스트의 길이를 어떻게 알 수 있는가?

- 다음 CPU 버스트의 길이를 예측
- 이전의 CPU 버스트들의 길이를 지수 평균(exponential average)하여 예측

1. t_n = actual lenght of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$.

다음 CPU 버스트의 길이 추정 (2)

$\alpha = 1/2$이고, $\tau_0 = 10$일 때의 지수 평균



CPU 버스트(t_i)		6	4	6	4	13	13	13	· · ·
"추정" (τ_i)	10	8	6	6	5	9	11	12	· · ·

지수 평균 (Exponential Average) 예

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - 최근 값은 고려하지 안음
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - 최근의 CPU 버스트만 참조
- 일반적으로

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + \cdots + (1-\alpha)^j \alpha\, t_{n-j} + \cdots + (1-\alpha)^{n+1} \tau_0$$

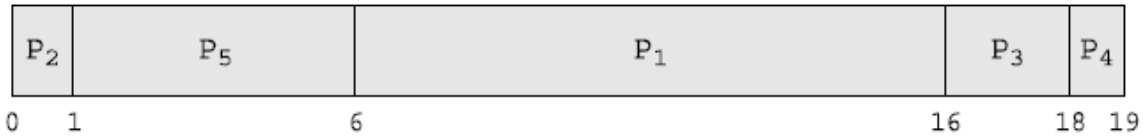
  - α 와 $(1-\alpha)$가 모두 1보다 작거나 같기때문에 각 후속의 항은 그 전 항보다 가중치(weight)가 작게된다.

우선순위 스케줄링 (Priority Scheduling)

- 우선순위 스케줄링 알고리즘은 우선순위 숫자가 각 프로세스들에게 연관
  - CPU는 가장 높은 우선순위를 가진 프로세스에게 할당
  - 비선점형, 선점형
  - SJF는 예측된 다음 CPU 버스트 길이가 우선순위가 되는 우선순위 스케줄링 알고리즘의 특별한 경우
  - 내부적으로 시간 제한, 메모리 요구, 열린 파일의 수, 평균 입/출력 버스트의 평균 CPU 버스트에 대한 비율 등 우선순위의 계산에 사용
  - 외부적으로는 프로세스의 중요성, 컴퓨터 사용을 위해 지불되는 비용의 유형과 양 등이 우선순위의 계산에 사용
  - 낮은 우선순위 프로세스들이 CPU를 무한히 대기하는 기아 상태 (starvation) 발생이 단점
    - 오랫동안 대기하는 프로세스들의 우선순위를 점진적으로 증가시키는 노화(aging) 기법 사용하여 해결

우선순위 스케줄링 예

프로세스	버스트 시간	우선순위
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

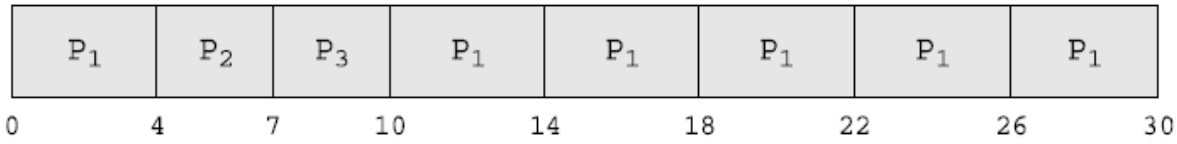


라운드 로빈 스케줄링 (Round-Robin Scheduling)

- 각 프로세스는 시간 할당량(time quantum) 또는 시간 조각(time slice)이라고 하는 작은 단위의 시간을 획득하고, 이 시간이 지나면 프로세스는 선점되고 준비완료 큐의 꼬리에 추가
  - 시분할(대화형) 시스템에 적합
  - 시간 할당량은 일반적으로 10에서 100 밀리초
  - 준비 완료 큐에 n 개의 프로세스가 있고 시간 할당량이 q이면, 각 프로세스는 자신의 다음 시간 할당량이 할당될 때까지 $(n-1) \times q$ 시간 이상을 대기하지는 않음
- RR 알고리즘의 성능
  - 시간 할당량이 매우 크면, RR 정책은 선입 선처리 정책과 같음
  - 시간 할당량이 최소한 문맥 교환 시간(10 마이크로초 미만)에 비해 더 커야 함
    - 작으면 오버헤드가 너무 높게 됨

라운드 로빈 스케줄링 예

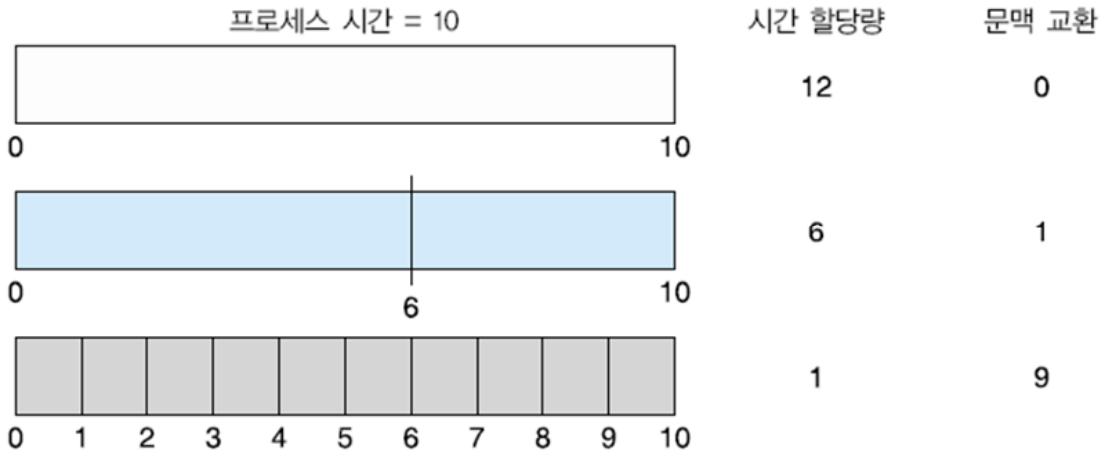
프로세스	버스트 시간
P_1	24
P_2	3
P_3	3



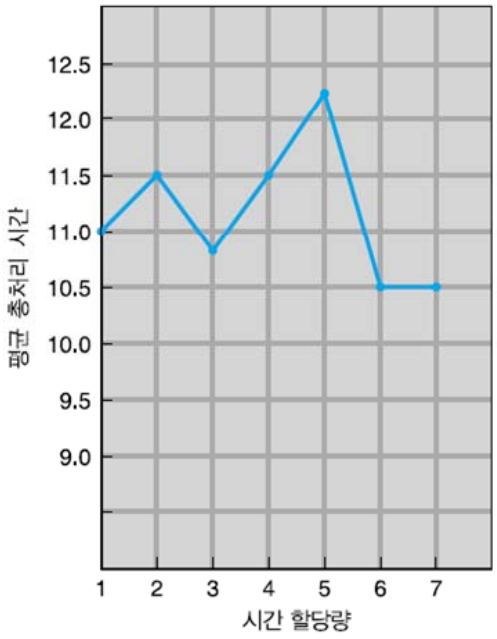
- CPU 버스트 시간의 단위는 밀리초
- 시간 할당량을 4 밀리초

- 평균 대기 시간 = (6+4+7)/3 = 5.66 밀리초

시간 할당량과 문맥 교환 시간



시간 할당량의 크기와 총처리 시간(turnaround time)



프로세스	시간
P_1	6
P_2	3
P_3	1
P_4	7

다단계 큐 스케줄링 (1) (Multilevel Queue Scheduling)

- 다단계 큐 스케줄링 알고리즘은 준비 완료 큐(ready queue)를 다수의 별도의 큐로 분류
  - 포어그라운드(foreground, 대화형)
  - 백그라운드(background, 일괄 처리) 프로세스
- 각 큐는 자신의 스케줄링 알고리즘을 수행
  - 포그라운드 : RR, 백그라운드: FCFS
- 큐와 큐 사이에 스케줄링
  - 고정 우선순위의 선점형 스케줄링
    - 포그라운드 큐는 백그라운드 큐보다 절대적으로 높은 우선순위 부여
  - 큐들 사이에 시간을 나누어 사용 (time slice)
    - 각 큐는 CPU 시간의 일정량을 받아서 자기 큐에 있는 다양한 프로세스들을 스케줄
      - 포그라운드 큐는 RR을 위해 CPU 시간의 80% 할당, 백그라운드 큐는 FCFS로 CPU 시간의 20%를 할당

다단계 큐 스케줄링 (2)

다단계 피드백 큐 스케줄링 (Multilevel Feedback Queue Scheduling)

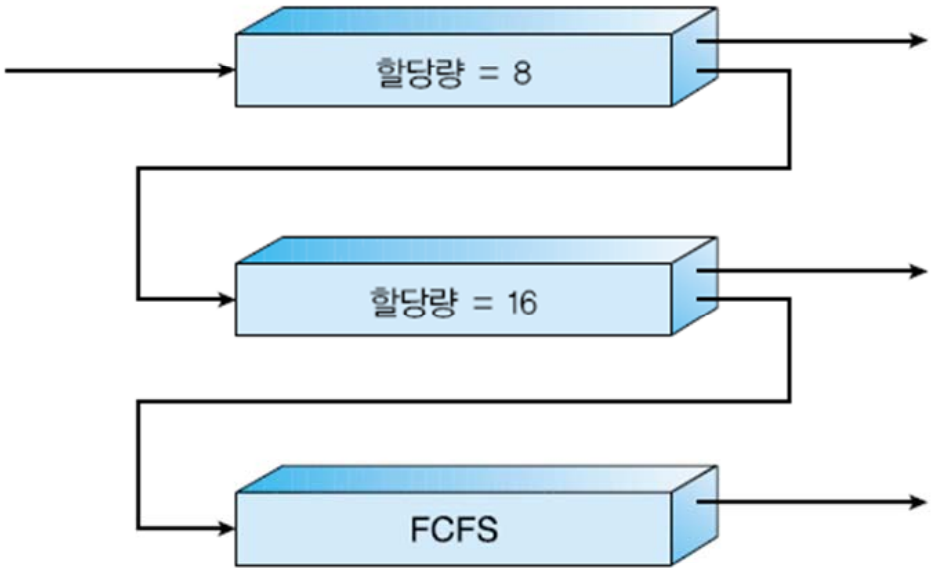
- 다단계 피드백 큐 스케줄링 알고리즘에서는 프로세스가 큐들 사이를 이동하는 것을 허용
  - 큐들 사이 이동으로 노화(aging) 구현하여 기아 상태를 예방
- 다단계 피드백 큐 스케줄러는 다음의 매개변수에 의해 정의
  - 큐의 개수
  - 각 큐를 위한 스케줄링 알고리즘
  - 한 프로세스를 높은 우선순위 큐로 올려주는 시기를 결정하는 방법
  - 한 프로세스를 낮은 우선순위 큐로 강등시키는 시기를 결정하는 방법
  - 프로세스가 서비스를 필요로 할 때 프로세스가 들어갈 큐를 결정하는 방법

다단계 피드백 큐 스케줄링 예 (1)

- 3개의 큐
  - Q_0 – RR, 8 밀리초 의 시간 할당량(quantum)
  - Q_1 – RR, 16 밀리초 의 시간 할당량
  - Q_2 – FCFS

- 스케줄링
  - 준비 완료 큐로 들어오는 프로세스는 큐 0에 넣어지고, 8 밀리초의 시간 할당량이 주어짐. 만약 프로세스가 이 시간 안에 끝나지 않는다면 큐 1의 꼬리로 이동
  - 큐 0이 비어 있다면, 큐 1의 머리에 있는 프로세스에게 16 밀리초의 시간 할당량이 주어짐. 이 프로세스가 완료되지 않는다면, 선점되어 큐 2에 넣어짐
  - 큐 0과 큐 1이 비어 있을 때만 큐 2에 있는 프로세스들이 FCFS 방식으로 실행

다단계 피드백 큐 스케줄링 예 (2)



스레드 스케줄링 (Thread Scheduling)

- 사용자 수준과 커널 수준 스레드는 스케줄링에서 차이
  - 운영체제에서는 스케줄되는 대상은 프로세스가 아니라 커널 수준 스레드
- 지역 스케줄링 (local scheduling)
  - 스레드 라이브러리가 사용자 수준 스레드를 가용한 LWP 상에서 스케줄링
    - LWP(light weight process)
      - 사용자 스레드와 커널 스레드 사이에 존재하는 중간 자료구조
      - 사용자 스레드를 실행하기 위해 스케줄하는 가상의 처리기
  - 프로세스-경쟁 범위 (process-contention scope, PCS)
    - 동일한 프로세스에 속한 스레드들 사이에서 CPU를 경쟁
- 광역 스케줄링 (global scheduling)
  - CPU 상에 어느 커널 스레드를 스케줄할 것인지 결정
  - 시스템-경쟁 범위 (system-contention scope, SCS)

Pthread 스케줄링

- 스레드 생성 시 PCS, SCS를 지정하는 API 제공
  - PTHREAD_SCOPE_PROCESS는 PCS 스케줄링을 적용하여 스레드를 스케줄
  - PTHREAD_SCOPE_SYSTEM은 SCS 스케줄링을 적용하여 스레드를 스케줄
- 경쟁 범위의 정책의 정보를 얻어내고 지정하는 함수
  - pthread_attr_getscope(pthread_attr_t *attr, int *scope)
  - pthread_attr_setscope(pthread_attr_t *attr, int scope)

```c
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread t tid[NUM THREADS];
   pthread attr t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread exit(0);
}
```


다중 처리기 스케줄링 (Multiple-Processor Scheduling)

- 여러 개의 CPU가 있는 다중 처리기에서 스케줄링은 더욱 복잡해짐
  - 부하 공유(load sharing)가 가능
- 프로세서(처리기)들의 기능이 동일한(homogeneous) 경우를 가정

- 다중 처리기 스케줄링 주요 이슈 소개
  - 대칭/비대칭 다중 처리 (asymmetric/symmetric multiprocessing)
  - 처리기 친화성 (processor affinity)
  - 부하 균등화 (load balancing)
  - SMT (Simultaneous Multithreading)

대칭/비대칭 다중 처리

- 비대칭 다중 처리(asymmetric multiprocessing)
  - 주 서버(master server)라는 하나의 처리기가 모든 스케줄링 결정과 입/출력 처리 그리고 다른 시스템의 활동을 취급
  - 다른 처리기들은 다만 사용자 코드만을 수행
  - 단지 한 처리기만 시스템 자료 구조를 접근하여 자료 공유의 필요성을 배제하기 때문에 간단

- 대칭 다중 처리(symmetric multiprocessing, SMP)
  - 각 처리기가 독자적으로 스케줄링
  - 스케줄링은 각 처리기의 스케줄러가 준비 완료 큐를 검사해서 실행할 프로세스를 선택
  - 여러 개의 처리기가 공동 자료 구조를 접근하여 갱신 가능
  - 거의 모든 현대 운영체제들은 SMP를 지원

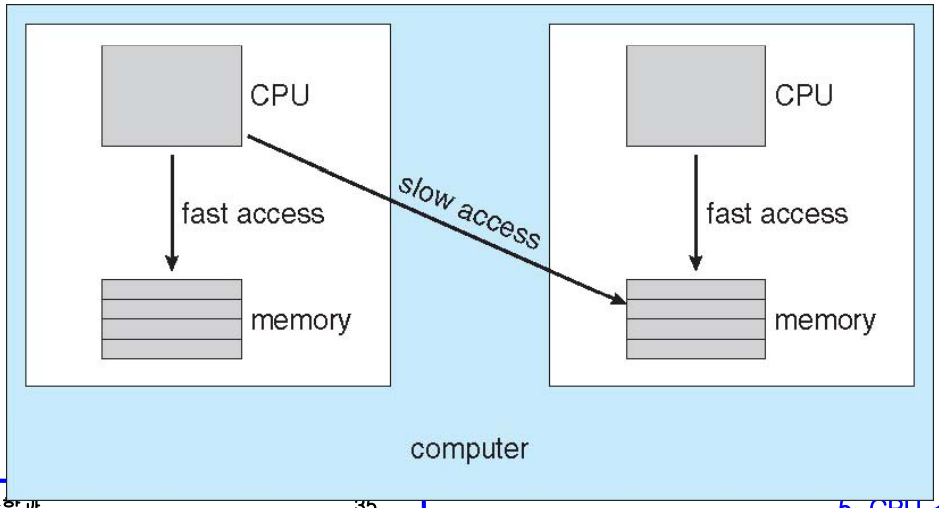
처리기 친화성 (Processor Affinity)

- SMP 시스템은 한 처리기에서 다른 처리기의 이주(migration) 를 피하고 대신 같은 처리기에서 프로세스를 실행을 시도하고, 이 현상을 처리기 친화성(processor affinity) 이라고 함
  - 이주 시 캐시를 무효화하고 다시 채우는 비용이 많이 듬

- 약한 친화성 (soft affinity)
  - 운영체제가 동일한 처리기에서 프로세스를 실행시키려고 노력하는 정책을 가지고 있지만 보장하지는 않음
- 강한 친화성 (hard affinity)
  - 시스템 호출을 통하여 프로세스가 다른 처리기로 이주되지 않도록 지정

NUMA와 CPU 스케줄링

- NUMA(Non-Uniform Memory Access) 구조
  - 비균등 메모리 접근 구조
  - 운영체제 스케줄러와 메모리 배치 알고리즘을 연동하여 특정 CPU에 친화성을 갖는 프로세스를 해당 CPU에스케줄



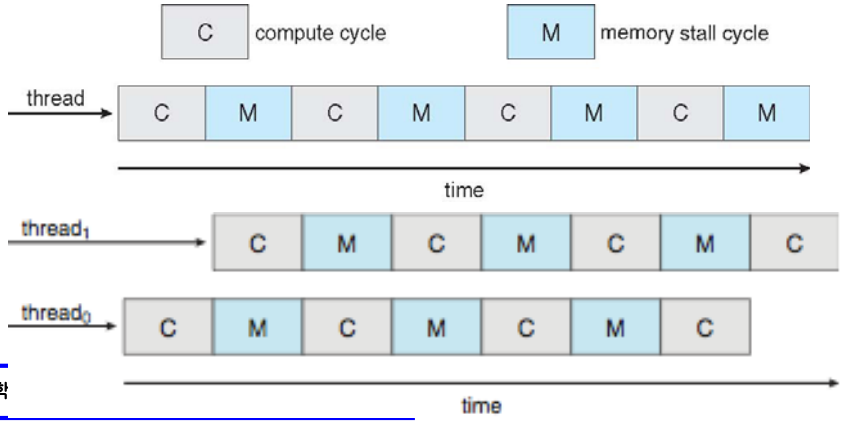
부하 균등화 (Load Balancing)

- 부하 균등화(load balancing)는 SMP 시스템의 모든 처리기 사이에 부하가 고르게 배분되도록 시도

- push 이주(migration)
  - 특정 태스크가 주기적으로 각 처리기의 부하를 검사
  - 만일 불균형 상태로 밝혀지면 과부하인 처리기에서 쉬고 있거나 덜 바쁜 처리기로 프로세스를 이동(또는 push)시킴으로써 부하를 분배
- pull 이주 방식
  - 쉬고 있는 처리기가 바쁜 처리기를 기다리고 있는 프로세스를 pull할 때 발생
- Linux는 200 밀리초마다(push 이주) 또는 처리기의 실행 큐가 비게되면(pull 이주) 자신의 부하 균등화 알고리즘을 실행

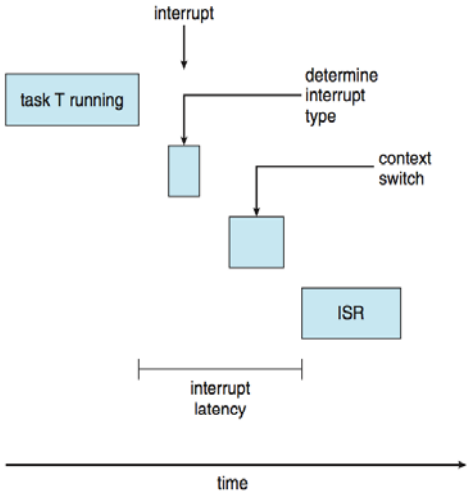
다중 코어 프로세서 (Multicore Processors)

- 최근의 경향은 하나의 물리적인 칩 안에 여러 개의 처리기 코어를 장착
  - 속도가 빠르고 적은 전력 소모
  - SMT (Simultaneous Multithreading), Hyperthreading
- 코아 당 다수의 스레드 스케줄
  - 한 스레드의 메모리 멈춤(memory stall) 시 다른 스레드 스케줄



실시간 CPU 스케줄링 (1) (Real-Time CPU Scheduling)

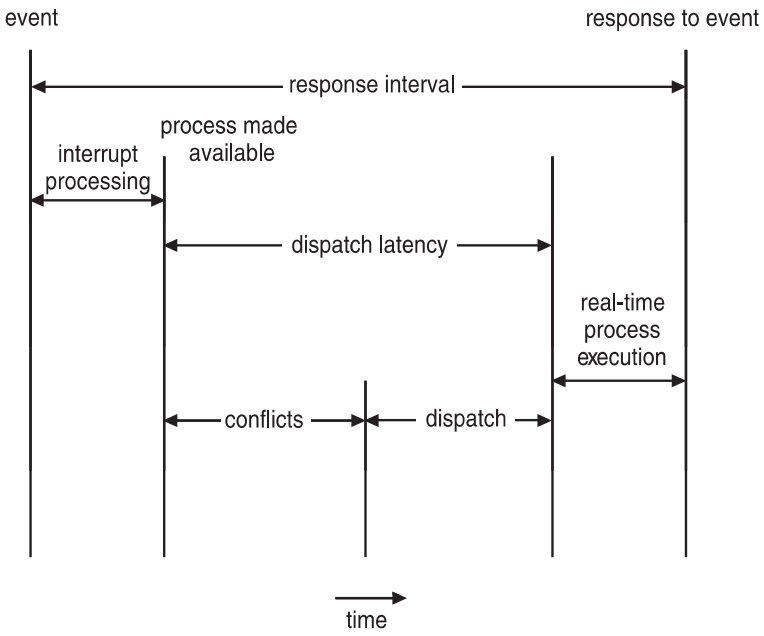
- 연성 실시간 시스템 (soft real-time system)
  - 중요한(critical) 실시간 프로세스가 스케줄 된다는 보장 없음
- 경성 실시간 시스템 (hard real-time system)
  - 마감시한(deadline) 내에 태스크가 서비스되어야 함



- 두 종류의 지연이 성능에 영향을 미침
  1. 인터럽트 지연
     인터럽트 도착에서부터 인터럽트 서비스까지의 시간
  2. 디스패치 지연
     현재 프로세스가 CPU에서 물러나고 다른 프로세스를 스케줄하는 시간

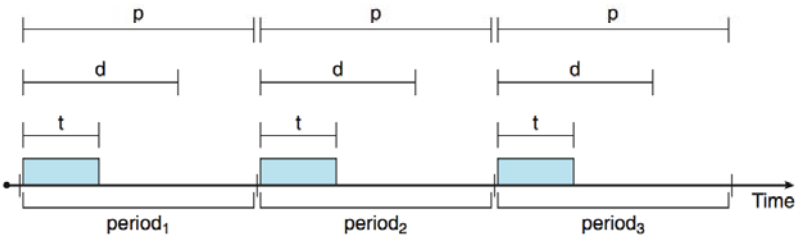
실시간 CPU 스케줄링 (2)

- 디스패치 지연 시 상충 단계 (conflict phase)
  1. 커널 모드에서 실행되고 있는 어떤 프로세스도 선점(preemption)
  2. 높은 우선순위의 프로세스가 필요한 자원(현재 낮은 우선순위의 프로세스가 점유한 자원) 해제(release)



실시간 스케줄링 – 우선순위 기반

- 실시간 운영체제의 스케줄러는 선점(preemptive)을 이용한 우선순위 기반의 스케줄링(priority-based scheduling) 알고리즘을 지원해야함
- 실시간 운영체제의 프로세스의 특성
  - 프로세스들을 주기적(periodic)이고 일정한 간격으로 CPU를 필요로 함
  - 각 프로세스는 고정된 수행시간 t, 마감시간 d, 주기 p가 정해져 있음
  - $0 \le t \le d \le p$
  - 주기적인 태스크의 빈도(rate)는 1/p

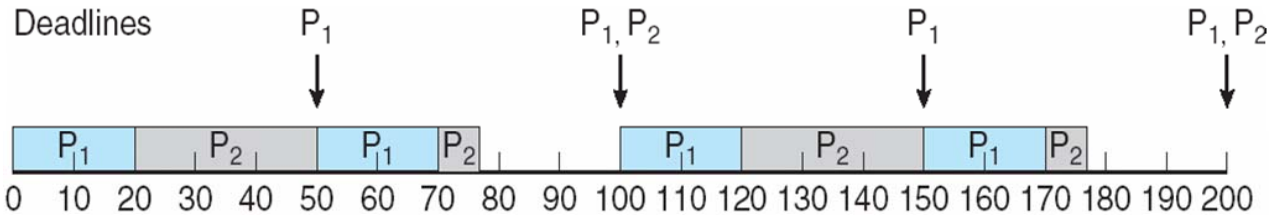


실시간 스케줄링 – Rate Monotonic 스케줄링 (1)

- 선점 가능한 정적 우선순위 정책 사용하여 주기적인 태스크 스케줄
  - 주기에 반비례하여 우선순위 배정
  - 주기가 짧으면 높은 우선순위, 길면 낮은 우선순위가 배정

- 두 개의 프로세스 P1, P2 예 1
  - P1 프로세스
    - 주기 p1=50, 수행시간 t1=20, CPU 이용률 = t1/p1 = 20/50 = 0.40
  - P2 프로세스
    - 주기 p2=100, 수행시간 t2=35, CPU 이용률 = t2/p2 = 35/100 = 0.35
  - 각 프로세스의 마감시간은 다음 주기가 시작하기 전임
  - 총 CPU 이용률 = 0.4+0.35= 0.75, 75%

실시간 스케줄링 – Rate Monotonic 스케줄링 (2)

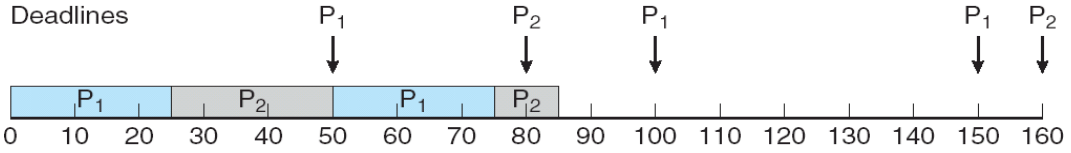


- 주기 p1=50, 수행시간 t1=20
- 주기 p2=100, 수행시간 t2=35

- 만약 P2가 높은 우선순위라면 다음과 같이 P1이 마감시간을 만족시키지 못함
  - p.262 그림 5.16

실시간 스케줄링 – Rate Monotonic 스케줄링 (3)

- 두 개의 프로세스 P1, P2 예 2
  - P1 프로세스
    - 주기 p1=50, 수행시간 t1=25, CPU 이용률 = t1/p1 = 25/50 = 0.50
  - P2 프로세스
    - 주기 p2=80, 수행시간 t2=35, CPU 이용률 = t2/p2 = 35/80 = 0.44
  - 각 프로세스의 마감시간은 다음 주기가 시작하기 전임
  - 총 CPU 이용률 = 0.5+0.44= 0.95, 95%
  - p2의 마감시간을 충족 시키지 못함



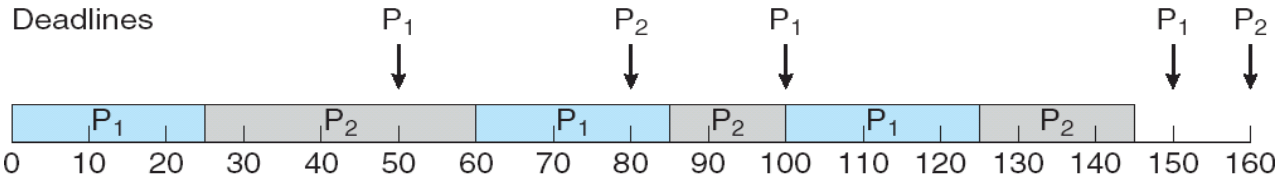
- Rate Monotonic 스케줄링은 CPU 이용률 한계로 CPU 자원을 최대화하여 사용하는 것이 불가능

실시간 스케줄링 – Earliest-Deadline-First 스케줄링 (1)

- Earliest-deadline-first (EDF) 스케줄링은 마감시간에 따라 우선순위를 동적으로 부여
  - 마감시간이 빠를수록 우선순위는 높고, 늦을 수록 낮아짐

- 두 개의 프로세스 P1, P2 예 2
  - P1 프로세스
    - 주기 p1=50, 수행시간 t1=25, CPU 이용률 = t1/p1 = 25/50 = 0.50
  - P2 프로세스
    - 주기 p2=80, 수행시간 t2=35, CPU 이용률 = t2/p2 = 35/80 = 0.44

실시간 스케줄링 – Earliest-Deadline-First 스케줄링 (2)



- 시간 50에서 P2의 마감시간(80)이 P1의 마감시간(100, 두번째)보다 더 빠르기때문에 P2의 우선순위가 높아 선점 당하지 않음
- 시간 100에서 P1의 마감시간(150, 세번째)가 P2의 마감시간(160, 두번째) 보다 더 빠르기때문에 P2는 선점됨

❑ EDF는 모든 프로세스가 마감시간을 만족하도록 스케줄링이 가능
- 프로세스가 자신의 마감시간을 스케줄러에게 동적으로 알려 주어야 함

실시간 스케줄링 – 일정 비율의 몫 스케줄링

❑ 일정 비율의 몫 스케줄링(proportional share scheduling)은 모든 응용들에게 T개의 시간의 몫을 할당
- 한 응용이 N개의 시간 몫을 할당 받으면 모든 프로세스의 시간 중 N/T 시간을 할당 받게 됨

❑ A B C 3개의 프로세스, T=100 예
- A = 50, B = 15, C = 20 의 시간의 몫 할당
  - A는 처리기의 50%, B는 15%, C는 20% 할당
- 새로운 프로세스 D가 30의 시간의 몫 요구시 승인 제어기는 D의 시스템 진입을 거부

POSIX 실시간 스케줄링

- POSIX는 실시간 컴퓨팅 용으로 POSIX.1b 확장을 제공
- 실시간 스레드를 관리하는 API 제공
- 실시간 스레드와 관련하여 두 개의 스케줄링 클래스 정의
  - SCHED_FIFO
    - FIFO 큐를 사용하여 먼저 온 것을 먼저 서비스하는 정책에 따라 스케줄
    - 스레드들이 같은 우선순위 시간 조각(time-slicing)이 없음
  - SCHED_RR
    - 스레드들이 같은 우선순위 시간 조각(time-slicing)을 갖는 것을 제외하고는 SCHED_FIFO와 비슷
- 스케줄링 정책에 관한 정보를 저장하고 얻어내는 두 개의 함수 제공
  - `pthread_attr_getsched_policy(pthread attr t *attr, int *policy)`
  - `pthread_attr_setsched_policy(pthread attr t *attr, int policy)`


```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
 /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread exit(0);
}
```

리눅스 스케줄링 (버전 2.5)

- 리눅스 커널 버전 2.5는 O(1) 상수 복잡도의 스케줄링 시간
  - 선점형, 우선순위 기반
  - 두 가지 우선순위 영역: 시분할(time-sharing), 실시간(real-time)
  - 실시간 영역은 0 ~ 99, nice 영역은 100 ~ 140 값 범위
  - 낮은 값의 우선순위가 높음
  - 우선순위가 높을 수록 더 큰 시간 조각(time slice, quantum)을 부여
  - 시간 조각이 남아 있는 길이 만큼 태스크는 실행 가능(runnable, active)
  - 시간 조각이 남아 있지 않으면(expired), 모든 다른 태스크들이 자신의 시간 조각을 다 사용할 때까지 실행 가능하지 않음
  - 실행 가능한 모든 태스크들은 per-CPU 런큐(runqueue) 자료구조로 관리
    - 두개의 우선순위 배열 (active, expired)
    - 우선순위에 따라 태스크들 인덱싱
    - Active 태스크가 없는 경우 2개의 배열 교환
  - 잘 동작하지만 대화형 프로세스의 응답 시간 느림

리눅스 스케줄링 (버전 2.6.23+) (1)

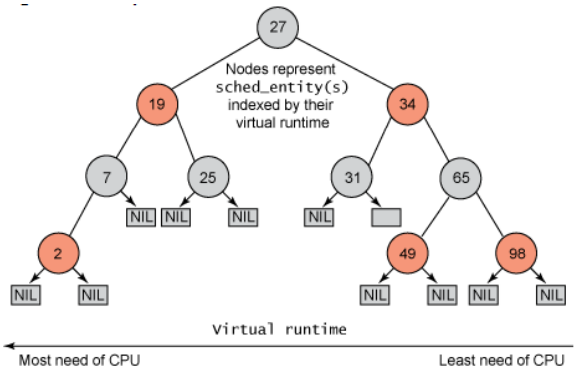
- CFS (Completely Fair Scheduler)
- 스케줄링 클래스
  - 각 태스크는 특정 우선순위 가짐
  - 스케줄러는 가장 높은 우선순위 클래스에서 가장 높은 우선순위의 태스크를 선택
  - 고정된 시간 조각 할당 보다는 CPU 시간에 비례하여 시간 조각 지정
  - 기본적으로 두 개의 스케줄링 클래스가 있으며, 다른 클래스 추가도 가능 -> 디폴트, 실시간
- 시간조각은 -20~+19 사이의 nice 값 기반으로 계산
  - 낮은 값이 높은 우선순위
  - 태스크가 최소한 한 번 실행되기까지의 간격인 목적 지연시간(target latency) 계산
  - 활성 태스크의 수가 일정 임계값보다 많아지면 목적 지연시간 증가

리눅스 스케줄링 (버전 2.6.23+) (2)

- CFS는 태스크 당 가상 실행시간(virtual run time) , vruntime 관리
  - 태스크의 우선순위에 기반한 감쇠 지수(decay factor)와 관련됨
    - 낮은 우선순위의 태스크는 높은 우선순위의 태스크보다 감쇠율이 높음
  - 보통의 우선순위(nice 값이 0) 태스크의 경우 가상 실행 시간은 실제 물리적 실행 시간과 같음
  - 실제 실행시간 200ms 인 경우 예
    - 보통의 우선순위 (nice = 0): vruntime = 200
    - 높은 우선순위 (nice < 0): vruntime < 200
    - 낮은 우선순위 (nice > 0): vruntime > 200
- 가장 작은 가상 실행시간을 갖는 태스크를 다음에 실행된 태스크로 선택
  - 높은 우선순위의 태스크는 낮은 우선순위의 태스크를 선점할 수 있음

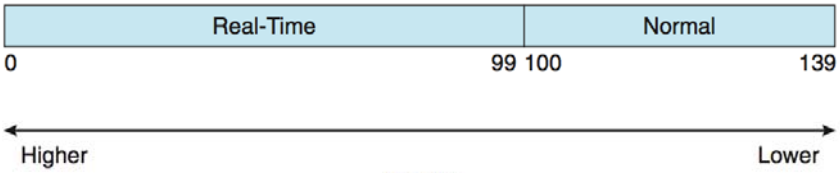
리눅스 스케줄링 (버전 2.6.23+) – CFS 성능

- 리눅스 CFS 스케줄러는 모든 실행 가능한 태스크는 키 값으로 vruntime을 갖는(가상 실행시간으로 정렬된) 균형 이진 탐색 트리(balanced binary search tree)인 적-백 트리(red-black tree)를 구성하여 관리
  - 실행 가능한 태스크는 트리에 추가
    - 입출력 봉쇄되어 실행 가능하지 않는 경우 트리에서 제거
  - 왼쪽 노드가 가장 작은 키 값 (가장 높은 우선순위)
  - 균형 트리이므로 탐색 시간은 O(logN), N은 노드의 갯 수



리눅스 스케줄링 (버전 2.6.23+) (3)

- POSIX.1b에 기반한 실시간 스케줄링
  - 실시간 태스크는 보통의 태스크보다 높은 우선순위로 실행
- 실시간 태스크 우선순위 영역, 보통의 태스크 우선순위 영역으로 구분
  - 실시간 태스크는 0에서 99 사이의 영역의 정적 우선순위
  - 보통의 태스크는 100에서 139까지의 영역 우선순위
  - 두 영역이 하나의 전역 우선순위(global priority) 구조로 사상
    - 전역 영역에서는 작은 값이 상대적으로 높은 우선순위를 가짐
    - -20의 nice 값은 우선순위 100에 사상
    - +19는 우선순위 139에 사상



알고리즘의 평가

- 특정 시스템을 위한 알고리즘을 어떻게 선택하는가?
  1. 알고리즘을 선택하는 기준(criteria) 정의
     - CPU 이용률, 응답 시간, 처리량 등
  2. 알고리즘을 평가

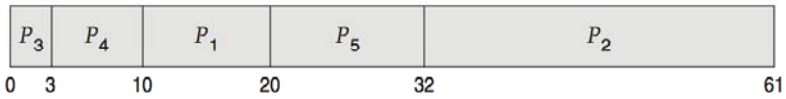
- 결정론적 모델링 (Deterministic Modeling)
  - 분석적 평가 (analytic evaluation) 유형
  - 주어진 작업 부하에 대한 알고리즘의 성능을 평가하는 공식이나 값을 생성하기 위해 주어진 알고리즘과 시스템 작업 부하를 이용
  - 시간 0에 도착한 다섯개의 프로세스 예 (시간은 ms)

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

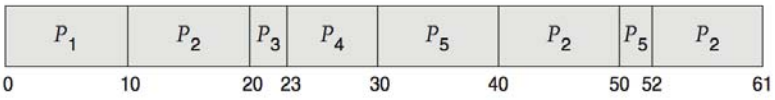
결정론적 평가 (Deterministic Evaluation)

- 각 알고리즘에 대해 최소의 평균 대기시간을 계산
- 결정론적 모델은 단순하고 빠르지만, 입력으로 정확한 숫자를 요구하고, 평가 결과도 단지 이들 입력 값에만 적용
  - 선입 선처리(FCFS): (0+10+39+42+49)/5 = 28ms

  - 비선점 최단작업 우선(SJF): (10+32+0+3+20)/5 = 13ms



  - 라운드 로빈(RR): (0+32+20+23+40)/5 = 23ms



큐잉 모델 (Queuing Model)

- 실제 시스템에서 실행되는 프로세스는 수시로 변하기 때문에 결정론적 모델링을 사용할 수 있는 프로세스들의 정적인 집합은 없음
- 큐잉 모델 (Queuing Model)
  - 프로세스의 도착 시간, CPU와 입출력 버스트들을 확률적인 분포로 기술
    - 보통 지수적(exponential)이며 평균에 의해 기술
    - 이들 분포로부터 평균 처리량, 이용률, 대기 시간 등을 계산
  - 컴퓨터 시스템은 서버들의 네트워크로 기술되고, 각 서버는 대기 프로세스 들의 큐를 가지고 있음
    - CPU는 준비 완료 큐를 갖는 서버, 입출력 시스템은 장치 큐를 갖는 서버
    - 도착률과 서비스율을 알면 이용률, 평균 큐 길이, 평균 대기 시간 등을 계산할 수 있음
  - 이러한 연구 분야를 큐잉 네트워크 분석(queuing-network analysis)라고 함

큐잉 모델 – Little's Formula

- Little's Formula
  - 시스템이 안정 상태라면 큐를 떠나는 프로세스의 수는 도착하는 프로세스의 수와 같아야 함

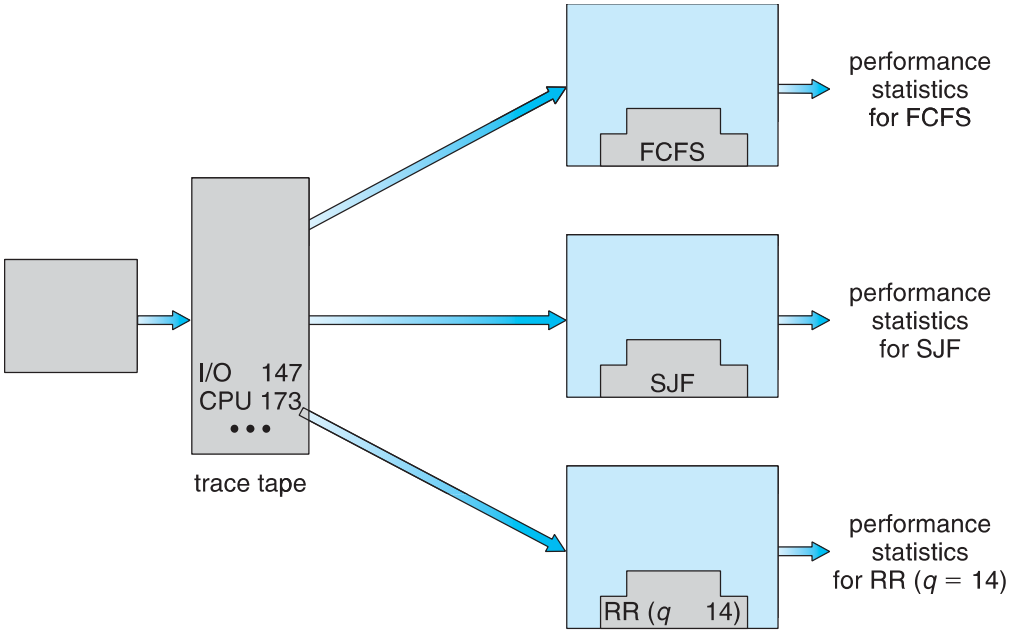
$n = \lambda \times W$

- n =평균 큐 길이
- W = 큐에서의 평균 대기 시간
- λ = 새로운 프로세스들이 큐에 도착하는 평균 도착률
- 예를들어 초 당 7개의 프로세스가 평균적으로 도착하고, 큐에 통상 14개의 프로세스가 있음을 안다면, 프로세스 당 평균 대기 시간은 2초로 계산

모의실험 (Simulation)

- 큐잉 모델은 제한적
  - 복잡한 알고리즘이나 분포와 관련된 수학적 분석이 어려워 실제 시스템에 대한 근사치만을 산출
- 모의실험(simulation)이 스케줄링 알고리즘을 더 정확하게 평가
  - 컴퓨터 시스템 모델을 프로그래밍
  - 시간을 나타내는 변수 사용
  - 알고리즘의 성능을 나타내는 통계들을 수집
  - 모의실험을 구동하기 위한 자료들은 여러 가지 방법으로 생성
    - 확률 분포(probability distribution)에 따른 난수 발생기
      - 확률 분포에 따라 프로세스, CPU 버스트 시간, 도착, 출발 등을 생성
    - 분포는 수학적 경험적으로 정의
    - 추적 테이프(trace tape)를 사용하여 실제 시스템의 실제 사건들의 순서를 기록

모의실험에 의한 CPU 스케줄러의 평가



구현 (Implementation)

- 모의실험도 정확성에 한계
- 실제 운영 환경 하에서의 평가를 위해 실제 시스템에 실제 알고리즘을 삽입하여 실행
  - 높은 비용, 높은 위험성
  - 알고리즘이 사용되는 환경 변화
- 가장 융통성 있는 스케줄링 알고리즘은 시스템 관리자 또는 사용자에 의해 알고리즘이 조정 가능해야 함
- 또 다른 접근 방법은 우선순위를 변경하는 API 사용
  - 알고리즘이 사용되는 환경 변화

과제: 연습문제

- 연습문제 5.5
- 연습문제 5.7
- 연습문제 5.8
- 연습문제 5.22

실습 과제

- 다음 프로그램에서 스레드 함수에 임의의 작업을 추가하여 실행하고 결과를 분석하라.
  - p.252 그림 5.8 Pthreads 스케줄링 API
  - p.266 그림 5.20 POSIX 실시간 스케줄링 API