5장. CPU 스케줄링

목표

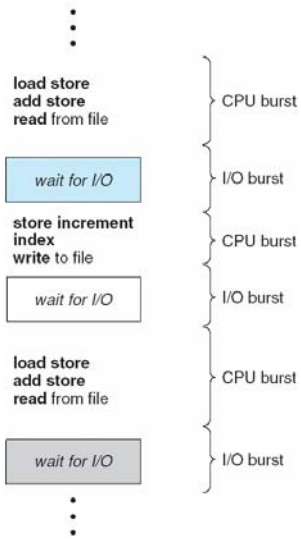
- multiprogramming 운영체제의 기반인 CPU 스케줄링 소개
- 다양한 CPU 스케줄링 알고리즘
- CPU 스케줄링 알고리즘 선택을 위한 평가 기준
- 스케줄링 알고리즘 사례

5.1 기본 개념

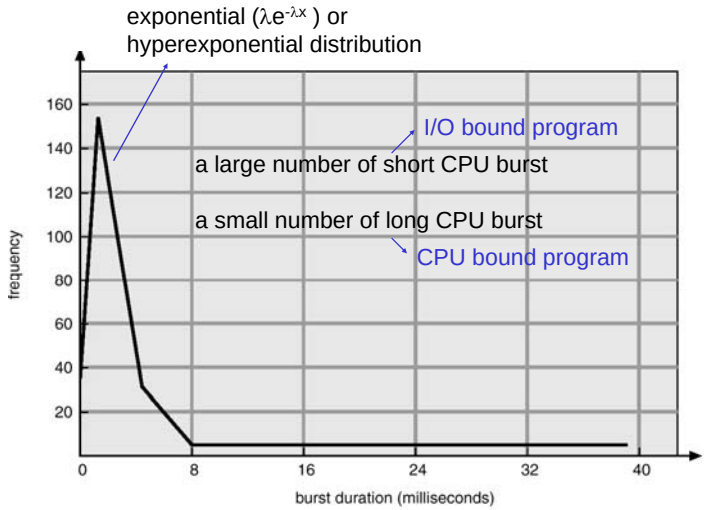
- multiprogramming의 목적
  - CPU 이용률 최대화
- CPU-I/O Burst Cycle
  - 프로세스 실행은 **CPU 실행**과 **I/O 대기**의 사이클로 구성됨

CPU burst I/O burst

- CPU burst 분포
  - (see next page)

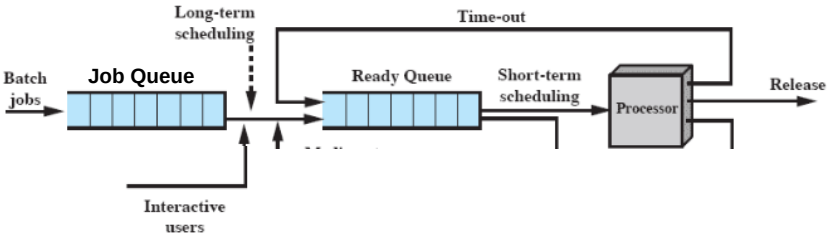


CPU-burst 시간의 분포도



CPU Scheduler

- CPU scheduler (short-term scheduler)
  - ready queue에 있는 프로세스들 중 하나를 선택하여 이 프로세스에게 CPU를 할당함



CPU Scheduling 시점

- CPU scheduling 결정은 process가 다음 상황일 때 발생 가능함
  1. running 상태에서 waiting 상태로 전환 (예. I/O wait, child termination wait)
  2. running 상태에서 ready 상태로 전환. (e.g. time-out)
  3. waiting 상태에서 ready 상태로 전환. (e.g. I/O completion, event occur)
  4. Terminate.
- **비선점(non-preemptive) 스케줄링** – cooperative 스케줄링
  - 1번, 4번 경우는 선택의 여지가 없으며, 이 경우에만 스케줄링
    - 반드시 스케줄링하여 새 프로세스를 선택해야 함
    - 프로세스는 종료하거나 block될 때까지 CPU를 계속 점유
  - (예) windows 3.x, 예전 Mac OS
- **선점(preemptive) 스케줄링**
  - 모든 경우에 스케줄링이 가능(2, 3번 경우 포함)
  - CPU 독점을 방지하거나(timer 사용), 프로세스 우선순위를 반영하고자 할 때 2, 3번의 경우(ready queue가 변화)에 스케줄링 할 수 있음
  - (예) 대부분의 현대 OS

Preemptive 스케줄링의 문제점과 해결책

- 공유 데이터의 일관성(consistence) 유지 문제
  - 선점 스케줄링 방식에서, process는 프로세스는 데이터를 변경하는 도중에 다른 프로세스에게 선점되어 변경된 데이터를 저장하기 전에 CPU를 내어놓을 수 있다.
  - 다수의 프로세스가 데이터를 공유할 때에 경쟁적으로 데이터를 변경하면 데이터 일관성이 유지되지 않을 수가 있다. → 경쟁조건
  - **해결책** : 공유 데이터 접근에 대한 조정이 필요 → 동기화 방식 (6장)
- user mode에서의 preemption
  - 선점형 스케줄링을 하는 운영체제에서, 한 프로세스가 데이터를 변경하는 동안 선점되어 다른 프로세스가 같은 데이터를 읽거나 수정한다면, 데이터 일관성이 유지되지 않을 수 있음
  - 사용자 프로그램은 운영체제가 제공하는 **동기화 방식**을 사용하여 데이터 일관성 문제가 발생하지 않도록 작성해야 한다.

Kernel mode에서의 preemption

- Kernel mode에서의 공유 데이터 접근 문제
  - 모든 커널 루틴은 커널 데이터를 공유함
  - 커널은 system call을 통하여 요청된 프로세스의 작업을 처리할 수 있으며, 공유 데이터를 접근하는 커널 루틴이 실행되는 동안에, **인터럽트**로 인해서 다른 커널 루틴에게 선점되면 공유 데이터 일관성 유지가 되지 않을 수 있다.
  - 커널에서의 이러한 문제 발생은 시스템 전체에 영향을 주므로 위험함
- 운영체제 커널에서의 preemption 처리 방법
  - 비선점형 커널 – 커널 내에서의 preemption을 허용하지 않음
    (1) system call이 완료되거나 (2) I/O block이 발생할 때까지 기다린 후에 context switching을 수행
    - 실시간 컴퓨팅을 지원하는 데 부적합
  - 선점형 커널 – 커널 내에서 preemption을 허용
    - 커널 내에서 공유 데이터 접근에 대한 동기화를 사용하여 커널을 작성해야 함
    - 실시간 컴퓨팅 지원에 적합

디스패처(Dispatcher)

- Dispatcher
  - CPU의 제어권을 CPU 스케줄러가 선택한 프로세스에게 주는 모듈
  - 다음 작업 수행
    - context 스위칭
    - CPU 동작 모드를 user mode로 전환
    - 선택한 프로세스가 다시 시작하도록, user program의 적절한 위치로 이동(jump)
- Dispatch 지연(latency)
  - 한 프로세스를 정지하고, 다른 프로세스의 수행을 시작할 때까지 소요되는 시간
  - dispatch latency은 가능한 한 작아야 함(빠르게 동작)

5.2 Scheduling 기준(Criteria)

- CPU 이용률(utilization)
  - 0 – 100% (CPU를 가능한 한 바쁘게 유지)
- 처리량(Throughput)
  - 단위 시간당 수행이 완료된 프로세스
- 총 처리시간(Turnaround time)
  - 프로세스를 실행하는 데 소요된 시간
- 대기시간(Waiting time)
  - ready queue에서 대기하는 시간 (CPU 실행과 I/O 대기가 아닌 시간)
- 응답시간(Response time)
  - 대화형 시스템에서 요청을 한 후에 (첫 번째) 응답을 받을 때까지의 소요시간 (최종 출력을 얻는 시간이 아님))

최적화 기준

- 스케줄링 알고리즘 최적화 기준
  - CPU 이용률과 처리량 : 최대화
  - 총 처리 시간, 대기 시간, 응답 시간 : 최소화
- 최적화하는 값
  - 대부분의 경우 평균값을 최적화
  - 일부 경우에는 평균값 대신에 최대값 또는 최소값을 최적화
  - 대화형 시스템은, 응답시간의 변동폭(variance)를 최소화
    → 합리적이고 예측 가능한 응답시간 제공

5.3 스케줄링 알고리즘(Scheduling Algorithm)

- 선입선처리(First-Come First-Serve:FCFS) 스케줄링
- 최단작업우선(Shortest-Job-First:SJF) 스케줄링
- 우선순위(Priority) 스케줄링
- 라운드로빈(Round Robin:RR) 스케줄링
- 다단계 큐(Multilevel Queue) 스케줄링

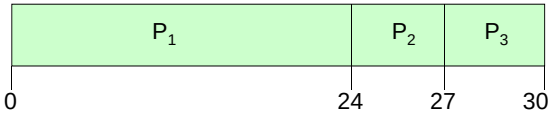
First-Come, First-Served (FCFS) Scheduling

- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Arrival order: P_1 , P_2 , P_3 (arrival time t=0)

The *Gantt Chart* for the schedule

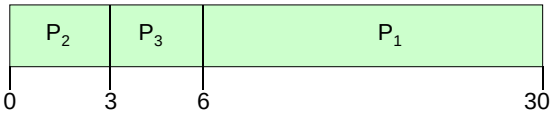


  - 대기시간: P_1 = 0; P_2 = 24; P_3 = 27
  - 평균대기시간: (0 + 24 + 27)/3 = 17

FCFS Scheduling (계속)

- Arrival order: P_2 , P_3 , P_1 . (arrival time t=0)

The Gantt chart for the schedule is:



  - 대기시간 : P_1 = 6; P_2 = 0, P_3 = 3
  - 평균대기시간 : (6 + 0 + 3)/3 = 3
    → 이전의 경우보다 더 좋음
- 호위효과(Convoy effect)
  - 긴 프로세스 뒤에 짧은 프로세스들이 있는 경우에
    짧은 프로세스는 긴 프로세스의 CPU busrt가 끝날 때까지 기다려야 함
    → 짧은 프로세스들이 먼저 처리되도록 할 때보다 CPU와 장치 이용률이 저하됨

Shortest-Job-First (SJF) Scheduling

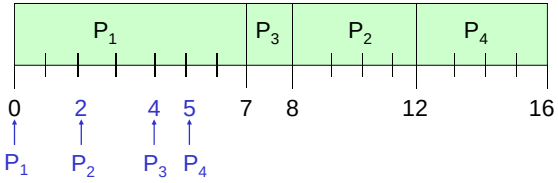
- 최단작업 우선(**SJF**) 스케줄링
  - Shortest Process Next(SPN), Shortest Request Next(SRN)라고도 함
  - 프로세스는 다음 CPU burst 길이가 연관되며
    최단 다음 CPU burst를 갖는 프로세스를 스케줄링
  - 문제점 : 다음 CPU burst를 알기 어려움 → 해결책 : 예측(*prediction*)
- 두 가지 방식의 SJF 스케줄링
  - 비선점 SJF – process는 CPU burst를 끝낼 때까지 선점되지 않음
  - 선점 SJF – (새 프로세스의 CPU burst < 현재 프로세스의 잔여시간)
    이면 현재 프로세스가 선점되어 스케줄링
    → Shortest-Remaining-Time-First (**SRTF**).
    "preemptive SJF = SRTF"
- SJF는 대기시간이 최적임
  - 최소 평균 대기시간을 제공.

(예) Non-Preemptive SJF

- Example

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

- SJF (non-preemptive)



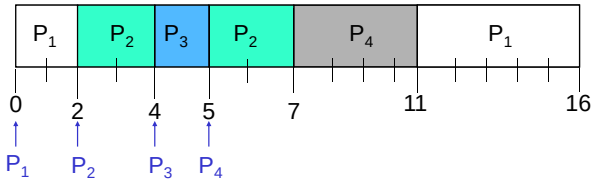
  - 평균대기시간 = (0 + 6 + 3 + 7)/4 = 4

(예) Preemptive SJF

- Example

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

- SJF (preemptive)



  - 평균대기시간 = (9 + 1 + 0 +2)/4 = 3

다음 CPU Burst 길이 예측

- 다음 CPU Burst 길이를 미리 알 수 없음
- 해결책 : 다음 CPU Burst 길이 예측
  - 이전 CPU Burst 길이를 사용하여 다음 CPU Burst 길이를 예측
    - 가정: 다음 CPU Burst 길이는 이전의 값과 비슷할 것이다.
- 예측값 : 지수 평균(exponential average) 사용
  - 다음 예측값 = τ_n (이전 예측값)과 t_n (이전 CPU Burst)의 지수 평균

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

  - initial τ_0 : 상수 또는 전체 평균으로 정의

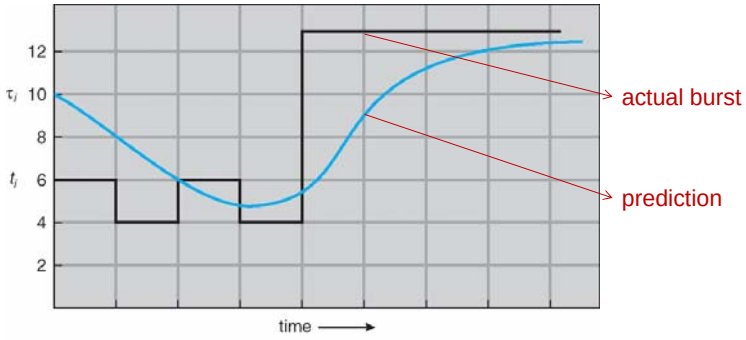
1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$

지수 평균

- 지수 평균의 예
  - α =0: $\tau_{n+1} = \tau_n$ ➔ 최근 값을 고려하지 않고, 이전 예측값 사용
  - α =1: $\tau_{n+1} = t_n$ ➔ 최근 값만 사용하고, 이전 예측값 사용하지 않음.
  - 일반적으로 α =1/2 ➔ 최근 값과 이전 예측값의 평균을 사용
- $0 \le \alpha$, $(1 - \alpha) \le 1$ ➔ τ_{n+1} 은 τ_n 또는 t_n 보다 가중치가 작다.
- 지수 평균식 확장

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha\, t_{n-j} + \ldots + (1-\alpha)^n \alpha\, t_0 + (1-\alpha)^{n+1}\tau_0$$

Next CPU Burst 길이 예측

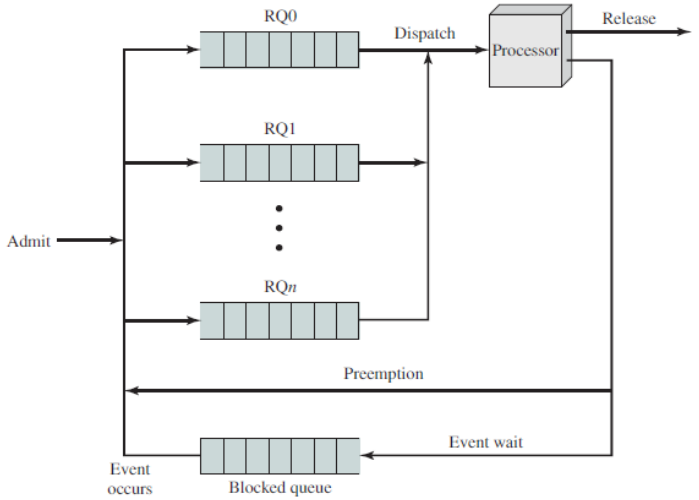


CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

우선순위(Priority) Scheduling

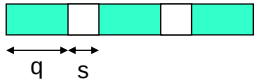
- 우선순위 스케줄링
  - 프로세스는 우선순위 번호와 연관됨
    - 대개 우선순위가 높을 수록 작은 우선순위 번호를 가짐
  - CPU는 가장 높은 우선순위를 가진 프로세스에게 CPU를 할당
- 우선순위 정의에 고려되는 요인
  - **내부적** : 시간 제한, 메모리 요구, open file 수, I/O와 CPU의 비율 등
  - **외부적** : 프로세스 중요성, 비용의 유형과 액수, 정치적 요인 등
- 두 가지 방식의 Priority Scheduling – 선점, 비선점
- SJF 스케줄링은 priority 스케줄링의 특별한 경우임
  - priority = 다음 CPU burst time의 예측값 (작을수록 높은 우선순위)
- 문제점 : **기아 상태**(starvation)/무한 봉쇄(indefinite blocking)
  - 낮은 우선순위의 process가 무한히 대기하여 수행되지 못할 수 있음
  - 해결책 ➔ **노화(aging)**
    - 오랫동안 대기하는 process는 우선순위를 점진적으로 증가시킴

Priority Queueing



Round Robin (RR) Scheduling

- Round Robin 스케줄링
  - 시분할 시스템을 위해서 설계됨 – CPU 공유
  - 각 프로세스에게 작은 양의 CPU 시간(time quantum) 할당
    - 크기: 대개 10-100 msec
  - 실행 프로세스는 이 시간 경과 후 **선점**되어 ready queue의 끝으로 이동
  - 타이머 인터럽트 사용
- 최대 대기 시간
  - n개의 ready process가 존재하고, time quantum *q* 일 때
    max. waiting time = (n-1)q
- 성능
  - *q* large ⇒ FIFO (FCFS)와 같게 됨
  - *q* small ⇒ *q*가 context switch time에 비해서는 커야 함.
    그렇지 않으면 switching overhead가 너무 크게 됨.
- 시간 할당량(time quantum)을 정하는 경험 법칙
  - 전체 CPU burst time의 80% 정도가 quantum time보다 짧도록 정함

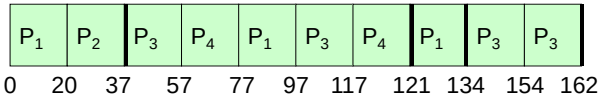


예: RR 스케줄링 (Time Quantum = 20)

- Example

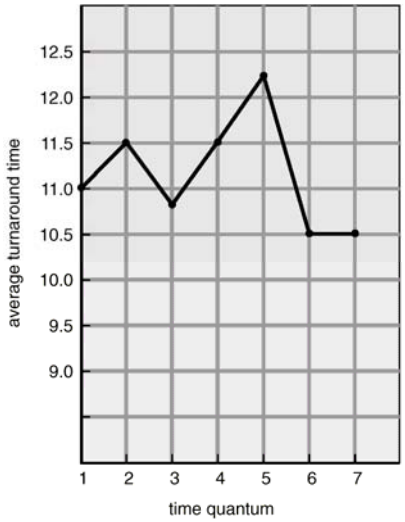
Process	Burst Time
P1	53
P2	17
P3	68
P4	24

- The Gantt chart is:



- 성능
  - SJF 보다는 평균 총 처리시간(turnaround time)이 더 크다.
  - 응답시간(response time)이 더 짧다.

Time Quantum과 Turnaround Time의 관계



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- time quantum 크기가 증가함에 따라서 평균 총처리시간이 반드시 개선되는 것은 아니다.

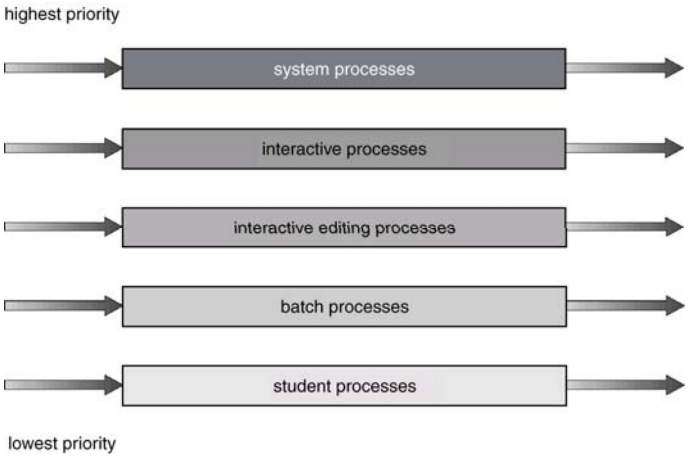
1 2 3 4 5 6 7
1 2 **3** 4 1 2 4 1 **2** 4 1 4 1 4 **1** 4 **4**

for q=1, turnaround time =
(3+9+15+17)/4 =44/4=11

다단계 큐 스케줄링

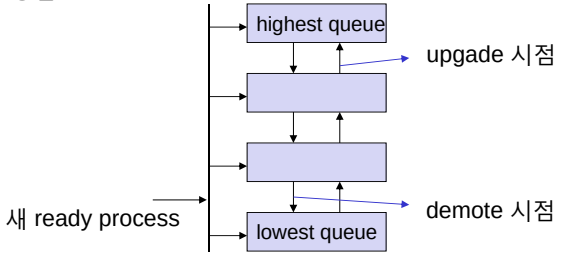
- 다단계 큐(Multilevel Queue) 스케줄링
  - Ready queue가 여러 개의 큐로 분할됨
  - 각 큐는 자신의 스케줄링 알고리즘 사용
    - (예) foreground (interactive)용 큐 – RR
      background (batch)용 큐 – FCFS
- 스케줄링은 큐들 간에도 있어야 함
  - 고정 우선순위 스케줄링
    - (예) foreground 작업을 모드 처리한 후에 background 작업을 수행
    - 기아 상태(*starvation*) 가능성
  - Time slice – 각 큐마다 CPU 사용량/비율을 정해서 할당
    - (예) foregound 큐에 80%, background 큐에 20% 할당

(예) 다단계 큐의 예

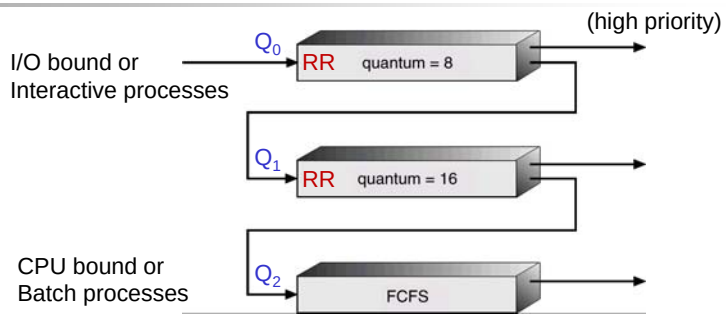


다단계 피드백(Multilevel Feedback) 큐 스케줄링

- 다단계 큐 스케줄링은 process 생성시에 하나의 queue에 영구할당
- 다단계 피드백 큐 스케줄링
  - process가 여러 큐들 사이의 이동하는 것을 허용함 ➔ aging 구현
- 다단계 피드백 큐 스케줄러의 매개변수
  - 큐 개수
  - 각 큐에 대한 스케줄링 알고리즘
  - 큐 승급/강등 시점
  - 진입할 큐 결정 방법



(예) 다단계 피드백 큐



- Example:
  - 새 작업 => queue Q_0 (q=8, FCFS)
  - 8 ms 내에 끝나지 않으면 => queue Q_1. (q=16, FCFS)
  - 추가 16ms 내에 끝나지 않으면 => queue Q_2.
  - priority: $Q_0 > Q_1 > Q_2$
    - Q_2 에 있는 프로세스는 Q_0 and Q_1 이 비어있을 때에만 수행됨
  - 작업 특성에 따라서 큐의 작업들이 자동적으로 분류됨

5.4 Thread 스케줄링

- Thread 스케줄링
  - 운영체제는 kernel-level thread들을 스케줄링
  - user-level 스케줄링은 thread library에 의해서 수행됨
- 스케줄링 경쟁 범위(contention scope)
  - **프로세스 경쟁범위(Process-contention scope:** PCS)
    - user-level thread를 가용 LWP 상에 스케줄링 (CPU스케줄링 아님)
    - 같은 process의 thread들 간에 스케줄링 경쟁
    - many-to-one 또는 many-to-many 맵핑 모델에서 사용
  - **시스템 경쟁 범위(System-contention scope:**SCS)
    - 커널이 kernel-level thread들을 CPU 스케줄링
    - system의 모든 thread들 간에 스케줄링 경쟁
    - one-to-one 맵핑 모델에서 사용
- Pthread 스케줄링 정책 – thread 생성 시에 지정 허용
  - PTHREAD_SCOPE_PROCESS: PCS scheduling (many-to-many)
  - PTHREAD_SCOPE_SYSTEM: SCS scheduling (one-to-one)

Pthread 스케줄링 API 사용 예

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *runner(void *param); /* the thread runs in this function */

int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];      /* the thread identifier */
    pthread_attr_t attr;            /* set of attributes for the thread */

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm PROCESS or SYSTEM */
    /* On Linux, Mac OS X, only PTHREAD_SCOPE_SYSTEM is supported.
     * Solaris supports both. */
    if (pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM) != 0)
        fprintf(stderr, "Unable to set scheduling scope\n");
    /* now 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);
}

void *runner(void *param)
{
    /* do some work */
    ...
    pthread_exit(0);
}
```

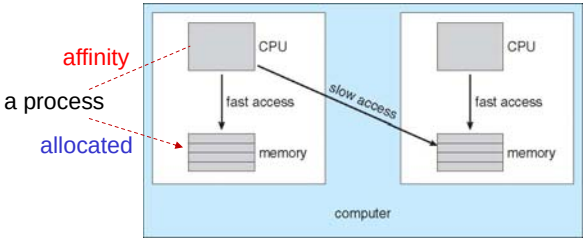
일부 시스템 에서는 특정 contention scope 값만 허용됨

5.5 다중 프로세서 스케줄링

- 다중 프로세서(Multiple-processor) 스케줄링
  - 여러 개의 CPU가 있는 경우에는 CPU 스케줄링이 더 복잡해짐
  - 여기서는 모든 프로세서가 동일한(homogeneous) 시스템을 가정함
- 다중 프로세서 스케줄링 접근 방법
  - **비대칭(Asymmetric)** 다중 처리
    - 한 프로세서(master processor)가 스케줄링, 입출력처리, 시스템 활동을 처리(운영체제 커널 수행)
    - 나머지 프로세서들은 user code만 수행함
      → 자료 공유 필요성을 배제하므로 간단한 설계
  - **대칭(Symmetric)** 다중처리(SMP)
    - 각 프로세서는 독자적으로 스케줄링(self-scheduling)
    - ready queue
      - 공동 큐 – 모든 프로세서가 함께 사용 (자료 공유 문제)
      - 분리 큐 – 프로세서마다 분리된 자신의 큐를 사용 (자료 공유 배제) → 거의 모든 현대 운영체제에서 사용

프로세서 친화성(Affinity)

- **프로세서 친화성(Processor Affinity)**
  - process가 현재 실행 중인 프로세서에서 다른 프로세서로의 이주(migration)를 피하고 다음 스케줄링에서도 현재 프로세서에서 계속 실행을 시도하는 것
  - 프로세서를 이동하면 **캐쉬 무효화**와 채우기를 해야 하므로 비용이 증가
- 프로세서 친화성 형태
  - **연성 친화성(soft affinity)** – 프로세서 지정하지 않음. 이주 가능
  - **강성 친화성(hard affinity)** – 프로세서(집합)을 지정.
  - 시스템의 형태(특히 주메모리 구조)가 프로세서 친화성에 영향을 줌
- NUMA(non-uniform memory access) 시스템과 CPU 스케줄링

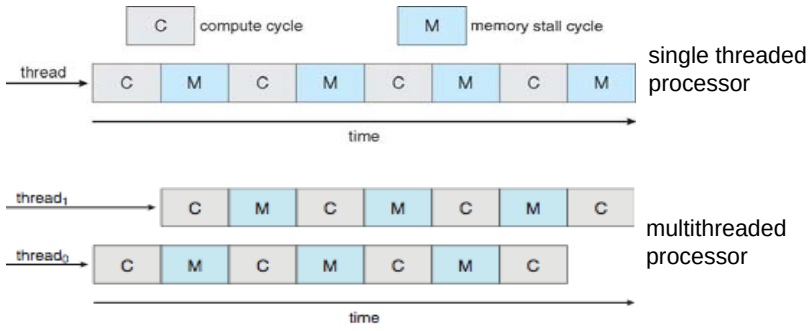


부하 균등화(Load Balancing)

- 부하 균등화(Load balancing)
  - 모든 프로세서들 간에 부하(작업)가 고르게 배분하려는 시도
- 공통 큐를 갖는 시스템에서 부하 균등화는 불필요
- 분리된 개인 큐를 갖는 시스템의 부하 균등화 방식
  - **push** migration
    - 특정 task가 주기적으로 각 프로세서의 부하를 검사
    - 과부하 프로세서가 발견되면 process들을 idle 또는 less-busy 프로세서로 이주시킴
  - **pull** migration
    - idle processor가 busy processor에서 대기 중인 process들을 자신에게로 이주시킴
  - 두 방식은 배타적이 아니며, 함께 사용할 수 있음
    - Linux는 200ms마다 push 이주 알고리즘을, ready queue가 비게 되면 pull 이주 알고리즘을 수행하여 부하 균등화 시도
- 부하 균등화는 프로세서 친화도의 장점과 상충됨

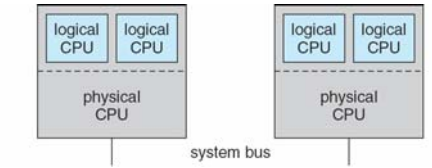
멀티쓰레드 프로세서

- 멀티쓰레드(Multithreaded) 프로세서
  - 하나의 물리적 프로세서(core)에 다수의 논리적 프로세서를 제공하는 프로세서. → 연산장치를 공유
  - 논리적 프로세서는 별도의 레지스터 집합을 가짐 → 빠른 context 전환
  - 메모리 접근 동안 연산장치를 다른 논리 프로세서가 이용함
    - 하드웨어가 논리 프로세서에게 물리 프로세서를 스케줄링



멀티 코어 프로세서

- 멀티코어(Multicore) 프로세서
  - 같은 칩에 다수의 프로세서 (multi-threaded) 코어를 보유한 프로세서



a multithreaded multicore processor

- 멀티 코어 프로세서 스케줄링
  - 운영체제 – thread에게 논리 프로세서(hardware thread)를 스케줄링
    → coarse-grained multithreading
  - CPU 하드웨어 – hardware thread에게 core를 스케줄링
    → fine-grained multithreading
  - 운영체제가 이러한 시스템에서 수행하는 것을 알고 있다면 알맞은 스케줄링 알고리즘을 사용할 수 있음 → 성능 향상

5.6 실시간 스케줄링

- 생략

5.7 운영체제 사례

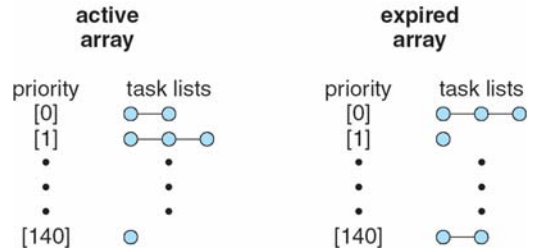
- Linux scheduling
- Windows scheduling
- Solaris scheduling

Linux 스케줄링 – 예전 버전

- kernel version 2.5 이전
  - 표준 UNIX scheduling algorithm의 변형을 사용
  - SMP systems을 위한 적절한 지원을 제공하지 않음
  - task 수가 증가함에 따라서 성능이 저하됨 – 규모 확장성 부족
- kernel version 2.5/2.6 : 상수시간에 실행. *O*(1) scheduling time
  - O(1) scheduling time - task 수와 관계없음.
  - SMP에 대한 향상된 지원 – 프로세서 친화도, 부하균등화 기능 포함
  - 선점형, 우선순위 기반 스케줄링
  - 두 가지 우선순위 영역 – 낮은 값이 높은 우선 순위
    - real-time: a real-time range (0-99) 0 : the highest priority
    - time-sharing: a **nice** value (100-140)
  - 다른 OS와 같지 않게, 우선순위가 높을 수록 큰 time quantum을 부여

Linux 스케줄링 – v2.5/2.6 (계속)

- Epoch 기반 스케줄링
  - time slice가 남아 있으면 task는 실행가능(runnable) → **active**
  - time slice가 남아있지 않으면, 다른 task가 그들의 time slice를 모두 사용할 때까지 실행가능하지 않음. → **expired**
- 모든 runnable tasks는 per-CPU **runqueue** 자료구조로 관리함
  - Two priority arrays – active, expired (그림 참조)
  - 우선 순위를 index로 사용하여 active array가 참조됨
  - active task가 없으면, active와 expired 배열을 서로 교환함
- 잘 동작하지만 interactive process의 응답시간이 느림

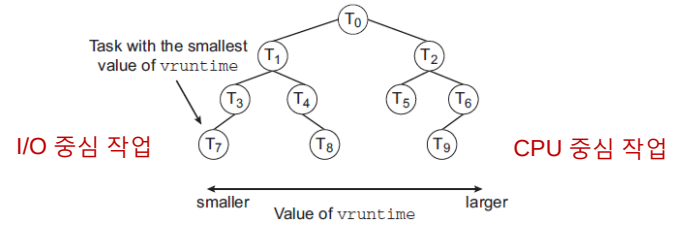


Linux 스케줄링 - Version 2.6.23+

- Completely Fair Scheduler(CFS) – 완전 공평 스케줄러
- 스케줄링 클래스
  - 각 task는 특정 우선순위 가짐
  - 스케줄러는 가장 높은 클래스에서 가장 우선순위가 높은 task 선택
  - 고정된 양의 time quantum 대신에 우선순위(nice값)에 따라서 CPU 시간 비율을 결정
  - 두 개의 스케줄링 클래스, 다른 클래스도 추가 가능
    - default – CFS 스케줄링
    - real-time
- Time quantum 계산
  - -20 부터 +19까지의 nice value를 기반으로 계산 (-20이 높은 우선순위)
  - **target latency** 계산 – 모든 수행 가능한 task가 최소 한 번 실행할 수 있는 시간 간격. CPU 시간 비율은 target latency 값으로부터 할당됨
  - target latency는 default 값과 최소값을 가지며 active task의 수가 임계값 이상으로 증가하면 target latency가 증가할 수 있음

Linux 스케줄링 - Version 2.6.23+ (계속)

- CFS 스케줄러는 task 별로 **가상 실행시간** (virtual run time)을 관리
  - 변수 vruntime 사용
  - task의 우선순위에 따라 감쇠지수(decay factor)와 정해짐
    - 낮은 우선순위가 감쇠율이 높음(> 1)
  - 보통 우선순위 작업: 가상 실행시간 = 실제 실행시간
  - 낮은 우선순위 작업: 가상 실행시간 > 실제 실행시간
  - 높은 우선순위 작업: 가상 실행시간 < 실제 실행시간
- 가장 작은 가상 실행시간을 갖는 task를 선택하여 스케줄링
  - ready 상태의 task들은 vruntime을 key 값으로 하여 이진 균형 트리인 red-black tree로 관리 → 빠른 탐색 (O(logN)연산)



Windows 스케줄링

- a priority-based, preemptive 스케줄링
- Windows 우선순위

variable class: 1-15

priority class →	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

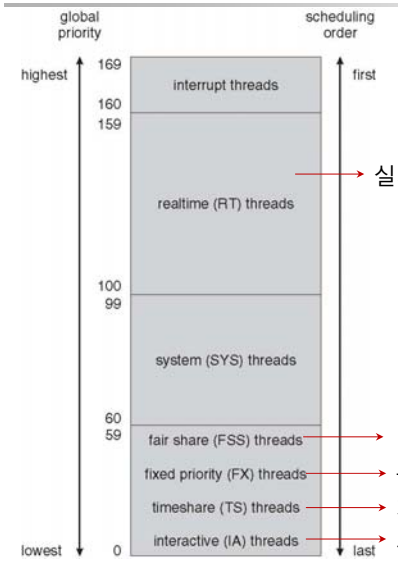
relative priority within each of the priority classes

- time quantum 만료 → 우선순위 낮아짐
- wait 해제 → 우선순위 높여줌 (wait 이유에 따라서 증가 정도가 다름)
- foreground process가 background process보다 더 높은 우선순위(3배)

Solaris 스케줄링

- 우선순위 기반 스케줄링
- 6개의 스케줄링 클래스
  1. time sharing (TS) – default scheduling class
  2. interactive (IA) – window applications
  3. real time (RT) – real-time processes
  4. system (SYS) – kernel threads (e.g. scheduler, paging daemon)
  5. fair share (FSS) } introduce with Solaris 9
  6. fixed priority (FP) }
- 각 클래스마다 다른 우선순위, 다른 스케줄링 알고리즘 존재
- Time sharing class
  - 다단계 피드백 큐 스케줄링을 사용하여 동적으로 priority 변경하고 다른 길이의 time slice를 할당
  - interactive process – higher priority, smaller time slice (response)
  - CPU-bound processes – lower priority, larger time slice (throughput)
- 전역 우선순위 – 클래스 고유의 우선순위를 전역 우선순위로 바꾸어서 스케줄링

Solaris scheduling (cont')



실시간 프로세스 – 높은 우선순위, 응답시간 보장

프로세스 집합(project라고 불림)에 할당

우선순위 대신에 **CPU 공유량**(share) 사용

우선순위 고정

기본 스케줄링 클래스

윈도우 응용 용 (더 나은 성능)

Solaris Dispatch Table

new priority

priority	time quantum	time quantum expired	return from sleep
0	200 ms	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

inverse relationship

TS와 IA class

time quantum expired:
→ priority를 낮춤

return from sleep:
→ priority를 높임

5.8 스케줄링 알고리즘 평가

- 특정 시스템을 위한 알고리즘 선택 방법
  - 알고리즘 선택 기준 정의
  - 알고리즘 평가
- 평가 방법
  - 결정론적(Deterministic) 모델:
    - 특정한 미리 정의된 부하 사용
  - Queueing 모델:
    - 부하를 확률 분포로 기술 → Queueing network의 수학적 분석
  - 시뮬레이션: 시스템 모델을 프로그래밍하는 작업 포함
    - 부하: (1) by random-number generator (2) trace tape
  - 구현(Implementation)
    - 실제 부하 사용