5장. CPU Scheduling
0. Basic Concepts
CPU Scheduling에 대한 기본적 개념
1. Scheduling Criteria
CPU Scheduling에 어떤 유형이 존재하는지?
2. Scheduling Algorithms
CPU Scheduling에 다양한 알고리즘 존재함
3. Thread Scheduling
Multi-Thread가 지원될 경우, Scheduling은 Thread 단위로 발생하므로
4. Multi-Processor Scheduling
CPU가 많은 환경, core가 많은 환경에서 Scheduling을 어떻게 하는지?
5. Real-Time CPU Scheduling
시간 제약이 있는 어떤 일을 수행할 때, 그 안에 맞추어 CPU Scheduling이 어떻게 되는지?
6. Operating Systems Examples
7. Algorithm Evaluation
여러 가지 알고리즘을 평가하는 방법에는 무엇이 있는지?

목적
-CPU Scheduling 알고리즘에 대해 알아보는 것(어떤 유형과 구체적인 알고리즘)
-Scheduling 기준에 따라 CPU Scheduling 알고리즘 평가
-CPU가 많은 환경, Core가 많은 환경에서는 어떻게 Scheduling이 일어나는가
-real-time 환경에서 어떻게 시간 제약에 맞추어 Scheduling하는지
-현존하는 OS에서 어떤 Scheduling 알고리즘을 지원하는가
-CPU Scheduling 평가에 대해 알아보는 것

0. Basic Concepts
프로세스가 수행되는 과정
프로그램이 수행될 때는 CPU를 주로 이용하는 작업을 할 수 있다. -> CPU burst

여기서 파일을 읽게 되면 file은 하드디스크(stable storage)에 주로 있기 때문에 I/O에서 파일을 읽어오는데 시간이 걸림(CPU는 빠르나 I/O는 상대적으로 훨씬 느려 오랜 시간 기다림)
I/O 일이 끝나 INT가 오면, 읽어온 데이터를 처리하고 그 밖의 (file write하면)CPU burst 시작되므로
대부분 작업들은 CPU일도 있지만, I/O를 기다리는 일 번갈아가며 작업 수행함

Prog은 CPU burst - I/O burst를 번갈아 가며 수행된다.

CPU Scheduling 개념
-CPU burst(CPU 할당 필요한 시점)에 있는 proc들에게 어떠한 순서로 CPU를 할당할 것인가?를 결정하는 것

CPU Scheduling의 목적
-CPU의 활용성(총 수행 시간에서 CPU가 얼마나 사용되느냐)을 높이는 것
이를 최대로 가져가기 위해 앞서 Multiprogramming 배움

Multiprogramming : 여러 proc가 동시에 메모리에 loading되는 것
A, B, C를 번갈아가며 수행할 때에 메모리에 이미 준비되어 있는 proc를 수행

CPU burst
-CPU 안에서 instruction을 수행하거나 Memory와 CPU 간의 데이터 이동
-CPU burst 시간은 CPU를 연속적으로 사용하는 시간의 길이
-CPU burst의 시간이 긴 것은 별로 없음을 그래프에서 확인 가능
(대부분의 prog은 짧은 시간의 CPU 시간 할당받아 자주 사용하는 형식)

CPU Scheduler
-CPU를 필요로 하는 proc(ready queue 내 존재)에게 CPU 사용권을 넘기는 것

CPU Scheduling이 일어나는 과정
a. Switches from running to waiting state ex. I/O wait
(CPU를 사용하던 proc가 CPU가 필요없는 시점)
b. Switches from running to ready state ex. timesharing 시스템 내 CPU 사용권 이동
(CPU를 사용하던 proc가 CPU 필요하지만 다른 proc를 위해 독점하지 않고 넘겨주는 것)
c. Switches from waiting to ready
(CPU를 사용하던 proc가 CPU 필요하지만 다른 proc를 위해 독점하지 않고 넘겨주는 것)
d. Terminates (CPU를 사용하던 proc가 CPU가 필요없는 시점)

a, d - nonpreeemptive Scheduling (비선취) : 어느 proc가 CPU 필요없는 순간에만 Scheduling할 수 있는 것 - 강제로 빼앗지 않는 것

b, c 등 다른 모든 것 preeemptive Scheduling (선취) : 돌던 proc를 독점 방지하고자 강제로 빼앗는 것
preeemptive가 우선순위를 명확히 따지기 때문에 성능이 더 좋음(기술적 어려움 존재)
-shared data 이용할 때
-어느 proc가 커널 모드에 있을 때
-민감한 OS 활동 중 CPU 사용권 뺏는 것의 위험성

Dispatcher
-Scheduling에 의해 proc 수행이 결정되었을 때, proc 간 교체를 해주는 동작
-switching context, switching to usr mode, usr prog 재시작을 위한 적절한 위치로의 jumping의 의미를 포함하고 있음
-P0의 모든 상태를 PCB에 잘 저장하고 P1의 PCB의 내용을 CPU에 잘 끌어와 P1 수행
이로 인해 Dispatch latency 발생(하나의 proc 멈추고, 한 proc 실행해야 하는 동작에서)

1. Scheduling Criteria
CPU Scheduling을 어떻게 해야 잘 하는 것인가?
Scheduling 성능 평가 기준(5가지 정도)
-CPU utilization
-Throughput : 단위 시간 내 수행 작업량 or 단일 작업 평균 시간 소모량
-Turnaround time : proc가 끝날 때까지 걸리는 평균 시간
-Waiting time : ready queue에 머무르는 총 시간
-Response time : proc가 시작되고 처음으로 CPU를 할당받을 때까지 걸린 시간

Scheduling Algorithm Optimization Criteria
@사용률 및 처리율 높임
-Max CPU utilization
-Max throughput
@시간 줄임
-Min turnaround time
-Min waiting time
-Min response time

2. Scheduling Algorithms
(1) FCFS(First- Come, First-Served) Scheduling (nonpreeemptive함)
-온 순서대로 서비스 받는 것(공평) ex. 식당, 은행 갔을 때
-Scheduling 구현이 쉬움
-Convoy effect : 긴 proc 뒤에 짧은 proc가 오면서 오래 대기하는 것
수행 시간이 짧은 proc가 먼저가서 수행할 경우에는 FCFS 좋음

※Gantt Chart : 시간(길이로 표현됨)과 해야할 일을 간단히 표시하는 차트

P1, P2, P3 예시 10p, 11p

(2) SJF(Shortest-Job-First) Scheduling (nonpreeemptive함)
-수행 시간이 짧은 proc를 먼저 수행하는 것
-Average waiting time 최소화 가능(time을 성능 평가 기준으로 보았을 때 최적임)
-어떤 작업이 있을 때, 이 작업의 길이(CPU burst의 length)가 짧은지 긴지 판단하기 어려움

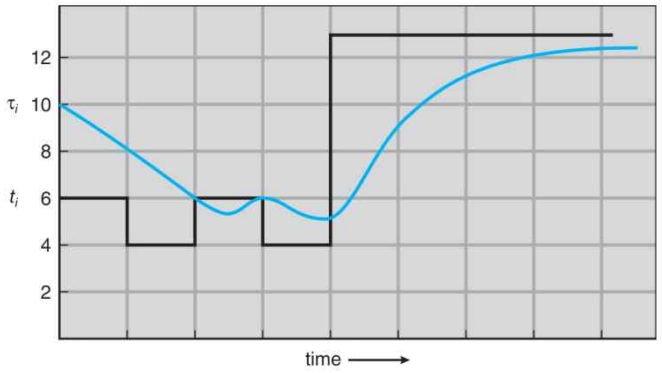
※Determining Length of Next CPU Burst
과거의 CPU burst를 근거로 평균(exponential averaging)을 내 그 길이를 예측할 수 있다
최신의 것에 비중을 더 두어 평균 산출함(1/2, 1/4, 1/8 등)
예측 과정

a. tn = actual length of n th CPU burst

b. $T_{n+1} = predicted\ value$ for 다음 $CPU\ burst$

c. $\alpha, 0 \leq \alpha \leq 1$ (알파로 비중 조정)

d. Define: $T_{n+1} = \alpha t_n + (1-\alpha) T_n$ (실측과 예측한 값의 평균 내서 다음 예측값 산출)



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

그림 1. Prediction of the Length of the Next CPU Burst

i) a = 0 //과거와 미래에 똑같은 비중을 둔다

ii) a = 1 //바로 직전 것만 보고서 미래를 예측한다

공식 확장

If we expand the formula, we get:

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

Since both α and $(1-\alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Shortest-remaining-time-first 예시 (기존 nonpreeemptive->preeemptive 함)
가장 짧은 proc가 와서 수행 중에, 남은 일보다 짧은 proc가 오면 그에게 CPU 사용권 넘김

(3) Round Robin (RR)
-CPU 시간을 정하여 그 짧은 CPU 시간 동안 서로 돌아가며 proc에게 CPU 할당하는 것
-time quantum(q) : 얼마나 많은 시간을 proc에게 할당할지 결정한 것 (적정:10-100msec)
따라서, n개의 proc있을 때, maximum wait time은 (n-1)q이다.
-중요점 : queue의 시간을 어떻게 가져가느냐에 따라
a. q large => FIFO
b. q small => q must be large with respect to context switch(이것이 너무 많아짐),

otherwise overhead is too high

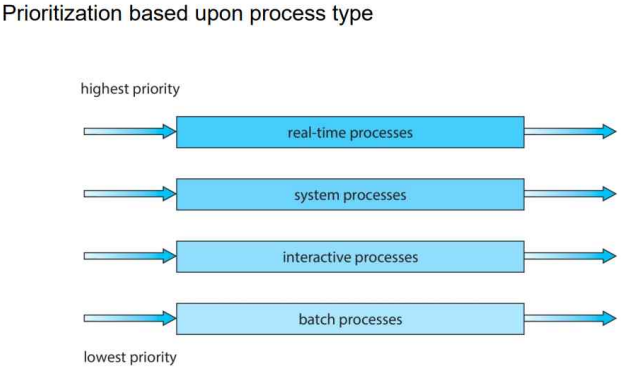
#Time Quantum and Context Switch Time
-time quantum을 적당히 조절하는 것이 중요함
-일반적 context switch overhead < 10μsec 이내(overhead 거의 무시할 수 있는 수준)
-time quantum이 너무 작아 context switch overhead로 작용하여 좋지 않음

#Turnaround Time
-time quantum이 변해감에 따라 Turnaround time(각 작업이 도착해서부터 끝날 때까지 걸리는 시간)을 평균을 낸 것
-RR(time quantum)은 CPU 시간을 골고루 부여하므로, Turnaround time를 줄이기 힘듦
-RR(time quantum)은 response time(응답시간)은 줄일 수 있음
-time quantum에 따라 average Turnaround time이 들쭉날쭉함

Priority Scheduling
-우선순위 높은 것을 먼저 수행하는 것
(FCFS-먼저 온 것, RR-오래 기다린 것 등 앞선 3가지 알고리즘이 해당됨)
-preemptive, nonpreemptive 구별 필요
-어떤 정책에 의해서 우선순위를 높이는 방법이 다름
-Starvation 문제 : 우선순위 낮은 proc가 CPU 할당 못받고 오랫동안 굶주릴 수 있음
-해결책 Aging : CPU 할당 못 받음에 따라 CPU 사용권에 대해 우선순위를 높여줌

Priority Scheduling w/ Round-Robin
-priority Scheduling을 하되, 우선순위가 동일한 proc 존재 시에 어떻게 해결하는가?
이에 대해 RR 방식으로 주어진 time quantum 내에 번갈아가며 수행

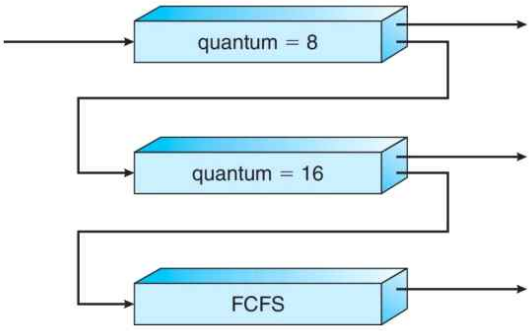
Multilevel Queue
-proc를 그룹으로 묶어서 우선순위 level을 만들어 놓고, 가장 높은 level의 group을 먼저 수행하며 해당 level 내 모든 proc 완료 시 다음 level 수행하는 것
-실제 우선순위 예시 : real-time proc > system proc > interactive proc > batch proc



Multilevel Feedback(demote 가능성 존재하므로) Queue

-queue에서 수행하다가 조건이 되면 다음 queue로 넘어가는 것
-여러 가지 parameter 값을 지정 가능
a. number of queues
b. scheduling algorithms for each queue
#어떤 조건, 순간에 upgrade되고 demote 되는지
c. method used to determine when to upgrade(위queue<-아래queue) a process
d. method used to determine when to demote(위queue->아래queue) a process
e. method used to determine which queue a process will enter
-여러 알고리즘의 장점을 융합시킨 알고리즘 형태

Example of Multilevel Feedback Queue



※조건
-Three queue 존재
Q0 : RR with time quantum 8 milliseconds
Q1 : RR time quantum 16 milliseconds
Q2 : FCFS (우선순위 낮은 것)
-> SJF는 아니지만, 짧은 proc가 CPU 우선권을 가지도록 만듦
(짧은 proc에 비교적 우선 순위가 높게 유지되도록 함)

3. Thread Scheduling
-user-level과 kernel-level threads 구분 필요
-U, K 사이 Light Weight Process 존재(커널 마다 하나)하여 U 동작하기 위해 LWP 할당받아야 하고, K 동작 위해서는 CPU 할당받아야 함.
-process-contention scope (PCS)
U thread를 어떻게 Scheduling해서 LWP 할당할 것인가를 결정하는 것
(한 proc 내 여러 thread 존재 시, 어떤 thread에게 LWP 할당할지?, 프로세스 내 경쟁)
-system-contention scope (SCS)
K thread를 어떻게 Scheduling해서 CPU 할당할 것인가를 결정하는 것 (시스템 내 경쟁)

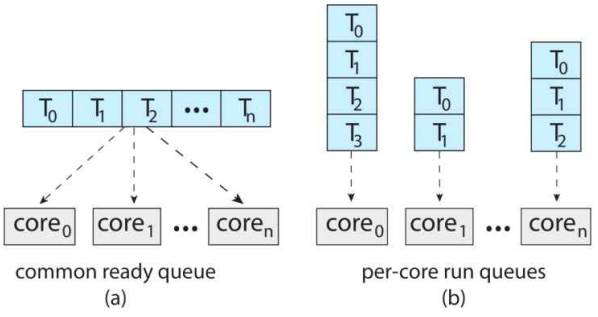
Pthread Scheduling
-PTHREAD_SCOPE_PROCESS(프로세스 스코프)
한 프로세스 내 어느 U Thread를 Scheduling할 것인가?
-PTHREAD_SCOPE_SYSTEM(시스템 스코프)
전 시스템 내 어느 K Thread를 Scheduling할 것인가?
-OS마다 어떤 Scope가 지원되는지 다름
Linux and macOS 오직 PTHREAD_SCOPE_SYSTEM만 지원함

Pthread Scheduling API
-Scheduling하는 Scope가 API로 제공됨
pthread_attr_init(&attr); //pthread 초기화
//적용되는 scheduling이 프로세스 스코프인지 시스템 스코프인지 확인
pthread_attr_getscope(&attr, &scope)
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); //스코프 설정하는 것
pthread_create(&tid[i],&attr, runner, NULL); //thread 생성
pthread_join(tid[i], NULL); //생성한 thread가 runner 실행 후에 종료되기를 기다리는 것

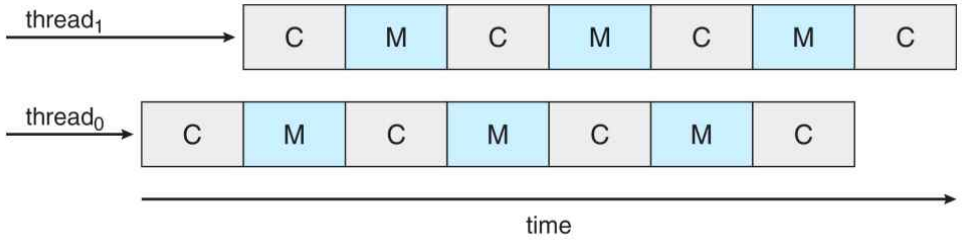
4. Multi-Processor Scheduling

Multi-Processor : CPU가 많다는 것
Multicore CPUs : 한 CPU 내 존재하는 core가 많다는 것
Multithreaded cores : 한 core 내 H/W thread를 동시에 보관하기 위한 요소 있는 것
(Multithread가 동시에 수행할 수 있는 환경 제공함, H/W적으로 parallel한 수행 가능 환경)
NUMA(Non Uniform memory access) systems : 메모리 접근시, 접근 시간 동일 X
Heterogeneous multiprocessing : 한 컴퓨터 내 다른 역할 수행하는 CPU 많이 존재

Symmetric multiprocessing (SMP)
-하는 역할이 같은(symmetric) CPU가 많이 존재한다는 것
a. common ready queue (일반적)
하나의 common ready queue(CPU 할당 기다리는) 내 존재하고 이 queue에서 하나씩 작업을 가져와서 여러 개의 core가 수행할 수 있는 것
b. per-core run queues (Multicomputer 환경)
대기 행렬이 CPU core마다 존재함



Multicore Processors
-최근 여러 core를 한 CPU 칩 내 위치시키는 것이 유행
-장점 : 빠르고 전력 소비가 적음
-쓰레드가 하나 존재한다면, 쓰레드는 computation 일과 memory access 일도 해야 함
-But, H/W적인 쓰레드가 여러 개 존재하면 computation, memory access 일을 교차하면서 실행시켜, CPU 사용률 높임(동시에 여러 쓰레드 수행되는 효과)



Multithreaded Multicore System
-Chip-multithreading (CMT) : CPU core 내 H/W thread 여러 개 존재하는 것
S/W thread가 H/W thread를 하나씩 차지하므로, 한 core 내 두 thread 동시 작업 가능
-Two levels of scheduling : 기존 (S/W -> H/W Thread로 Scheduling) level 하나에서 (H/W thread -> core Scheduling) level 하나 더 늘어났다고 생각하면 편함
*Usr -> Kernel thread 할당은 S/W thread 내 이야기(매핑 관계)

※H/W thread vs. S/W thread
-H/W 쓰레드는 os가 Scheduling 해줄 수 있는 최소 단위의 일(=명령어의 모음)을 말함.
1코어는 1 쓰레드를 지원하나 최근에는 2 쓰레드 이상 지원하는 경우가 흔함.

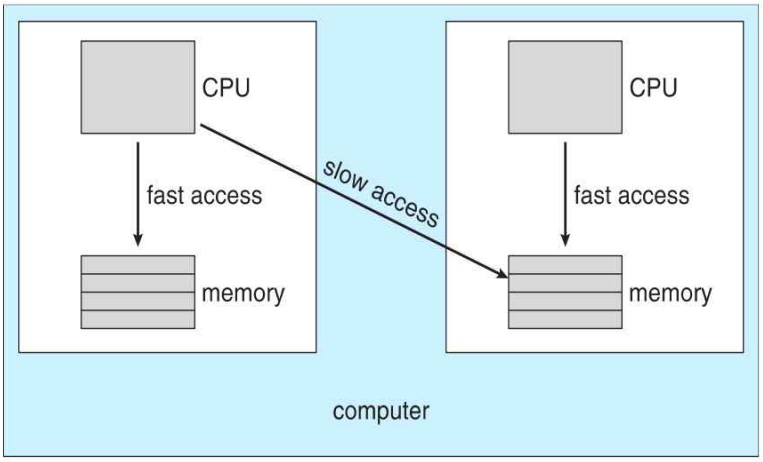
-S/W 쓰레드는 소프트웨어 상에서 병렬적으로 task를 나누고, 일을 할당할 때 쓰임.
이 쓰레드가 그대로 하드웨어 스레드에 올라가 사용되며, 따라서 소프트웨어 스레드가 100개 존재하여도 동시에 실행될 수 있는 쓰레드는 하드웨어 쓰레드 개수와 같음.
ex. 4코어 8 (H/W)쓰레드라는 것 = 상, 하권이 나뉜 4세트의 책과 같다(총 8책). 이 4세트를 가지고 도서관에서 100명의 사람에게 빌려줄 수 있으나, 한 번에 읽을 수 있는 사람은 8사람(S/W 쓰레드) 밖에 안 된다.

Multiple-Processor Scheduling - Load Balancing
-SMP 환경에서, 모든 CPU들은 효율적으로 load 수행해야 함
Load balancing 필요 : CPU 마다 load를 적절히 가지고 있어야 함
#방법 : 하나의 CPU와 다른 CPU의 작업 load량이 너무 차이나면 작업들이 이주함(많 -> 적)
a. Push migration : 작업 많은 CPU에서 일 없는 CPU 찾아 작업을 나눠주는 것
b. Pull migration : 할 일 없어서 작업 많은 CPU 찾아 작업을 달라고 요청하는 것

Multiple-Processor Scheduling - Processor Affinity
CPU가 여러 개 있는 환경에서 어떤 작업이 어떤 CPU에서 수행될 것인지를 결정할 때, 친밀도 or 호감도(이득을 챙길 수 있는) 높은 CPU에서 작업하도록 함

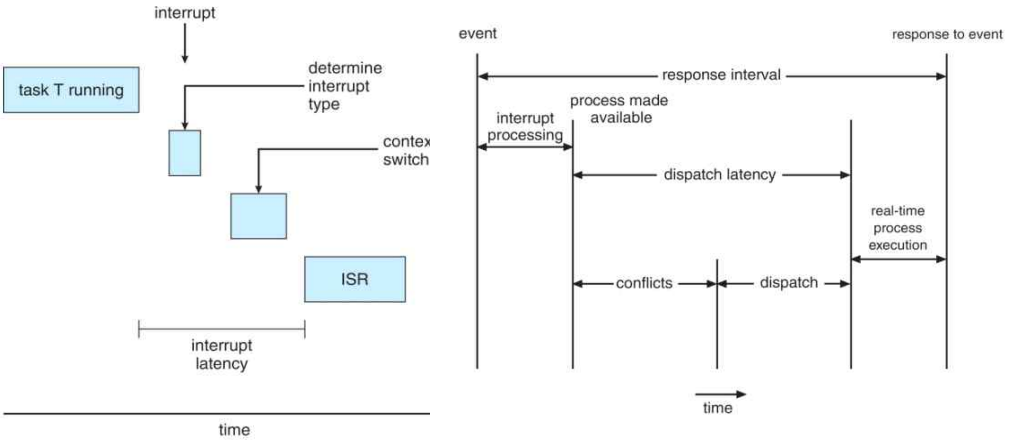
-processor affinity : 이전에 thread가 CPU에서 작동 시, 그 CPU의 cache에 data가 남아 있어(cache contents, 그 thread의 메모리 접근 저장), thread 실행 시 다시 그곳에서 동작
#친밀도의 2가지 종류
-Soft affinity : Cache의 역할을 향상키기고자 지정할 수 있으며, 보증없이 같은 CPU내 동작하도록 시도함
-Hard affinity : CPU의 집합 내 특정 CPU에서 process를 실행할 수 있도록 함
(자기 작업에 적합한 CPU 사용자가 직접 선택함)
*Load balancing도 processor affinity에 영향 줄 수 있으며, 이 이동으로 인해 CPU는 cache 정보 잃게 됨

NUMA and CPU Scheduling



-local 내 memory에는 빨리 접근할 수 있지만, remote 환경의 memory에는 접근 느림
-이런 NUMA 구조를 알고 Scheduling 수행하는 것이 좋음

5. Real-Time CPU Scheduling
-빠른 수행과 실시간 수행은 차이가 있음
-실시간 시스템은 시간 제약(Deadline)이 있어, 그 제약 내에서만 작업 수행한다면, 성능이 동일하게 높다고 판단함
#real-time 종류
a.Soft real-time systems : Deadline을 지나쳤을 시, 성능은 급격히 떨어지지만 쓸 만함
b.Hard real-time systems : Deadline을 지나쳤을 시, 성능이 없어지거나 위험함
-Event latency : 이벤트(INT, Dispatch 등) 발생 후, 서비스가 제공될 때까지 경과된 시간
@latency의 두 가지 종류
a. Interrupt latency
INT 왔을 때, INT type을 확인하고, context switch 후, ISR 시작 전까지 걸리는 시간
b. Dispatch latency
현재 CPU 내 process 종료 후, 다른 것과 switch시 걸리는 시간(confilct, dispatch로 구성)



※Dispatch latency의 Conflict [이 지연 시간 고려할 필요 있다는 의미]

a. 커널 모드에서 실행 중인 모든 프로세스를 선점하는 경우

b. 우선순위 높은 프로세스에 필요한 리소스를 우선순위 낮은 프로세스에 의해 해제된 경우 (low level proc가 access하고 있는 자원을 high level proc가 기다려야 할 경우도 있음)

Priority-based Scheduling

-실시간 Scheduling에서 사용하는 방식

-우선순위를 정하는 방법이 일반 CPU Scheduling과 차이 존재

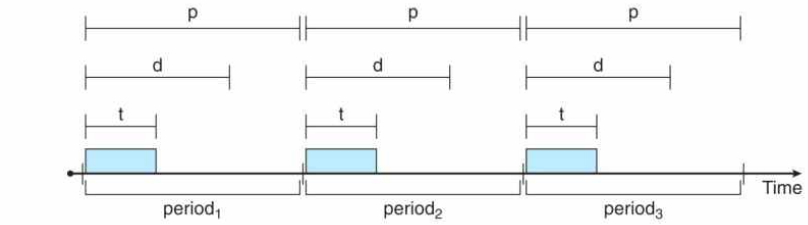
-real-time 시스템은 비주기적으로 발생하는 일도 존재하지만, 주기적으로 해야 할 일도 있음

-p(주기), t(CPU burst time), d(deadline), 일반적 p와 d는 동일한 경우 많음($0 \le t \le d \le p$)

Has processing time *t*, deadline *d*, period *p*

$0 \le t \le d \le p$

Rate of periodic task is $1/p$



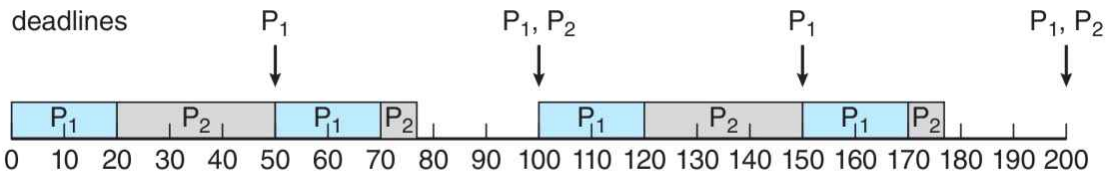
#주기적인 작업 처리할 때, Scheduling하는 방법

(1) Rate Montonic Scheduling

-주기가 짧으면, 우선순위를 높이는 scheduling 방법(=period 역순으로 우선순위 높임)

Shorter periods = higher priority, Longer periods = lower priority

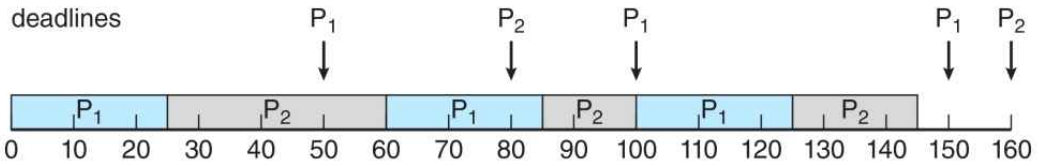
-simple하면서도 주기에 따라 고정적으로 수행하여, Real-Time Scheduling에 적합



P1 주기: 50, P2 주기: 100이라 P1 먼저 Scheduling하고 남은 시간에 P2 수행함

-어떤 Deadline 맞출 수 있는 Scheduling이 존재해도, R.M 스케줄링이 실패하는 경우 있음
>> 주기 짧은 작업에 우선순위 주다 보니, 그것을 우선하느라 다른 작업의 주기를 맞추지 못하고 Scheduling되는 경우

(2) Earliest Deadline First Scheduling (EDF) - preemptive
-Deadline을 기준으로, Deadline이 임박한 것의 우선순위를 높이는 Scheduling 방법
ex. 과제 제출 오후 6시 것을, 오후 11시 마감 작업보다 먼저 처리하는 것



-앞서 Rate Montonic Scheduling이 missing하는 Scheduling 해결 가능
-Deadline을 모두 만족시킬 수 있기 때문에, Optimal하다고 함(real-time중 제일 좋음)

(3) Proportional Share Scheduling
-CPU 시간을 가상으로 쪼개, 그 부분에 대해 proc마다 시간 할당하는 Scheduling 방법
-CPU 성능의 일정한 Portion을 어떤 Proc에게 고정으로 할당해주는 것
ex. 100의 성능을 가진 컴퓨터에 10씩 쪼개어 N개에 분배해주면, 한 AP 입장에서 10의 성능을 가진 컴퓨터가 나를 위해 독점적으로 사용할 수 있다는 것(T: 10, N: 100)

POSIX Real-Time Scheduling
보통 OS들은 Real-time 작업에 제일 높은 우선순위 부여하고 같은 우선순위일 경우 방식 선택 가능
a. SCHED_FIFO : FCFS 전략을 사용하여 쓰레드가 Scheduling됨
b. SCHED_RR : 동일한 우선순위의 쓰레드를 위한 time-slicing이 발생함
(이것 제외하고는 SCHED_FIFO와 유사)

Scheduling 정책을 얻거나 설정하기 위한 두 가지 함수(앞선 보았던 것들과 유사한 API)
a. pthread_attr_**getsched_policy**(pthread_attr_t *attr, int *policy)
OS에서 디폴트로 어떤 Scheduling policy를 사용하고 있는지 알려줌
b. pthread_attr_**setsched_policy**(pthread_attr_t *attr, int policy)
이것으로 FIFO, RR 중 프로그래머가 원하는 Scheduling policy 설정 가능함

POSIX Real-Time Scheduling API(관련 과제 없을 시 삭제)

```
#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");
}
```

```
POSIX Real-Time Scheduling API (Cont.)
/* 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);
}
```

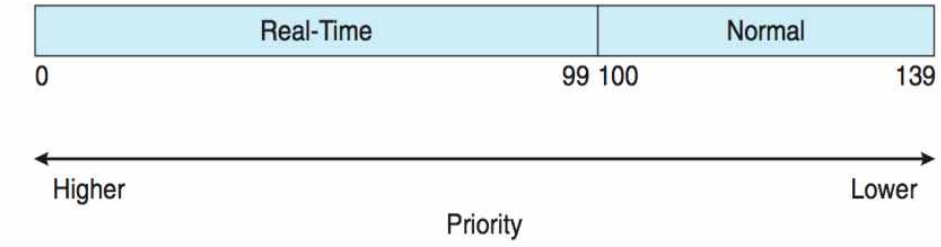
6. Operating Systems Examples

(1) Linux scheduling (2.5 버전)
-Preemptive, priority based scheduling 제공
-time-sharing과 real-time이라는 두 가지 우선순위 범위(range) 부여]
ex. Real-time range from 0 to 99
-우선순위 감소시키는 nice값 존재(상대적 우선순위 조정) ex. nice value from 100 to 140
-priority 작은 값이 우선순위가 높음을 의미함
-우선순위 높은 것에 대해서 timequantum(q)을 더 많이 줌
-Task 상태로는 active(할당받은 시간 남음)와 expired(할당받은 시간 전부 소모) 존재

모든 Task expired 시, expired set 내 다시 CPU 시간 할당하여 active 상태로 동작시킴
-모든 동작 가능한 Task들은 CPU별 runqueue 데이터 구조에서 추적함
-response time 관점에서 우선순위 낮은 것도 기다리면 기회가 온다는 장점이 있지만, interactive한 proc가 너무 오래 기다리게 된다는 단점 존재

(1)-2 버전 2.6.23 이후
-Completely Fair Scheduler (CFS)
Scheduler는 높은 Scheduling class(2가지 class)에 있는 높은 우선순위 task를 선택함
(real-time의 경우는 높은 class에 속하며 선택받는 것에 유리함)
@Scheduling classes
real-time이나 system에 대해서는 우선순위 등급을 높게 부여하고, 일반 따라 우선순위 등급이 존재 [default (일반적인 prog), real-time]
@Time Quantum
-고정된 시간 할당이 아니라, CPU 시간의 비율에 기반하고 있음
-nice 값(-20~+19)에 기반하여 계산됨(낮은 값이 높은 우선순위를 가짐)
-Target Latency : 모든 task는 적어도 한번은 수행할 수 있는 시간 간격(공평성 제공)
만약 active task가 증가하면 이 시간 간격은 증가할 수 있음
-virtual run-time (변수 vruntime) : 가상 실행 시간
얼마나 CPU 시간 할당받았는지를 결정함
자신이 CPU 시간을 많이 할당받았으면 vruntime이 크고 우선순위가 떨어지며, vruntime이 작다는 것은 그동안 CPU 시간을 많이 할당받지 못했다는 것(우선순위↑)
vruntime 또한 동일하게 적용하는 것 X라, decay factor를 두어 가중치 다르게 적용
<동일 CPU 시간 할당받더라도 우선순위가 높은 것(decay factor 작음)은 vruntime이 조금씩 올라가고, 우선순위가 낮은 것(decay factor 큼)은 vruntime을 많이 높임>
▶proc 성격마다 다른 우선순위 (level) class을 부여하지만, 그 level 내에서는 공평성 강조

#Linux Scheduling 예시 (Real-time 영역, Normal 영역 존재)
-Real-time tasks은 static priorities(0~99) 가지고 있음(Normal 영역은 뒤에 덧붙여 존재)
-Nice 값 -20은 전역 우선순위 100에 해당되고, Nice 값 +19 전역 우선순위 139에 해당함



-Scheduling domain : 서로 균형 이룰 수 있는 CPU core의 집합 단위
하나의 CPU 내 core가 여러 개 존재하는 domain이라는 구역으로 나누어, domain 내 존재하는 core들이 빨리 access할 수 있는 메모리(cache 공간) 주는 것
목적 : 가끔 발생하는 일에 대한 cache 효과 극대화

(2) Windows scheduling
-priority-based이며, preemptive scheduling 적용함
-우선순위 높은 thread(or task) 먼저 수행됨
-스케줄러로 Dispatcher 사용
-Thread은 아래 3가지 case 발생 전까지 동작할 수 있음
a. blocks b. uses time slice c. preempted by higher-priority thread
-Real-time threads는 non-real-time을 뺏을 수 있음
-32 단계 priority 체계(Variable class:1~15, real-time class: 16-31)
-우선순위 0은 memory-management 쓰레드
(각 우선 순위에 대한 queue 존재하며, 실행 가능한 쓰레드 없다면 idle 쓰레드 실행)
-Priority class 내에서 상대적 우선순위(class 내에서 조정 가능한 상대적 값) 존재
(REALTIME을 제외하고 모두 가변적)
-Priority class + 상대 우선 순위 => 숫자 우선순위 부여하기도 함
-Windows에서는 wait 상태에서 ready로 들어간 것은 우선순위를 높일 수 있음
(Foreground는 background에 비해 우선순위를 3배까지 높일 수 있음)

	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

그림 13. Windows Priorities(6가지 class 존재)

*user-mode scheduling (UMS)
Applications create and manage threads independent of kernel
For large number of threads, much more efficient
UMS schedulers come from programming language libraries like
C++ Concurrent Runtime (ConcRT) framewor

(3) Solaris(Unix의 일종) scheduling
-Priority-based scheduling
-Class 6개 존재 [Time sharing(default) (TS), Interactive (IA), Real time (RT), System (SYS), Fair Share (FSS), Fixed priority (FP)]
-주어진 쓰레드는 한 번에 하나의 클래스에만 존재 가능하며, 각 class마다 own scheduling algorithm 가지고 있음
-여기서는 priority값이 크면, 우선순위가 높음
-return from sleep : 그동안 sleep 상태로 있어 CPU 할당 못받아 우선순위 높여줌
-Class 마다 우선순위 band가 존재하며, 다른 OS와 유사하게 쓰레드 동작시켜줌
a. blocks b. uses time slice c. preempted by higher-priority thread

priority	time quantum	time quantum expired	return from sleep
0	200	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

그림 14. Solaris Dispatch Table

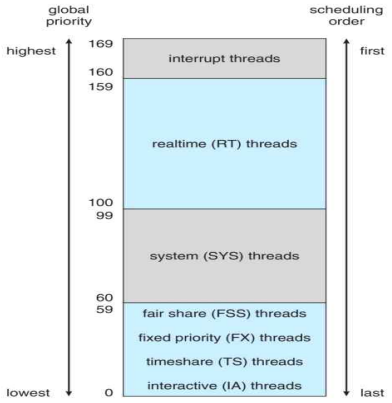


그림 15. Solaris Scheduling

7. Algorithm Evaluation

OS마다 detail한 OS 알고리즘이 달라 성능 측정 및 평가해야 할 필요성 O

#성능 측정 및 평가 방법

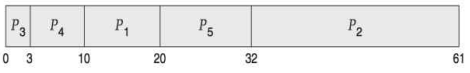
(1) Deterministic modeling(Deterministic Evaluation)

정해진 입력값(수치)을 가지고 성능 평가를 실제로 진행해보는 것

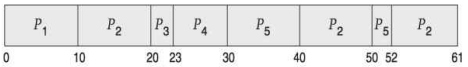
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

FCS is 28ms:

Non-preemptive SFJ is 13ms:



RR is 23ms:



(2) Queueing Models(서비스를 받는 Queue(=ready queue), Server(=CPU) 존재)

어떤 확률 분포적인 것을 가지고 다양한 성능 평가를 계산하는 것(여러 이론 존재)

Little's Formula (n, W, λ 각각 모두 다른 분포에 속할 수 있으며, 계산 가능)

-n = average queue length

-W = average waiting time in queue

-λ = average arrival rate into queue

-Little's law : 정상 상태에서 [대기열 떠나는 프로세스 == 도착하는 프로세스] 라는 법칙

도출 => n = λ x W (두 개만 알면 나머지 하나 구할 수 있음)

(3) Simulations

실제 구현하는 것은 아니지만, 모의 실험을 수행해보는 것

-작업의 성격이 어떤 지, 언제 도착하고 얼마나 수행 시간이 필요한지 등의 변수에 대해서 난수 값나 수학적인 분포, 실제 시스템을 모니터링하여 얻은 값을 대입

(4) Implementation

직접 구현하여 수행해보는 것(시간 및 비용이 많이 드나, 정확한 측정 가능)