**운영체제 정리본**

2015253039 권진우

5장 세마포부터

**세마포** : 뮤텍스 락의 개념을 포함한 특수한 자료구조

- Mutex Lock보다 정교하고 강력한 **프로세스 동기화(실행시간을 맞추는것)도구**

- 세마포 S

- 특별한 **표준 동기화 연산**을 통해서만 접근 가능한 정수 변수

- **세마포 값 = 사용 가능한 자원 수**(1,2,3,4,….)

Semaphore 연산

- Semaphore값 **초기화(최대 가용자원 갯수) EX.자원이 5개면 세마포 초기값 = 5, 프로세스가 사용시마다 1개씩 감소**

(Lock의 경우 1개가 돌면서 진행 되지만 세마포는 여러 개의 락으로 생각)

- 두 개의 원자적(**atomic, 끊어지지 않게 한번에 연산**) 연산

- **P(test)** = Wait, Acquire(대기, 진입)

- **V(increment)** = Signal,Release(반납)

**P(S)** **<= 세마포 자원 사용 요청**

While (S <= 0)

; // busy wait(자원이 없을 때 루프돌며 대기)

// now S >0(세마포 자원이 있을 때)

S=S-1; (세마포 값 **감소**, 프로세스가 **자원 사용**)

**V(S)** **<= 세마포 자원 사용 후 반납**

S=S+1;(세마포 값 *증가, 자원 반납*)

**\* 각 각 함수(S=S+1,S=S-1)들은 원자적으로 사용되어야 한다.**

세마포의 용도

- **상호배제 구현** –> 2진 세마포(세마포 값을 1로 초기화) = **Binary Semaphore**

- **유한 개수 자원** 접근, 한정된 동시진행(concurrency) -> **Counting Semaphore**

**-** 프로세스 동기화 : Signaling

Binary Semaphore**(2진 세마포)**

- **상호배제를 구현하기** 위해 사용(Mutex Lock과 유사)

- 세마포 값을 ‘1’로 둔다. 🡺 가능한 세마포 값(자원갯수) : 0 or 1

- 임계구역 접근 제어에 사용된다.

Semaphore S = 1;(초기화)

프로세스1 프로세스2

P(S); P(S);

CS구역 CS구역

V(S); V(S);

\***세마포는 1일 때 Unlock, 0일 때 Lock**(남은 자원의 개수를 의미하므로)

\**Mutex락은 0일 때 Unlock, 1일 때 Lock*(**세마포랑 반대**)

카운팅 세마포

- **한정된 동시성 제공가능** = **유한 개의 자원 제공**(세마포 값 1,2,3,4,5,…)

- **Semaphore값 : 현재 사용가능한 자원 개수 의미**

**Semaphore S = n;(초기화, 최대 N개 프로세스가 병행실행 가능)**

**프로세스1 프로세스2 프로세스3 ….**

P(S); P(S); P(S);….

CS구역 CS구역 CS구역….

V(S); V(S); V(S);….

\*여러 개의 프로세스가 각각 CS구역에 동시에 따로 진입하여 수행**(동시에 진입 가능한 프로세스 개수가 한정(초기화)되기 때문에 각각 프로세스별로 CS구역 진입 가능)**

일반적인 동기화

- EX. 프로세스1의 A부분 실행 후 프로세스2의 B부분 실행(순서에 맞춰서)

코드

Semaphore S = 0;(0으로 초기화)

A (실행)

V(S); 🡺 P(S,wait) <S가 0이었다가 V(S)로 반납할 때 세마포값 증가 🡺 P(S)가 세마포 자원 받아서 실행>

B (실행)

\*프로세스2는 프로세스1의 V(S)실행까지 기다렸다가 실행을 구현

\*V()는 자원반납이므로 S=S+1, P()는 자원을 사용하므로 S=S-1;

세마포 구현

- Busy Waiting 세마포(자원이 여러개) = Spin Lock과 유사하게 구현(Lock(자원) 1개를 돌려가며 사용)

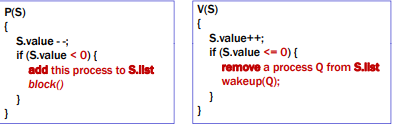
- **No-Busy Waiting 세마포** : 프로세스 Block-Wakeup 방법 이용

- 세마포 자원을 획득할 수 없을 때(S=0) 세마포 대기 큐에 자기 프로세스를 추가하고 자신을 **Block**시킴 🡺 세마포 자원이 사용 가능하게 되면**(S>0, V(S))** 세마포 대기 큐에 한 프로세스를 꺼내서 Ready큐로 이동 🡺 **대기 중인 프로세스 1개 Wakeup**

No Busy Waiting 세마포 구현

- **Value**=세마포 값 , **List**=세마포를 대기하는 프로세스 리스트(프로세스 대기 큐)

코드



(**P(S), V(S) 각각 Atomic 연산**이 되어야함, P(S)에서 S.value --; 세마포 값(자원 개수)을 **먼저 빼고** 대기 큐에 넣고 block시킨다. V(S)에서 **먼저 자원 개수를 증가**시키고 증가시켰을 때 자원이 1개이상 존재하면 대기 큐에 대기 프로세스가 없었다는 뜻, 0이하이면 대기 프로세스가 있었다는 뜻이므로 리스트에서 프로세스를 하나 깨워서 실행)

-세마포 값을 먼저 감소 = 음수 가능 🡺 **음수 크기**는 **대기 큐의 프로세스 개수**

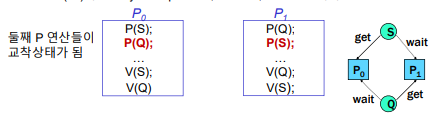
**\*세마포 검사와 감소 순서가 Busy Waiting과 반대**

EX. **S.Value = -4 면 대기중인 프로세스 4개**

- 세마포를 사용한 코드는 **교착상태, 고아** 발생가능

- **교착상태** = Deadlock(진행이 안되는 상황)

- 2개 이상의 프로세스가 **1개의 자원을 점유**하고 있고 각 프로세스들은 일을 수행하려면 서로 **상대방의 자원을 필요로 하는 상황**, 끝없는 **기다림**(즉, 프로세스가 **자원 2개 필요로** 하는 상황)

P0 : S 🡺 Q, P1 : Q 🡺 S

-기아 : 무한 봉쇄

- 대기 List를 FIFO(큐형)방식이 아닌 LIFO(스택형)방식으로 작성되면 일부 프로세스가 무한대기가 걸릴 수 있음

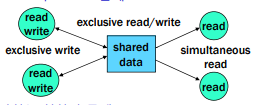
\*FIFO : First in First Out

**동기화 문제**

1 유한 버퍼 문제 : 두 프로세스가 1개 자료 공유



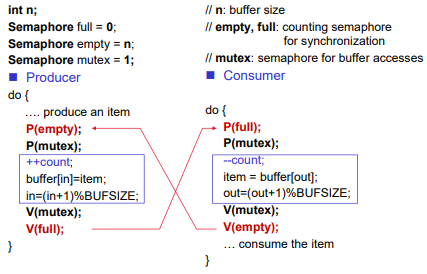
2 Readers와 Writers 문제 : 여러 프로세스가 1개 자료 공유

**쓰는동안 읽기 불허, 동시 쓰기 불허,** 동기 읽기 가능

3 **식사하는 철학자 문제** : 여러 프로세스가 여러 자료 공유



-*유한버퍼문제*



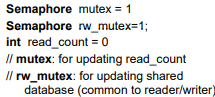
\*full=자원이 모두 사용중, empty=자원 빈자리 수(사용가능),

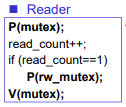
**P(S):S=S-1; , V(S):S=S+1**;<위에서는 n으로 사용>

**네모구역 = CS구역**

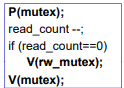
*Readers-Writers 문제*(세마포에서)

🡸Binary 세마포

rw\_mutex=1;(공유 데이터에 대한 Lock변수),

🡸**첫** Reader, 최초의 reader는 공유 데이터 사용권한 획득(2번째부터 그냥ㄱ)

Read\_count++; = reader의 개수

🡸**마지막** Reader, 마지막 reader가 Mutex공유데이터에 대한 권한 반납

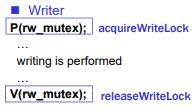
\*------(첫R)

---------

----------

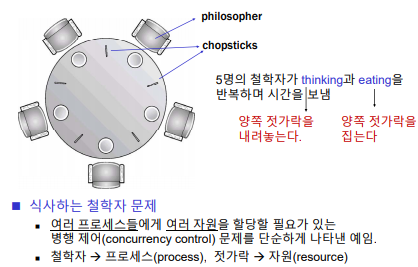
------------\*(마지막R) Reader는 다음과 같이 동시 읽기 허용

**\*은 공유데이터 상용 권한 획득 및 반납**

Writer는 Lock을 필요로 함**(동시쓰기 불가, 동시 읽기+쓰기 불가)**

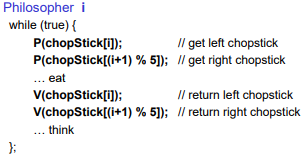
**동시 읽기/쓰기 불가인 이유** : read가 지속될 때 write가 중간에 진입하여 정보를 업데이트를 할 수가 없음 = writer고아상태(**reader가 진행 될 때 writer의 진입의사 여부 체크**)

*식사하는 철학자 문제 :* ***여러 개의 프로세스가 여러 자원을 공유할*** *때 생기는 문제*



다음장에 코드

Semaphore Chopstick[5] = (1,1,1,1,1); 🡸 세마포 자원 5개 초기화



I : 자원 번호

(i+1) % 5 : 오른쪽 자원

- But, **교착상태(deadlock)**가능성 존재 : 모든 철학자(프로세스)가 왼쪽 젓가락을 집고 있는 상태

(**무한대기**가 됨)

\*교착상태 해결법 : (Best)**젓가락 2개를 모두 집을 수 있을 때 집게 함(사용 전 젓가락 상태 확인)**

자원(젓가락)-1만큼의 프로세스(철학자)가 실행.

홀수 철학자는 왼쪽 먼저, 짝수 철학자는 오른쪽 먼저 집는다.

**5.8 모니터**

- 프로그래머가 세마포 사용 실수하면 문제 발생 가능(3가지 동기화 문제) 🡺 교착상태,기아 발생

\*잘못된 세마포 사용 예

- P와 V연산 순서 반대(상호배제위반) , V대신 P사용(P P 🡸교착상태 발생) , P또는 V누락(CS에 진입X, 상호배제 위반) \*모니터의 장점 : 세마포 잘못 사용 가능성 제로

병행 프로그래밍을 위한 지원

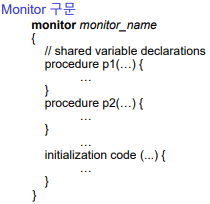
- 병행 프로그래밍이 순차 프로그래밍 보다 어려움

- 세마포 같은 기능을 직접 사용치 않고 병행 프로그래밍**(멀티 태스킹,쓰레딩)**에서 **고수준의 상호배제 지원기능 개발(언어 차원에서 지원 =** 세마포 사용하지만 프로그래머가 작성하는 것이 아닌 프로그램이 대신 지원**)** \*프로그래밍언어의 Syntax차원, 함수 차원 등등

프로세스 동기화를 위한 편리하고 효율적인 방법을 제공하는 **고수준(Language 수준/차원)의 동기화 추상화 데이터형(공유자원을 안전하게 사용하기 위해)**(ADT) – Data와 Procedure 포함

🡺 상호배제 보장

Monitor의 예

<-데이터와 프로시저를 같이 정의한 구문

🡸공유데이터 초기화 코드

컴파일러의 역할 : 프로세스가 동시에 진입하지 못하게 사용자 대신 처리

**조건변수(Condition Variables) : 모니터**에서 **동기화에 사용되는 특별한 자료구조형**

-조건변수를 사용하는 연산 : wait(), signal() 🡸 P, V

- wait 호출 프로세스는 **타 프로세스가 signal을 호출할** 때까지 **block**

- signal 연산은 정확히 **하나의 프로세스만** 재개(중지된 프로세스가 없으면 아무것도 X)

 🡸wait 3개 다 실행시키려면 signal 3개 필요

조건변수와 세마포 차**이점**

- 세마포의 V 연산은 세마포 값에 영향을 주지만 모니터는 그런거 없음

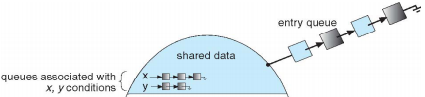
- *조건변수* : *wait를 만나면 무조건 signal이 호출할 때까지 기다림*

- **세마포** : **자원의 빈자리가 없을 때(가능자원 0)부터 wait상태** 들어감, 자원이 있으면 바로 진행

모니터 내에서의 동기화

- 조건변수에 대한 두 연산을 사용하여 수행, **조건변수 : wait가 풀리기 위해서는 signal 신호조건 필요**

조건변수를 갖는 모니터



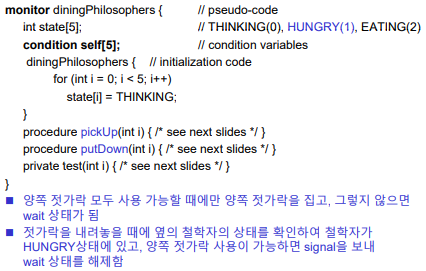
<X,Y는 큐를 기다리는 대기 리스트>

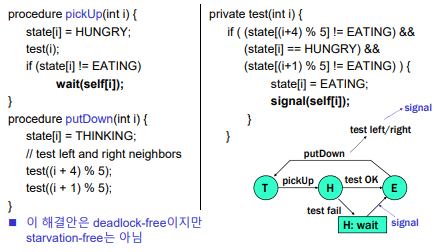
모니터에서는 식사하는 철학자 문제(Deadlock=교착상태) 해결

Int state[5]; 🡸 5명의 상태

Condition self[5]; 🡸젓가락 5개의 상태(조건)

Hungry(1) = 젓가락 확인 단계

(이 코드는 시험에 안낸다고 했음)



PickUp = 젓가락 집은 상태

Test((i+4) % 5), Test((i+1) % 5) 는 양 옆 젓가락 검사(i+4대신 i-1도 가능)

Wait 과 signal 함수 있음

\***사용 전 양쪽 확인**하고 진입함으로써 **교착상태 해결**

\* 모니터와 같은 기능이 여러 언어에서 사용 됨

**동기화(Synchronized) Statement**

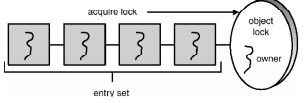
- 동기화 메소드 : 메소드를 **한번에 한 프로시저만 실행**(사용)

- 동기화 메소드 호출은 Lock 소유를 필요로 함(Lock 사용)

공유 데이터에 대한 병행접근을 **직렬화**하여 경쟁조건 발생 방지

\***프로시저가 메소드 진입 🡺 Lock시작 🡺 CS구역 같은 상태 🡺 메소트 탈출과 함께 Lock 해제**

\*직렬화 : 동기화 메소드 사용 예



**Wait와 notify Method**

-wait 메소드 : **object lock을 해제**?하고 **쓰레드를 Block상태**로 설정하고 **wait상태**에 놓임

-notify 메소드 : wait 상태들 중 임의의 쓰레드를 선택해서 entry(진입) set 으로 이동시키고 쓰레드를 Runnable 상태로 설정<V()와 비슷>

W | W | W | N |있을 때, Notify가 다른 쓰레드(W)를 깨움, W는 메소드를 기다리는 쓰레드

**\*Java**의 wait, notify는 모니터의 wait, signal과 유사하지만 **조건변수(Condition V)를 사용하지는 않음(Object당 1개의 조건 변수를 사용하는 것과 같음)**

*Wait/Notify 메소드를 사용한 insert와 remove(중요\*\*)*

**동기화 예**

Windows, Linux, Solaris, pthread library(쓰레드간 동기화를 위한 API 자원)

**윈도우 동기화**

-윈도우 커널의 Global 공유자원에 대한 접근 보호

- 단일 프로세서 시스템 : 일시적 interrupt mask(인터럽트 불가)

- **다중 프로세서 시스템** : **Spinlock 사용(멀티 코어에서는 모든 Core를 인터럽트 불가만드는것이 오버헤드)**

-Spinlock을 갖고 있는 동안 선점되지 않음

**커널 외부 쓰레드 동기화**

- Dispatcher objects 제공 – Mutex, 세마포, 조건변수 제공

- CS 구역 Object : User모드 Mutex, 커널 개입X

- 다중 프로세서 시스템에서 **처음에는 Spinlock**(가까운 시일내에 lock획득이 예상될 때 사용) 사용하다가 **오래 기다려지면 커널** **Mutex를 할당하고** Block.(커널에서 관리시작)

따라서경쟁이 있을 때만 커널이 개입(경쟁이 없으면 가까운 시일 내에 spinlock 을 획득하여 사용)

**리눅스 동기화**

**-Atomic** 정수 연산 제공

(커널 내부 동기화)



위의 기능 제공

**Solaris 동기화**

**- 적응적 Mutex** : spinlock 또는 sleeping lock**<짧은 코드 세그먼트에서 데이터보호를 위해 사용>**

-처음에는 Spinlock 사용하다가 오래 기다려지면 Block(윈도우와 비슷하지만 방식이 다름)

-Lock을 다른 쓰레드가 소유 -> Lock이 방출 될 때까지 sleep=block

-긴 코드 세그먼트용 : 조건변수, 세마포

-reader-writer locks

**Pthread 동기화(라이브러리)\*\*\*\*\***

- Pthreads **API**는 **사용자 수준에서** 프로그래머에게 제공되는 함수

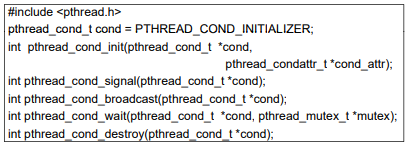
- **운영체제와는 독립적**

**기능 : Mutex, 조건변수** + 확장(버전마다 사용 가능여부 상이) : *세마포, Read-Write Lock, Spinlock*

\*P54~P64코드 따로 보고 이해만

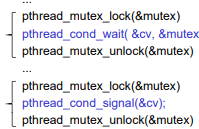
\*OS마다 조금씩 다른 방식으로 동기화 구현

**Pthread 조건변수**



Destroy : condition 자체를 없앨 때 사용

Wait, Signal 사용

Synchronize기능이 별도로 없기에 Lock 사용

🡸Mutex 얻은 상태에서 wait

Mutex 락을 걸고 wait, signal을 호출해야함

\*Mutex락 : Lock을 이용한 상호배제 락(Mutual Exclusive)

\*Spinlock : Busy Waiting을 하는 Mutex락

**5.11 교착상태**

**-**교착상태에 대한 기술방법 + 교착상태 예방법

시스템 모델

-**프로세스 : 자원(유한갯수)을 사용하기 위해 경쟁**하는 능동적인 프로그램

자원

**물리적 자원** : CPU 사이클, 메모리 공간, I/O장치 등(보이는)

*논리적 자원* : 파일, 세마포, 뮤텍스 락, 스핀락, Monitor 등(보이지 않는)

**프로세스의 자원이용 과정**

1 **요청** : 사용 가능한 자원 요청, 사용할 수 없을 시 대기

2 **사용** : 자원 사용

3 **해제,반납** : 자원 사용 후 자원 반납

- 자원 요청/사용과 반납 시스템 호출 : open()-close(), request()-release(), allocate()-free(),

wait()-signal() <= 세마포, 조건변수

**교착상태** : 프로세스가 자원을 1개씩 가지고 서로 상대방의 자원을 필요로 할 때 발생(2개의 자원을 활용해야 진행이 될 수 있는 프로세스) => '무한 대기'

EX. P1이 DVD를 점유하고 프린터 요청 + P2는 프린터를 점유하고 DVD요청 => 진행X, Block

**교착상태 특징**

필요조건 **4**가지

1. **상호배제** : 한번에 한 프로세스만 사용할 수 있는 자원(*CS*)이 존재

2. **점유하며 대기** : 적어도 1개의 자원을 점유하고 타 프로세스의 점유된 자원 대기

3. **비선점** : 자원을 강제로 뺏을 수 없음, 자발적으로 방출된 자원만 사용 = 대기

\*선점 : 상대방의 자원을 강제로 빼앗아 사용

4. **순환대기** : 순환적으로 대기 P0->P1->P2->P3->P0...(**식사하는 철학자**)

교착상태 해결법 : **4가지 중 1가지 원인만 제거**

**자원할당 그래프**

- **V** : vertex(점), **E** : edge(선)**<---------** **방향성이 있음**

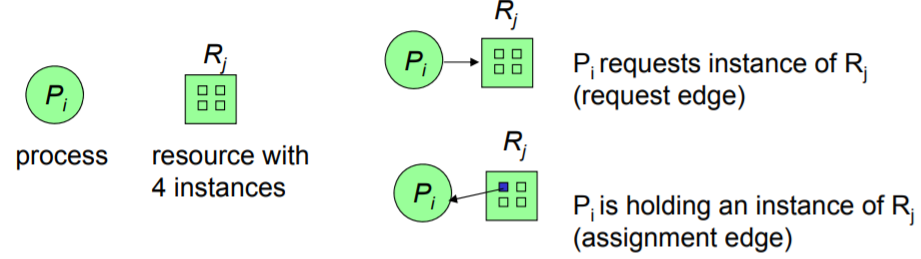
- vertex 집합 V = 두 종류의 집합(점/선) P, R로 구분

- P : 프로세스 집합, R : 자원 집합

**P ---> R : 프로세스가 자원요청** <Request edge>

**R ---> P : 자원이 프로세스에 할당** <Assignment edge>

\*4개의 자원과 하나의 프로세스가 요청/할당 예시



**자원 할당 그래프에서 교착상태(Deadlock) 알아보기**

- 자원 할당 그래프가 Cycle이 없을 때 => No deadlock, 문제없음

- **자원 할당 그래프가 Cycle이 있을 때**

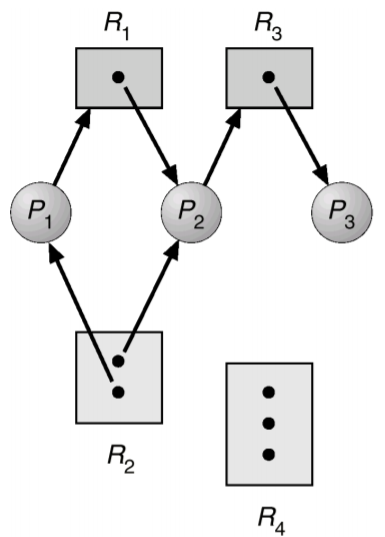
=> **자원 유형 당 1개의 instance 보유 => deadlock(무조건)**

=> *자원 유형 당 여러개 instance 보유* => *deadlock의 가능성* 존재

\*(Cycle=순환하는 선 의 수가 instance 수보다 작으면 No deadlock)

== **Cycle 수 < instance 수 : No deadlock**

**Cycle 수 >= instance 수 : deadlock**(교착상태)

(예시), 그래프 보고 deadlock 가능성 판단하기

R1,R3 : **자원 당 instance** 1개

R2 : instance 2개, R3 : instance 3개

-> **자원 4종류**

P1,P2,P3 : 프로세스 3개

P->R : 자원 요청

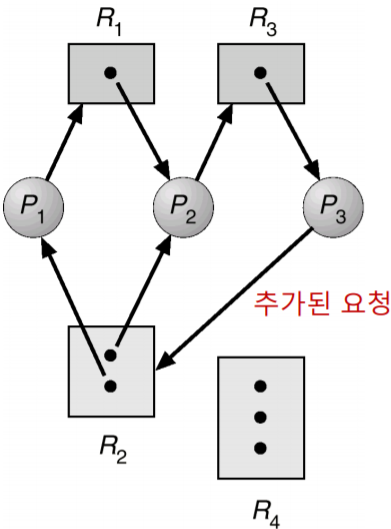
*R->P : 자원 할당*(프로세스에)

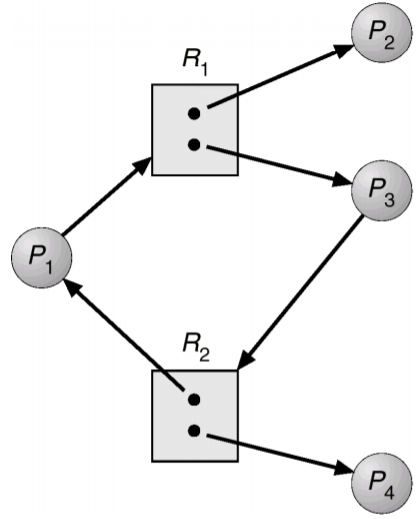
**<옆의 그래프에는 Cycle이 없음>**

**<Cycle은 화살표가 순환해야 함>**

**사이클을 갖는 자원 할당 그래프 예시(Deadlock 발생)**

\*교착상태의 자원할당 그래프(Cycle 2개 존재) \**Cycle 그려보기*



사이클 1개, 자원 당 인스턴스 2개 => 교착상태 발생X

교착상태 처리 : 예방, 회피, 탐지 및 복구, 무시

1 **예방 또는 회피** : 교착상태를 애초에 불허

-예방 : **교착상태 4가지 필요조건 중 하나 발생 막기**

-회피 : **추가 정보(실행전 자원이 사용가능한지 확인)로** 프로세스 대기여부 결정

2 **탐지 및 복구**

-**교착상태 허용,** Deadlock 추적 알고리즘 + Deadlock 복구(Ex. 프로세스Exit) 알고리즘 필 요

-보안 측면에서 사용됨

3 **무시** : 커널의 교착상태는 처리, **사용자 모드의 교착상태 무시**

-Deadlock은 자주 발생되지 않기 때문에 예방 또는 탐지/복구를 하지 않음

-**교착상태(Deadlock)허용**, 교착상태 발생 시 사용자 **수작업으로 복구**

-**대부분 운영체제에서 사용 기법 -> Kernel내 교착상태는 발생하지 않게 설계 및 예방/ BUT, 사용자 프로그램의 교착상태 문제는 무시 -> 사용자가 프로세스 종료**

-> *무시 기법 사용하는* ***이유*** : 교착상태 가능성은 매우 낮기 때문에 **매번** 교착상태를 예방하거나 탐지 및 복구 하는 것{ **오버헤드 발생** }이 시스템의 성능(속도)측면에서 **비효율적**이기 때문.

**교착상태 예방**

- 4가지 중 하나 막기

1. **상호배제 조건 거부** : 자원을 동시에 같이 쓰게 함[EX. 읽기 전용 파일은 동시에 읽어 도 문제 발생X ], **(*단점* : 동시 공유가 불가한 자원에 대해서는 불가능)**

2. **점유 대기 조건 거부** : 실행 전 모든 자원 할당

- **실행 전** 필요한 **모든 자원 요청** 및 할당

- 점유 자원을 반납 한 후에 반납한 자원과 추가 자원 요청

- *단점* : 자원 이용률이 낮아진다.(비효율) + 기아상태 가능성

3. **비선점 조건 거부** : 선점 허용

- 현재 점유한 모든 자원을 반납하고 대기 => 프로세스는 반납한 자원과 추가요청 자원 의 대기리스트에 추가 => **모든 자원을 할당 받을 수 있을 때 재개**

- **요청한 자원을 이용할 수 없을 때 대기상태가** 됨 => 자원이 타 프로세스에게 선점됨 => **선점 된 자원과 요청한 자원을 할당 받을 수 있을 때 재개**

- *단점* : 자원 저장/복원이 용이한 경우만 적용가능(CPU레지스터, 메모리)

4. **순환 대기 조건 거부** : **여러 자원이 필요한 경우 오름차순으로 자원을 요청** => 자원 을 얻는 순서가 일정하게 적용 됨 => **0,1,2,3,4...(자원) 순으로 자원을 할당 요청(자원 3 을 할당 받은 상태에서 추가요청은 4,5,6,7... 순으로 자원 요청, 0,1,2 자원 요청하려면 반납후 대기)**

- 가장 많이 사용되는 예방법

교착상태 예방의 단점

- 1. 장치 이용률 저하 : 순환대기 조건 거부의 경우 프로세스가 자원을 모두 할당 받지 못할 시 타 프로세스가 자원을 내어줄 때까지 사용되지 않고 있는 자원까지 기다려야 함

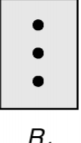
**\* 자원: 0,1,2,3중에 0,1이 할당 중일 때 다른 프로세스가 자원1,4 를 필요로 하면 할당된 1이 반납 될때 까지 자원4도 가져가지 못하고 대기**

- 2. 시스템 처리율 감소

**교착상태 예방/회피 방법**

- 현재 자원들이 어떻게 할당되어 있는지 추가정보 필요 -> 순환 대기 조건이 발생하지 않도록 자원 요청 제어

Safe 상태 = No Deadlock / Unsafe 상태 = Deadlock 가능성 존재

**단일 인스턴스 자원 - 자원 할당 그래프 사용**하여 교착상태 회피 알고리즘 구성

 // 단일 // 여러 인스턴스 자원

여러 인스턴스 갖는 자원 - Banker's 알고리즘 사용 - 생략함

자원할당 그래프 알고리즘

- 예약 Edge : 잠재적으로 미래에 요청 가능성이 있는 Edge(선)은 점선으로 표시

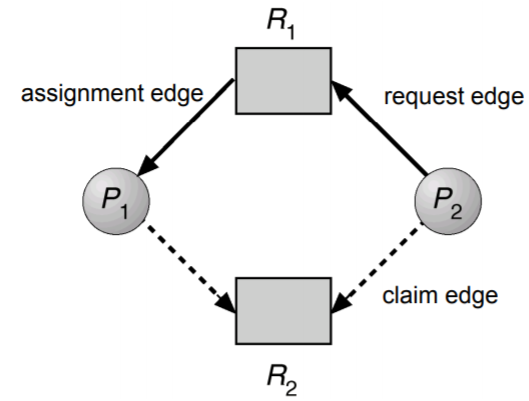
P --------------------> R //**자원요청**

p - - - - - - - > R //**자원 요청 가능성(점선)**

R---------------------> P // 자원 할당

P0--------------------> P1 // ***대기 그래프(P0이 P1에 할당(점유)된 자원 대기\*\****

=> **자원 요청 선**을 **할당 선으로 변경**하는 것이 자원 할당 그래프에 **Cycle을 형성하지 않을 때** 승인/허용 \* Cycle 수 >= instance 수 : Deadlock 가능성

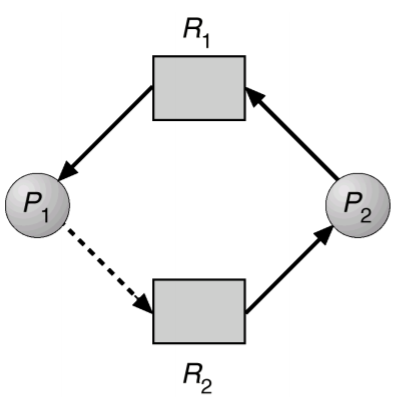


P1은 R1을 할당 받음

P2는 R1에 자원 요청

**점선은 요청 가능성 존재** Edge



<=만약, 이 상태에서 P1이 R2요청하면 교착상태 발생하므로 애초에 P2의 R2 요청은 허용X

P1은 R1을 할당 받음

P2는 R1 자원을 요청

P2는 R2의 자원을 할당 받음(불가)

<= Cycle이 생김

**\*Cycle은 점선까지 포함하여 생각함**

**따라서, P2의 R2요청은 승인되지 않음**

**교착상태 탐지(2가지 유형)/복구**

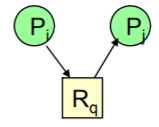
- 단일 인스턴스 자원 유형(배움) <= 한번에 한 프로세스만 사용 가능(CS)

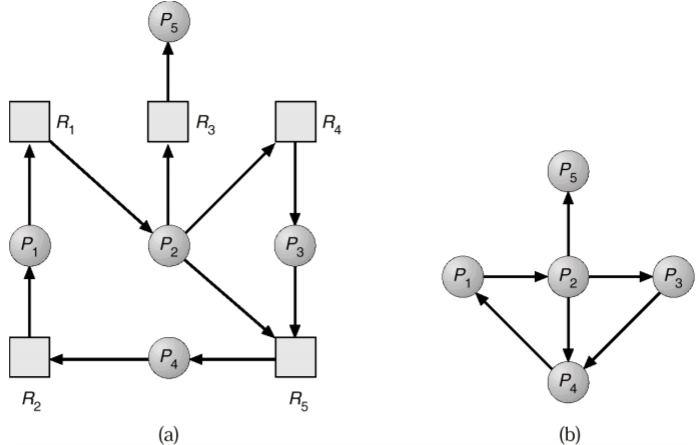
- 다수의 인스턴스를 갖는 자원 유형(안배움)

단일 인스턴스 자원 유형

- Wait-for graph를 이용하여 **cycle 존재 여부(출발점으로부터 시작하여 다시 출발지로 돌아올 수 있는지)** 검사

- O(n^2)시간 소요

 **==**



==>

탐지 알고리즘의 수행 빈도

- 얼마나 자주 교착상태가 발생하는가? + 발생 시 몇개의 프로세스가 영향을 받는가?

방법 1 : 요청이 즉시 승인되지 않으면 바로 탐지 알고리즘 수행 = 오버헤드 큼, 유발 Prcss 탐지

**방법 2** : 일정 시간(1시간)마다 탐지 알고리즘 수행 =교착상태를 유발한 마지막 프로세스 탐지 불가

방법 2 많이 사용

교착상태에서의 **복구**

교착상태 복구 방법

1. 프로세스 종료(2가지 방법)

2. 자원 선점

종료 1. 교착상태 모든 프로세스 중지(Abort)

2. 교착상태가 제거 될 때 까지 하나씩 중지 = 부분 종료 방식

선점 - 교착상태가 제거 될 때 까지 프로세스의 자원을 선점하여 타 프로세스에게 제공

- Victim 프로세스 발생 5장 끝

**6장**

CPU 스케쥴링

개념

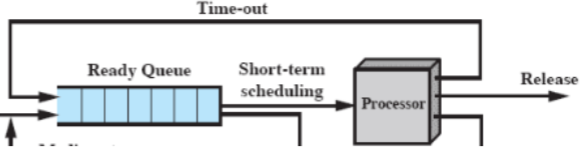
1. 멀티프로그래밍의 목적 = CPU 사용률 극대화

2. CPU-I/O Burst Cycle

- 프로세스 실행은 **CPU 실행(CPU burst) + I/O대기(I/O burst)**로 구성

- I/O bound 프로그램 = CPU burst 시간이 짧음/ **CPU bound 프로그램 = CPU burst 시간이 김**

**-CPU 스케쥴러** = **단기 스케쥴러** 라고도 함 = 빈번히 발생하는 스케쥴링



<Ready큐에 있는 프로세스 중 하나 선택해서 프로세서에 할당>

\*장기 스케쥴러는 Disk에서 메모리의 Ready 큐로 옮기는 것 = 현대에는 사용자의 영역으로 변함

\*중기 스케쥴러는 메모리의 프로세스(Ready)와 Disk의 프로그램과 바꾸는 것(가상화 사용가능)

**CPU 스케쥴링 시점\*\***

**1**. Running상태에서 *Wait로 전환*할 때 : **100% 스케쥴링 필요**(다음 Running 선택)

**2**. Running 상태에서 ***Ready로 전환*** : Ex. Timer의 Time-out

**3**. Waiting 상태에서 ***Ready로 전환* :** Ex. I/O 처리완료

**4**. Terminated(**종료**되서 반납을 기다림) : **100% 스케쥴링 필요**

- 비선점 스케쥴링(호랭이 담피시절) : **1,4번 경우에만 스케쥴링 필요** + **반드시 스케쥴링을 통해** 새 **프로세스 선택**

- 선점 스케쥴링(현대) : **모든 경우 스케쥴링 가능(**1,2,3,4번 경우) + **CPU 독점 방지(**Timer), **우선순위 반영** 할 때 2,3번 스케쥴링***(Ready 큐에 변동이 있을 때 스케쥴링) = 우선순위 반영, 독점방지***

**선점(Preemptive) 스케쥴링(경쟁 조건 발생)의 문제점/해결책**

- **공유 데이터 일관성 문제** : 프로세스는 **데이터 변경 중**에 **자원이 타 프로세스에게 선점**되어 변경된 데이터를 저장하기 전에 CPU를 내놓을 수 있다. -> 경쟁적으로 데이터 변경 시 일관성이 유지되지 않을 수 있다. = 잘못된 데이터가 사용 될 수 있음

- **해결책** : 프로세스 **동기화** = 데이터 접근에 대한 동시성 제어

유저 모드에서의 선점 : **비선점 스케쥴링의 운영체제(OS)에서** 한 프로세스가 데이터를 변경하는 동안 선점되어 다른 프로세스가 같은 데이터를 읽거나 수정한다면 데이터 일관성 유지X

**Kernel 모드에서의 선점**

- 커널 모드에서 공유 데이터 접근

- 커널은 SYSTEM-Call 을 통해 요청된 프로세스의 작업을 처리한다. -> **공유 데이터를 쓰는 커널 루틴 동안 인터럽트로 인해 다른 커널 루틴에게 선점되면 데이터 일관성 유지X**

-시스템 전체에 영향을 주므로 위험

운영체제 커널에서 선점 처리 방법

- **비선점 커널** : 커널 내에서 **선점을 허용치 않음(**커널 작성이 쉬움**)**

허용하는 경우 **1**. Sys-call 처리 완료, **2**. 커널 루틴 처리 중 I/O block이 발생하여 대기할때

-> 실시간 컴퓨팅에서는 부적합

\*유저모드에서는 선점이 이루어질 수 있지만 커널모드에서는 불가

- **선점 커널** : 커널 내에서 **선점 허용(**커널 작성이 어려움**)**

**커널 내에서 공유 데이터 접근(**CS구역 설정**)에 대한 동기화(**세마포, 조건변수, Mutex 등 등**)를 사용**하여 커널 설계

-> 실시간 컴퓨팅에 적합

\*유저모드, 커널모드 모두에서 선점이 이루어질 수 있음

<프린트 그림보기>

디스패처(Dispatcher)

- **CPU 제어권을** CPU 스케쥴러가 선택한 프로세스에게 주는 모듈

-Context Switching 수행(CPU동작모드 유저모드로 전환), 유저프로그램의 위치 이동

디스패처 지연(Latency) : 한 프로세스를 정지 ~ 다른 프로세스 수행 시작

- 가능한 빨라야함 <- 오버헤드(사용자를 위한 성능적인 입장에 있어서 유용한 작업이 아님)이기 때문에

CPU 스케쥴링의 기준 **5**가지

- **CPU 이용률** / 처리량(**시간당 수행이 완료**된 프로세스) / 총 **처리시간(프로세스 당**) / **대기시간**(Ready큐에서) / 응답시간(대화형 시스템에서 **요청~응답 받기까지 소요시간**)

스케쥴링 최적화

- CPU 이용률, 처리량 : 최대화 / 총 처리시간, 대기시간, 응답시간 : 최소화 가 BEST!

- 대화형 시스템은 **응답시간의 변동 폭 최소화** = 일정한 응답시간 제공

***스케쥴링 알고리즘 5가지***

- 선입 선출 스케쥴링(FCFS = FIFO) = 선착순

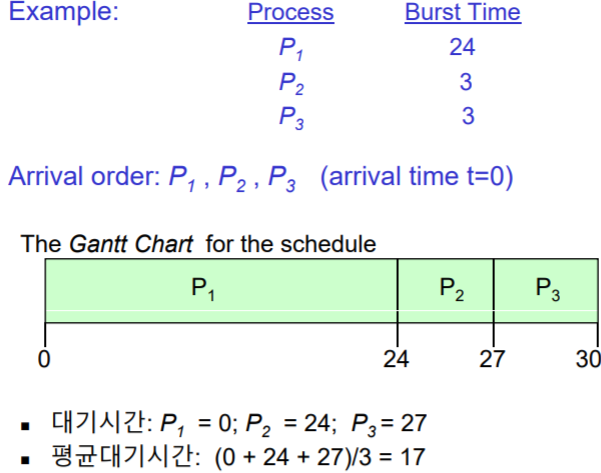
- 최단작업우선 스케쥴링(SJF) : **Short Job First**

- 우선순위 스케쥴링 : Priority

- 라운드 로빈 스케쥴링(RR) : Round Robbin

**- 다단계 큐(우선순위 별) 이용 스케쥴링 + 피드백**

**\*선입선출 스케쥴링(FCFS)** : 아무리 비슷한 시간에 들어와도 모든 프로세스는 1자로 **나열되어 순서대로 처리**

<- **대기시간 기준,** 총 처리시간 : 30

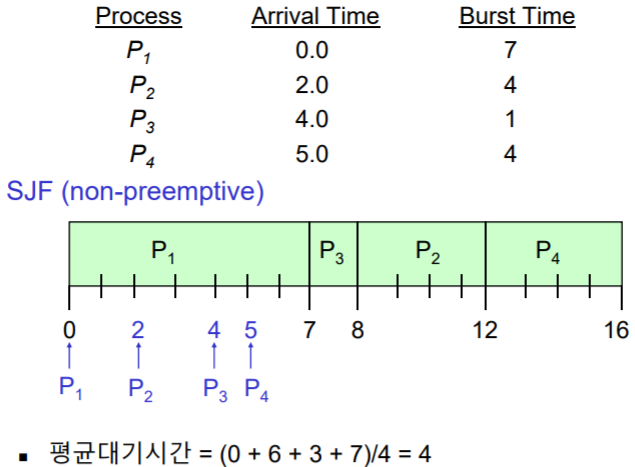
\***대기시간** : **서비스 요청부터** **~ 프로세스 처리 완료 시까지 Ready큐에 대기하는 시간**

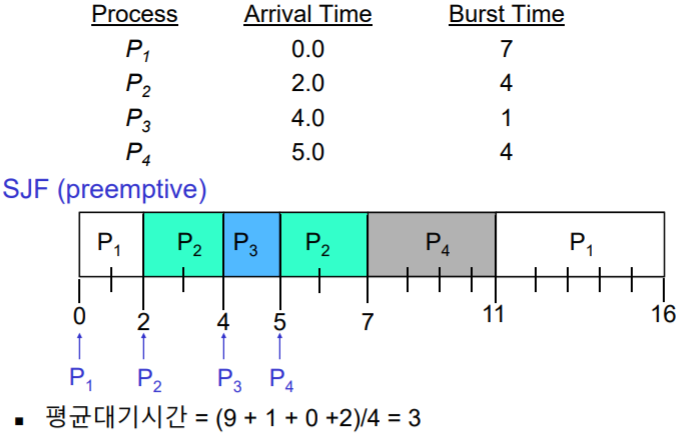
**\*최단작업우선 스케쥴링(SJF** = SPN = SRN) : **평균 대기시간이 가장 짧은 방식**

**-** CPU **최단 다음 CPU burst를 갖는 프로세스를 스케쥴링**

- **문제점** : CPU burst를 사전에 알기 어려움 -> **예측**방법 사용(뒤에 계산법 나옴)

- 2가지 방식 존재 : *비선점 SJF*, 선점 SJF = SRTF(Shortest-Remaining Time First)

비선점 SJF

선점 SJF

P1 : 2~11 까지 대기 = 9, P2는 4~5까지 대기 = 1 ....

Arrival Time 은 Ready큐에 프로세스가 실리는 시간

평균대기시간이 가장 짧음 = 선점 SJF

\* **Context Switching이 더 자주 일어남 <- 해당 오버헤드 발생 증가** 가능성있음

대기시간 : Ready 큐의 대기시간 = (Arrival Time ~ 프로세스 종료까지) - CPU burst 시간

<계산 문제 나올 수 있음> , <예시 만들어서 풀어보기>

**CPU burst 길이 예측** : 최단작업 스케쥴링(SJF)에서 프로세스마다 CPU burst 길이를 **사전에 알 수 없기 때문에 예측 방법 사용**

- 지수 평균 값 사용 : 다음 예측 값 = 이전 예측 값과 이전 CPU Burst 의 지수평균

 \***a**는 0~1 사이 비중을 두고 싶은 만큼 사용(보통 0.5)

\***a**는 **이전 CPU burst값** 의 비중

Ex.a = 0 , 이전 burst값 고려 않고, 이전 예측 값을 사용

a = 1 , 이전 CPU burst 값을 사용, 예측 값 사용X

확장 식 :

\*초기 예측 값은 지정해주어야 한다.

**\*우선순위 스케쥴링**

- 우선순위가 높을 수록 보통 작은 번호를 가짐

- 우선순위 **정의(정하는데) 고려 요인 =** 많다

**내부적** : 시간제한, 메모리 요구, Open file수, I/O와 CPU 비율 등

**외부적** : 프로세스 중요성, 비용, 사용자 지정

우선순위 스케쥴링 2가지 방식 : **선점** 스케쥴링, **비선점** 스케쥴링

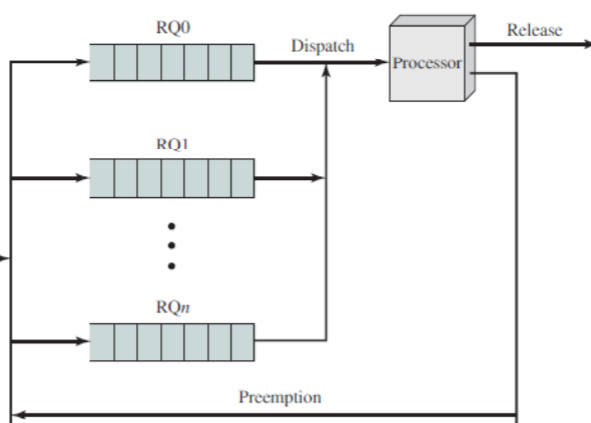
문제점1 : **낮은 우선순위**의 Process가 **무한히 대기**하여 수행되지 못할 수 있음

해결책 : **Aging** 방법 : **오래** Ready 큐에 **대기하는 프로세스의 우선순위를 점차 증가**

**\***SJF(최단작업우선 스케쥴링)은 우선순위 스케쥴링 중 하나이다.

문제점2 : **대기 프로세스 개수가 늘어날수록** 큐를 탐색하여 우선순위가 높은 것을 찾는데 **시간이 오래걸림**

해결책 : 우선순위 별로 **Ready 큐를 여러개** 만듦(큐마다 우선순위를 가짐, Ex.0순위 큐, 후순위 큐)

RQ0 우선순위 최상위 큐

**\*라운드 로빈(RR) 스케쥴링**

- 시분할 시스템에서 응답시간 최소화를 위해 설계 (Time Sharing)

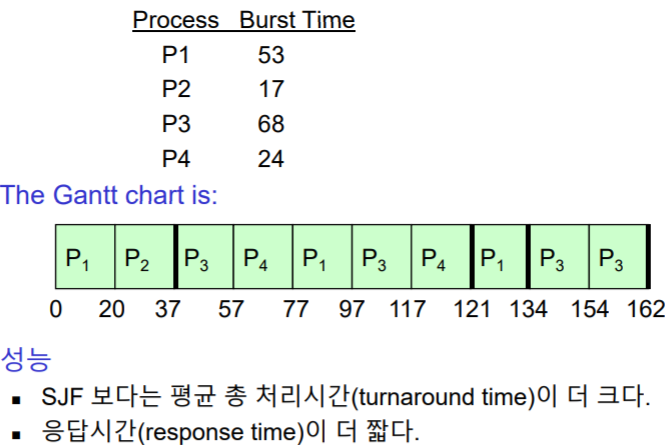
- 각 프로세스에게 작은 양의 CPU 시간을 고정적으로 할당 -> 시간 경과 후 선점되어 Ready큐로 이동(보통 Timer : 10~100msec)

- **Time Quantum(CPU 할당시간)**이 **크면** FIFO와 동일(모든 프로세스가 한 time quantum에 완료)

- Time Quantum이 **작으면** Context Switching 오버헤드가 너무 크게 됨

- 최적 시간 할당량(Time Quantum) : **CPU burst time의 80%정도**를 Time Quantum으로 정함??

RR스케쥴링의 예



단점

장점

Time Quantum = 20일 때

응답시간은 대화형 작업, 여러 사용자 동시 사용시 중요

**\*다단계 큐 스케쥴링**

- 여러 개의 큐로 분할

- 각 큐는 자신의 스케쥴링 알고리즘 사용 : interactive용=RR(응답시간 최소), batch용=FCFS, 우선순위용=우선순위(Priority)

**스케쥴링은 큐들 간에도 존재** : **우선순위가 높은 큐가 수행에 있어서 빈틈이 없으면** 낮은 우선순위 큐의 프로세스는 Block됨(수행을 못함)= **기아상태 가능성** 존재

따라서, Time-Slice : 각 큐마다 CPU 사용/비율을 정함 /Ex. foreground큐 : 80%,background큐 : 20%

다단계 피드백 큐 스케쥴링

- 다단계 큐 스케쥴링은 Process 생성시 하나의 큐에 영구 할당

- 다단계 큐 스케쥴링은 Process 생성시 하나의 큐에 할당 되었다가 상황에 따라 큐간 이동가능

- Aging 구현

**필요한 정책(매개변수)**

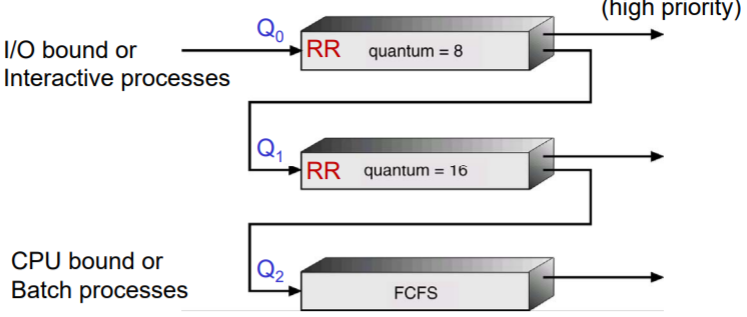
**1**. 큐 갯수

**2**. 각 큐를 위한 스케쥴링 알고리즘

**3**. 큐 승급/강등 시점

**4**. 프로세스가 서비스를 필요로 할 때 진입 큐 결정 방법

<강등만 존재하는 다단계 피드백 큐 예시>



Time Quantum시간을 나눠서 큐를 만듦으로써 I/O bound 프로그램과 CPU bound 프로그램이 작업 특성에 따라 자동적으로 분류되게 된다. RR 방식(8) - RR 방식(16) - 선입선출(FCFS) 방식

우선순위 : Q0 > Q1 > Q2

Thread 스케쥴링**(Many to 1 / 1 to 1 / Many to Many)**

- **운영체제**는 Kernel 레벨 쓰레드들을 스케쥴링

- User 레벨 스케쥴링은 **Thread library**에 의해 수행

**스케쥴링 경쟁 범위**

- **프로세스 경쟁 범위**(PCS: Process-contention scope) **: 한 프로세스의 Thread들 간에** 스케쥴링 경쟁(CPU 스케쥴링이 아님)

- 가용 **LWP 상에서 스케쥴링** 됨 \*가용 LWP 란? LightWeight Process, LWP는 커널과 사용자 프로세스 를 연결하는 인터페이스 역할

- **Many to One, Many to Many** 에서 사용

- **시스템 경쟁 범위**(SCS) : 시스템 전체가 스케쥴링 경쟁범위

- 커널이 Kernel-level thread들을 CPU 스케쥴링

- **One to One** 모델에서 사용

\* Pthread 에서 스케쥴링 사용 - thread생성 시 지정

-> PTHREAD\_SCOPE\_PROCESS : 프로세스 경쟁 범위

-> PTHREAD\_SCOPE\_SYSTEM : 시스템 경쟁 범위

Pthread 스케쥴링의 예시

- if(pthread\_attr\_setscope(&attr, **PTHREAD\_SCOPE\_SYSTEM**) !=0) 과 같이 **스케쥴링 범위 지정**

다중 프로세서 스케쥴링

- 다중프로세서 **스케쥴링은 공유 자원 문제, 동기화로** 인해 단일 프로세서 스케쥴링보다 어렵다.

- 다중 프로세서 스케쥴링 접근 방법

- **비대칭 다중처리(Asymmetric)** **: 코어별로 수행하는 작업이 다름**, Ex. 한 프로세서는 스케쥴링, 입출력처리, 커널 수행/ 다른 프로세서는 유저모드 프로세스 수행

장점 : ***커널*이 1개의 코어만 사용하므로 동기화 문제를 쉽게 해결 + 공유 데이터 문제 해결**

**- 대칭 다중처리(Symmetric) :** 각 프로세서는 **독자적으로 스케쥴링**되어 어느 프로세서든지 **모든 역할 수행 가능**, **한 프로세스**를 **동시에 여러 CPU에서 처리 가능**

대칭 다중처리의 2가지 방법

- 공동 큐 : 모든 프로세서가 큐를 공동 사용 -> 동기화가 철저해야함(동기화 처리로 오버헤드 발생)

- 분리 큐 : 프로세서마다 분리된(독자적) 큐 사용 : 자료 공유 자동적 배제(현대 OS 사용)

\***분리 큐**를 사용하면 공동 큐의 **자원 공유 문제 해결 가능** : **CPU마다 자신의 큐가 존 재(**CPU0 : 큐0 / CPU1 : 큐1 / CPU2 : 큐2 / CPU3 : 큐3)

공동 큐 - CPU0, CPU1, CPU2, CPU3 ============== Queue

**분리 큐 - CPU0 ==================Queue<독자적 큐 존재로 공유 자원 문제 해결>**

**CPU1===================Queue**

**CPU2===================Queue**

**CPU3===================Queue**

**프로세서(Core) 친화성\*\*** : **현재 실행 중인 프로세서에서 다른 프로세서로의 큐 이동을 피하고 현재 프로세서에서 다시 실행되기를 시도**하는 것 -> **캐시무효화를 방지할 수도** 있게 됨 <- 해당 코어의 캐시내용이 보존 될 경우 다시 실행 되었을 때 캐시를 그대로 **재사용** 가능(효율적)

\*캐시 메모리 : CPU들의 공용 캐시(보통 L3 캐시)가 있고 CPU 코어마다 독자적인 캐시 (L1, L2 캐시) 존재

프로세스 친화성 형태

- 연성 친화성 : 이주O, 하지만 기본적으로는 현재 프로세서에서 계속 머무르려는 경향

- *강성 친화성 : 이주X,* 특정 프로세서 지정

**시스템의 형태/구조가 프로세서 친화성에 영향을 줄 수 있다.**

**Ex.** NUMA 시스템 : 강성 친화성을 가지는 시스템으로 메모리의 부분(PART)마다 각 CPU코어로 액세스 속도가 다른 부분이 있다. 메모리 : ㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁ(메모리 셀 공간)

CPU : CPU1 CPU2

\*구조에 따라 특정 부분 액세스가 빠르고 느릴 수 있다.

부하 균등화 : **프로세서들**은 개개의 **독립된 분리 큐**(데이터 공유 문제[Ex.동시접근, 일관성 유지]를 없애기 위해 사용)를 가지고 있는데 단점으로는 특정 레디 큐에 작업이 몰리는 **'부하 불균형'**이 발생할 수 있음. -> 해결을 위해 **부하 균등화** 사용

**\*부하 균등화 : 모든 프로세서들 간에 부하(작업)가 균등하게 배분하려는 것**

따라서, **공통 큐**를 갖는 시스템에서는 **불필요**

**- push migration**방식 : *OS의 판단*에 따라 결정

- 큐에 있는 프로세스의 작업량을 미리 알 수 없기 때문에 특정 task가 **주기적으로 각 프로세서의 부하를 검사 ->** 과부하 프로세서(큐)가 발견되면 **해당 프로세스**들을 -> **less-busy/Idle 상태 프로세서**로 **이주**

**- pull migration**방식 : CPU에서 작업을 끌어옴, *CPU가 작업요청*

- Idle 프로세서가 busy 프로세서 레디 큐에서 프로세스를 가져옴(이주)

\*두 방식 모두 **동시에 사용 가능**

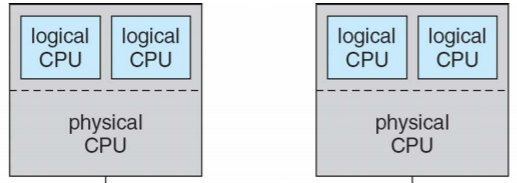
장점 : 프로세스를 이주시키므로써 프로세서의 **작업이 고르게 분배**되어 **CPU를 효율적**으로 이용률을 높힌다. -> 프로세서 친화도(같은 프로세서에서 계속 실행하려는 시도)의 장점(캐시 보존)과 **반대**

멀티쓰레드 프로세서 : OS의 스케쥴링이 아닌 자동적 스케쥴링

Ex. Intel의 하이퍼쓰레딩 기술 = 한 코어 안에 Thread가 2~4개가 들어감으로써 실제 **연산장치(ALU = 물리적 연산장치)**는 하나지만 **쓰레드(논리적 프로세서)**를 여러개 두어 연산장치를 공유하여 쓰는 프로세서 <**1코어 : 1물리 프로세서 + 2~4논리 프로세서**>

- 쓰레드는 독립적인 레지스터를 가짐 <- **빠른 Context Switching을 위해**

- **하이퍼 쓰레딩** 사용 방식 : 한 쓰레드(논리 프로세서)가 수행 중 메모리에 접근하는 동작을 하면 다른 쓰레드가 ALU(물리 프로세서)를 이용 -> 연산장치를 최대로 이용 + 빠른 context switching

하지만 한 코어에서 thread들이 **병렬처리되지는 못함**, 각자 다른 코어의 쓰레드끼리는 병렬처리지만 한 코어 안의 쓰레드끼리는 병행

\*CPU bound 프로세스 : ***CPU 사용***이 많음(CPU를 한번 잡으면 길게 점유)

I/O bound 프로세스(**대화형**) **: *I/O사용***이 많아서 CPU 사용시간은 짧음 -> CPU bound 보다 **우선순위가 더 높음**(I/O bound 프로세스가 실핼시간이 짧아서)

여러 운영체제에서의 스케쥴링 예시 **<운영체제 별로 특징/핵심 Idea만 이해하기>**

Linux 스케쥴링(옛날)

- Kernel버전 2.5이전 : Unix scheduling 알고리즘 변형 사용

- SMP(Symmetric 멀티 프로세서)지원 미약 -> 프로세스 수 증가에 따른 성능 저하(Scalability가 낮음) 스케쥴링해서 실행하는 시간 : **O(n)** <<- n은 프로세서 수

- Kernel버전 2.5, 2.6 : 상수시간에 실행 **O(1)** <- **스케쥴링 시간**이 프로세스 수가 증가해도 일정

- 프로세스 수가 많아져도 스케쥴링 시간 변화X + SMP 지원(부하 균등화, 프로세서 친화도)

-***선점형 우선순위 기반 스케쥴링***

- **O(1)**이 될 수 있는 이유 : 애초에 프로세스 수행 후 **반납 할 때 우선순위에 맞춰서 큐의 적절한 위치로 반납** -> 이후 스케쥴링이 따로 크게 필요치 않음

- 우선순위 : 낮은 값을 더 높은 우선순위로 함(보통의 경우와 **반대**)

\* 우선순위 : **Real time(**선점형 SJF**) > time-sharing(**RR**)**

**- 우선순위가 높을 수록 -> 큰 time Quantum 부여** -> 그룹단위로 작업이 처리됨

|----------------------------| <- 프로세스

|-------| **\* 우선순위가 높을수록 time slice가 많다.**

|-------------|

|-------------------------|

각 프로세스의 time slice를 모두 처리할 때까지 기다렸다가 모든 작업의 time slice가 처리되면, 다시 채워서 스케쥴링 <- 한 프로세스가 작업 완료 후 너무 오래 기다릴 수도 있는 단점이 있다.

**Linux 스케쥴링 2.6.23+은 P41,42 직접 보기\*\*\* 중요!**

Windows 스케쥴링

***- 선점형 우선순위 기반 스케쥴링***

- **time Quantum 만료** -> 우선순위 **하락**시켜줌

- wait 해제(깨어날 때,Ex. **I/O대기**[I/O대기로 들어가면서 자원이 선점 당함]**가 끝날 때**) -> 우선순 위 상승시켜줌

-> **I/O bound 프로세스(I/O 대기를 자주함) 일수록 wait 횟수가 증가(=wait 해제 횟수 증가)하므로 우선순위가 자연스럽게 높아진다**. 따라서 I/O bound > CPU bound 우선순위 구현

- 우선순위 : foreground > background(3배)

Solaris 스케쥴링

***- 우선순위 기반 스케쥴링***

- 6개의 스케쥴링 클래스

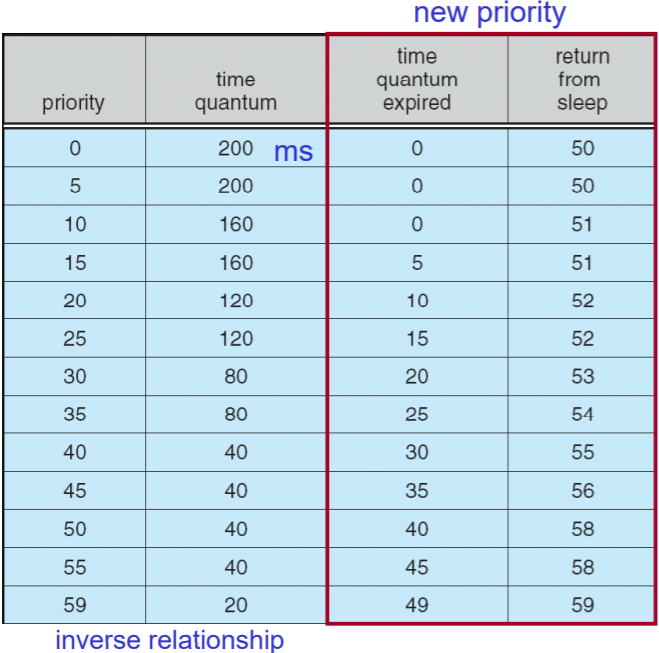
1. time-sharing(대화형)

2. interactive

3. real time 등등

- 다단계 피드백 큐 스케쥴링을 사용 -> 동적으로 우선순위 변경 + 다른 길이의 time slice 할당 -> 다른 길이로 할당 의미 : **I/O bound에게는 짧은 time slice**(우선순위 높음) / **CPU bound 는 긴 time slice 할당(**우선순위 낮음**)**

- Solaris Dispatch Table



\* 숫자가 크면 우선순위가 높음(Solaris 에서)

\* return from sleep(wait)일때 우선순위를 높여줌

\* **time Quantum**은 우선 순위가 낮을 수록(CPU) 길다/ 우선 순위가 높을 수록(I/O) 짧다.

**\* Time Quantum이 짧을수록 Wait상태로 자주 들어가므로 더 많이 Wait(sleep)에서 깨어나고 우선순위가 더 자주 많이 높아지게 된다 <-** 자연적

\* time Quantum expired : 만료된 횟수를 나타냄

스케쥴링 알고리즘 3가지 평가 방법

1. 결정론적 모델 : 미리 정의 된 부하 사용(고정)

2. Queueing 모델 : 수학적으로 부하를 계산하여 예측해서 큐에 작업 분배

3. 시뮬레이션 : 시스템 모델의 부하를 실제 부하 COUNT를 구해서 스케쥴링 알고리즘에 적용

번외 : OS를 구현해서 부하정도를 실제로 확인하고 보완 -> 매우 힘듦  **6장끝!**