

# CH5 : CPU Scheduling

## 5.1 Basic Concepts

## 5.2 Scheduling Criteria

## 5.3 Scheduling Algorithms

## 5.4 Thread Scheduling

## 5.5 Multi-Processor Scheduling

## \* about CPU Scheduling

① is basis of multi-programmed OSs  
 (→ P 여러개 실행하는 OS에서는 기본)  
 병렬작업 X, 여러개 실행

- ② OS schedule kernel-level thread  
 ↳ User-level (X) thr lib
- ③ process sche → thread sche
- ④ run "on a CPU" = "on a CPU's core"
- ⑤ 일관, core 1개로 가정

## 5.1 Basic Concept

: Single CPU core  
 → one processor run at a time  
 Multiprogramming : CPU 사용률 최대화  
 위해 Process 항상 실행  
 ↑  
 Process 실행 → I/O request → CPU wait...  
 Several process in memory at a time  
 (Everytime one process has to wait  
 another process takes over CPU)

## Scheduling

- ① fundamental OS function
- ② almost all resources are scheduled before use

## 5.1.1 CPU-I/O Burst Cycle

\* CPU 스케줄링은 프로세스 단위에  
 의해 결정 (I/O burst < CPU burst)  
 ?:?:

### (1) Process Execution

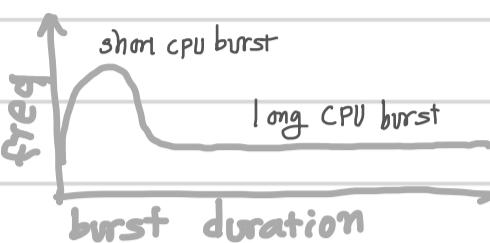
= CPU exec + I/O wait

= CPU burst → I/O burst → CPU →  
 시작할 때  
 마지막  
 final CPU burst → I/O...  
 end with sys request to execution

= Process에 I/O burst가 많을 때  
 CPU burst가 많은지에 따라  
 고려해서 scheduling \*

## (2) frequency curve & CPU burst

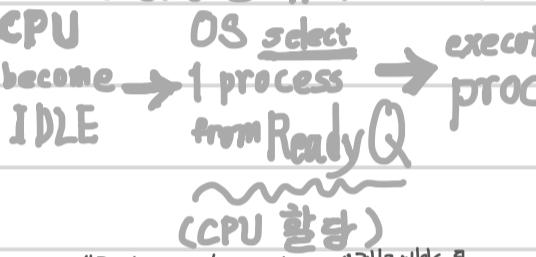
: duration of CPU burst tend to have  
 "frequency curve"



→ 보통 한 App에서 CPU를 끊거나 하는 것들이 많지 않다. frequency curve

\* I/O bound program → short CPU bursts  
 CPU bound " " → long CPU "

## 5.1.2 CPU SCHEDULER



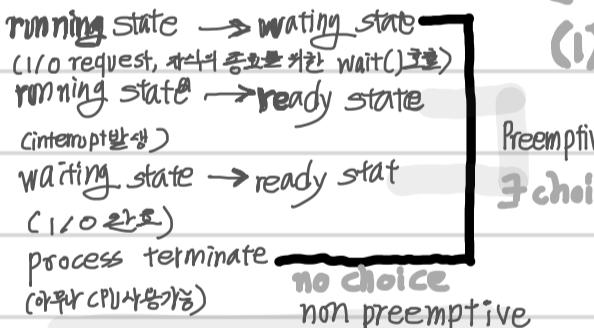
→ CPU 스케줄링은 readyQ에서 선택하거나  
 멀티태스킹 process로부터 Select → CPU 할당.

\* ready Q는 FIFO Q, Priority Q, tree, LL  
 Unordered Linked List

\* Ready Q 내의 레코드들은 Process의 PCB

(Process Control Block) \*  
 ↳ ready Q에서 스케줄링 Algo에 따라 실행할 대상

## 5.1.3 Preemptive Non P Scheduling



(1)

Preemptive  
 choice

no choice  
 non preemptive



## (2) nonpreemptive 비선원

(release 때까지 기다려야 함)  
 by termination, switching to the wait state

## nonpreemptive sched Algo

- ① access to data
- ② preemption in kernel mode \*
- ③ interrupts during crucial OS activity \*

⇒ 3 가지를 고려해야 함. (→ 절대 OS 서비스)

## ① Access to shared data - preemptive

this result in race condition

(프로세스간 데이터 공유시 경쟁초기 초래)

2) read data in inconsistent state (누락)

(동일 Data를 다른 프로세스가 write, 다른读后 read)

## ② Kernel mode - preemption

kernel mode → preemption → 쿠레발생.  
 (OS의 자리를 받아 처리하는 kernel의 중요한 data를 바꿈)

1) Preemption은 커널로드에 영향을 줌.

따라 syscall 처리 등장, 커널은 Proc를 위한 활동으로 바뀜  
 (change important data 000 kernel modifying 00000 process preemption)  
 ⇒ I/O Q

## 2) Design of OS Kernel

### non preemptive kernel

커널이 syscall complete를 기다리며,  
 그 프로세스가 block 되기를 기다림.

kernel structure is simple  
 (커널 data structure가 inconsistent상태에서 신뢰성 X)

real time computing 지원 X (실시간 X)

### Preemptive kernel :

: Prevent race condition in shared data

## ③ Interrupts

1) interrupt can occur anytime (ignored by kernel X)

(↳ ignore: overwritten)  
 input lost

2) interrupt 영향받는 code는 guarded by simultaneous use

## 5.1.4 Dispatcher

CP니 스케줄러가 선택한 Process에게 CPU core의  
 control을 넘기는 역할 (= program)

### Function of Dispatcher

① 문맥교환 (one proc to another)

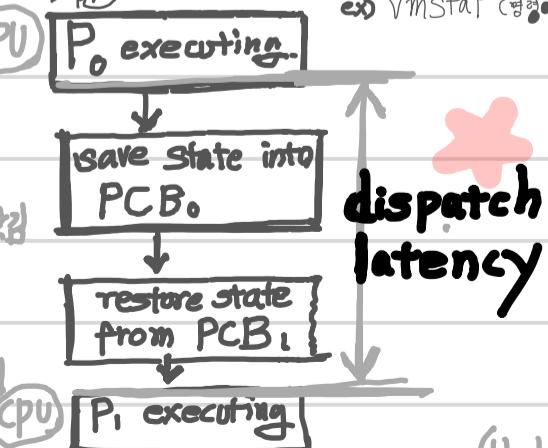
② 사용자 프로그램 전환

③ Jump to proper loc to resume program

: context switching 때마다 흐름으로 최대한 빨리 수행되야 함

Dispatch latency \* : dispatcher가 한  
 프로세스를 맡았고 다른 프로세스를 실행하기 위한  
 시간

ex VMSTAT (영역)



(blocking for)

\* voluntary context switch : CPU 양도, I/O

nonvoluntary context switch : CPU 뺏김, (preempt by H prior)

## 5.2 Scheduling Criteria

ready Q의 process → select → CPU 할당  
→ 실행

### (1) CPU UTILIZATION

: keep CPU busy, (real sy: 40% lightly loaded  
(ex) command Linux -top) 90% heavily loaded

### (2) Throughput 처리량

: # of completed proc per time unit

### (3) Turn around time 처리시간

: how long it takes to execute the proc

ready-Q 대기시간, 실행시간, 10시간 포함

(internal from proc 대기 to completion)

### (4) Waiting time 대기시간

(readyQ에 대기한 시간 합) : 스케줄링 알고리즘  
proc → readyQ에 대기 시간에만 영향 미침.

### (5) response time 응답시간

(output 빨리 만들고, 빠른 대응 시간을 세우고 싶은)

time from request 지향 ~ first response produced

↓ ex) interactive sys (output the response)

maximize CPU utilization, throughput

minimize turnaround + waiting + response +

(most case → optimize 광고 축정과 최적화)

(reasonable, predictable response + ⇒ faster, highly variable)

## 5.3 Scheduling Algo : CPU burst

(readyQ의 어떤 Proc이 CPU core를 할당 // 가정: Core 1)

### I. FCFS First Come First Served 가장 코드 간단

① CPU를 차지한 proc가 CPU를 차지한 쿠션에 PCBS 연결

② FiFo Q : Process가 readyQ에 들어가면 큐 끝에 PCB 연결

CPU free → head의 process가 할당

③ average waiting time is often long, not minimal

④ CPU burst가 매우 짧으면, average waiting time 매우 짧아짐.

\* Gantt Chart : illustrate schedule, proc별 start/finish + proc.

⑤ Convoy Effect 효과 : 모든 proc가单一 매우 짧은 proc

→ CPU를 기다림.

E) CPU bound proc (CPU intensive) ⇒ CPU other proc

→ move to I/O device Ready Q finish I/O, wait short CPU burst → move back to I/O Q

① lower device utilization (idle → 이용률↓)

② non-preemptive, troublesome for interactive sys

### 2. SJF Shortest Job First

① the smallest next CPU burst 처리 (同\*FCFS)

② Process 전자결정 X, CPU burst \*

③ Optimal : 평균 대기시간 최소

④ next CPU burst로 이동할 때의 CPU 스케줄링에 구현 X

⑤ Preemptive SJF = shortest-remaining-time-first SA

: new proc의 next CPU burst ⇒ 실행 중 proc의 남은 CPU burst

\* 이전 CPU burst 측정 결과를 차수별로 하여 미루.

$T_{n+1} = \alpha T_n + (1-\alpha) T_n$

$T_1 = \alpha t_0 + (1-\alpha) T_0$

$T_2 = \alpha t_1 + (1-\alpha) T_1$

↓ 예측된 most recent ... ↓

next CPU burst

\* α 가중치 ( $0 \leq \alpha \leq 1$ )

\*  $T_0$  타우의 초기값

① 상수

② process를 실행하는 데 일반적으로 평균 CPU burst time

### 4 RR round-robin

similar to FCFS  
preemptive

① 시설이 process 간 전환 가능하도록, 신호호출

timeslice, time quantum 같은 틀이 기본적

② readyQ → 첫 번째 Q를 끌어 한개 시간 단위

동안 CPU 할당 시간 단위 interrupt

\* Fifo Q인 대로 yQ에서 Process로 → 각각 timer 발생

→ 프로세스를 dispatch

③ CPU burst < time quantum → voluntarily

CPU release

④ CPU burst > time quantum → timer go off

→ context switch tail of readyQ

⑤ \* average waiting time is long

⑥ readyQ에 대기 중 proc의 time quantum 만큼

⑦ Preemptive \*이다 ★ 한번 끌어온다

⑧ 최대 ( $n-1$ ) \* q 시간 단위 만큼 머기

⑨ time quantum 이렇게 설정하는데 이 알고리즘

⑩ extremely large → same: FCFS

" small → context switch + overhead 발생"

⑪ \* time quantum ≥ context switching

→ \* context switching 0 + Q의 10%, CPU 시간의 10%를 context switching에 사용

⑫ \* CPU burst의 80%는 time quantum

보다 짧아야! ⇒ Proc의 80%는

time quantum 내에 끝내도록 time quantum을 설정!

⑬ 처리시간은 Time Quantum 크기로 관리한다.

most process' CPU burst + ≤ time quantum

→ 평균 처리시간 확장.

Add context-switch time : time quantum

각각, 평균 처리시간이 더 증가한다.

time quantum이 증가해도 process 간의

평균 처리시간은 반드시 항상 유지되는 것을.

process | time

P<sub>1</sub> 6

P<sub>2</sub> 3

P<sub>3</sub> 1

P<sub>4</sub> 7

10.75

3 : (13+6+7+1)

10.5

6 : (6+9+10+1)

time quantum

high priority

interactive IO 처리

real-time processes RR

system processes

interactive processes

foreground process 사용자 상호작용

batch processes

background process 사용자 인수 FCFS로

lowest priority

8 1/2 6 1/2 10

$T_{n+1} = \alpha T_n + (1-\alpha) T_n$

$\alpha = 1/2$

$T_0 = 10$

$T_1 = 8$

$T_2 = 6$

$T_3 = 6$

$T_4 = 5$

$T_5 = 9$

$T_6 = 11$

$T_7 = 12$

$\alpha < 1, 1-\alpha < 1$

→ Each successive term has less weight than its predecessor.

→ 후속 항은 이전 항보다 가중치가 작음.

가장 가중치가 큰 항

가장 가중치가 작은 항

→ associated with zt process

integer

### 5. Priority Sche

우선순위 높은 process에 CPU 할당

FIFO

SUFE는 우선순위 스케줄링의 일반적 경우

Priority : 다음 CPU burst의 역할

Internal P & External P

Internal (운영체계 자체에 특징 있는 경우)

: 시간제한, 메모리 요구, 사용한 파일의 수, 10대 CPU 비율

External (OS 외부에서 기준에 정의)

: 프로세스 중요도, 비용, 작업 우선순위, 상대 위치

④ preemptive priority scheduling

Nonpreemptive priority sche: 일시적이

⑤ Problem: infinite blocking, starvation

(\* blocked : ready to run, waiting for CPU)

: 부하가 큰 시스템에서, higher-priority proc

가지도록 하면, CPU를 차지하는 proc

→ 블록 상태에 걸리거나, sys crash나 loose

⑥ Solution: aging, RR + priority S

Aging : 주기적으로 waiting process

의 priority를 1등급 시킬.

ex) priority 127 → 243 → 0

combine RR + Priority S

: 가장 우선순위 높은 process 실행

→ 같은 우선순위 RR

⑦ OCN) search to determine highest p.

6. Multilevel Q.Sche

① 우선순위별로 큐를 따로 운영

= Separate Qs + RR + Priority

② priority 가 장기적으로 활용

→ Process는 실행시간 동안 같은 큐에 존재

③ process를 type에 따라 개별 큐로 분할

개기 위해 사용 가능

④ proc type에 따라 다른 응답 시간 요구

→ 유형별로 자신만의 스케줄링 알고리즘.

⑤ Scheduling among Qs

fixed-priority preemptive sche

(absolute priority) → Starvation 1st

time slice (CPU time) among Qs

ex) \* Foreground - background round robin

: foreground Q - RR로 80%, CPU +

background Q - FCFS = 20% CPU +

high priority

quantum = 8 RR

큐 1 quantum = 16 RR

큐 2 FCFS

low priority

queue 0 queue 1 queue 2

Only when queue 0 is empty it executes processes in queue 1.

• (큐0이 high 레벨일 때만 큐1에 low 있는 process가 실행됨)

Processes in queue 2 will be executed only if queues 0 and 1 are empty.

A process that arrives for queue 1 will preempt a process in queue 2.

• (큐1에 high 도착한 process는 큐2에 low 있는 process를 선점함)

If a process in queue 0 does not finish within time quantum(8),

it is preempted and is moved to the tail of queue 1.

To prevent starvation, a process that waits too long in a lower-priority queue may gradually be moved to a higher-priority queue.

• (우선순위가 낮은 큐에서 오래 기다린 프로세스는 우선순위가 높은 큐로 점차 이동)

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less.

• (8ms 이하의 CPU burst를 갖는 process에게 가장 높은 priority가 부여됨)

Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.

Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.

• (8~24ms인 process도 빨리 서비스 받음; 더 짧은 process보다는 우선순위가 낮음)

Long processes automatically sink to queue 2 and are served in FCFS

order with any CPU cycles left over from queues 0 and 1.

• (긴 process는 큐2로 이동하여 큐0,1로부터 남는 CPU cycle에 대해 FCFS로 서비스 받음)

## 5.4 ThreadSche

(introduced threads to process model distinguishing between user-level, kernel-level thread)

(1) OS의 kernel-level thread가 스케줄링

thread가 user-level thread 관리, 채널 전환을 포함

(2) to run on CPU, user

# 5.4 thread scheduling

## 1. contention scope

- (5) User-level-thread과 Kernel-level-thread의 차이  
: 스케줄 방식에 따른 캐시 PCS, SCS  
**PCS (USER-L-T를 SCHE)**  
Process Contentionscope  
프로세스 내 스레드간 CPU 경쟁
- ① (many-to-one, many-to-many)  
: thread lib가 user-level thread를 available LWP에서 실행하도록 스케줄링
  - ② 다른 proc에 속한 thread간 CPU 경쟁
  - ③ PCS according to priority 를 원칙으로 Proc 선택
  - ④ PCS가 실행중인 thread 선택

### SCS (KERNEL-L-T를 SCHE)

System ContentionScope 시스템내 스레드간 CPU 경쟁

- ① C systems using one-to-one model  
: 어떤 kernel-level thread를 CPU에 스케줄링하려면 → SCS로만 SCHEDULE!
- ② System의 모든 thread간 CPU 경쟁
- ③ thread lib가 available LWP에 스케줄 → OS가 LWP의 kernel thread를 CPU core에 스케줄링
- ④ window, linux, SCS로만 스레드를 스케줄링

## 2. Pthread Scheduling

### (1) POSIX Pthread API

- PCS or SCS 를 사용할 수 있게 API 제공  
(thread 생성 때 PCS/SCS를 선택하는 옵션)
- ex) PTHREAD\_SCOPE\_PROCESS : PCS 스케줄링  
PTHREAD\_SCOPE\_SYSTEM : SCS 스케줄링

```

int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    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); // SCS scheduling policy

    for (i = 0; i < NUM_THREADS; i++)           /* create the threads */
        pthread_create(&tid[i], &attr, runner, NULL);
    for (i = 0; i < NUM_THREADS; i++)           /* now join(wait) on each thread */
        pthread_join(tid[i], NULL);
}

void *runner(void *param) { /* Each thread will begin control in this function */
    /* do some work ... */
    pthread_exit(0);
}

```

이 글을 보고, 어느 스케줄링이 해당하는지 알 수 있다.

/\* set the default attributes \*/

thread library

Processor CMT

Operating System View

Figure 5.10 Pthread scheduling API

## 5.5 Multi-Processor Scheduling

- \* multiple CPUs - load sharing - thread parallel
- \* Multi processor : 물리적인 processor 뒤에 With each single core
- System Architectures

Multicore CPUs, Multi threaded cores, NUMA Sys - identical

Heterogeneous multi processing (다른) ↔ Homogeneous

### (1) CPU Schedule in multiprocessor Sys

#### - Asymmetric Multi processing

- ① Single processor (master server) + Other Processor
- All scheduling Decision (I/O processing, sys activities) User code만 실행
- ② Simple (장점) : 한 개 core만 사용자 요구에 접근, data sharing 배제
- ③ 단점 : Master Server의 Bottleneck 가능성이 ↑ 퍼포먼스 ↓

### Symmetric Multi Processing SMP 특징 실행

- ① each processor - self scheduling (multiprocessor 특성)
- (각 processor별 스케줄링이 readyQ 확인, 실행할 thread 선택)

#### ② 스케줄 해야 할 thread Organization (관례)

- 1) 공통 Ready Q : 모든 코어가 공통 Ready Q에 있어 캐시
- 경쟁 조건 발생: 같은 스레드를 스케줄하지 않도록, 스레드가 Q에 대입하지 않도록 보장
- 2) 개별 thread Q : 각 Processor가 개별적인 thread Q를 가짐
- (Per core nm Q) : 각 Processor가 개별 thread Q에서 쓰리드 스케줄.
- 쿠모 유도 인한 경쟁 조건이 없음 : more efficient use of cache memory
- ③ 모든 Processor들은 workload 균등화하는

### MultiCore Processor 의

### (2) Memory Stall

- ① Memory Stall : processor가 메모리 접근 시 data가 not-available 해서 기다리며 시간낭비함  
(CPU로이가 다른 일의 X, 기다리기)

- ② 이유 : 현재 Processor는 memory 빈대 예쁘고, Cache miss 때문 (Accessing data, not in cache memory, cache data가 없는 경우 ⇒ memory에서 가져와야 함)

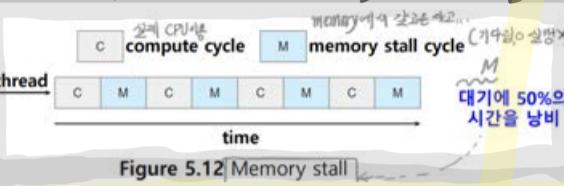


Figure 5.12 Memory stall

### Dual-threaded Processing Core

The execution of thread 0 and the execution of thread 1 are interleaved

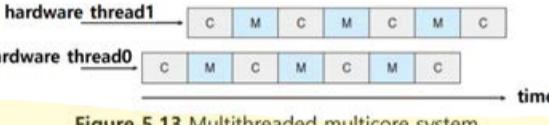


Figure 5.13 Multithreaded multicore system

SMT = 하이퍼스레드

: core에 여러 스레드  
(Modern Hardware Design)

### remedy

### (3) Multithreaded Processing Core

- ① 몇 가지 hardware thread를 한 core에 할당 (한 core에 할당되어 쓰이는 thread)

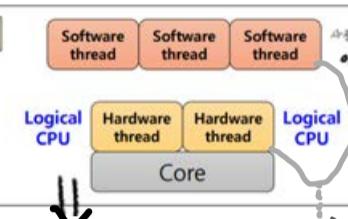
- ② Memory Stall remedy (\*stall = 증발) core가 다른 한 hardware thread가 예상 기다리게 되면 → 코어가 안된다.

### ③ OS Perspective 관점

hardware thread는 각각의 architectural state 를 가짐 ⇒ instruction pointer, register set

(예들이 실제 CPU Core에 할당되어 실행됨)

OS는 hardware thread를 logical CPU로 보고 예상됨



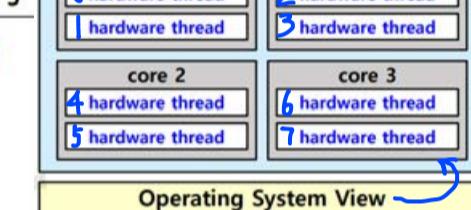
### CHIP MULTITHREADING 테크닉

: Processor가 4개 computing core 포함, 각 core가 2개

하드웨어 thread 포함, ⇒ OS 관점에서 (logical) CPU 8개

사용자 software thread 이를 2개 쟁탈되어

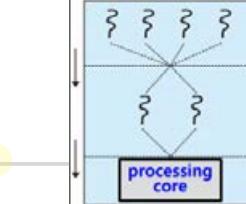
Processor CMT



### Operating System View



### (4) 멀티쓰레딩 방법 2개

software thread  
hardware thread이해하고  
할고

(for dual-threaded processing core)

## 1) 2 Different level 스케줄링

first level scheduling decision by OS

- ① OS가 hardware thread에게 할당될 software thread를 선택하는 단계

- ② OS가 스케줄링 Algo를 선택

2nd level of scheduling

- ① 각 CORE가 실행할 hardware thread를 선택하는 단계

(여기서 각각이 실행할 hardware thread를 결정하는 단계)

- ② Simple RR : hardware th → processing core

- ③ 하드웨어 thread의 dynamic urgency value

부여 (0: lowest urgency ~ 7: highest)

5 different events trigger thread switch

(event! → th switching logic compare urgency of 2 threads → select highest urgency → processor core)

아니요!

## 2) OS의 스케줄러가

자원 공유하고 있다면, 효과적 스케줄링 가능

- a) CPU has 2 processing core with 2 hardware threads

- ① 각각, 2 소프트웨어 thread가 system all running

→ 동일 core or 서로 다른 core all run 가능

- ② 같은 core에 schedule되면, 자원 공유로 끌려갈 수 있음.

- ③ OS가 자원 공유를 한다면,

share하지 않는 logical processor에 share

### core를 걸게됨

처리 시간이 많이 걸리는 이벤트인 경우

a thread executes on a core until a long-latency event, such as a memory stall occurs.

(이벤트 발생 전까지 한 core에서 실행됨)

The core must switch to another thread to begin execution.

(다른 스레드를 실행)

The cost of switching between threads is high

since the instruction pipeline must be flushed before the other thread can begin execution on the processor core.

(core에 다른 스레드가 실행되기 전에 명령 pipeline이 비워져야 함 → switching cost 높음)

Once this new thread begins execution, it begins filling the pipeline with its instructions.

(새 스레드 실행 시, pipeline이 새 명령어들로 채워짐)

### ② Fine-grained (or interleaved) multithreading 더 짧은 단위

switches between threads at a much finer level of granularity.

typically at the boundary of an instruction cycle.

(instruction cycle 경계에서 switch가 일어남)

Architectural design of fine-grained systems includes logic for thread switching

(구조적 설계에서 스레드 교환 회로를 포함 → 비용이 적음)

As a result, the cost of switching between threads is small.

→ 이렇게 있다 정도로 알면됨

104

### (5) Multicore Processor 세팅

피지컬 코어의 자원들 (캐시, pipelines)

하드웨어 쓰레드간 공유되며 약간

따라서, processing core는 한 번에

하나의 hardware thread만 실행됨

그래서, 멀티쓰레드 multicore Processor는

캐시 (시화물) 2개 | level의 스케줄링 필요

104

### ① software threads 선택 (scheduling)

### ② hardware threads 선택 (logical processors)

logical CPU

### 스케줄링 정리

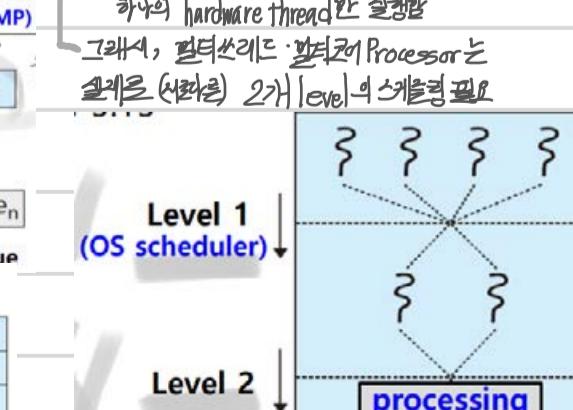


Figure 5.15 Two levels of scheduling (dual-threaded processing core)

Figure 5.10 Pthread scheduling API

SMP Symmetric Multi Processing

Processor CPU 코어

운영체계

SMP 시스템이 여러 물리적 Processor

들을 제공하여 여러 Processors

별로 실행되도록

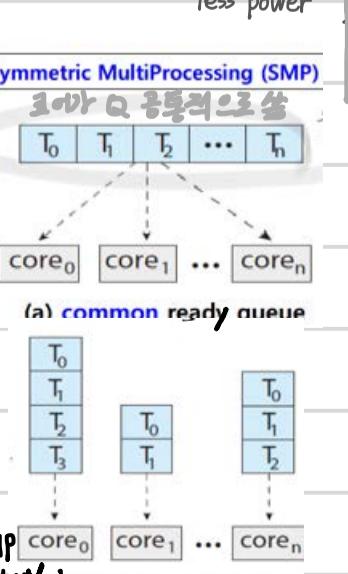
각 core는 구현 상으로 처리: OS에

즉각적으로 logical core가 있는 것처럼

보여진다.

Multicore Processor를 사용하는

SMP 시스템은 faster, less power



# CH4 Threads & Concurrency

## 4.1 Overview

## 4.2 multicore Programming

## 4.3 multithreading Models

## 4.4 Thread Libraries

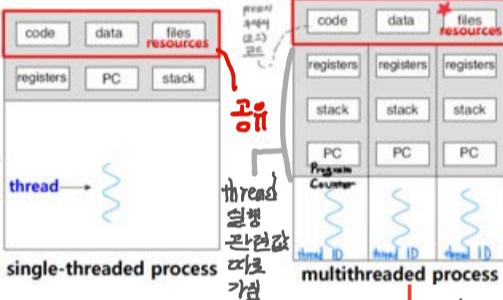
## 4.5 Implicit Threading

## 4.6 Threading Issues

## 4.7 Operating System Ex

### 4.1 Overview : Thread

- ① thread: process 여러개 할 때 쓰는 것, CPU 사용 기본 단위
- ② 동일 프로세스 속에는 서로 thread들 각각 code, data, resource, file signal 등
- ③ 유닉스: thread ID, PC (Program Counter), register set, stack 등
- 쓰레드마다 다르게 실행되고 있고, 어제에 실행하고 있는 데다 다른 쪽으로 가는
- ④ Proc가 여러 thread가 있어, → 동시에 하나 이상의 task 실행



- ⑤ Ex) App ..... requested to perform similar tasks
- web server ← request webpage, image, sound
- \* single threaded process (one client per time)
- client wait long

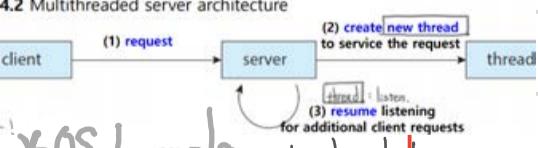
### Old Solution: SingleThreaded Process

- ① Server accept request
- ② Create separate process to service request
- ③ process creation: time consuming, intensive
- 세부에 보면, O(n) head
- ④ 새 Process가 기본 process와 동일한 처리

### New Solution: multithread Process

(one process with multiple thread)

- ①
- ② server create separate threads client request
- ③ request is made → server create new thread, (for the request) and resume listening for additional request



- ① Linux Systems: system boot time이 kernel thread 들이 생성될 때 그들의 task들 → their tasks
- : managing devices, memory management, interrupt handling
- ② 명령어
  - ps -ef (리눅스 셋팅에 실행되는 kernel thread)
  - kthread (PID=2): 모든 커널 thread의 parent

## ⑥ Benefits of multithread Programming

### (1) Responsiveness

사용자에게 빠른 응답성

- ① continued execution, especially user interface ex) 버튼 클릭 후 다음 단계로
- ② interactive App multithread: 프로그램이 계속 실행되어 응답성이 증가
- ③ 사용자가 time-consuming 작업 버튼 클릭 (Separate asynchronous thread ↔ single thread) remain responsive

### (2) Resource Sharing

① Process는 shared memory, msg passing 으로만 자원 공유

- ② thread는 속한 Process의 memory와 자원 공유 (동일 주제간 내부 여러 쓰레드가资源共享 가능)
- ③ resource sharing → faster, lighter

### (3) Economy

① Proc 실행에 따른 메모리와 시스템 리소스 사용량 대비 효율

- ② thread의 실행은 threads → economical: create, context-switching

### (4) Scalability 확장성

① core가 많아지면 single-thread process는 한 프로세스에서 비단 실행될

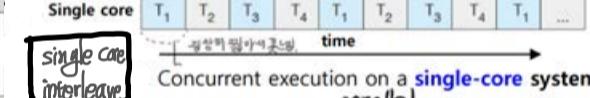
- ② 쓰레드는 다른 core에서 Parallel 실행 가능 → 각각 쓰레드마다 하나의 코어에 할당

## 4.2 MultiCore Programming

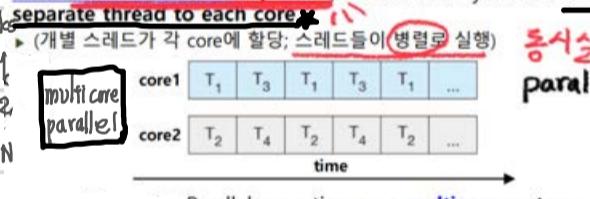
An application with four threads interleave 기회 넘기

On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time, because the processing core is capable of executing only one thread at a time.

▶ (processing core가 한번에 하나의 스레드만 실행, 스레드들은 interleaving됨)



On a system with multiple cores, however, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core.



## 1. Single & multicore system

## 2. Concurrency & parallelism

- ① concurrent system (병행, 별렬X)
  - : several tasks make progress

- ② parallel system (별렬)
  - : several tasks 동시 실행

(\* concurrency without parallel 가능)

## 3. Programming challenge for multicore system

(하나의 APP 안에서 서로 다른 core에 병렬적으로 실행하기 위해 고려해야 할 것)

### ① identifying tasks (task 허용)

: 독립된, parallel task로 나누어지는 영역(영역별로, 이상적으로 병렬적/ 병렬 수행 가능)

### ② Balance 고려

: task들이 공동하게 처리되도록 (작업이 여러 core에 걸쳐서 가능)

### ③ Data Splitting

: App의 task를 나누어주는 것(여기서 task가 사용하는 data도 core들에 나누어주어야 함)

### ④ Data Dependency (작업에 대한 Data Write X)

: task에서 접근하는 data는 task들 사이에 의존성이 있으니 확인이 필요함

: task가 data dependency가 있는 경우: 프로그래밍은

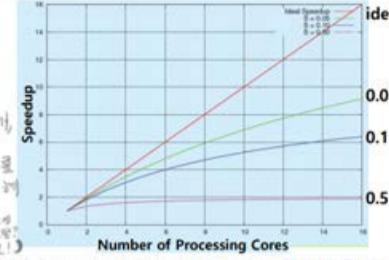
: data dependency를 관리해 주고, task 실행이 일관화될 것을 보장해야 한다.

## 4 Amdahl's Law

Performance  $\rightarrow$  core 개수를 주관으로 높이는 것보다  
AMDAHL'S LAW task를 core에 차례로 실행하는 것과  
parallel components. If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follows:

$$S: \text{Serially} \quad \text{parallel} \quad \text{speedup} \leq \frac{1}{S + (1-S)/N}$$

As an example, assume we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores, we can get a speedup of 1.6 times. If we add two additional cores (for a total of four), the speedup is 2.25 times. Below is a graph illustrating Amdahl's Law in several different scenarios.



One interesting fact about Amdahl's Law is that as N approaches infinity, the speedup converges to 1/S. For example, if 50 percent of an application is performed serially, the maximum speedup is 2.0 times, regardless of the number of processing cores we add. This:

## 5. 2 types of Parallelism

### ① Data Parallelism: Distribution of Data across multiple core

### ② Task Parallelism: "if tasks"

→ 2 types of Parallelism은 서로 배타적이지 않고, 같은 쓰레드 사용 가능

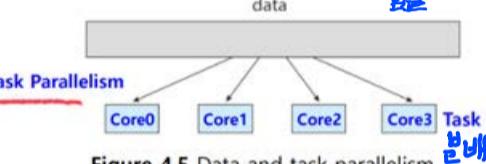
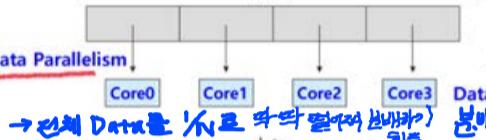


Figure 4.5 Data and task parallelism

### (1) Data Parallelism

: 여러 core에 같은 data의 subset를 보내, 각 core에서 동일 연산을 실행  
(ex) 배열, 반복문의 몇몇 연산

### (2) Task Parallelism

: 여러 core에 task 보내  
distributing not data but tasks(tasks)

: 각 쓰레드는 별도로 고려하지 않음  
각 쓰레드는 동일 data, 다른 data에 대해서 연산

## 4.3 Multithreading Models

### 1. Support For Thread

- ① User thread: 커널 관리 없이 실행, 커널 지원 없어 관리, user-level thread Lib 지원

- ② Kernel thread: (ex: window, linux, mac)  
OS가 직접 지원, 관리

- ③ User thread + kernel thread 사이에 관계가 존재하는 경우

- : many to one, one to one, many to many relationship

### (1) Many-to-one Model

- ① map many user-level thread to 1 kernel thread

- ② 한 번에 한 쓰레드만 커널에 접근, 병렬 수행 X

(한 thread blocking system all → 전체 proc block)

③ Thread management → user space의 thread Lib로

④ Multiple processing core가 이를 활용 X (기억안쓰임)

### (2) One-to-One Model

- ① user, kernel thread 1대 1로 map

- ② 다른 thread → Parallel로 실행 가능 (multiprocessor)

(单一의 blocking syscall을 하더라도 다른 스레드가 실행 가능)

- ③ many-to-one보다 greater concurrency

- ④ user thread 생성과 kernel thread 생성

→ kernel thread 수 증가 → 시스템 부담

(ex: Linux, OS, Window ... 여러 프로그램에서 사용 가능)

### (3) Many-to-Many Model

- ① user-level threads >= kernel threads

kernel thread 수는 특정 App이나 기계마다 달라짐

(ex: 어떤 1-1 thread mapping이 있는 경우)

추가로 커널 코어가 필요할 때.

### 2. Effect of Many-to-Many on Concurrency

- ① many-to-one 모델 단점

- 프로그램마다 원하는 만큼 thread 생성 가능

- 병렬 수행 X, 한 번에 하나의 스레드를 큐에.

### ② one-to-one 모델 단점

: 너무 많은 thread가 생성되지 않도록 신경써야함.

### ③ many-to-many 단점

커널 코어는 한 번에 만큼만 thread 생성 가능

→ 사용되는 kernel thread가 병렬로 수행시,

스레드가 blocking sys call을 수행시,

kernel이 다른 스레드를 실행할.

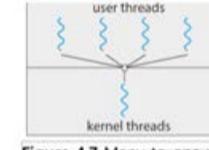


Figure 4.8 One-to-one model

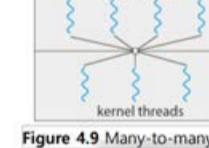


Figure 4.9 Many-to-many

\*thread = logical core



# CH3. Process

## 3.1 Proc Concept

## 3.2 Proc Scheduling

## 3.3 Operations on processes

## 3.4 Interprocess Communication

## 3.5 IPC in Shared-Memory Sys

## 3.6 IPC in Message-Passing Sys

## 3.7 Ex of IPC system

## 3.8 Communication in Client-Server System

### \* Process Management

① Process : 실행되는 프로그램, 흘러가는 process의 일정.

② Process는 현재 실행 중인 것, 사용하는 자원을 갖는 (CPU time, Memory file, I/O devices)

③ 프로세스는 다른 사용자에게 작업을 하는 단위를 차지하는 (시스템은 프로세스의 관점)

\* OS process와 UserProcess 동시에 실행된다 : OS process는 커널에서 실행되는 리소스(Sys Code)를 실행하는 process

UserProcess는 사용자 program을 실행하고 있는 부분이 UserProcess

④ core에 있는 (P), 동시에 Parallel하게 thread하는 개체가 여러 개 core에서 동시에 실행.

## 3.1 Process Concept

① CPU activities = Process = CPU를 사용하는 다양한 작업

① batch system  $\rightarrow$  time sharing system

(기억을 일정, 일정 처리 시스템, 시설화 시스템)

② single-user system에서도 멀티프로세싱 환경이 실행 가능.

③ embedded 환경(multitasking 지원 X)에서도 멀티프로세싱 환경을 지원해야.

## 2) Process

① Current status of process ('activity')

program counter(PC)가 있음

(주로 실행해야 하는 경쟁에 보여지는 가짐)

processor의 register(P 실행하는 Data)

- 보유한, 계산 결과값을 저장)

② process의 memory layout.

① text section : executable code

② data section : global variable

③ heap section : memory that is (run time)

dynamically allocated

④ stack section : temporary data structure

$\rightarrow$  push & pop invoking function

(function parameter, return address, local variable)

max ↓ stack ↑ 할당

↑ heap ↓ 할당

global variable

(OS) 0 text ↓ executable code

(3) Stack, heap 등의 grow, shrink

OS must ensure do not overlap each other

(4) program is passive entity : 프로그램 자체로

정적 어드레스를 포함하는 파일 (실행 가능한 파일)

(5) Process is active entity : 관리되는 리소스

다음 실행 명령어를 나타내는 program counter를 가짐

(6) 실행 파일이 메모리에 로드될 때, 프로그램이 process가 된다.

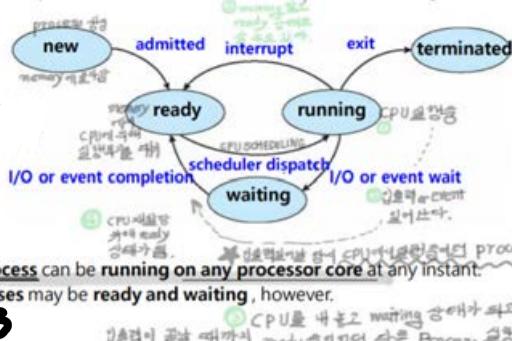
(7) 두 Process가 동일 program과 관련

$\Rightarrow$  두 개의 독립된 실행 환경으로 인해 (동일 코드)

$\Rightarrow$  txt section이 equivalent 해도

data, heap, stack sections are vary

(8) process 실행은 process 실행 가능



## (3) multiprogramming

① core 주변의 process 수가 많을 경우, process가 wait해야 할

② degree of multiprogramming

: 현재 실행되는 process 수

③ multiprogramming과 time sharing

수개의 실행 가능한 프로그램을 Balance하는 것

(Balancing 위해 프로세스의 일정과 통합하기)

④ I/O bound process, CPU bound process

(I/O intensive) 예상된 사용량

을 예상해 Balance

## 1) Scheduling Queues

OpProc 스텝업  $\rightarrow$  ready(Q)에 삽입

: core에 할당되어 ready / wait

② ready Q는 Linked List 구조로

(들이 오는 순서대로 연결됨)

: 큐의 head는 첫 PCB의 모티브 가짐,

각 PCB는 각각 PCB에 대신 그 모티브

PCB3 → PCB4 → PCB5

head tail

$\leftarrow$  ready Q / wait Q

## 2) Wait Q

① CPU 할당  $\rightarrow$  실행  $\rightarrow$  종료

interrupt or wait for event

(특정 event I/O에 대해서 wait)

CPU process  $\rightarrow$  device에 I/O request

$\rightarrow$  process가 wait  $\rightarrow$  wait Q

(device가 process보다 속도 느림)

② event가 처리 요청  $\rightarrow$  wait Q에 삽입

## 3) Process가 CPU 사용

위해 어떻게 스케줄링?

(\* O : resource that serve the queue)

④ CPU-scheduling info : process priority, pointers to scheduling queues, scheduling parameters

⑤ Memory-management information

: load된 process와 memory 접근 여부

⑥ Accounting info : 해당 process를 user가

얼마나 사용하고 있는지  $\rightarrow$  유료화되는 경우

꼭 필요하다.

⑦ I/O status info : 1) process가 어떤 file

로부터 data를 읽어온다, 쓴다.

(어떤 file/장치에서 읽어올 때 어떤 file에 씌어가는지)

## 5) Thread

\* core 1개 가짐

(1) process :单一 thread를 실행하는 program

(2) single thread : 하나의 친구가 하나의 task를

실행함 (thread가 하나라면)

ex) 코딩에서 혼자 앱하고 혼자 같이 동시에 실행 X

(3) 한 process가 여러 실행 thread를

가지고 한 번에 하나의 task를 실행할

$\Rightarrow$  thread supportSystem에는 PCB가

각 thread에 대한 정보를 포함하도록 확장됨.

## 4) Process Control Block

= task control block : process와 관련된

제어와 정보를 가지고 있는 block (작고 크다)

① Process State 상태

② Program Counter

(현재 값, 저장 공간)

③ CPU registers

: 실제 실행 후, 영향이

걸친 하위 및 필요한 등급의

값들을 register에 저장

$\Rightarrow$  interrupt 발생하면,

해당 Process가 저장하고 있던 그대로로 돌아오기, state info가 ... 저장되어 있어

reschedule 후 실행

(Process가 실행되는 동안 PCB에 의해)

④ CPU-scheduling info : process priority,

pointers to scheduling queues, scheduling parameters

⑤ Memory-management information

: load된 process와 memory 접근 여부

⑥ Accounting info : 해당 process를 user가

얼마나 사용하고 있는지  $\rightarrow$  유료화되는 경우

꼭 필요하다.

⑦ I/O status info : 1) process가 어떤 file

로부터 data를 읽어온다, 쓴다.

(어떤 file/장치에서 읽어올 때 어떤 file에 씌어가는지)

## 3.2 Proc Scheduling

multiprogramming의 목표 : maximize CPU utilization

(한정된 process를 실행되도록하여 CPU 사용률 최대화)

time sharing의 목표 :

process가 여러 CPU로 차례로 switch  $\rightarrow$  program이

interact 함

(1) CPU scheduler가 여러 Process를 차례로 플레이

한번에 하나의 Process 실행

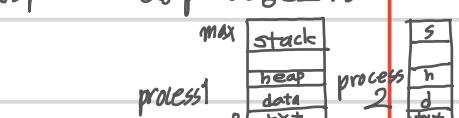
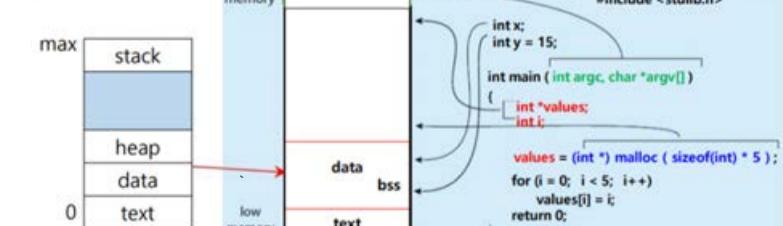
(2) System with Single CPU core  $\leftrightarrow$  multi core sys

only 1 process at a time

multiple process at a time

The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.

A separate section is provided for the argc and argv parameters passed to the main() function.

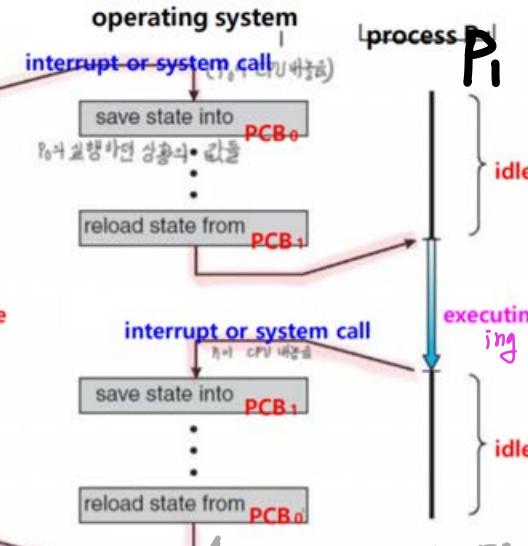


The GNU size command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is memory, the following is the output generated by entering the command size memory:

text 1158 data 284 bss 8 dec 1450 filename memory sum (txt, data, bss)

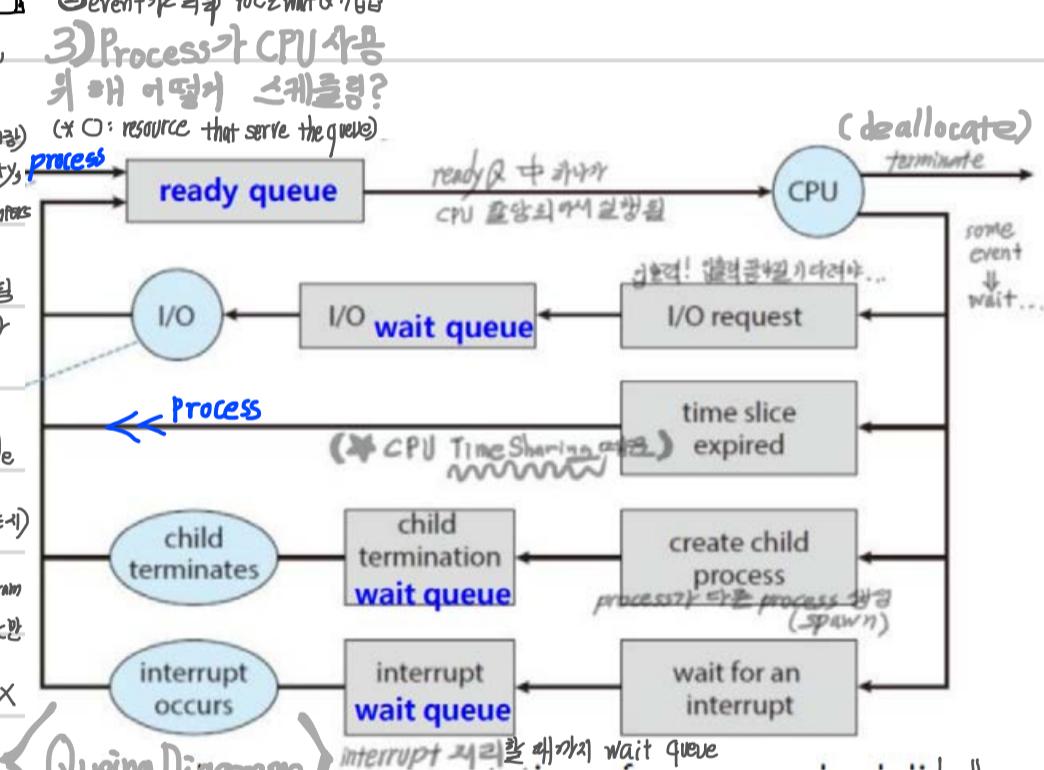
## process P0

## process P1



⑤ context switch time은 overhead이다.  
작게 하려면, system do no useful work while switching  
context switching 시간은 하드웨어 차원에 크게 좌우됨  
OS가 복잡할수록 context switching 시간은 많아짐

© 2020, 손도희, all right reserved



## 4) Swapping

개념만 가르기!

→ swap out 디스크 디스크 swap in 디스크

$\leftarrow$  memory에 process가 놔두면서 load 함

Swapping

① 메모리가 초과 사용되어 해제되어야 할 때면 플로우

② 메모리에 proc를 제거해 다음화 정도 degree of Multiprogramming 낮추는 것

③ 나중에 다시 메모리로 재사용이 가능한 경우에 재설정

- Context는 process의 PCB에 구현되어 있음.
- “는 value of CPU register, process state, memory management info”
- CPU 코어를 다른 Process로 switch할 때 context switch 됨
- state save of P : kernel이 PCB에 old process의 context에
- state restore : 실행할 new proc의 saved context를 load 함 of different P

\*parallel != concurrent: 여러 개 Process가 동시에 실행되는 행정실무

## 3.30 Operations on Process

- 프로세스들은 동시에 Concurrently 수행되며 동시에 생성/삭제
- 프로세스는 실행 중 터미널로 사용자 Proc (child) 생성 가능
- PID: 각 프로세스의 고유한 정수 (unique integer)
- kernel 내부 index로 접근 용으로 사용 가능

### 1) Linux OS의 ProcessTree (Process = Task)

- ① systemd가 모든 process를 root parent process (부모는 user process, 자식은 child process)
- ② 부팅되면 systemd로부터 나머지 Process들이 생성 (\*: daemon) (후에 세운 제3자 Process)
- PS -el PStree: 명령어

### 1. Process Creation

- ① 자식 Proc 생성 당시 Child Proc가 자원을 요구함.

- ① Child Proc가 OS로부터 자원 할당 받음
- ② 부모 Proc 자원의 Subset으로 계약됨
- 부모가 자식들간 자원 분할 / 자식들이 자원 공유
- 이유: Proc가 너무 많은 child 생성으로 → 사전 예약 부족

- ② Child Proc에 필요한 Data 전달

- ① 자식 생성 시, Parent P가 초기 Data 전달
- 스크립트 파일에 파일 hw.c 내용 출력 Process
- 생성 시, Parent P로부터 파일명을 input으로 받음.
- name of output device 출력 장치로

- ② OS가 child P(생성 시)에게 resource 전달

- (→ 추가 자원 필요)

ex) 자식은 두 Open file 만 전달 받아들여야 한다면 전달

- ③ 부모/자식 프로세스의 실행

- (자식 Process 실행 후에는 자원 반환해야 함)

- ① 부모/자식 병렬 실행 (독립화, concurrently 실행)

- ② 부모가 자식이 종료될 때까지 기다림 (Wait 했다가)

- wait child terminate

- ④ 부모/자식 Process의 주소공간 사용 \*

- 부모가 자식을 생성하면, 생성되는 시스템 공간 부여 해서

- 가져온 Address Space → 자식 간의 new program load

- ① 자식이 부모의 Duplication (fork() system call)

- 자식이 부모와 동일한 프로그램과 Data를 가짐.

- ② 자식 Proc가 자신에게 load되는 새 프로그램을 가짐

- (자식 간의 새 프로그램 실행할 때 load)

→ 그래서 그 새 프로그램 실행하기

### 2. How

#### 1) PID = fork()

- 부모 Proc가 fork() 하는 System call, 새 자식 Proc 생성

- ① (새 자식 Proc는 부모의 주소공간 복사본 가져옴)

- (→ new proc는 original 주소공간의 복사본으로 구성됨)

- ② parent와 child가 초기 통신하기 함.

- ③ fork() 후 두 프로세스는 계속 실행됨

- 즉, concurrently 어떤 부모가 자식 wait

- \*pid = fork();

- 부모 Proc에서는 pid를 부모 Proc에서는 pid를 return, 자식 pid를 return

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

- exit(0);

- <parent code>

- wait(pid);

#### How Fork Works

- int pid = fork();

- if (pid == 0) {

- <child code>

