

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

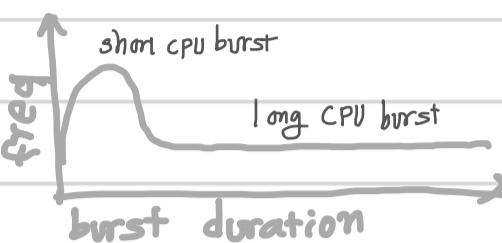
$$= \text{CPU exec} + \text{I/O wait}$$

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

= Process의 I/O burst가 끝나자
 CPU burst가 끝나자 이어서
 고려 해서 Scheduling *

(2) frequency curve & CPU burst

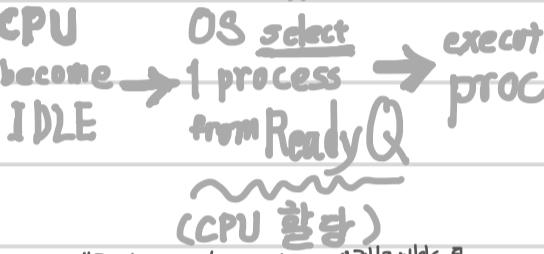
: duration of CPU burst tend to have
 "frequency curve"



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

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

5.1.2 CPU SCHEDULER



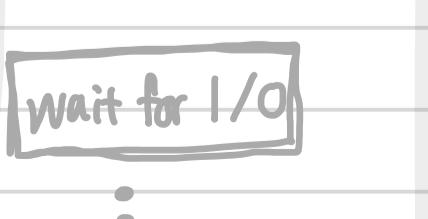
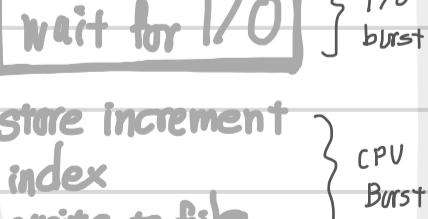
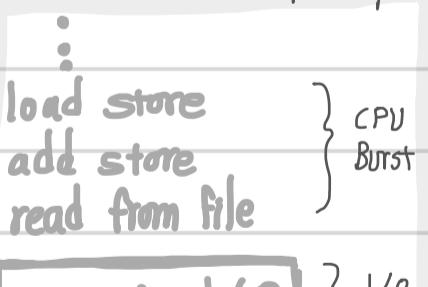
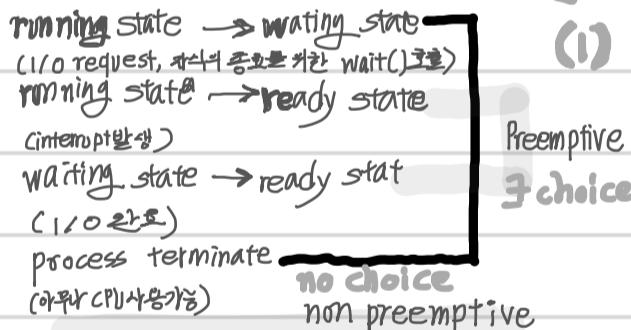
→ CPU 스케줄링은 readyQ에서 선택을 하거나
 미션리스트에서 process로부터 Select → CPU 할당.

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

* Ready Q 내의 레코드들은 Process의 PCB임
 (Process Control Block) *

↳ ready Q에서 스케줄링 Algo에 따라 실행할 대상

5.1.3 Preemptive Non P Scheduling



(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 커널설계에 영향을 줌.
 예) syscall 처리 동안, 커널은 Proc를 위한 활동으로 바뀜
 (change important data 000 kernel modifying 00000 process preemption)
- 2) Design of OS Kernel

nonpreemptive kernel

커널이 syscall complete를 기다리며,
 그 프로세스가 block 되기를 기다림.

kernel structure is simple
 (커널 data structure가 inconsistent상태에서 신뢰성X)
 real time computing 지원X (실시간X)

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

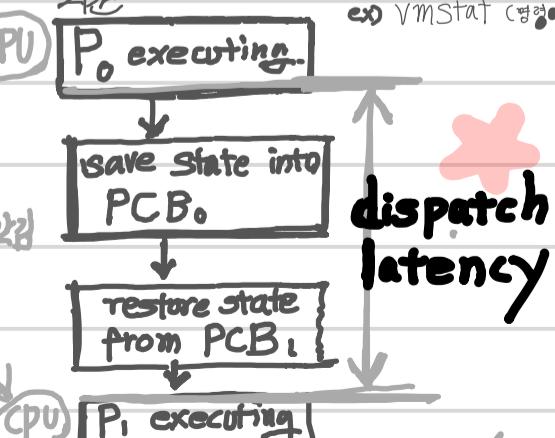
5.1.4 Dispatcher

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 초월적으로 최대한 빨리 수행되야
 Dispatch latency * : dispatcher가 한
 프로세스를 맡았고 다른 프로세스를 실행하기 위한
 시간

ex VMSTAT (영역)



* 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 대응을 통해 시간 줄이기)

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 으로 (同 FCFS)

② Process 전략으로 X, CPU burst *

③ Optimal : 평균 처리시간 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

3. 現代 SJF & Approximate SJF

: next CPU burst는 pre(CPU burst)의 길이가 비슷하다.

* 이를 CPU burst 측정 결과를 적용해 예측.

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

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

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

↓
예측은 most recent ...
next CPU burst

* α 가 증가 (0 ≤ α ≤ 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 → 전환 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%로,

CPU 시간의 10%를 context switching에 사용

⑫ * CPU burst의 80%는 time quantum

보다 짧아야! ⇒ Proc의 80%는

time quantum 내에 끝내도록 time quantum 설정!

⑬ 처리시간은 Time Quantum 크기로 관리한다.

most process' CPU burst + ≤ time quantum

→ 평균 처리시간 확장.

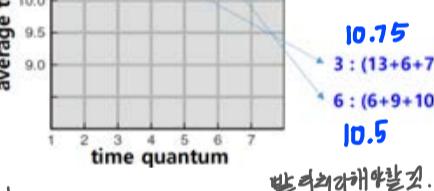
Add context-switch time : time quantum

각각, 평균 처리시간이 더 증가한다.

time quantum이 증가해도 process 실행의

평균 처리시간은 반드시 항상 유지되는 것을.

process	time
P ₁	6
P ₂	3
P ₃	1
P ₄	7



10.75
3 : (13+6+7+1)
6 : (6+9+10+1)
10.5

highest priority
interactive IO 처리량

real-time processes RR

system processes

interactive processes foreground process 사용자상호작용

batch processes background process 사용자인식

lowest priority FCFS

발전기해야 할 것.

interactive IO 처리량

foreground Q - RR로 80%, CPU +

background Q - FCFS = 20% CPU +

high priority quantum = 8 RR

큐1 quantum = 16 RR

큐2 FCFS

low priority

8 1/2 6 1/2 10 $\alpha = 1/2$

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

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

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

↓
예측은 most recent ...
next CPU burst

* α 가 증가 (0 ≤ α ≤ 1)

* T_0 타우의 초기값

① 상수

② process를 실행하는 데 일반적으로 평균 CPU burst time

5 PrioritySche

integer

similar to FCFS preemptive

① 중요도 높은 process에 CPU 할당

② SJF는 중요도의 스케줄링의 일반적 경우

Priority : 다음 CPU burst의 역할

③ Internal P & External P

Internal (운영체계 내부에 있는 큐)

: 시간제한, 메모리 요구량,

사용한 파일의 수, 10대 CPU 비율

External (OS 외부에서 가진 큐)

: 프로세스 종료, 비용, 작업 완료, 성과 보상

④ preemptive priority scheduling

Nonpreemptive priority sche: 일시적이거나

⑤ Problem: infinite blocking, starvation

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

: 부하가 큰 시스템에서, higher-priority proc

가지도록 하면, CPU를 차지하는 proc

→ 블록되거나 시스템이 막힐 수 있음

⑥ Solution: aging, RR + priority S

Aging: 주기적으로 waiting process

의 priority를 1등차 시킬.

ex) priority 127 → 243 → 0

combine RR + Priority S

: 가장 우선도 높은 process 실행

→ 같은 우선도의 RR

⑦ O(n) search to determine highest p.

6 Multilevel Q.Sche

① 우선순위별로 큐를 따로 운영

= Separate Qs + RR + Priority

② priority가 높아질수록

→ Process는 실행시간동안 같은 큐에 존재

③ process를 type에 따라 개별 큐로 분할

개별로 사용 가능

④ proc type에 따라 다른 실행 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 queue

: foreground Q - RR로 80%, CPU +

background Q - FCFS = 20% CPU +

high priority quantum = 8 RR

큐1 quantum = 16 RR

큐2 FCFS

low priority

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

• (큐0이 high 빌 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 빠리 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.

• (길 process는 큐2로 이동하여 큐 0,1로부터 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

5.4 thread scheduling

1. contention scope

(5) User-level-thread과 Kernel-level-thread의 차이
: 스케줄 방식에 따른 캐시 PCS, SCS
Process Contentionscope
PCS (USER-L-T를 SCHE)
Contenitonscope
프로세스 내 스케줄링 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 스케줄링 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 스케줄링 → OS가 LWP의 kernel thread를 CPU core에 스케줄링
- ④ window, linux, SCS로만 스케줄링

2. Pthread Scheduling

(1) POSIX Pthread API

PCS or SCS 를 사용할 수 있게 API 제공
(thread 실행할 때 PCS/SCS를 선택하는 방향성)
ex) PTHREAD_SCOPE_PROCESS : PCS스케줄링
PTHREAD_SCOPE_SYSTEM : SCS스케줄링

```
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

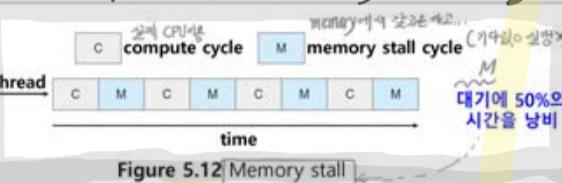
    for (i = 0; i < NUM_THREADS; i++) // error
        pthread_create(&tid[i], &attr, runner, NULL);
    for (i = 0; i < NUM_THREADS; i++) /* now join(wait) on each thread */
        pthread_join(tid[i], NULL);
}

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

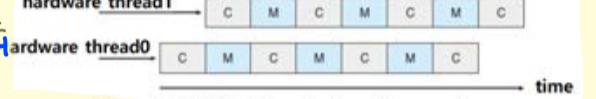
MultiCore Processor 의

(2) Memory Stall

- ① Memory Stall : processor가 메모리 접근 시 data가 not-available 상태로 기다리며 시간낭비함
(CPU로이가 다른 일의 X, 기다리기)
- ② 이유 : 한 core가 memory 빌드에 바쁘고, cache miss 때는 (accessing data, not in cache memory, cache data가 없는 경우 → memory에서 가져와야 함)



Dual-threaded Processing Core
The execution of thread 0 and the execution of thread 1 are interleaved



remedy

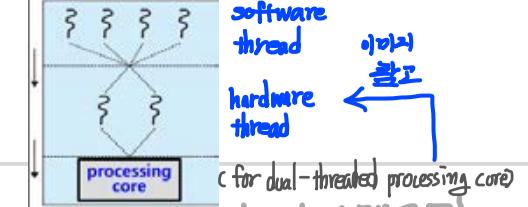
: core에 여러 스레드
(Modern Hardware Design)

(3) Multithreaded Processing Core

- ① 몇몇 hardware thread를 한 core에 할당 (한 core에 할당되어 쓰이는 thread)
- ② Memory Stall remedy (*stall = 증발) core가 다른 한 hardware thread가 필요하지 않으면 → thread로 전환 → 코어가 안 훈련.

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



1) 2 Different level 스케줄링

first level scheduling decision by OS

- ① OS가 hardware thread에게 할당될 software thread를 선택하는 단계
- ② OS가 스케줄링 Algo. 선택

2nd level of scheduling

- ① 각 CORE가 실행할 hardware thread를 선택함
(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)
여기 포함

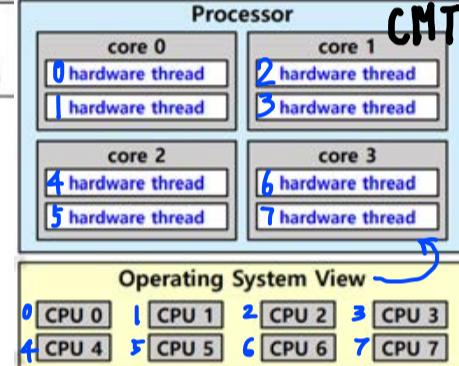
2) OS의 스케줄러가 자원 공유를 안다면, 효과적 스케줄링 가능

- a) CPU has 2 processing core with 2 hardware threads
- b) 1번, 2 번 스트레이트 thread가 system all running

- 통일 core or 서로 다른 core가 run 가능
- ② 같은 core에 schedule되면, 자원공유로 끄려질 수 있음.
- ③ OS가 자원공유를 안다면, share하지 않는 logical processor에 share

CHIP MULTITHREADING 테크닉

: Processor가 4개 computing core 포함, 각 core가 2개 hardware thread 포함 ⇒ OS 관점에서 (logical) CPU 8개



(4) 멀티쓰레딩 방법 2가지

① Coarse-grained multithreading

- ▶ 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

- ▶ switches between threads at a much finer level of granularity.
- ▶ typically at the boundary of an 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.

더 많은 텁 term

104

5.5 Multi-Processor Scheduling

* multiple CPUs → load sharing → thread parallel
* Multi processor : 물리적인 processor 끼리 with each single core
System Architectures

Multicore cpus, Multi threaded cores, NUMA Sys - identical
Heterogeneous multi processing (heterogeneous) ↔ 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 (간접) : 한 개 core만 사용자 주제로 접근, data sharing 배제

- ③ 단일 : 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 들어가지 않도록 보장

2) 개별 thread Q : 각 Processor가 개별적인 thread Q를 가짐

→ 큐 공유로 인한 경쟁 조건 없음 : more efficient use of cache memory

- ③ 개별 thread Q after workload 나뉨

④ 모든 Processor들은 workload 균등화하는 Balancing Algo 사용

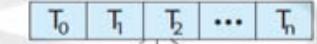
SMP Symmetric Multi Processing

* Processor CPU 쪽
운영체계의 멀티프로세서를 제공하여 여러 Process들이 병렬로 실행되도록 한다.
각 core는 구현 상으로는 OS에 대해서 logical core가 있는 것으로 보여진다.

Multicore Processor를 사용하는 SMP 시스템은 faster, less power

Symmetric MultiProcessing (SMP)

1) 각각 Q 공통적으로 실행



(a) common ready queue



(b) per-core run queues

피지컬 코어의 자원을 (캐시, pipelines)는

하드웨어 츠리드간 공유되어야 함

→ 따라서, processing core는 한 번에

하나의 hardware thread만 실행됨

그래서, 멀티쓰레드 멀티프로세서는

스레드 (사화를) 2개 Level의 스케줄링 필요

© 2020, 손도희, all right reserved

① software threads 선택

(scheduling queue)

② hardware threads 선택

(logical processors)

logical CPU

스케줄링 정리

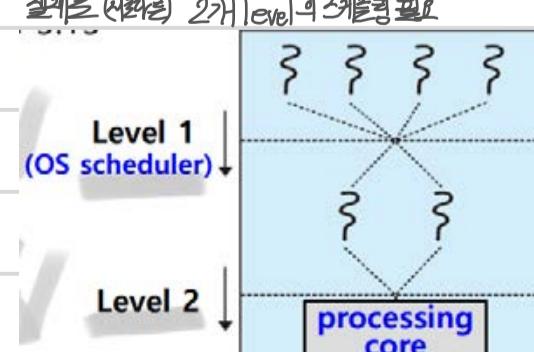


Figure 5.15 Two levels of scheduling
(dual-threaded processing core)

CH4 Threads & Concurrency

4.1 Overview

4.2 multicore Programming

4.3 multithreading Models

4.4 Thread Libraries

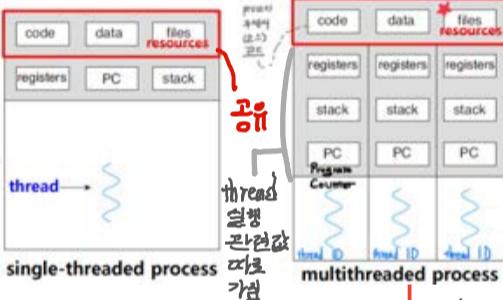
4.5 Implicit Threading

4.6 Threading Issues

4.7 Operating System Ex

4.1 Overview : Thread

- ① thread: process 여러개 할 때 프로세스, 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 자체 프로그램 있고, 어제에 실행하고 있는 데리고 가짐
- 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

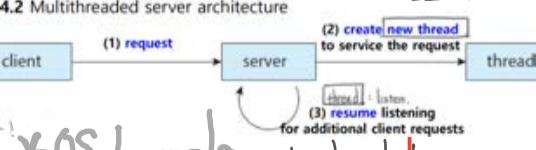
Old Solution: SingleThreaded Process

- 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



* OS kernels - multi threaded

- ① Linux Systems: system boot time이 kernel th 들이 생성됨 → their tasks
- : managing devices, memory management, interrupt handling

- ② 경쟁력
 - : PS -ef (리눅스 셋팅에 실행되는 kernel thread)
 - k-thread (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 단계

- ② interactive App multithread: 프로세스가 처리 일정이 증가

- ③ 사용자가 time-consuming 작업 버튼 클릭 (Separate asynchronous thread ↔ single thread)

- remain responsive

- un responsive

(2) Resource Sharing 자원공유

- 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 경제성

- ① Proc 실행에 따른 메모리와 자원 활용에 대한 부담 ↓

- ② thread의 실행 등록 threads

- economical: create, context-switching

(4) Scalability 확장성

- ① core가 많아도 single-thread process는 한 프로세스에서만 실행됨

- ② 쓰레드는 다른 core에서 Parallel 실행 가능

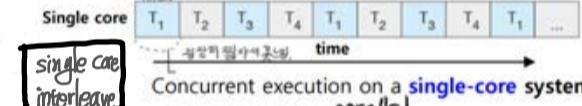
- 각각 쓰레드마다 하나의 코어에 할당

4.2 Multi Core Programming

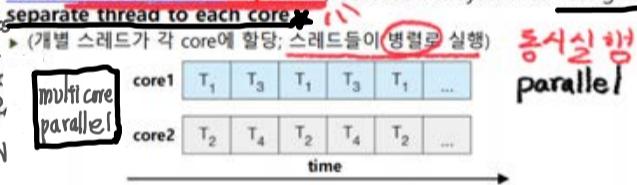
An application with four threads interleaved 가능 넣기

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 interleave됨)



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, 병렬X)

- : several tasks make progress

- ② parallel system (병렬)

- : several tasks execute in parallel

- (* concurrency without parallelism 가능)

3. Programming challenge for multicore system

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

- ① identifying tasks (task mapping)

- : 독립된, parallel task로 나누어지는 영역 찾기, 이상적으로 병렬적/ 병렬 수행 가능

- ② Balance 고려

- : task들이 균등하게 처리되도록 (작업이여도 균등)

- : 기저로 각 작업에 별도 core 사용은 가능 X

- ③ Data Splitting

- : App의 task를 나누어주는 것

- : 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: 프로세스는 data dependency를 수행하도록, task 실행이 동기화될 것을 보장해야 함

4 Amdahl's Law

Performance \rightarrow core 개수를 주관으로 높이는 것보다
AMDAHL'S LAW task를 core에 대해서 실행되는 것과
additional computing cores to an application that has both serial (nonparallel) and 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:

$$S = \text{@serial}$$

(nonparallel)

potential performance

speedup

$\leq \frac{1}{S + (1-S)/N}$

ideal

speedup

1.6

times

2.2

times

2.0

times

1.5

times

1.3

times

1.2

times

1.1

times

1.0

times

0.9

times

0.8

times

0.7

times

0.6

times

0.5

times

0.4

times

0.3

times

0.2

times

0.1

times

0.0

times

ideal

speedup

1.0

times

0.5

times

0.25

times

0.125

times

0.0625

times

0.03125

times

0.015625

times

0.0078125

times

0.00390625

times

0.001953125

times

0.0009765625

times

0.00048828125

times

0.000244140625

times

0.0001220703125

times

0.00006103515625

times

0.000030517578125

times

0.0000152587890625

times

0.00000762939453125

times

0.000003814697265625

times

0.0000019073486328125

times

0.00000095367431640625

times

0.000000476837158203125

times

0.0000002384185791015625

times

0.00000012020928955078125

times

0.000000060104644775390625

times

0.0000000300523223876953125

times

0.00000001502616119384765625

times

0.000

4.4 Thread Lib

(ThreadLib : 개발자에게 스레드를 생성/관리하기 위한 API를 제공)

(1) ThreadLib 2가지 구현방법

① User-level threads library

[커널스페이스], User Space Only threadlib를 제공
할수 있는 User Space에서의 local 함수로, SysCall X

② Kernel-level thread Library

[OS에 의해 직접 지원하는 kernel-level lib 사용]

(Trap, Interrupt 일어나야 하는 경우)

lib를 통한 API에서 할수 있는 SysCall을 가짐

1. 2 strategies

for creating multiple threads

(1) Asynchronous Threading 비동기

① parent는 child 생성과 동시에 실행하기

② parent와 child는 동시에/독립적 실행

③ 독립적 이므로 data sharing이 가능하지 않음

(2) Synchronous Thr 동기

① parent must wait for children terminate

② 각각의 스레드들은 concurrently 실행되고,

종료되면 parent가 수행됨

③ 각각의 스레드는 parent와 join함수,

모든 자식들이 join되어야 parent가 실행되거나

④ 스레드는 상당한 베타 공유가 필요함.

(자식이 처리한 결과 Data → 부모가 처리)

ex parent thread가 자식들의 계산결과 처리

이 APP를 thread를 사용해 구현하는 코드 살펴보기. $sum = \frac{N(N+1)}{2}$

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int sum;          /* 공유 데이터 */
void *runner(void *param);    /* threads call this function */
int main(int argc, char *argv[]) {
    pthread_t tid;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join(tid, NULL);
    printf("sum = %d\n", sum);
}

void *runner(void *param) {
    int i, upper=atoi(param);
    sum=0;
    for (i=1; i<upper; i++)
        sum += i;
    pthread_exit(0);
}
```

Figure 4.11 Multithreaded C program using the Pthreads API

(여러 스레드를 처리(wait)하도록 for loop 내에 join 연산을 포함시)

• 10 threads

```
#define NUM_THREADS 10
/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];
for (int i=0; i<NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Figure 4.12 Pthread code for joining ten threads

한번은 하나하나 들려서 join 되도록.

2. Thread creation with 3Lib

(1) POSIX : Pthreads

(2) Window : windows.h

(3) Java Threads (2 techniques)

① create new class : derive from Thread Class

 ↳ override run() 메소드

② define a class that implements

 the Runnable interface

 ↳ implements Runnable

Thread creation involves creating a Thread object and passing it as an instance of a class that implements Runnable, followed by invoking start() method on the Thread object.

Thread worker = new Thread(new Task());

worker.start();

Java

The join() method can throw an InterruptedException, which we choose to ignore.

```
try {
    worker.join(); // 스레드 종료시까지 대기
} catch (InterruptedException ie) {
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join0:
```

여러 스레드를 위해 Figure 4.13 수정

If the parent must wait for several threads to finish, the join() method can be enclosed in a for loop similar to Pthreads.

```
#include <windows.h>
```

```
#include <stdio.h>
```

```
DWORD Sum; /* 공유 데이터 */
```

```
void for windows
DWORD WINAPI Summation(LPVOID Param) {
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i=1; i<=Upper; i++)
        Sum += i;
    return 0;
}
```

```
int main(int argc, char *argv[]) {
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    Param = atoi(argv[1]);
}
```

```
ThreadHandle = CreateThread(
    NULL,
    0,
    Summation,
    &Param,
    0,
    &ThreadId);
/* create the thread */
```

```
WaitForSingleObject(ThreadHandle, INFINITE);
CloseHandle(ThreadHandle);
printf("sum = %d\n", Sum);
/* default security attributes */
/* default stack size */
/* thread function */
/* parameter to thread function */
/* default creation flags */
/* returns the thread identifier */
/* wait for the thread to finish */
/* close the thread handle */
// child 종료 후 child 가 계산한 sum 출력
```

47

// win API

CreateProcess(...);

WaitForSingleObject();

/* data is shared by the thread(s) */

DWORD : unsigned 32-bit integer

/* The thread(child) will execute */

Figure 4.13 Multithreaded C program using the Windows API

// parent thread main() 부터 thread

/* global data */

/* create the thread */

/* default security attributes */

/* default stack size */

/* thread function */

/* parameter to thread function */

/* default creation flags */

/* returns the thread identifier */

/* wait for the thread to finish */

/* close the thread handle */

// child 종료 후 child 가 계산한 sum 출력

47

여러 쓰레드

WaitForMultipleObjects(N, THandles, TRUE, INFINITE);

size N array of thread HANDLE objects

Four parameters

자식개수?

① The number of objects to wait for

② A pointer to the array of objects

③ A flag indicating whether all objects have been signaled

④ A timeout duration (or INFINITE)

(The parent thread can wait for all its child threads to complete.)

Parent가 기다리다가 전체를 받아온다

44

*parallel != concurrent: 여러 개의 Process가 동시에 실행되는 행정 실행

병렬 프로세스를 동시에 Concurrently 수행하고 동시에 생성/삭제

3.30 Operations on Process

- 프로세스들은 동시에 Concurrently 수행하고 동시에 생성/삭제

- 프로세스는 실행 중 터미널로 사용자 Proc (child) 생성 가능

PID: 프로세스 식별 (unique integer)

→ kernel 내부 index로 접근 용으로 사용 가능

1) Linux OS의 ProcessTree (Process = Task)

① system()가 모든 process를 root parent process (부모는 user process, 자식은 child process)

② 부팅되면 Systemd로부터 나머지 Process들이 생성

* daemon (후기 세번 째 공통한 Process)

* PS -el PStree: 명령어

현재 실행 중 Process들을 → 현재 active한 process들

1. Process Creation

(1) 자식 Proc 생성 당시 Child Proc가 차원을 요구함.

① Child Proc가 OS로부터 차원 할당 받음

② 부모 Proc 차원의 Subset으로 계약됨

부모가 자식들간 차원 분할 / 자식들이 차원 공유

이유: Proc가 너무 많은 child 생성으로 → 사전과 부족

(overloading) 예방

(2) Child Proc에 필요한 Data 전달

① 자식 생성 시, Parent P가 초기 Data 전달

→ 스크립트 파일에 파일 hw.c 내용 출력

: 생성 시, Parent P로부터 파일 명을 input으로 받음.

name of output device 출력 장치로

(3) 부모/자식 프로세스의 실행

(자식 Process 실행 후에는 차원 할당 필요)

① 부모/자식 병행 실행 (독립화, concurrently 실행)

② 부모가 자식이 종료될 때까지 기다림 wait 했다가

wait child terminate

(4) 부모/자식 Process의 주소 공간 사용 *

부모가 자식을 생성하면, 생성되는 시스템 공간은 부모에게

가져갈 Address Space → 차식 간의 new program load

① 차식이 부모의 Duplication (fork() system call)

→ 차식이 부모와 동일한 프로그램과 Data를 가짐.

② 차식 Proc가 차식에게 load되는 새 프로그램을 가짐

(차식 간의 새 프로그램 실행 할 것을 load)

→ 그래서 그 새 프로그램 실행 가능

2. How

1) PID = fork()

부모 Proc가 fork() 하는 Syscall(일정)하면, 새 차식 Proc 생성

① (새 차식 Proc는 부모의 주소 공간 복사본 가져옴)

(new proc는 original 주소 공간의 복사본으로 구성됨)

② parent와 child가 별개 통신하기 함.

③ fork() 후 두 프로세스는 계속 실행됨

→ 즉, concurrently 어떤 부모가 차식 wait

*pid = fork();

부모 Proc에서는 pid로

자식 Proc에서는 (값이) pid를 return, 차식 pid가 return됨.

(Process 2) pid=25

How Fork Works

int pid = fork();

if (pid==0) {

<child code>

exit(0);

<parent code>

wait(pid);

UNIX kernel

→ Unix 이렇게 차원을 가짐

자식

부모 차원은 차원을

갖고 차원을

갖고 차원을

*빨간 글자 → SysCall

(한글로 표기되었던...)

→ 차원을 갖는다.

3.3 Operations On Process

2. Process Termination

프로세스의 종료 Self

- ① 자신의 프로그램 → exit() system call로 OS에 자신의 상태 표시하여 종료
- ② wait() SysCall로 대기중인 부모 process에 자신의 process의 상태값 status value를 return 된다.
- ③ 프로세스 자식들은 OS에 의해 항상 해제되고 죄수된다.

2) 다른(부모) 프로세스에 의한 종료

Parent 부모 Proc는 자식 Proc 종료 가능

- ① Terminate Process() 같은 sys call로 / 다른(부모) 프로세스 종료 시 kill window 끝. → 반드시 부모에게만 한다.

- ② Proc 종료시키기 위해 부모는 자식 ID 알아야

3) 사용자에 의한 종료

: 사용자나 외각 환경은 이를 통해 종료된다

- * 4) 부모가 자식 Proc 종료시키는 이유 (도)
 - ① 자식이 할당된 자원을 초과 사용할 경우 (부모가 자식의 상황에 감지하는 mechanism 필요)
 - ② 자식에게 할당된 task가 더 필요 : task queue
 - ③ 부모가 exit() 자식이 계속 실행되는 것을 OS가 활용하지 않는 경우

Cascading termination initiated by the OS

- Some systems do not allow a child to exist if both.
 - (부모가 종료되면 자식이 존재할 수 없음) → OS가 허용 X (자식도 실행 중)
- If a process terminates (either normally or abnormally), then all its children must also be terminated. (모든 자식이 종료되어야 함)

5) Cascading termination

- A parent process may wait for the termination of a child process by using the wait() system call. (부모는 자식이 종료되었음을 기다림)

```
pid_t pid; // pid of the terminated child
int status; // exit status of the child
pid = wait(&status);
```

(종료되는 자식의 pid가 return 됨 → 부모는 어느 자식이 종료됐는지 식별)
 pid = wait(&status);

- 비교) We can terminate a process by using the exit() system call, providing an exit status as a parameter:

자식에 대한 종료 → exit(1); // 1인 상태 (status)로 exit

→ 자식 부모가 종료 (> *값에 따라 자식은 종료, 정상 종료) 부모 종료 = 정상 종료.

* Process의 exit status → process table에 저장되므로

(체크) 프로세스 데몬의 항목은 부모가 WAIT() 호출할 때까지 남아 있어야 한다.

6) Zombi proc (도)

- ① 종료되었으나 부모가 아직 wait() 호출 안한 process X
- ② Proc를 종료하면 끊은 시간 동안 둘째 상태가 됨.
- ③ 부모가 wait() 호출하면, 좀비 프로세스의 pid와 프로세스 테이블 항목

(PID가 Process table에 남아있고) * (정상 종료되어 소멸된다)

7) Orphan Process * init == systemd

- ① Parent P 없는 child P의 상태: 부모가 wait() 호출 X, terminate 장애.
- ② (Unix) init process (부모proc)를 orphan P의 사용으로 자장하여 해결
- ③ init의 주의로 WAIT() 호출 → orphan 프로세스의 exit status 수신
 - orphan proc의 id와 proc 테이블 항목 차이

8) Android Proc Hierarchy (라온드 기록)

☞ 2가지로: 예전엔 시스템 자원화석(Proc)을, → 부모进程의 하위

* 프로세스 상태는 종료로 되어 있다면 → 어떤 App(프로세스), 예상상태에서

foreground-visible-service-background-empty (5가지)로 나누어

Hierarchy of process classifications (From most to least important)

- Foreground process: The current process visible on the screen, representing the application the user is currently interacting with. (사용자와 상호작용)
- Visible process: A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process). (Foreground 프로세스가 참조하는 활동을 수행)
- Service process: A process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music)
- Background process: A process that may be performing an activity but is not apparent to the user. (사용자가 인식 못함)
- Empty process: A process that holds no active components associated with any App

Empty process: A process that holds no active components associated with

3.4 Interprocess Communication

프로세스간 COMMUNICATION

1) Process Concurrently 실행 in OS

① process가 Independent 할 수 있으므로

: 다른 process는 서로 다른 프로세스로 통작 **data sharing**

② cooperating: data sharing

: share → 영향을 주고 받을 수도

cooperating proc require IPC

2) interprocess communication

: process 간 서로 다른 COMMUNICATION

* Cooperate (data share) 형태

* IPC 가 필요: IPC Model 2가지

① Shared Memory Model

: region of shared memory between Cooperating Proc

read, write exchange info → communication

② msg-passing-mode

: message queue (msg Queue)

(a) Shared memory

(b) Message passing

Figure 3.11 Communications models

© 2020, 손도희,
all right reserved

- ① data 양 줄 때
- ② no conflicts
- ③ easier to implement

MSG - Passing in distributed system

(시스템에 여러 대의 노드)

④ time consuming

(syscall 필요 → 커널 개입 필요)

⑤ Shared Memory

⑥ faster

(2대 이상 proc 가능)

⑦ syscall 필요 아예 없지만,

커널 개입 필요 X.

⑧ 동기화된 메모리 segment

상호작용 proc의 주소공간에 추가

⑨ 동기화된 segment를 사용하여 통신

원하는 process들은 각각 address space

에 attach 해야 함

⑩ * Data 정의, 위치는 OS가 아니라

프로세스마다 각각 결정됨.

⑪ 프로세스들은 동시에 동일한

스페이스를 사용할 수 있음.

5) Sol 2 - msg passing

- ① Provide 2 operation: send(msg), recv(msg)

② process가 보낸 msg 크기 2개

③ fixed size (시스템 구현 수단으로)

④ variable size (시스템 구현 복잡)

⑤ programming task simpler

→ trade off 존재

(3) communication link 풀

↳ Physical O / Logical Link O /

(send(), receive() operation)

① Direct Communication *

: 직접 통신 — 통신상대 명시(1:1)

Indirect Communication

: 간접 통신 — mail box 사용

번호: (owned by OS/Proc)

② Synchronous (기다리는*)

Asynchronous (안 기다리는*)

communication

③ Automatic/Explicit Buffering

(4) Naming

: 통신하는 Proc끼리 가리키는 방법

Direct Comm

① Proc 이름 명시해야함 * recv(P, msg)

② Communication Link 직접 연결

↳ 2proc彤이 1link

only 1link (1in) → 통신 경로

가리키는 방법 찾기 어렵음.

③ Symmetry in addressing (Both 1등명)

Asymmetry: sender와 recipient를

(receiver id, msg) pair name

④ symmetry, asymmetry 연결 (id)

: process def의 제한된 그룹성 (역할)

프로세스 id변경시 그룹의 process 정의

old id에 대신 new id로 수정

hard coding은 간접기법보다 바깥의 X

Indirect Communication

① mail box (or port)로 msg 보냄. (크리에이터)

mail box의 proc들이 msg 수신권한 갖기

mail box는 (integer) unique id, owner

同一个 mailbox은 Proc가 동시에 (여러대) 있다.

Proc가 shared mail box 사용해 통신 가능

(5) Synchronization

① blocking send/recv

↳ Synchronous

block

수신할 때까지 send하기 → 수신 후 송신

msg available까지 recv하기 → 수신수신 (block)

② non blocking

send : send → resume

recv : receive → valid msg/Null

→ 4가지 2가지: block send/block recv

③ Producer-Consumer 문제에 적용 가능

(6) Buffering

: Buffer capacity에 따른 Case 나누기

exchanged msg는 temporary Q에

① zero-capacity (Q= maxLen 0)

msg mail box에 버퍼에 있는 경우

→ 빈 공간 찾기 (block)

sender는 receiver가 필요로 하는 대로 깨끗이 Block (여성간단히)

② bounded-capacity (finite len N)

N/가지 까지 사용 가능

Q full X: msg는 Q에 들어

Sender execute without wait

link full: sender must block until space available

③ unbounded capacity (infinity)

any # of msg can wait in it

sender never block

producer merely invoke send() call

④ wait until msg is delivered to receiver or mailbox

⑤ consumer invoke recv()

if block full msg is available

Figure 3.12 The producer process

message next_produced;

while (true) {

/* produce an item in next_produced */

while ((in + 1) % BUFFER_SIZE == out)

; /* full → do nothing */

buffer[in] = next_produced;

in = (in + 1) % BUFFER_SIZE;

}

/* consume the item in next_consumed */

item next_consumed;

while (true) {

receive (next_consumed);

/* consume the item in next_consumed */

next_consumed = (in + 1) % BUFFER_SIZE;

in = (in + 1) % BUFFER_SIZE;

}

/* produce an item in next_produced */

send (next_produced);

/* consume the item in next_consumed */

next_consumed = (in + 1) % BUFFER_SIZE;

in = (in + 1) % BUFFER_SIZE;

}

/* produce an item in next_produced */

send (next_produced);

/* consume the item in next_consumed */

next_consumed = (in + 1) % BUFFER_SIZE;

in = (in + 1) % BUFFER_SIZE;

}

/* produce an item in next_produced */

send (next_produced);

/* consume the item in next_consumed */

next_consumed = (in + 1) % BUFFER_SIZE;

in = (in + 1) % BUFFER_SIZE;

}