운영체제

(Operating System)

양희재

(hjyang@ks.ac.kr)

컴퓨터공학과

2015

경성대학교

목 차

1 서론

1.1 운영체제의 정의

. PC를 구입하면
- Windows 8, Linux, MS-DOS, MacOSX, iOS

. 운영체제: Operating System

. 운영체제가 없는 컴퓨터?
- 컴퓨터: 프로세서와 메모리
- 전원을 켜면 어떤 일이? 휘발성 메모리 - 야생마
- 프로그램을 실행하려면?
- 여러 개의 프로그램을 동시에 실행시키려면?
- 프린터에 인쇄 명령을 내리려면?
- 하드 디스크에 저장하려면?

. 운영체제란?
- 컴퓨터 하드웨어를 잘 관리하여
  (프로세서, 메모리, 디스크, 키보드, 마우스, 모니터, 네트워크, 스피커, 마이크, GPS, …)
- 성능을 높이고 (Performance)
- 사용자에게 편의성 제공 (Convenience)

. 운영체제: 컴퓨터 하드웨어를 관리하는 프로그램
- Control program for computer
- cf. CP/M

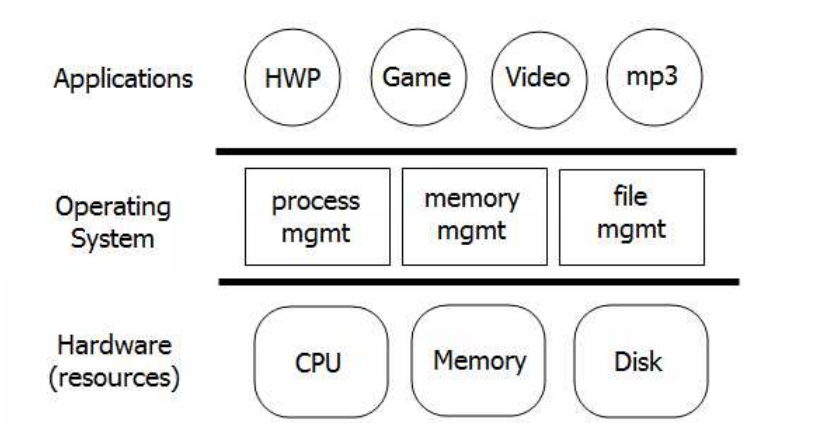
. 컴퓨터 구조
- 프로세서, 메모리 (ROM, RAM), 디스크, 입출력장치 ...

. 부팅(Booting)
- POST (Power-On Self-Test)
- 부트로더 (Boot loader)

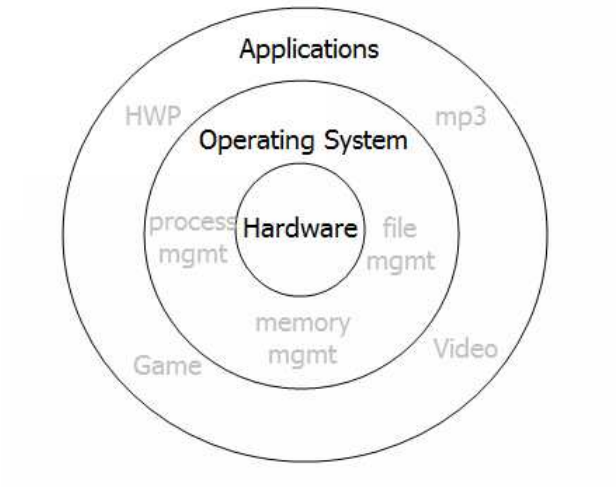
. 운영체제
- 관리(Management) 프로그램
- 프로세서, 메모리, 디스크, 입출력장치의 관리

– 커널 (kernel) + 명령 해석기 (shell, command interpreter)

. 운영체제의 위치



또는



. 운영체제 vs 정부 (Government)

– 자원 관리자 (resource manager)

– 자원 할당자 (resource allocator)

– 주어진 자원을 어떻게 가장 잘 활용할까? 국토, 인력, 예산

– 정부가 직접 일하지는 않는다. 민간기업이 일한다.

– 정부 부서: 행정자치부, 교육부, 국방부, 건설교통부, 농림부, 외교부, …

– 운영체제 부서: 프로세스 관리, 메모리 관리, 파일 관리, 입출력장치 관리, …

1.2 운영체제의 역사

. 컴퓨터 역사: 1940년대 말~
- 하드웨어 발전 ☞ 운영체제 기술 발전

, Batch processing system (일괄처리시스템)
- Card reader > memory > processing > line printer
- 기술발전: resident monitor

. Multiprogramming system (다중프로그래밍 시스템)
- 컴퓨터는 비싼 자원
- 빠른 CPU, 느린 i/o → 메모리에 여러 개의 job
- 기술발전: CPU scheduling, 메모리 관리, 보호

. Time-sharing system (시공유 시스템)
- 강제 절환, 시분할 시스템, interactive system (대화형)
- 기술발전: 가상 메모리, 프로세스간 통신, 동기화

. OS 기술 천이
- 컴퓨터 규모별 분류
 . Supercomputer > Mainframe > Mini > Micro
 . Supercomputer > Server > Workstation > PC > Handheld > Embedded
- 고성능컴퓨터의 OS 기술이 Handheld/Embedded 까지
 . Batch processing
 . Multiprogramming
 . Timesharing

1.3 고등 운영체제
. 고등 컴퓨터 구조 (Advanced Computer Architectures)
 ☞ 고등 운영체제의 등장

가) 다중 프로세서 시스템 (Multiprocessor system)
- 병렬 시스템 (parallel system)
- 강결합 시스템 (tightly-coupled system)
- 3가지 장점: performance, cost, reliability
☞ 다중 프로세서 운영체제 (Multiprocessor OS)

나) 분산 시스템 (Distributed system)
- 다중 컴퓨터 시스템 (multi-computer system)
- 소결합 시스템 (loosely-coupled system)
☞ 분산 운영체제 (Distributed OS)

다) 실시간 시스템 (Real-time system)
- 시간 제약: Deadline
- 공장 자동화 (FA), 군사, 항공, 우주
☞ 실시간 운영체제 (Real-time OS = RTOS)

1.4 인터럽트 기반 시스템
. Interrupt-Based System
- 현대 운영체제는 인터럽트 기반 시스템!

. 부팅이 끝나면?
- 운영체제는 메모리에 상주 (resident)
- 사건(event)을 기다리며 대기: 키보드, 마우스, …

. 하드웨어 인터럽트 (Hardware interrupt)
- 인터럽트 결과 운영체제 내의 특정 코드 (ISR) 실행
- Interrupt Service Routine 종료 후 다시 대기

. 소프트웨어 인터럽트 (Software interrupt)
- 사용자 프로그램이 실행되면서 소프트웨어 인터럽트 (운영체제 서비스 이용 위해)
- 인터럽트 결과 운영체제 내의 특정 코드 실행 (ISR)
- ISR 종료 후 다시 사용자 프로그램으로

. 인터럽트 기반 운영체제
- 운영체제는 평소에는 대기 상태
- 하드웨어 인터럽트에 의해 운영체제 코드 (ISR) 실행
- 소프트웨어 인터럽트에 의해 운영체제 코드 실행
- 내부 인터럽트(Interrnal interrupt)에 의해 운영체제 코드 실행
- ISR 종료되면 원래의 대기상태 또는 사용자 프로그램으로 복귀

1.5 이중모드

. 한 컴퓨터를 여러 사람이 동시에 사용하는 환경
- 또는 한 사람이 여러 개의 프로그램을 동시에 사용
- 한 사람의 고의/실수 프로그램이 전체 영향
  . STOP, HALT, RESET 등
  . 사용자 프로그램은 STOP 등 치명적 명령 사용 불가하게!
- 사용자 (user) 모드 vs 관리자 (supervisor) 모드
- 이중 모드 (dual mode)
- 관리자 모드 = 시스템 모드 = 모니터 모드 = 특권 모드
- Supervisor, system, monitor, priviliged mode
. 특권 명령 (privileged instructions)
- STOP, HALT, RESET, SET_TIMER, SET_HW, …

. 이중 모드 (dual mode)
- 레지스터에 모드를 나타내는 플래그(flag)
- 운영체제 서비스 실행될 때는 관리자 모드
- 사용자 프로그램 실행될 때는 사용자 모드
- 하드웨어/소프트웨어 인터럽트 발생하면 관리자 모드
- 운영체제 서비스가 끝나면 다시 사용자 모드

. 일반적 프로그램의 실행
- 프로그램 적재 (on memory)
- user mode → (키보드, 마우스) → system mode (ISR) → user mode → (모니터, 디스크, 프린터) → system mode (ISR) → user mode → .....

1.6 하드웨어 보호
. 입출력장치 보호 (Input/output device protection)
. 메모리 보호 (Memory protection)
. CPU 보호 (CPU protectiopn)

(1) 입출력장치 보호
. 사용자의 잘못된 입출력 명령
- 다른 사용자의 입출력, 정보 등에 방해
- 예: 프린트 혼선, 리셋 등
- 예: 다른 사람의 파일 읽고 쓰기 (하드디스크)

. 해결법
- 입출력 명령을 특권명령으로: IN, OUT
- 입출력을 하려면 운영체제에게 요청하고 (system mode 전환),
- 운영체제가 입출력 대행; 마친 후 다시 user mode 복귀
- 올바른 요청이 아니면 운영체제가 거부
. 사용자가 입출력 명령을 직접 내린 경우?
- Privileged instruction violation

(2) 메모리 보호
. 다른 사용자 메모리 또는 운영체제 영역 메모리 접근
- 우연히 또는 고의로
- 다른 사용자 정보/프로그램에 대한 해킹
- 운영체제 해킹
. 해결법
- MMU 를 두어 다른 메모리 영역 침범 감시하도록 (Memory Management Unit)
- MMU 설정은 특권명령: 운영체제만 바꿀 수 있다
. 다른 사용자 또는 운영체제 영역 메모리 접근 시도?
- Segment violation

(3) CPU 보호
. 한 사용자가 실수 또는 고의로 CPU 시간 독점
- 예: while (n = 1) …
- 다른 사용자의 프로그램 실행 불가
. 해결법
- Timer 를 두어 일정 시간 경과 시 타이머 인터럽트
- 인터럽트 → 운영체제 → 다른 프로그램으로 강제 전환

1.7 운영체제 서비스
. 프로세스 관리
. 주기억장치 관리
. 파일 관리
. 보조기억장치 관리
. 입출력 장치 관리
. 네트워킹
. 보호
. 그외

(1) 프로세스 관리

. Process management

. 프로세스 (process)

– 메모리에서 실행 중인 프로그램 (program in execution)

. 주요기능

– 프로세스의 생성, 소멸 (creation, deletion)

– 프로세스 활동 일시 중지, 활동 재개 (suspend, resume)

– 프로세스간 통신 (interprocess communication: IPC)

– 프로세스간 동기화 (synchronization)

– 교착상태 처리 (deadlock handling)

(2) 주기억장치 관리

. Main memory management

. 주요기능

– 프로세스에게 메모리 공간 할당 (allocation)

– 메모리의 어느 부분이 어느 프로세스에게 할당되었는가 추적 및 감시

– 프로세스 종료 시 메모리 회수 (deallocation)

– 메모리의 효과적 사용

– 가상 메모리: 물리적 실제 메모리보다 큰 용량 갖도록

(3) 파일 관리

. File management

. Track/sector 로 구성된 디스크를 파일이라는 논리적 관점으로 보게

. 주요기능

– 파일의 생성과 삭제 (file creation & deletion)

– 디렉토리(directory)의 생성과 삭제 (또는 폴더 folder)

– 기본동작지원: open, close, read, write, create, delete

– Track/sector 와 file 간의 매핑(mapping)

– 백업(backup)

(4) 보조기억장치 관리

. Secondary storage management

– 하드 디스크, 플래시 메모리 등

. 주요기능

– 빈 공간 관리 (free space management)

- 저장공간 할당 (storage allocation)
- 디스크 스케쥴링 (disk scheduling)

(5) 입출력 장치 관리
. I/O device management
. 주요기능
- 장치 드라이브 (Device drivers)
- 입출력 장치의 성능향상: buffering, caching, spooling

1.8 시스템 콜

. System calls
- 운영체제 서비스를 받기 위한 호출
. 주요 시스템 콜
- Process: end, abort, load, execute, create, terminate, get/set attributes, wait event, signal event
- Memory: allocate, free
- File: create, delete, open, close, read, write, get/set attributes
- Device: request, release, read, write, get/set attributes, attach/detache devices
- Information: get/set time, get/set system data
- Communication: socket, send, receive

. 예제: MS-DOS
- INT 21H
- 관련 자료 http://spike.scu.edu.au/~barry/interrupts.html
- 예제: 파일 만들기 (Create file)
☞ AH = 3CH, CX = file attributes, DS:DX = file name

. 예제: Linux
- INT 80H
- 관련 자료 http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html
http://www.digilife.be/quickreferences/qrc/linux%20system%20call%20quick%20reference.pdf
- 예제: 파일 만들기 (Create file)
☞ EAX = 8, ECX = file attributes, EBX = file name

2 프로세스 관리

2.1 프로세스

. 프로그램 vs 프로세스 (program vs process)

- process, task, job …
- program in execution: text + data + stack, pc, sp, registers, …
- 무덤 속 프로그램, 살아 움직이는 프로세스

. 프로세스 상태

- new, ready, running, waiting, terminated (그림)
- 프로세스 상태 천이도 (process state transition diagram)
- 상태 천이는 언제 발생?

. PCB: Process Control Block

- Task Control Block (TCB)
- 프로세스에 대한 모든 정보
- process state (running, ready, waiting, …), PC, registers,

MMU info (base, limit), CPU time, process id, list of open files, …

- 사람과 비유?

. 프로세스 대기열 (queue)

가) Job Queue

- Job scheduler
- Long-term scheduler

나) Ready Queue

- CPU scheduler
- Short-term scheduler

다) Device Queue

- Device scheduler

. Multiprogramming

- Degree of multiprogramming
- i/o-bound vs CPU-bound process

. Medium-term scheduler

- Swapping

. 용어

- Context switching (문맥전환)
- Scheduler
- Dispatcher
- Context switching overhead

2.2 CPU 스케쥴링

. CPU Scheduling

- Preemptive vs Non-preemptive (선점 (先占) : 비선점(非先占))

. Scheduling criteria

- CPU Utilization (CPU 이용률)
- Throughput (처리율)
- Turnaround time (반환시간)
- Waiting time (대기시간)
- Response time (응답시간)
- ...

. CPU Scheduling Algorithms

- First-Come, First-Served (FCFS)
- Shortest-Job-First (SJF)
- Shortest-Remaining-Time-First
- Priority
- Round-Robin (RR)
- Multilevel Queue
- Multilevel Feedback Queue

2.3 First-Come, First-Served (FCFS) Scheduling

. Simple & Fair

. Example: Find Average Waiting Time

Process	Burst Time (msec)
P1	24
P2	3
P3	3

- AWT = (0+24+27)/3 = 17 msec
- *cf.* 3 msec!

. Gantt Chart

. Convoy Effect (호위효과)

. Nonpreemptive scheduling

2.4 Shortest-Job-First (SJF) Scheduling

. Example:

Process	Burst Time (msec)
P1	6
P2	8
P3	7
P4	3

- AWT = (3+16+9+0)/4 = 7 msec
- cf. 10.25 msec (FCFS)

. Provably optimal

. Not realistic; prediction may be needed

. Preemptive or Nonpreemtive

cf. Shortest-Remaining-Time-First (최소잔여시간 우선)

. Example

Process	Arrival Time	Burst Time (msec)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- Preemptive: AWT = (9+0+15+2)/4 = 26/4 = 6.5 msec
- Nonpreemptive: 7.75 msec

2.5 Priority Scheduling

. Priority (우선순위): typically an integer number

- Low number represents high priority in general (Unix/Linux)

. Example

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- AWT = 8.2 msec

. Priority

- Internal: time limit, memory requirement, i/o to CPU burst, …
- External: amount of funds being paid, political factors, …

. Preemptive or Nonpreemptive

. Problem

- Indefinite blocking: starvation (기아)
- Solution: aging

2.6 Round-Robin (RR) Scheduling

. Time-sharing system (시분할/시공유 시스템)

. Time quantum 시간양자 = time slice (10 ~ 100msec)

. Preemptive scheduling

. Example

Process	Burst Time (msec)
P1	24
P2	3
P3	3

- Time Quantum = 4msec
- AWT = 17/3 = 5.66 msec

. Performance depends on the size of the time quantum (Δ)

- Δ → ∞ FCFS
- Δ → 0 Processor sharing (* context switching overhead)

. Example: Average turnaround time (ATT)

Process	Burst Time (msec)
P1	0
P2	3
P3	1
P4	7

- ATT = 11.0 msec (Δ = 1)
- ATT = 12.25 msec (Δ = 5)

2.7 Multilevel Queue Scheduling

. Process groups

- System processes
- Interactive processes
- Interactive editing processes

- Batch processes
- Student processes

. Single ready queue → Several separate queues
- 각각의 Queue 에 절대적 우선순위 존재
- 또는 CPU time 을 각 Queue 에 차등배분
- 각 Queue 는 독립된 scheduling 정책

2.8 Multilevel Feedback Queue Scheduling

. 복수 개의 Queue

. 다른 Queue 로의 점진적 이동
- 모든 프로세스는 하나의 입구로 진입
- 너무 많은 CPU time 사용 시 다른 Queue 로
- 기아 상태 우려 시 우선순위 높은 Queue 로

2.9 프로세스 생성과 종료

. 프로세스는 프로세스에 의해 만들어진다!
- 부모 프로세스 (Parent process)
- 자식 프로세스 (Child process)
 cf. Sibling processes
- 프로세스 트리 (process tree)

. Process Identifier (PID)
- Typically an integer number
 cf. PPID

. 프로세스 생성 (creation)
- fork() system call = 부모 프로세스 복사
- exec() = 실행파일을 메모리로 가져오기

. 예제: Windows 7 프로세스 나열

. 예제: Ubuntu Linux 프로세스 나열

. 프로세스 종료 (termination)
- exit() system call

- 해당 프로세스가 가졌던 모든 자원은 O/S 에게 반환 (메모리, 파일, 입출력장치 등)

2.10 쓰레드

. Thread

. 프로그램 내부의 흐름, 맥

```
class Test {
 public static void main(String[] args) {
        int n = 0;
        int m = 6;
        System.out.println(n+m);
        while (n < m)
                n++;
        System.out.println("Bye");
 }
}
```

. 다중 쓰레드 (Multithreads)

- 한 프로그램에 2개 이상의 맥
- 맥이 빠른 시간 간격으로 스위칭 된다 ⇒ 여러 맥이 동시에 실행되는 것처럼 보인다
- concurrent vs simultaneous

. 예: Web browser

- 화면 출력하는 쓰레드 + 데이터 읽어오는 쓰레드

. 예: Word processor

- 화면 출력하는 쓰레드 + 키보드 입력 받는 쓰레드 + 철자/문법 오류 확인 쓰레드

. 예: 음악 연주기, 동영상 플레이어, Eclipse IDE, …

. 한 프로세스에는 기본 1개의 쓰레드

- 단일 쓰레드 (single thread) 프로그램

. 한 프로세스에 여러 개의 쓰레드

- 다중 쓰레드 (multi-thread) 프로그램

. 쓰레드 구조

- 프로세스의 메모리 공간 공유 (code, data)
- 프로세스의 자원 공유 (file, i/o, …)
- 비공유: 개별적인 PC, SP, registers, stack

. 프로세스의 스위칭 vs 쓰레드의 스위칭

. 예제: 자바 쓰레드

- java.lang.Thread

- 주요 메소드

. public void run() // 새로운 맥이 흐르는 곳 (치환)

. void start() // 쓰레드 시작 요청

. void join() // 쓰레드가 마치기를 기다림

. static void sleep() // 쓰레드 잠자기

. Thread.run()

- 쓰레드가 시작되면 run() 메소드가 실행된다

⇒ run() 메소드를 치환한다.

```
class MyThread extends Thread {
 public void run() { // 치환 (override)
        // 코드
 }
}
```

. 예제: 글자 A 와 B 를 동시에 화면에 출력하기

- 모든 프로그램은 처음부터 1개의 쓰레드는 갖고 있다 (main)

- 2개의 쓰레드: main + MyThread

```
class Test {
 public static void main(String[] arg) {
    MyThread th = new MyThread();
    th.start();

    for (int i=0; i<1000; i++)
            System.out.print("A");
 }
}

class MyThread extends Thread {
 public void run() {
    for (int i=0; i<1000; i++)
            System.out.print("B");
 }
}
```

3 프로세스 동기화

. Process Synchronization

cf. Thread synchronization

3.1 Cooperating Processes

. Processes

- Independent vs. Cooperating
- Cooperating process: one that can affect or be affected by other processes executed in the system
- 프로세스간 통신: 전자우편, 파일 전송
- 프로세스간 자원 공유: 메모리 상의 자료들, 데이터베이스 등
- 명절 기차표 예약, 대학 온라인 수강신청, 실시간 주식거래

. Process Synchronization: Why?

- Concurrent access to shared data may result in data inconsistency
- Orderly execution of cooperating processes so that data consistency is maintained

. Example: BankAccount Problem (은행계좌문제)

- 부모님은 은행계좌에 입금; 자녀는 출금
- 입금(deposit)과 출금(withdraw) 은 독립적으로 일어난다.

☞ 코드-1 참조

. 입출금 동작 알기 위해 "+", "-" 출력하기

. 입출금 동작에 시간 지연 추가

```
void deposit(int amount) {
        // 변경된 값을 임시변수에 저장하고
        int temp = balance + amount;
        System.out.print("+");// 시간 지연 후
        balance = temp;// 잔액 업데이트
 }
 void withdraw(int amount) {
        // 변경된 값을 임시변수에 저장하고
        int temp = balance - amount;
        System.out.print("-"); // 시간 지연 후
        balance = temp;// 잔액 업데이트
 }
```

. 잘못된 결과값

- 이유: 공통변수(common variable)에 대한 동시 업데이트 (concurrent update)
- 해결: 한 번에 한 쓰레드만 업데이트하도록 → 임계구역 문제

```
// 코드-1
class Test {
 static final int MAX = 100; // 입출금 회수
 public static void main(String[] args) throws
          InterruptedException {
          // 은행계좌를 만들고
          BankAccount b =
           new BankAccount();
          // 부모 쓰레드와
          Parent p = new Parent(b, MAX);
          // 자식 쓰레드를 만든 후
          Child c = new Child(b, MAX);
          // 각각 실행시킨다.
          p.start();
          c.start();
          p.join();// 부모와 자식 쓰레드가
          c.join();// 각각 종료하기를 기다린다.
          System.out.println("Final balance = "
           + b.getBalance());// 최종 잔액 출력
 }
}
```

```
class BankAccount {
 int balance;
 void deposit(int amount) {
          balance = balance + amount;
 }
 void withdraw(int amount) {
          balance = balance - amount;
 }
 int getBalance() {
          return balance;
 }
}
```

```
class Parent extends Thread {
 BankAccount b;
 int count;
 Parent(BankAccount b, int count) {
          this.b = b;
          this.count = count;
 }
 public void run() {
          for (int i=0; i<count; i++)
                   b.deposit(1);
 }
}
```

```
class Child extends Thread {
 BankAccount b;
 int count;
 Child(BankAccount b, int count) {
          this.b = b;
          this.count = count;
 }
 public void run() {
          for (int i=0; i<count; i++)
                   b.withdraw(1);
 }
}
```

3.2 임계구역 문제

. The Critical-Section Problem

. Critical section

- A system consisting of multiple threads

- Each thread has a segment of code, called critical section, in which the thread may be changing common variables, updating a table, writing a file, and so on.

. Solution

- Mutual exclusion (상호배타): 많아야 한 쓰레드만 진입

- Progress (진행): 진입 결정은 유한 시간 내

- Bounded waiting (유한대기): 어느 쓰레드라도 유한 시간 내 진입

. 프로세스/쓰레드 동기화

- 임계구역 문제 해결 (틀린 답이 나오지 않도록)
- 프로세스 실행 순서 제어 (원하는 대로)

3.3 세마포

. Synchronization Tools (동기화 도구)

- Semaphores
- Monitors
- .....

. Semaphores (세마포)

- n. (철도의) 까치발 신호기, 시그널; U (군대의) 수기(手旗) 신호
- 동기화 문제 해결을 위한 소프트웨어 도구
- 네덜란드의 Edsger Dijkstra 가 제안
- 구조: 정수형 변수 + 두 개의 동작 (P, V)

. 동작

- 마치 스택(stack)과 같이 ...: push() & pop()
- P: Proberen (test) → acquire()
- V: Verhogen (increment) → release()

. 전체 구조

- 내부적으로는 프로세스(쓰레드)가 대기하는 큐(queue/list)가 포함되어있다.

```
class Semaphore {
 private int value;                  // number of permits
 Semaphore(int value) {
        …
 }
 void acquire() {                    // P
        …
 }
void release() {                     // V
        …
 }
}
```

. 세부 동작

```
 void acquire() {
        value--;
        if (value < 0) {
                add this process/thread to list;
```

```
                block;
        }
 }
 void release() {
        value++;
        if (value <= 0) {
                remove a process P from list;
                wakeup P;
        }
 }
```

. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

. 일반적 사용 (1): Mutual exclusion

- sem.value = 1; // # of permit

```
sem.acquire();
Critical-Section
sem.release();
```

. 예제: BankAccount Problem

☞ 코드-2

```
// 코드-2
import java.util.concurrent.Semaphore;

class BankAccount {
 int balance;
 Semaphore sem;
 BankAccount() {
        sem = new Semaphore(1);// 초기값 = 1
 }
 void deposit(int amount) {      // 입금
        try {
        sem.acquire();   // 진입 전: acquire()
        } catch (InterruptedException e) {}
        int temp = balance + amount;
        System.out.print("+");
        balance = temp;
        sem.release();   // 나온 후: release()
 }
 void withdraw(int amount) {     // 출금
        try {
        sem.acquire();   // 진입 전: acquire()
        } catch (InterruptedException e) {}
```

```
        int temp = balance - amount;
        System.out.print("-");
        balance = temp;
        sem.release();   // 나온 후: release()
 }
 int getBalance() {
        return balance;
 }
}
```

. 일반적 사용 (2): Ordering

- sem.value = 0; // # of permit

. 예제: BankAccount Problem

(1) 항상 입금 먼저 (= Parent 먼저)

상호배타를 위한 **sem** 세마포어 외에 실행순서 조정을 위한 **sem2** 세마포어를 추가한다. 프로그램이 시작되면 부모 쓰레드는 그대로 실행되게 하고, 자식 쓰레드는 초기 값이 0인 **sem2** 세마포어에 대해 **acquire()** 를 호출하게 하도록 **deposit()**, **withdraw()** 메소드를 각각 수정한다. 즉 자식 쓰레드가 먼저 실행되면 세마포어 **sem2**에 의해 블록되고, 블록된 자식 쓰레드는 나중에 부모 쓰레드가 깨워주게 한다.

(2) 항상 출금 먼저 (= Child 먼저)

☞ 위와 유사

(3) 입출금 교대로 (P-C-P-C-P-C- …)

블록된 부모 쓰레드는 자식 쓰레드가 깨워주고, 블록된 자식 쓰레드는 부모 쓰레드가 각각 깨워주도록 한다. 상호배타를 위한 **sem** 세마포어 외에 부모 쓰레드의 블록을 위해 **dsem** 세마포어를, 자식 쓰레드의 블록을 위해 **wsem** 세마포어를 각각 사용한다.

(4) 잔액이 항상 0 이상

출금하려는 액수보다 잔액이 작으면 자식 쓰레드가 블록되도록 하며 이후 부모 쓰레드가 깨워주게 한다. 상호배타를 위한 **sem** 세마포어 외에 **sem2** 세마포어를 사용하여 잔액 부족 시 자식 쓰레드가 블록 되도록 한다.

3.4 생산자-소비자 문제

. 전통적 동기화 문제 (Classical Synchronization Problems)
(1) Producer and Consumer Problem
- 생산자-소비자 문제
- 유한버퍼 문제 (Bounded Buffer Problem)
(2) Readers-Writers Problem
- 공유 데이터베이스 접근
(3) Dining Philosopher Problem
- 식사하는 철학자 문제

. 생산자-소비자 문제
- 생산자가 데이터를 생산하면 소비자는 그것을 소비
- 예: 컴파일러 > 어셈블러, 파일 서버 > 클라이언트, 웹 서버 > 웹 클라이언트
. 유한 버퍼 (bounded buffer)
- 생산된 데이터는 버퍼에 일단 저장 (속도 차이 등)
- 현실 세계에서 버퍼 크기는 유한
- 생산자는 버퍼가 가득 차면 더 넣을 수 없다.
- 소비자는 버퍼가 비면 뺄 수 없다.
☞ 코드-3

. 잘못된 결과
- 실행 불가, 또는
- count ≠ 0 (생산된 항목 숫자 ≠ 소비된 항목 숫자)
- 최종적으로 버퍼 내에는 0 개의 항목이 있어야
. 이유
- 공통변수 count, buf[] 에 대한 동시 업데이트
- 공통변수 업데이트 구간(= 임계구역)에 대한 동시 진입
. 해결법
- 임계구역에 대한 동시 접근 방지 (상호배타)
- 세마포를 사용한 상호배타 (mutual exclusion)
- 세마포: mutex.value = 1 (# of permit) ☞ 코드-4

```
// 코드-3
class Test {
 public static void main(String[] arg) {
          Buffer b = new Buffer(100);
          Producer p =
           new Producer(b, 10000);
          Consumer c =
           new Consumer(b, 10000);
          p.start();
          c.start();
          try {
                    p.join();
                    c.join();
          } catch (InterruptedException e) {}
          System.out.println(
"Number of items in the buf is "
                     + b.count);
 }
}
```

```
class Buffer {
 int[] buf;
 int size, count, in, out;
 Buffer(int size) {
          buf = new int[size];
          this.size = size;
          count = in = out = 0;
 }
 void insert(int item) {
          /* check if buf is full */
          while (count == size)
                    ;
          /* buf is not full */
          buf[in] = item;
          in = (in+1)%size;
          count++;
 }
 int remove() {
          /* check if buf is empty */
          while (count == 0)
                    ;
          /* buf is not empty */
          int item = buf[out];
          out = (out+1)%size;
          count--;
          return item;
 }
}
```

```
class Producer extends Thread {
 Buffer b;
 int n;
 Producer(Buffer b, int n) {
                    this.b = b;
                    this.n = n;
 }
 public void run() {
          for (int i=0; i<n; i++)
                    b.insert(i);
 }
}
```

```
class Consumer extends Thread {
 Buffer b;
 int n;
 Consumer(Buffer b, int n) {
          this.b = b;
          this.n = n;
 }
 public void run() {
          int item;
          for (int i=0; i<n; i++)
                    item = b.remove();
 }
}
```

```
// 코드-4
import java.util.concurrent.Semaphore;

class Buffer {
 int[]    buf;
 int      size, count, in, out;
 Semaphore mutex;

 Buffer(int size) {
```

```
        buf = new int[size];
        this.size = size;
        count = in = out = 0;
        mutex = new Semaphore(1);
 }
 void insert(int item) {
        while (count == size);

        try {
        mutex.acquire();
        } catch (InterruptedException e) {}
        buf[in] = item;
        in = (in+1)%size;
        count++;                    // increase item count
        mutex.release();
 }
 int remove() {
        while (count == 0);

        try {
        mutex.acquire();
        int item = buf[out];
        out = (out+1)%size;
        count--;
        mutex.release();
        return item;
        } catch (InterruptedException e) {
                return -1;          // dummy
        }
 }
}
```

. Busy-wait

- 생산자: 버퍼가 가득 차면 기다려야 = 빈(empty) 공간이 있어야
- 소비자: 버퍼가 비면 기다려야 = 찬(full) 공간이 있어야

. 세마포를 사용한 busy-wait 회피 ☞ 코드-5

- 생산자: empty.acquire() // # of permit = BUF_SIZE
- 소비자: full.acquire() // # of permit = 0

[생산자]

```
empty.acquire();
PRODUCE;
full.release();
```

[소비자]

full.acquire();

CONSUME;

empty.release();

```
// 코드-5
import java.util.concurrent.Semaphore;

class Buffer {
 int[]    buf;
 int      size, count, in, out;
 Semaphore mutex, full, empty;

 Buffer(int size) {
         buf = new int[size];
         this.size = size;
         count = in = out = 0;
         mutex = new Semaphore(1);
         full = new Semaphore(0);
         empty = new Semaphore(size);
 }
 void insert(int item) {
         try {
         empty.acquire();
//       while (count == size);

         mutex.acquire();
         count++;
         buf[in] = item;
         in = (in+1)%size;
         mutex.release();

         full.release();
         }
         catch (InterruptedException e) {
         }
 }
 int remove() {
         try {
         full.acquire();
//       while (count == 0);

         mutex.acquire();
         count--;
         int item = buf[out];
         out = (out+1)%size;
         mutex.release();
```

```
        empty.release();
        return item;
        }
        catch (InterruptedException e) {
                return -1;        // dummy
        }
 }
}
```

3.5 읽기-쓰기 문제

. Readers-Writers Problem

. 공통 데이터베이스

- Readers: read data, never modify it
- Writers: read data and modifiy it
- 상호배타: 한 번에 한 개의 프로세스만 접근 ☞ 비효율적

. 효율성 제고

- Each read or write of the shared data must happen within a critical section
- Guarantee mutual exclusion for writers
- Allow multiple readers to execute in the critical section at once

. 변종

- The first R/W problem (readers-preference)
- The second R/W problem (writers-preference)
- The Third R/W problem

3.6 식사하는 철학자 문제

. Dining Philosopher Problem

- 5명의 철학자, 5개의 젓가락, 생각 → 식사 → 생각 → 식사 …
- 식사하려면 2개의 젓가락 필요

. 프로그래밍 ☞ 코드-6

- 젓가락: 세마포 (# of permit = 1)
- 젓가락과 세마포에 일련번호: 0 ~ 4
- 왼쪽 젓가락 → 오른쪽 젓가락

```
// 코드-6
import java.util.concurrent.Semaphore;

class Philosopher extends Thread {
 int id; // philosopher id
 Semaphore lstick, rstick; // left, right chopsticks

 Philosopher(int id, Semaphore lstick, Semaphore rstick) {
```

```
        this.id = id;
        this.lstick = lstick;
        this.rstick = rstick;
 }
 public void run() {
        try {
        while (true) {
                lstick.acquire();
                rstick.acquire();
                eating();
                lstick.release();
                rstick.release();
                thinking();
        }
        }catch (InterruptedException e) { }
 }
 void eating() {
        System.out.println("[" + id + "] eating");
 }
 void thinking() {
        System.out.println("[" + id + "] thinking");
 }
}

class Test {
 static final int num = 5; // number of philosphers & chopsticks
 public static void main(String[] args) {
        int i;

        /* chopsticks */
        Semaphore[] stick = new Semaphore[num];
        for (i=0; i<num; i++)
        stick[i] = new Semaphore(1);

        /* philosophers */
        Philosopher[] phil = new Philosopher[num];
        for (i=0; i<num; i++)
                phil[i] = new Philosopher(i, stick[i], stick[(i+1)%num]);

        /* let philosophers eat and think */
        for (i=0; i<num; i++)
                phil[i].start();
 }
}
```

. 잘못된 결과: starvation

- 모든 철학자가 식사를 하지 못해 굶어 죽는 상황

. 이유 = 교착상태 (deadlock)

3.7 교착상태

. Deadlock

. 프로세스는 실행을 위해 여러 자원을 필요로 한다.

- CPU, 메모리, 파일, 프린터, ……
- 어떤 자원은 갖고 있으나 다른 자원은 갖지 못할 때 (e.g., 다른 프로세스가 사용 중) 대기해야
- 다른 프로세스 역시 다른 자원을 가지려고 대기할 때 교착상태 가능성!

. 교착상태 필요 조건 (Necessary Conditions)

- Mutual exclusion (상호배타)
- Hold and wait (보유 및 대기)
- No Preemption (비선점)
- Circular wait (환형대기)

. 자원 (Resources)

- 동일 형식 (type) 자원이 여러 개 있을 수 있다 (instance)
- 예: 동일 CPU 2개, 동일 프린터 3개 등

. 자원의 사용

- 요청 (request) → 사용 (use) → 반납 (release)

. 자원 할당도 (Resource Allocation Graph)

- 어떤 자원이 어떤 프로세스에게 할당되었는가?
- 어떤 프로세스가 어떤 자원을 할당 받으려고 기다리고 있는가?
- 자원: 사각형, 프로세스: 원, 할당: 화살표

. 교착상태 필요조건

- 자원 할당도 상에 원이 만들어져야 (환형대기)
- 충분조건은 아님!

. 예제: 식사하는 철학자 문제

- 원이 만들어지지 않게 하려면?

3.8 교착상태 처리

. Handling deadlocks

- 교착상태 방지 (Deadlock Prevention)
- 교착상태 회피 (Deadlock Avoidance)
- 교착상태 검출 및 복구 (Deadlock Detection & Recovery)
- 교착상태 무시 (Don't Care)

(1) 교착상태 방지

. Deadlock Prevention

. 교착상태 4가지 필요조건 중 한 가지 이상 결여되게

- 상호배타 (Mutual exclusion)
- 보유 및 대기 (Hold and wait)
- 비선점 (No preemption)
- 환형 대기 (Circular wait)

. 상호배타 (Mutual exclusion)

- 자원을 공유 가능하게; 원천적 불가할 수도

. 보유 및 대기 (Hold & Wait)

- 자원을 가지고 있으면서 다른 자원을 기다리지 않게
- 예: 자원이 없는 상태에서 모든 자원 대기; 일부 자원만 가용하면 보유 자원을 모두 놓아주기
- 단점: 자원 활용률 저하, 기아 (starvation)

. 비선점 (No preemption)

- 자원을 선점 가능하게; 원천적 불가할 수도 (예: 프린터)

. 환형대기 (Circular wait)

- 예: 자원에 번호부여; 번호 오름차순으로 자원 요청
- 단점: 자원 활용률 저하

(2) 교착상태 회피

. Deadlock Avoidance

- 교착상태 = 자원 요청에 대한 잘못된 승인 (≒ 은행 파산)

. 예제

- 12개의 magnetic tape 및 3개의 process
- 안전한 할당 (Safe allocation)

Process	Max needs	Current needs
P0	10	5
P1	4	2
P2	9	2

– 불안전한 할당 (Unsafe allocation)

Process	Max needs	Current needs
P0	10	5
P1	4	2
P2	9	3

. 운영체제는 자원을 할당할 때 불안전 할당 되지 않도록

– 불안전 할당 → 교착상태

– 대출전문 은행과 유사: Banker's Algorithm

(3) 교착상태 검출 및 복구

. Deadlock Detection & Recovery

– 교착상태가 일어나는 것을 허용

– 주기적 검사

– 교착상태 발생 시 복구

. 검출

– 검사에 따른 추가 부담 (overhead): 계산, 메모리

. 복구

– 프로세스 일부 강제 종료

– 자원 선점하여 일부 프로세스에게 할당

(4) 교착상태 무시

. 교착상태는 실제로 잘 일어나지 않는다!

– 4가지 필요조건 모두 만족해도 …

– 교착상태 발생 시 재시동 (PC 등 가능)

3.9 모니터

. Monitor

– 세마포 이후 프로세스 동기화 도구

– 세마포 보다 고수준 개념

. 구조

– 공유자원 + 공유자원 접근함수

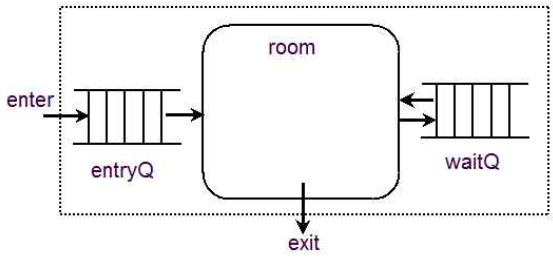
– 2개의 queues: 배타동기 + 조건동기

– 공유자원 접근함수에는 최대 1개의 쓰레드만 진입

– 진입 쓰레드가 조건동기로 블록되면 새 쓰레드 진입가능

– 새 쓰레드는 조건동기로 블록된 쓰레드를 깨울 수 있다.

- 깨워진 쓰레드는 현재 쓰레드가 나가면 재진입할 수 있다.



. 자바의 모든 객체는 모니터가 될 수 있다.

- 배타동기: synchronized 키워드 사용하여 지정
- 조건동기: wait(), notify(), notifyAll() 메소드 사용

```
class C {
 int value, …;
 synchronized void f() {
        …
 }
 synchronized void g() {
        …
 }
 void h() {
        …
 }
}
```

. 일반적 사용 (1): Mutual exclusion

```
synchronized {
        Critical-Section
}
```

. 예제: BankAccount Problem

```
class BankAccount {
 int balance;
 synchronized void deposit(int amount) {
        int temp = balance + amount;
        System.out.print("+");
        balance = temp;
        notify();// 자식 쓰레드를 깨워준다.
 }
 synchronized void withdraw(int amount) {
        // 잔액이 부족하면 블록된다.
        while (balance < amount)
         try {
```

```
                wait();
            }
            catch (InterruptedException e) {}
        int temp = balance - amount;
        System.out.print("-");
        balance = temp;
 }
 int getBalance() {
        return balance;
 }
}
```

. 일반적 사용 (2): Ordering

P1	P2
wait(); S1;	 S2; notify();

. 예제: BankAccount Problem

- 항상 입금 먼저 (= Parent 먼저)
- 항상 출금 먼저 (= Child 먼저)
- 입출금 교대로 (P-C-P-C-P-C- …)

. 예제: 생산자-소비자 문제

```
class Buffer {
 int[] buf;
 int size, count, in, out;

 Buffer(int size) {
        buf = new int[size];
        this.size = size;
        count = in = out = 0;
 }
 synchronized void insert(int item) {
        while (count == size)
        try {
                wait();
        } catch (InterruptedException e) {}
        buf[in] = item;
        in = (in+1)%size;
        count++;
        notify();
 }
 synchronized int remove() {
        while (count == 0)
        try {
```

```
                wait();
        } catch (InterruptedException e) {}
        int item = buf[out];
        out = (out+1)%size;
        count--;
        notify();
        return item;
 }
}
```

. 예제: 식사하는 철학자 문제

```
class Chopstick {
 private boolean inUse = false;

 synchronized void acquire() throws InterruptedException {
        while (inUse)
                wait();
        inUse = true;
 }
 synchronized void release() {
        inUse = false;
        notify();
 }
}
```

4 메모리

4.1 메모리 역사

. 메모리 종류
- Core memory
- 진공관 메모리
- 트랜지스터 메모리
- 집적회로 메모리: SRAM, DRAM

. 메모리 용량
- 1970년대: 8-bit PC, 64KB
- 1980년: 16-bit IBM-PC, 640KB → 1MB → 4MB
- 1990년: 수MB → 수십 MB
- 2000년~현재: 수백 MB → 수 GB

. 언제나 부족한 메모리
- 기계어/어셈블리어 → C언어 → 자바, 객체지향형 언어
- 숫자 처리 → 문자 처리 → 멀티미디어 처리 → Big Data

. 메모리 용량 증가 vs 프로그램 크기 증가
- 언제나 부족한 메모리

. 어떻게 메모리를 효과적으로 사용할 수 있을까?
- 메모리 낭비 없애기
- 가상 메모리 (virtual memory)

4.2 메모리 주소

. 주소(Address) + 데이터(Data)

. 프로그램 개발
- 원천파일 (Source file): 고수준언어 또는 어셈블리언어
- 목적파일 (Object file): 컴파일 또는 어셈블 결과
- 실행파일 (Executable file): 링크 결과

. 개발 도구

- 컴파일러 compiler
- 어셈블러 assembler
- 링커 linker
- 로더 loader

. 프로그램 실행
- code + data + stack

. 실행파일을 메모리에 올리기
- 메모리 몇 번지에?
- 다중 프로그래밍 환경에서는?

. MMU 사용
- 재배치 레지스터 (Relocation register)

. 주소 구분
- 논리주소 (logical address) vs 물리주소 (physical address)

4.3 메모리 낭비 방지
- Dynamic Loading
- Dynamic Linking
- Swapping

(1) 동적 적재 (Dynamic Loading)
. 프로그램 실행에 반드시 필요한 루틴/데이터만 적재
- 모든 루틴(routine)이 다 사용되는 것은 아니다 (예: 오류처리)
- 모든 데이터(data)가 다 사용되는 것은 아니다 (예: 배열)
- 자바: 모든 클래스가 다 사용되는 것은 아니다
- 실행 시 필요하면 그때 해당 부분을 메모리에 올린다.
. cf. 정적 적재 (Static loading)

(2) 동적 연결 (Dynamic Linking)
. 여러 프로그램에 공통 사용되는 라이브러리
- 공통 라이브러리 루틴(library routine)를 메모리에 중복으로 올리는 것은 낭비
- 라이브러리 루틴 연결을 실행 시까지 미룬다.
- 오직 하나의 라이브러리 루틴만 메모리에 적재되고,

- 다른 애플리케이션 실행 시 이 루틴과 연결(link)된다.
. cf. 정적 연결 (Static linking)
- 공유 라이브러리 (shared library) . Linux 또는,
- 동적 연결 라이브러리 (Dynamic Linking Library) - Windows

(3) 스와핑 (Swapping)
. 메모리에 적재되어 있으나 현재 사용되지 않고 있는 프로세스 이미지
- 메모리 활용도 높이기 위해 Backing store (= swap device) 로 몰아내기
- swap-out vs. swap-in
- Relocation register 사용으로 적재 위치는 무관
- 프로세스 크기가 크면 backing store 입출력에 따른 부담 크다.

4.4 연속 메모리 할당
. Contiguous Memory Allocation

. 다중 프로그래밍 환경
- 부팅 직후 메모리 상태: O/S + big single hole
- 프로세스 생성 & 종료 반복 ☞ scattered holes

. 메모리 단편화 (Memory fragmentation)
- Hole 들이 불연속하게 흩어져 있기 때문에 프로세스 적재 불가
- 외부 단편화 (external fragmentation) 발생
- 외부 단편화를 최소화 하려면?

. 연속 메모리 할당 방식
- First-fit (최초 적합)
- Best-fit (최적 적합)
- Worst-fit (최악 적합)

. 예제
- Hole: 100 /500 / 600 / 300 / 200 KB
- 프로세스: 212 417 112 426 KB

. 할당 방식 성능 비교: 속도 및 메모리 이용률
- 속도: first-fit
- 이용률: first-fit, best-fit

. 외부 단편화로 인한 메모리 낭비
- 1/3 수준 (사용 불가)
- Compaction : 최적 알고리즘 없음, 고부담
- 다른 방법은?

4.5 페이징
. Paging

. 프로세스를 일정 크기(=페이지)로 잘라서 메모리에!
- 프로세스는 페이지(page)의 집합
- 메모리는 프레임(frame)의 집합
- 페이지를 프레임에 할당

. MMU 내의 재배치 레지스터 값을 바꿈으로서
- CPU 는 프로세스가 연속된 메모리 공간에 위치한다고 착각
- MMU 는 페이지 테이블 (page table) 이 된다.

. 논리주소 (Logical address)
- CPU 가 내는 주소는 2진수로 표현 (전체 m 비트)
- 하위 n 비트는 오프셋(offset) 또는 변위(displacement)
- 상위 m-n 비트는 페이지 번호

. 주소변환 (Address translation)
- 논리주소 → 물리주소 (Physical address)
- 페이지 번호(p)는 페이지 테이블 인덱스 값
- p 에 해당되는 테이블 내용이 프레임 번호(f)
- 변위(d)는 변하지 않음
- 변환 그림 참조

. 예제 (1)
- Page size = 4 bytes
- Page Table: 5 6 1 2
- 논리주소 13 번지는 물리주소 몇 번지?

. 예제 (2)
- Page Size = 1KB

- Page Table: 1 2 5 4 8 3 0 6
- 논리주소 3000번지는 물리주소 몇 번지?
- 물리주소 0x1A53 번지는 논리주소 몇 번지?

. 내부 단편화 (Internal Fragmentation)
- 프로세스 크기가 페이지 크기의 배수가 아니라면,
- 마지막 페이지는 한 프레임을 다 채울 수 없다.
- 남은 공간 = 메모리 낭비

. 페이지 테이블 만들기
- CPU 레지스터로
- 메모리로
- TLB (Translation Look-aside Buffer) 로
- 척도: 테이블 엔트리 개수 vs 변환 속도

. 연습: TLB 사용 시 유효 메모리 접근 시간
- Tm = 100ns, Tb = 20ns, hit ratio = 80%
- Teff = hTb + (1-h)(Tb+Tm) = ?

. 보호 (Protection): 해킹 등 방지
- 모든 주소는 페이지 테이블을 경유하므로,
- 페이지 테이블 엔트리마다 r, w, x 비트 두어
- 해당 페이지에 대한 접근 제어 가능

. 공유 (Sharing): 메모리 낭비 방지
- 같은 프로그램을 쓰는 복수 개의 프로세스가 있다면,
- Code + data + stack 에서 code 는 공유 가능 (단, non-selfmodifying code = reentrant code = pure code 인 경우)
- 프로세스의 페이지 테이블 코드 영역이 같은 곳을 가리키게

4.6 세그멘테이션
. Segmentation
. 프로세스를 논리적 내용(=세그멘트)으로 잘라서 메모리에 배치!
- 프로세스는 세그멘트(segment)의 집합
- 세그멘트의 크기는 일반적으로 같지 않다.
. 세그멘트를 메모리에 할당

- MMU 내의 재배치 레지스터 값을 바꿈으로서
- CPU 는 프로세스가 연속된 메모리 공간에 위치한다고 착각
- MMU 는 세그멘트 테이블 (segment table) 이 된다.

. 논리주소 (Logical address)
- CPU 가 내는 주소는 segment 번호(s) + 변위(d)

. 주소변환
- 논리주소 → 물리주소 (Physical address)
- 세그멘트 테이블 내용: base + limit
- 세그멘트 번호(s)는 세그멘트 테이블 인덱스 값
- s 에 해당되는 테이블 내용으로 시작 위치 및 한계값 파악
- 한계(limit)를 넘어서면 segment violation 예외 상황 처리
- 물리주소 = base[s] + d
- 변환 그림 참조

. 예제
- 논리주소 (2,100) 는 물리주소 무엇인가?
- 논리주소 (1, 500) 은 물리주소?

Limit	Base
1000	1400
400	6300
400	4300
1100	3200
1000	4700

. 보호 (Protection): 해킹 등 방지
- 모든 주소는 세그멘트 테이블을 경유하므로,
- 세그멘트 테이블 엔트리마다 r, w, x 비트 두어
- 해당 세그멘트에 대한 접근 제어 가능
- 페이징보다 우월!

. 공유 (Sharing): 메모리 낭비 방지
- 같은 프로그램을 쓰는 복수 개의 프로세스가 있다면,
- Code + data + stack 에서 code 는 공유 가능 (단, non-selfmodifying code = reentrant code = pure code 인 경우)
- 프로세스의 세그멘트 테이블 코드 영역이 같은 곳을 가리키게

- 페이징보다 우월!

. 외부 단편화 (External Fragmentation)
- 세그멘트 크기는 고정이 아니라 가변적
- 크기가 다른 각 세그멘트를 메모리에 두려면 = 동적 메모리 할당
- First-, best-, worst-fit, compaction 등 문제

. 세그멘테이션 + 페이징
- 세그멘테이션은 보호와 공유면에서 효과적
- 페이징은 외부 단편화 문제를 해결
- 따라서 세그멘트를 페이징하자! ☞ Paged segmentation
- 예: Intel 80x86

5 가상 메모리

. Virtual Memory

. 물리 메모리 크기 한계 극복

- 물리 메모리보다 큰 프로세스를 실행?

  e.g. 100MB 메인 메모리에서 200MB 크기의 프로세스 실행

. 어떻게?

- 프로세스 이미지를 모두 메모리에 올릴 필요는 없다.
- 현재 실행에 필요한 부분만 메모리에 올린다!
- 오류 처리 제외, 배열 일부 제외, 워드프로세스에서 정렬, 표 기능 제외 ☞ 동적 적재(dynamic loading)과 비슷한 개념

5.1 요구 페이징

. Demand Paging

- 프로세스 이미지는 backing store 에 저장
- 프로세스는 페이지의 집합
- 지금 필요한 페이지만 메모리에 올린다(load) ☞ 요구되는 (demand) 페이지만 메모리에 올린다

. 하드웨어 지원

- valid 비트 추가된 페이지 테이블
- backing store (= swap device)

. 페이지 결함

- Page Fault
- 접근하려는 페이지가 메모리에 없다 (invalid) = 페이지 부재
- Backing store 에서 해당 페이지를 가져온다.
- Steps in handling a page fault

. 용어

- pure demand paging
- prepaging

. 비교: swapping vs demand paging

. 유효 접근 시간

. Effective Access Time

- p: probability of a page fault = page fault rate
- Teff = (1-p)Tm + pTp

. 예제

- Tm = 200 nsec (DRAM)
- Tp = 8 msec (seek time + rotational delay + transfer time)
- Teff = (1-p)200 + p8,000,000 = 200 + 7,999,800p
- p = 1/1,000 ☞ Teff = 8.2usec (40배 느림)
- p = 1/399,990 ☞ Teff = 220nsec (10% 느림)

. 지역성의 원리

- Locality of reference
- 메모리 접근은 시간적, 공간적 지역성을 가진다!
- 실제 페이지 부재 확률은 매우 낮다.

. 다른 방법

- HDD 는 접근 시간이 너무 길다 ☞ swap device 로 부적합
- SSD 또는 느린 저가 DRAM 사용

5.2 페이지 교체

. Page Replacement

- 요구되어지는 페이지만 backing store 에서 가져온다.
- 프로그램 실행 계속에 따라 요구 페이지가 늘어나고,
- 언젠가는 메모리가 가득 차게 된다 ☞ Memory full!
- 메모리가 가득 차면 추가로 페이지를 가져오기 위해
- 어떤 페이지는 backing store 로 몰아내고 (page-out)
- 그 빈 공간으로 페이지를 가져온다 (page-in)
- 용어: victim page

. Victim Page

- 어느 페이지를 몰아낼 것인가?
- i/o 시간 절약을 위해
- 기왕이면 modify 되지 않은 페이지를 victim 으로 선택
- 방법: modified bit (= dirty bit)

. 여러 페이지 중에서 무엇을 victim 으로?

- Random

- First-In First-Out (FIFO)
- 그외
- 용어: 페이지 교체 알고리즘 (page replacement algorithms)

. 페이지 참조 스트링 (Page reference string)
- CPU 가 내는 주소: 100 101 102 432 612 103 104 611 612
- Page size = 100 바이트라면
- 페이지 번호 = 1 1 1 4 6 1 1 6 6
- Page reference string = 1 4 6 1 6

. Page Replacement Algorithms
- FIFO (First-In First-Out)
- OPT (Optimal)
- LRU (Least-Recently-Used)

(1) First-In First-Out (FIFO)
. Simplest
- Idea: 초기화 코드는 더 이상 사용되지 않을 것
. 예제
- 페이지 참조 스트링 = 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
- # of frames = 3
- 15 page faults
. Belady's Anomaly
- 프레임 수 (= 메모리 용량) 증가에 PF 회수 증가

(2) Optimal (OPT)
. Rule: Replace the page that will not be used for the longest period of time
. 예제
- 페이지 참조 스트링 = 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
- # of frames = 3
- 9 page faults
. Unrealistic
- 미래는 알 수 없다!
  cf. SJF CPU scheduling algorithm

(3) Least-Recently-Used (LRU)

. Rule: Replace the page that has not been used for the longest period of time
- Idea: 최근에 사용되지 않으면 나중에도 사용되지 않을 것
. 예제
- 페이지 참조 스트링 = 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
- # of frames = 3
- 12 page faults

. Global replacement
- 메모리 상의 모든 프로세스 페이지에 대해 교체

. Local replacement
- 메모리 상의 자기 프로세스 페이지에 대해 교체

. 성능 비교
- Global replacement 가 더 효율적일 수 있다.

5.3 프레임 할당
. Allocation of Frames

. CPU utilization vs Degree of multiprogramming
- 프로세스 개수 증가 ☞ CPU 이용률 증가
- 일정 범위를 넘어서면 CPU 이용률 감소
- 이유: 빈번한 page in/out
- 쓰레싱 Thrashing: i/o 시간 증가 때문

. 쓰레싱 극복
- Global replacement 보다는 local replacement
- 프로세스당 충분한/적절한 수의 메모리(프레임) 할당

. 정적 할당 (static allocation)
- 균등 할당 (Equal allocation)
- 비례 할당 (Proportional allocation)

. 동적 할당 (dynamic allocation)
- Working set model
- Page fault frequency

- etc.

. Working set model
- Locality vs working set
- Working set window
- Woking set 크기 만큼의 프레임 할당

. Page-Fault Frequency (PFF)
- Page fault 발생 비율의 상한/하한선
- 상한선 초과 프로세스에 더 많은 프레임 할당
- 하한선 이하 프로세스의 프레임은 회수

5.4 페이지 크기
. Page size
- 일반적 크기: 4KB ~ 4MB
- 점차 커지는 경향

. 페이지 크기 영향
- 내부 단편화
- Page-in, page-out 시간
- 페이지 테이블 크기
- Memory resolution
- Page fault 발생 확률

. 페이지 테이블
- 원래는 별도의 chip (TLB 캐시)
- 기술 발달에 따라 캐시 메모리는 on-chip 형태로
- TLB 역시 on-chip 내장

6 파일 할당

. 컴퓨터 시스템 자원 관리
- CPU: 프로세스 관리 (CPU 스케쥴링, 프로세스 동기화)
- 주기억장치: 메인 메모리 관리 (페이징, 가상 메모리)
- 보조기억장치: 파일 시스템

. 보조기억장치 (하드 디스크)
- 하드디스크: track (cylinder), sector
- Sector size = 512 bytes, cf. Block size
- 블록 단위의 읽기/쓰기 (block device)
- 디스크 = pool of free blocks
- 각각의 파일에 대해 free block 을 어떻게 할당해줄까? ☞ 파일 할당

. 파일 할당 (File Allocation)
- 연속 할당 (Contiguous Allocation)
- 연결 할당 (Linked Allocation)
- 색인 할당 (Indexed Allocation)

6.1 연속 할당
. Contiguous Allocation
- 각 파일에 대해 디스크 상의 연속된 블록을 할당
- 장점: 디스크 헤더의 이동 최소화 = 빠른 i/o 성능
- 옛날 IBM VM/CMS 에서 사용
- 동영상, 음악, VOD 등에 적합
- 순서적으로 읽을 수도 있고 (sequential access 순차접근)
- 특정 부분을 바로 읽을 수도 있다 (direct access 직접접근)

. 문제점
- 파일이 삭제되면 hole 생성
- 파일 생성/삭제 반복되면 곳곳에 흩어지는 holes
- 새로운 파일을 어느 hole 에 둘 것인가? ☞ 외부 단편화!
- First-, Best-, Worst-fit
- 단점: 외부 단편화로 인한 디스크 공간 낭비
- Compaction 할 수 있지만 시간 오래 걸린다 (초창기 MS-DOS)

. 또 다른 문제점
- 파일 생성 당시 이 파일의 크기를 알 수 없다 ☞ 파일을 어느 hole 에?
- 파일의 크기가 계속 증가할 수 있다 (log file) ☞ 기존의 hole 배치로는 불가!
- 어떻게 해결할 수 있을까?

6.2 연결 할당
. Linked Allocation
- 파일 = linked list of data blocks
- 파일 디렉토리(directory)는 제일 처음 블록 가리킨다.
- 각 블록은 포인터 저장을 위한 4바이트 또는 이상 소모

. 새로운 파일 만들기
- 비어있는 임의의 블록을 첫 블록으로
- 파일이 커지면 다른 블록을 할당 받고 연결
- 외부 단편화 없음!

. 문제점
- 순서대로 읽기 = sequential access
- Direct access 불가
- 포인터 저장 위해 4바이트 이상 손실
- 낮은 신뢰성: 포인터 끊어지면 이하 접근 불가
- 느린 속도: 헤더의 움직임

. 개선: FAT 파일 시스템
- File Allocation Table 파일 시스템
- MS-DOS, OS/2, Windows 등에서 사용
- 포인터들만 모은 테이블 (FAT) 을 별도 블록에 저장
- FAT 손실 시 복구 위해 이중 저장
- Direct access 도 가능!
- FAT 는 일반적으로 메모리 캐싱

6.3 색인 할당
. Indexed Allocation
- 파일 당 (한 개의) 인덱스 블록 (데이터 블록 외에)
- 인덱스 블록은 포인터의 모음
- 디렉토리는 인덱스 블록을 가리킨다.

- Unix/Linux 등에서 사용

. 장점
- Direct access 가능
- 외부 단편화 없음

. 문제점
- 인덱스 블록 할당에 따른 저장공간 손실
- 예: 1바이트 파일 위해 데이터 1블록 + 인덱스 1블록

. 파일의 최대 크기
- 예제: 1블록 = 512바이트 = 4바이트 × 128개 인덱스, 128 × 512바이트 = 64KB
- 예제: 1블록 = 1KB = 4바이트 × 256개 인덱스, 256 × 1KB = 256KB
- 해결 방법: Linked, Multilevel index, Combined 등

7 디스크 스케쥴링

. Disk Scheduling

. 디스크 접근 시간

- Seek time + rotational delay + transfer time
- 탐색시간 (seek time) 이 가장 크다.

. 다중 프로그래밍 환경

- 디스크 큐(disk queue)에는 많은 요청(request)들이 쌓여있다.
- 요청들을 어떻게 처리하면 탐색시간을 줄일 수 있을까?

. 디스크 스케쥴링 알고리즘

- FCFS (First-Come First-Served)
- ???

7.1 FCFS Scheduling

. First-Come First-Served

- Simple and fair

. 예제

- 200 cylinder disk, 0 .. 199
- Disk queue: 98 183 37 122 14 124 65 67
- Head is currently at cylinder 53
- Total head movement = 640 cylinders
- Is FCFS efficient?

7.2 SSTF Scheduling

. Shortest-Seek-Time-First

- Select the request with the minimum seek time from the current head position

. 예제

- 200 cylinder disk, 0 .. 199
- Disk queue: 98 183 37 122 14 124 65 67
- Head is currently at cylinder 53
- Total head movement = 236 cylinders

. 문제점

- Starvation
- Is SSTF optimal? No! (e.g., 53 → 37 → … = 208 cyl)

7.3 SCAN Scheduling

. Scan disk

- The head continuously scans back and forth across the disk

. 예제

- 200 cylinder disk, 0 .. 199
- Disk queue: 98 183 37 122 14 124 65 67
- Head is currently at cylinder 53 (moving toward 0)
- Total head movement = 53+183 cylinders (less time)

. 토론

- Assume a uniform distribution of requests for cylinders
- Circular SCAN is necessary!

. SCAN Variants: C-SCAN, LOOK, C-LOOK

. C-SCAN

- Treats the cylinders as a cicular list that wraps around from the final cylinder to the first one

. LOOK

- The head goes only as far as the final request in each direction
- Look for a request before continuing to move in a given direction

. C-LOOK

- LOOK version of C-SCAN

. Elevator Algorithm

- The head behaves just like an elevator in a building, first servicing all the requests going up, and then reversing to service requests the other way

부록: 연습문제

* 서론

1. a) 이중모드(dual mode)란 무엇을 의미하는가? b) 이중모드는 컴퓨터에서 어떻게 구현하는가? c) 이중모드를 사용한 입출력장치(i/o devices) 보호 방법에 대해 설명하여라.

2. a) PCB (process control block) 이란 무엇을 의미하는가? b) PCB 에 저장되는 주요 정보의 예를 5가지 소개하라.

3. a) 시스템 콜(system call)과 소프트웨어 인터럽트(software interrupt)는 어떤 관련성이 있는가? b) 유닉스/리눅스 운영체제에서 fork() 시스템 콜은 어떤 용도로 사용되는가? c) 자신이 알고 있는 유닉스/리눅스 시스템 콜의 종류를 세 가지 나열하고 간략히 설명하라.

4. 프로세스의 상태는 new, ready, running, waiting, terminated 등 다섯 종류로 나눌 수 있다. a) 프로세스 상태 변화를 보여주는 상태천이도(state transition diagram)를 그려라. b) ready 와 waiting 상태는 어떻게 다른지 구분하여 설명하라.

5. 운영체제 내에는 프로세스가 순서를 기다리는 여러 개의 줄(queue), 즉 a) ready queue, b) job queue, c) device queue, d) semaphore queue 등이 있다. 위 네 가지 줄의 목적에 대해 각각 간략히 설명하고, 각 줄의 연결을 나타내는 연결도(queueing diagram)를 그려라.

6. 다음 용어의 의미를 간략히 설명하라.

- child process
- command interpreter
- context switch
- distributed operating system
- interrupt service routine
- job scheduler
- kernel
- MMU
- parent process
- privileged instruction
- process control block
- ready queue
- real-time OS
- real-time system
- shell

- software interrupt
- swapping
- system call
- time-sharing system

* CPU 스케쥴링

7. a) preemptive (선점형) scheduling 과 nonpreemptive (비선점형) scheduling 이 어떻게 서로 다른지 설명하여라. b) Round-Robin scheduling 이 preemptive scheduling 에 속하는 이유는 무엇인가? c) 실시간 운영체제가 일반적으로 preemptive scheduling 을 사용하는 이유는 무엇인가?

8. 프로세스 P_1, P_2, P_3 의 CPU burst time 은 각각 6, 2, 3 msec 이며, 세 프로세스는 각기 다른 시간에 ready queue 에 도착했다. 즉 P_1 은 0 msec에, P_2 는 2 msec에, P_3 는 3 msec 에 각각 도착했다. CPU 스케쥴링 방식으로 a) Non-preemptive SJF 알고리즘을 사용 시 CPU utilization (이용률) 은 얼마인가? b) FCFS 알고리즘 사용 시 average waiting time (평균대기시간)은 얼마인가? c) Preemptive SJF 알고리즘 사용 시 average turnaround time (평균반환시간)은 얼마인가? d) time quantum 이 2msec 인 Round-Robin 알고리즘 사용 시 throughput (처리율)은 얼마인가? 단, 각 문제마다 반드시 Gantt chart를 그리고 계산식을 보여야 하며 단위도 나타내어야 한다.

9. 프로세스 P_1, P_2, P_3 의 CPU burst time 은 각각 3, 1, 4 msec 이며, 세 프로세스는 각기 다른 시간에 ready queue 에 도착했다. 즉 P_1 은 0 msec에, P_2 는 4 msec에, P_3 는 5 msec 에 각각 도착했다. Time quantum 이 2 msec 인 round-robin scheduling 을 적용할 때 a) CPU utilization (이용률), b) average waiting time (평균대기시간), c) average turnaround time (평균반환시간), d) throughput (처리율)을 각각 구하라. 단, 반드시 Gantt chart를 그리고 계산식을 보여야 한다.

10. 프로세스 P_1, P_2, P_3 의 CPU burst time 은 각각 4, 2, 8 msec 이며, 세 프로세스는 각각 다른 시간에 ready queue 에 도착했다. 즉 P_1 은 0 msec에, P_2 는 1 msec에, P_3 는 5 msec 에 각각 도착했다. CPU scheduling 에 대한 아래 물음에 답하라 (※ 각 경우마다 Gantt chart를 그리고 수식도 적어라. 답은 계산하지 않아도 되지만, 반드시 단위는 기록해야 한다).

(a) FCFS 스케쥴링을 사용하면 평균 대기시간(average waiting time)은 얼마인가?

(b) 선점형(preemptive) SJF 스케쥴링을 사용할 때 평균반환시간(average turnaround time)은 얼마인가?

(c) Time quantum 이 무한대 (∞) 인 Round-Robin scheduling을 적용하면 평균반환시간은 얼마인가?
(d) 비선점형(nonpreemptive) 우선순위(priority) 스케쥴링을 사용하면 처리율(throughput)은 얼마인가? 단, 프로세스 P_1, P_2, P_3 의 우선순위는 각각 3, 2, 1 이며, 숫자가 작을수록 우선순위가 높다.

11. 프로세스 P_1, P_2, P_3 의 CPU burst time 은 각각 5, 3, 10 msec 이다. CPU scheduling 에 대한 아래 물음에 답하라 (※ 각 경우마다 Gantt chart를 그리고 수식도 적어라. 답은 계산하지 않아도 된다).
(a) 세 프로세스가 동일한 순간에 ready queue 에 도착했다고 가정하자. FCFS 스케쥴링을 사용하면 평균 대기시간(average waiting time)은 얼마인가?
(b) Time quantum 이 3 msec 인 Round-Robin scheduling을 적용하면 평균대기시간은 얼마인가?
(c) 이번에는 세 프로세스가 서로 다른 시간에 ready queue 에 도착했다고 가정하자. 즉 P_1 은 0 msec에, P_2 는 1 msec에, P_3 는 5 msec 에 각각 도착했다. 선점형(preemptive) SJF 스케쥴링을 사용할 때 평균반환시간(average turnaround time)은 얼마인가?
(d) 앞의 c)번 문제에서 비선점형(nonpreemptive) SJF 스케쥴링을 사용하면 평균반환시간은 얼마인가?

12. 프로세스 P_1, P_2, P_3 의 CPU burst time 은 각각 4, 10, 6 msec 이다. CPU scheduling 에 대한 아래 물음에 답하라 (※ 각 경우마다 Gantt chart를 그리고 수식도 적어라. 답은 계산하지 않아도 된다).
(a) 세 프로세스가 동일한 순간에 ready queue 에 도착했다고 가정하자. FCFS 스케쥴링을 사용하면 평균 대기시간(average waiting time)은 얼마인가?
(b) 우선순위(priority) 스케쥴링을 사용하면 평균반환시간(average turnaround time)은 얼마인가? 단, 프로세스의 CPU burst time 이 길수록 우선순위가 높다고 가정한다.
(c) Time quantum 이 5 msec 인 Round-Robin scheduling을 적용하면 평균대기시간은 얼마인가?
(d) 이번에는 세 프로세스가 서로 다른 시간에 ready queue 에 도착했다고 가정하자. 즉 P_1 은 0 msec에, P_2 는 5 msec에, P_3 는 8 msec 에 각각 도착했다. 선점형(preemptive) 우선순위 스케쥴링을 사용할 때 평균대기시간과 CPU 이용률(utilization)은 각각 얼마인가? 단, P_1, P_2, P_3 의 우선순위는 각각 1, 5, 3 이며, 숫자가 작을수록 우선순위가 높다고 가정한다.

13. 프로세스 P_1, P_2, P_3 의 CPU burst time 은 각각 3, 15, 5 msec 이다. CPU scheduling 에 대한 아래 물음에 답하라 (※ 각 경우마다 Gantt chart를 그리고 수식도 적어라. 답은 계산하지 않아도 된다).
(a) 세 프로세스가 동일한 순간에 ready queue 에 도착했다고 가정하자. 평균 대기시간(average waiting time)이 가장 짧도록 하려면 어떤 순서로 스케쥴링 하여야 하는가? 가장 짧은 평균 대기시간을 구하라.
(b) SJF scheduling 을 사용하면 처리율(throughput)은 얼마인가?
(c) Time quantum 이 5 msec 인 Round-Robin scheduling을 적용하면 평균반환시간(average turnaround time)은 얼마인가?
(d) 이번에는 세 프로세스가 서로 다른 시간에 ready queue 에 도착했다고 가정하자. 즉 P_1 은 0 msec에, P_2 는 4 msec에, P_3 는 5 msec 에 각각 도착했다. 선점형(preemptive) SJF scheduling 을 사용할 때 평균 대기시간과 CPU 이용률(utilization)은 각각 얼마인가?

14. 다음 용어의 의미를 간략히 설명하라.
- aging
- convoy effect
- multi-level queue scheduling
- preemptive scheduling

* 쓰레드
15. 프로세스(process)와 쓰레드(thread)의 유사점 및 차이점을 각각 설명하라.

16. a) 다중 쓰레드(multi-thread) 프로그램이란 무엇을 의미하는가? b) 자신이 알고 있는 다중 쓰레드 프로그램의 예를 간략히 소개하라.

* 자원할당도
17. 3개의 프로세스 P_1, P_2, P_3 와 3개의 자원 R_1, R_2, R_3 가 있다. R_1, R_2 는 단일 인스턴스(instance), 즉 동일종류 1개씩의 자원이며, R_3 는 동일 종류 2개의 인스턴스로 구성된다. P_1 은 R_1 을 가지고 있으면서 R_3 하나를 얻으려 대기하고 있고, P_2 는 R_2 와 R_3 하나씩 가지고 있으면서 R_1 을 얻으려 대기하고 있다. P_3 는 R_3 하나를 사용 중에 있으며, 사용이 끝나는 대로 종료될 예정이다. a) 현재의 자원 할당 상황을 보여주는 자원할당도(resource allocation graph)를 그려라. b) 현재 자원 할당 상황은 교착상태에 해당되는가? 그 이유도 설명하라.

* 프로세스 동기화

18. 임계구역 문제(critical section problem)에서 상호배타(mutual exclusion) 조건이란 무엇을 의미하는가?

19. a) 프로세스 동기화(process synchronization)란 무엇을 의미하는가? b) 세마포의 내부 구조를 그림으로 나타내고 간략히 설명하라. c) 어떤 세마포에 다섯 개의 프로세스가 블록(block)되어있다고 가정하자. 이때 세마포 내부의 정수 값(value)은 얼마인가?

20. 생산자-소비자 문제는 mutex, empty, full 등 세 가지 세마포를 사용하여 해결할 수 있다. mutex 는 상호배타 목적, empty 와 full 은 각각 버퍼의 빈 공간 및 차있는 공간에 대한 접근목적으로 사용한다. a) 버퍼에서 데이터를 빼내어 소비하는 동작을 위 세마포를 포함한 코드로 작성하라. b) 생산자와 소비자는 프로세스, 각 세마포는 자원이라고 가정하자. 생산자는 버퍼에 대한 접근 허용을 기다리고 있고, 소비자는 버퍼에서 데이터를 빼내어 소비하는 상황을 나타내는 자원할당도(resource allocation graph)를 그리라. 단, 버퍼 크기는 10이고 그 중 8개가 차있다고 가정한다.

21. 고전적 동기화 문제인 생산자-소비자 문제(producer-consumer problem)에 대한 다음 물음에 답하라. a) 생산자-소비자 문제란 무엇인가? b) 이 문제의 해결을 위해 세 개의 세마포(semaphore)가 사용된다. 사용되는 세마포의 용도를 각각 설명하여라. c) 각 세마포의 초기 값은 각각 얼마인가? 그 이유도 설명하라.

22. 전통적 동기화 문제인 생산자-소비자 문제를 세마포(semaphore)를 사용하여 해결할 수 있다. 아래 코드는 생산자, 소비자의 코드를 대략적으로 나타낸 것으로 full, empty, mutex 등 세 개의 세마포를 사용한다. a) ① ~ ④ 에 들어갈 세마포의 이름은 무엇이며 역할은 각각 무엇인가? b) 위 세 개의 세마포 초기값은 각각 얼마인가? 이유도 설명하여라. 단, 사용하는 버퍼의 길이는 5 라고 가정한다. c) 버퍼가 가득 차서 생산자는 버퍼가 비워지기를 기다리고 있으며, 소비자는 버퍼에서 1개 항목을 빼내고 있는 중이다. 이 상태를 보여주는 자원할당도(resource allocation graph)를 그리라. 단, 생산자와 소비자는 프로세스로, 세 개의 세마포는 자원으로 나타낸다고 가정한다.

```
// 생산자 코드                    // 소비자 코드
①.acquire();                    ③.acquire();
②.acquire();                    ④.acquire();
insert to the buf;              remove from the buf;
mutex.release();                mutex.release();
full.release();                 empty.release();
```

23. 고전적인 동기화 문제 중 유한 버퍼 문제(bounded buffer problem)에 대한 아래 물음에 답하라.

a) 유한 버퍼 문제가 무엇인지 간략히 설명하라.
b) 유한 버퍼 문제에서 생산자(producer)의 코드를 간략히 작성하라.
c) 위의 b)문제에서 사용되는 모든 세마포의 용도를 각각 설명하고 초기값이 얼마인지도 설명하라.

24. 아래 그림은 식사하는 철학자 문제(Dining philosopher problem)에서 사용되는 젓가락을 모니터로 구현한 자바 코드이다. acquire() 는 식사를 위해 젓가락을 잡는 동작이며, release() 는 식사를 마치고 젓가락을 놓는 동작을 의미한다. 이 코드의 ①, ②를 각각 완성하라.

```
class Chopstick {
 private boolean inUse = false;
 synchronized void acquire()
         throws InterruptedException {
         ①
         inUse = true;
 }
 synchronized void release() {
         inUse = false;
         ②
 }
}
```

25. 다음 용어의 의미를 간략히 설명하라.
- busy wait
- critical section

* 실행순서

26. 프로세스 P_1, P_2, P_3 의 코드는 각각 다음과 같다. 세마포(semaphore)를 사용하여 아래 조건이 각각 만족되도록 프로세스의 코드를 수정하라. 세마포의 초기값도 나타내어야 한다. 세마포는 한 개 또는 여러 개를 사용할 수 있다.

P_1: S1
P_2: S2
P_3: S3

(a) S1 이 끝나야 S2 나 S3 가 실행된다. S2, S3 의 순서는 상관없다.
(b) S1 과 S2 가 모두 끝나야만 S3 가 실행된다. S1, S2 의 순서는 상관없다.
(c) S1, S2, S3 의 순서대로 실행된다. 즉 S1 → S2 → S3 의 순서를 따라야 한다.

27. 프로세스 P_1 은 정수형 변수 n을 매 반복시마다 1씩 증가시키고, 프로세스 P_2 는 매 반

복시마다 동일한 변수를 1씩 감소시킨다 (이래 코드 참조). a) 프로세스 동기화에서 임계영역(critical-section)이란 무엇을 의미하는가? b) 이 문제에서 임계영역은 어느 곳인가? c) 이 문제가 항상 올바른 결과를 내도록 세마포(semaphore)를 사용하여 임계구역 문제를 해결하라. d) CPU 스케쥴링 방식에 관계없이 항상 P_1 프로세스가 P_2 프로세스보다 먼저 시작될 수 있도록 세마포를 사용하여 프로그램을 수정하라.

```
int n = 0;

P1: for (int i=0; i<100; i++)
        n++;
P2: for (int i=0--; i<100; i++)
        n--;
```

28. 아래 문제에서 변수 n, i, s 는 각 프로세스의 지역변수이며, value 는 모든 프로세스가 공통적으로 사용하는 전역변수이다. 프로세스 P_1 과 P_2 의 코드는 각각 다음과 같다.

P_1
```
while (true) {
        value = value + n;
        n++;
}
```

P_2
```
while (true) {
        for (i=0; i<100; i++)
                s = s + i;
        value = value - s;
}
```

(a) 프로세스 동기화 문제에서 임계구역(critical section)이란 무엇을 의미하는가?

(b) 위 P_2 프로세스의 코드 내용 중 임계구역에 해당되는 부분은 어디인가? 이유도 설명하라.

(c) 세마포(semaphore)를 사용하여 P_1, P_2 코드의 임계구역 문제를 해결하라. 세마포의 초기 값도 나타내어라.

(d) 프로세스 스케쥴링 방식과 관계없이 항상 P_1 이 P_2 보다 전역변수 value 값을 먼저 업데이트 하도록 세마포를 사용하여 위 P_1, P_2 의 코드를 수정하여라. 세마포의 초기 값도 나타내어라.

29. 동일한 은행계좌에 대해 프로세스 P_1 과 P_2 는 입금, P_3 는 출금을 한다고 가정하자. 입금 동작은 deposit(), 출금 동작은 withdraw() 라는 함수 내에서 일어난다.

(a) 다음 코드는 P_1 과 P_2 가 입금을 마칠 때까지 P_3 가 출금을 하지 못하도록 세마포 sem 을 사용하여 제어하는 코드이다. ① 번에 들어갈 코드는 무엇인가? 또한 세마포 sem 의 초기값은 얼마인가? 그 이유도 함께 설명하여라.

P_1:
```
deposit(); sem.release();
```

P_2: deposit(); sem.release();

P_3: ①; withdraw();

(b) 이번에는 P_1 이 입금을 마쳐야만 P_2 도 입금을 할 수 있도록 위 코드를 수정하여라. P_3 는 P_1, P_2와 관계없이 출금할 수 있다. 사용하는 세마포의 초기값을 결정하고 그 이유도 함께 설명하여라.

(c) P_1 이 입금한 후 P_2 가 입금하고, 마지막으로 P_3 가 출금하도록 위 코드를 수정하여라. 필요하다면 여러 개의 세마포를 사용해도 된다. 이때도 세마포의 초기값을 결정하고 그 이유도 설명하여라.

30. 동일한 은행계좌에 대해 프로세스 P_1 은 입금, P_2 와 P_3 각각 출금을 한다고 가정하자. 입금 동작은 deposit(), 출금 동작은 withdraw() 라는 함수 내에서 일어난다.

(a) 다음 코드는 P_1 이 입금을 마칠 때까지 P_2, P_3 가 출금을 하지 못하도록 세마포 sem 을 사용하여 제어하는 코드이다. ① 번에 들어갈 문장은 무엇인가? 또한 세마포 sem 의 초기값은 얼마인가? 그 이유도 함께 설명하여라.

P_1: deposit(); sem.release(); sem.release();

P_2: sem.acquire(); withdraw();

P_3: ①; withdraw();

(b) 이번에는 P_2 가 출금을 마쳐야만 P_3 도 출금을 할 수 있도록 코드를 수정하여라. P_1 은 P_2, P_3 와 관계없이 입금할 수 있다. 사용하는 세마포의 초기값을 지정하고 그 이유도 함께 설명하여라.

(c) P_1 이 입금한 후 P_2 가 출금하고, 마지막으로 P_3 가 출금하도록 코드를 수정하여라. 필요하다면 여러 개의 세마포를 사용해도 된다. 이때도 세마포의 초기값을 지정하고 그 이유도 설명하여라.

* 교착상태

31. 어떤 컴퓨터 시스템에서 현재 3개의 프로세스 P_1, P_2, P_3 가 실행 중이며, 이 시스템에는 3개의 자원 R_1, R_2, R_3 가 제공되고 있다. P_1 은 R_1 을 사용 중이며 R_2 를 사용하기 위해 기다리고 있다. P_2 는 R_2 를 사용 중이며 R_3 를 사용하기 위해 기다리고 있다. P_3 는 R_3 를 사용 중이며, R_2 을 사용하기 위해 기다리고 있다. R_1 은 3개의 instance 로 구성되며, R_2, R_3 는 각각 한 개의 instance 로 구성된다.

a) 이 시스템의 자원 할당도 (resource-allocation graph) 를 그리라. b) 교착상태 (deadlock)가 일어나기 위한 4가지 필요조건을 각각 간략히 설명하라 c) 이 시스템은 교착상태에 놓여져 있는가? 그 이유도 설명하라.

32. a) 교착상태(deadlock)가 일어나기 위한 네 가지 필요조건 중 보유 및 대기(hold and wait) 란 무슨 의미인가? b) 식사하는 철학자 문제(dining philosopher problem)에서 보유 및 대기 조건이 만족되지 않게 하려면 어떻게 해야 하는가?

33. 식사하는 철학자 문제(dining philosopher problem)에서 다섯 명의 철학자에 대해 각각 0번부터 4번까지 반시계 방향으로 번호를 부여했다.
a) 0, 2번 철학자는 식사 중이며, 1, 3번 철학자는 생각 중이다. 4번 철학자는 식사를 위해 한 쪽 젓가락은 손에 잡았지만 다른 쪽 젓가락은 아직 잡지 못했다. 이 상태를 나타내는 자원할당도(resource allocation graph)를 그리라.
b) 위의 a)문제에서 그린 자원할당도를 볼 때 교착상태(deadlock)가 일어날 것인가? 이유도 설명하라.

34. 교착상태(deadlock)에 대한 다음 물음에 답하라. a) 교착상태란 무엇인가? b) 교착상태가 일어나기 위한 네 가지 필요조건을 각각 간략히 설명하라.

35. 다음 용어의 의미를 간략히 설명하라.
- banker's algorithm
- deadlock avoidance
- dining philosopher problem
- monitor
- readers/writers problem

* 메모리 관리
36. 메모리 관리 기법 중 동적 적재(dynamic loading)와 동적 연결(dynamic linking)은 각각 무엇을 의미하는가?

37. 다음 용어의 의미를 간략히 설명하라.
- backing store
- best-fit
- dynamic linking
- first-fit
- relocation register

* 페이징/세그멘테이션
38. 페이징 시스템에서 한 페이지의 크기가 항상 2의 지수승인 이유는 무엇인가?

39. 페이징을 사용하는 운영체제에서 메모리의 특정 부분에 대한 접근을 보호하는 방식에

대해 설명하라.

40. 페이지 테이블 내용이 순서대로 1 7 0 2 5 와 같을 때 다음 페이지 크기에 대해 물리 주소 300번지는 논리주소 몇 번지에 해당되는가? a) 페이지 크기 = 64바이트 b) 페이지 크기 = 256바이트

41. 메모리 관리 기법으로 페이징(paging)을 사용하는 컴퓨터를 가정하자. 한 페이지의 크기가 512 바이트이며, 페이지 테이블의 내용이 순서대로 4, 0, 1, 2, 3, 5 등과 같을 때, (a) 논리주소가 16진수 0x123 라면 물리주소는 얼마인가? (b) 물리주소가 십진수 1500 이라면 논리주소는 얼마인가?

42. 페이지 크기가 16바이트인 페이징 시스템에서 페이지 테이블의 내용이 순서대로 아래와 같다.

1 6 2 7 3 8 4 9 5 0

a) 논리 주소 50번지는 물리 주소 몇 번지에 해당되는가? b) 물리 주소가 이진수로 1011001 이라면 논리 주소는 얼마인가? c) 30바이트 크기의 프로세스를 메모리에 배치한다면 내부 단편화(internal fragmentation)로 인한 메모리 손실은 몇 바이트인가?

43. 페이징(paging)을 사용하는 메모리 관리 시스템에서 페이지 테이블 내용은 순서대로 다음과 같다.

1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

논리주소(logical addres)가 16진수 0x3456 일 때 아래 페이지 크기에 대해 물리주소(physical address)는 16진수로 각각 얼마인지 답하라. a) 1KB b) 8KB

44. 컴퓨터에서 사용되는 주소는 논리주소와 물리주소로 나눌 수 있다. a) 논리주소와 물리주소란 각각 무슨 의미인가? b) 논리주소와 물리주소 등 두 개의 주소를 분리함으로써 얻을 수 있는 잇점은 무엇인가? c) 세그멘테이션을 사용하는 시스템에서 논리주소는 (s,d)로 나눌 수 있다. 단, s는 세그멘트 번호이며 d 는 변위(displacement)이다. 세그멘트 테이블 내용이 아래 그림과 같을 때 물리주소 2014번지에 대응되는 논리주소 s 와 d 의 값은 각각 얼마인가?

	base	limit
0	1000	2000
1	0	1000
2	3000	1500
3	5000	500

45. a) 세그멘테이션(segmentation) 방식이 페이징(paging) 방식에 비해 잘 사용되지 않는 가장 큰 이유는 무엇인가? b) 페이징을 사용하는 세그멘테이션(paged segmentation)이란 무엇이며, 이 방식의 장단점은 각각 무엇인가?

* 가상메모리

46. 요구페이징을 사용하는 시스템에서 페이지 참조 스트링이 다음과 같다. 1, 2, 3, 4, 5, 1, 2, 3. 다음 페이지 교체 알고리즘에 대해 각각 몇 회의 페이지 부재가 발생할까? 단, 메모리는 3개의 프레임으로 구성된다고 가정한다. a) OPT b) LRU

47. 요구 페이징을 사용하는 어떤 시스템에 5개의 메모리 프레임이 있으며, 이 프레임에는 현재 0, 1, 3번 째 페이지 내용이 들어있고 나머지는 비어있다. 페이지 크기는 8바이트라고 가정한다. CPU 에서 다음과 같은 16진수 바이트 주소를 낼 때 아래 물음에 답하라.

2 1 15 9 1E 26 23 24 28 12 11 1 8 A C 24 1A 1C

a) 위 주소를 페이지 참조 스트링(page reference string)으로 나타내어라.
b) 위 페이지 참조 스트링에 대해 LRU 페이지 교체 알고리즘을 적용하면 몇 번의 페이지 결함(page fault)이 발생하는가? 최종적인 메모리 프레임 내용도 나타내어라.
c) OPT 교체 알고리즘을 사용하면 몇 번의 페이지 결함이 일어나는가? 최종적인 메모리 프레임 내용도 나타내어라.

48. 페이지 크기가 16바이트인 요구 페이징(demand paging) 시스템을 가정하자. 메모리는 4개의 프레임으로 구성된다. 이 시스템에서 CPU 는 아래 바이트 주소를 순서대로 낸다.

12 13 14 16 18 20 30 34 10 11 28 29 30 31 32 33 34 70 76 77 60 62 64 12 13

a) 처음에 모든 프레임은 비어있었다고 가정하자. 페이지 부재(page fault)가 4번 일어났을 때 페이지 테이블 내용을 나타내어라.
b) 만일 페이지 교체 알고리즘으로 LRU 를 사용한다고 가정하면 위 주소에 대해 몇 번의 페이지 부재가 발생하는가? 이유도 설명하여라.

49. 요구 페이징(demand paging) 기반의 메모리 관리 시스템에서 다섯 개의 프레임이 있으며 LRU 페이지 교체 알고리즘이 사용된다고 가정하자. 프레임은 처음에 모두 비어 있다.
a) 이 메모리는 두 개의 프로세스 P_1, P_2 가 각각 실행된다. 처음에는 P_1 이 실행되어 P_{10}, P_{11}, P_{13}, P_{11}, P_{10} 의 순서대로 페이지를 요구한다. 그 후 콘텍스트 스위칭이 일어나 P_2 가 실행되며 P_{20}, P_{21}, P_{23}, P_{24}, P_{20} 의 순서대로 페이지를 요구한다. P_2 가 요구한 페이지에 대해 몇 번의 페이지 부재(page fault)가 발생하는가? 단, 전역 페이지 교체(global page replacement)를 사용한다고 가정한다.
b) 이번에는 지역 페이지 교체(local page replacement)를 사용한다고 가정하고 위의 a)문

제를 풀어보라. 단, P_1 에게는 세 개의 프레임이, P_2 에게는 두 개의 프레임이 각각 할당된다.

50. 요구 페이징(demand paging) 시스템에서 페이지 크기(page size)에 대한 설명으로 옳지 않은 것을 모두 찾으라 (단, 오답을 선택하면 감점 있음).
a) 페이지 크기가 클수록 페이지 테이블 크기가 커진다.
b) 페이지 크기가 클수록 외부 단편화(external fragmentation)가 커진다.
c) 페이지 크기가 클수록 backing store 에 대한 입출력이 효율적이 된다.
d) 페이지 크기가 클수록 페이지 부재(page fault) 회수가 많아진다.

51. 다음 용어의 의미를 간략히 설명하라.
- Belady's anomaly
- demand paging
- demand segmentation
- dirty bit
- internal fragmentation
- local page replacement
- page fault
- thrashing
- TLB
- valid bit
- victim page
- virtual memory

* 파일 시스템
52. 다섯 개의 디스크 블록을 사용하는 어떤 파일이 있다. 이 파일은 디스크의 6, 3, 9, 2, 1 번째 블록에 순서대로 놓여있다. a) FAT (File Allocation Table) 방식을 사용하는 파일시스템일 때 이 파일을 나타내는 FAT 내용을 그림으로 나타내라. b) 색인할당 (indexed allocation) 방식을 사용하는 파일시스템일 때 이 파일의 인덱스 블록 구조를 그림으로 나타내라.

53. MS-DOS 등에서 사용되는 FAT 파일 시스템에서 FAT 테이블 내용이 아래와 같다고 하자.

14	0	1	-1	2	4	5	3	8	9	-1	11	6	12	-1	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

a) 디렉토리 내용에 따르면 어떤 파일이 디스크의 5번째 블록부터 놓여있다고 한다. 이 파

일은 몇 개의 블록을 사용하고 있는가? 단, FAT 테이블 내용에서 -1 은 파일의 끝을 의미한다고 가정한다.
b) FAT 파일 시스템의 장단점을 다음 측면에서 각각 설명하여라. ① 내부 단편화 ② 외부 단편화 ③ 접근 속도 ④ 안전성

54. 색인할당(indexed allocation) 방식을 사용하는 파일 시스템에서 디스크의 블록(block) 크기는 2KB 라고 가정하자. 블록 위치를 나타내는 인덱스(index) 또는 포인터(pointer) 크기가 4바이트라면 이 방식으로 할당할 수 있는 파일의 최대 크기는 몇 바이트인가? 단, 파일 당 하나의 인덱스 블록만 사용한다.

55. 10개의 블록(block)으로 이루어진 어떤 파일을 가정하자. 이 파일의 8번째 블록을 읽기 위해서는 몇 번의 디스크 접근이 필요한가? 이유도 설명하라. 다음 네 가지 파일 할당 방법에 대해 각각 답하라. 단, 파일 정보를 담은 디렉토리(directory)와 FAT는 운영체제 메모리에 이미 적재되어있다고 가정한다.
a) 연속할당 (contiguous allocation)
b) 연결할당 (linked allocation)
c) 색인할당 (indexed allocation)
d) FAT 파일 시스템

* 디스크 스케쥴링
56. 디스크 스케쥴링 방식으로 FCFS, SSTF, SCAN 을 각각 들 수 있다. a) 디스크에 대한 접근 요구가 매우 많을 때 SSTF 방식의 문제점은 무엇인가? b) 디스크에 대한 접근 요구가 매우 적을 때 이 세 방식의 성능을 각각 비교하라. 즉 각 방식마다 디스크 헤더가 움직인 총거리는 각각 어떻게 다른가?

57. 0번부터 99번까지 100 개의 실린더(cylinder)를 가진 디스크 드라이브가 있다. 디스크 헤더는 현재 80번째 실린더에 위치하고 있으며 직전에는 83번에 있었다. 디스크 큐(disk queue)에는 읽을 실린더 값이 순서대로 아래와 같이 들어있다고 가정하자.

24 50 82 72 20 96 37

현재 헤더 위치에서 시작하여 큐 내의 모든 실린더 내용을 읽으려 할 때 다음 디스크 스케쥴링 알고리즘에 대해 헤더가 움직여야 하는 전체 거리를 각각 계산하라. a) SSTF b) C-LOOK

58. 어떤 디스크 드라이브가 100개의 트랙을 가지고 있다고 가정하자. 디스크 헤더는 현재 30번째 트랙에 놓여있으며, 그 전에는 26번째 트랙에 있었다. 디스크 큐에는 다음과 같은 트랙 접근 요구가 순서대로 놓여있다.

25 10 40 85 5

다음 디스크 스케쥴링 알고리즘을 적용할 때 현재 위치에서 디스크 헤더가 움직여야 되는 총 이동거리는 각각 얼마인가? a) SSTF b) C-LOOK c) FCFS

59. 다음 용어의 의미를 간략히 설명하라.

- C-LOOK
- elevator algorithm
- indexed allocation
- SSTF