OS (기말대비)

▼ 병행성 문제

▼ 목차

- 병행성의 정의 **(다중프로그래밍, 멀티프로세서, 분산처리)**
- 용어정리 **(데드락, 라이브락, 상호배제, 임계영역, 경쟁상태, 기아상태)**
- 병행성 문제를 해결하기위한 기법
  - SW 기법
    - **Dekker 알고리즘**
      - Turn Variable
      - Busy Waiting
      - Busy Waiting Modified
      - Busy Flag Again
    - **세마포어 (→ 생산자-소비자 문제(유한버퍼), 식사하는 철학자)**
      - 이진세마포어
      - 범용세마포어
    - **모니터**
    - 메시지 Call (쓰레드끼리)
  - HW 기법
    - **인터럽트 불능화**
    - **특별한 명령어**
      - TestAndSet
      - Exchange (인텔계열.. ECHG)
- **교착상태**
  - 발생조건
    - Mutual Exclusion
    - Hold And Lock

        - Non Preemption
        - Circular Wait
    - 처리 :
        - **예방**
        - **회피 (RVCA 벡터/행렬)**
            - 프로세스 시작거부 →
            - 자원 할당거부 → 은행원 알고리즘 (Safe/Unsafe)
        - **발견**
            - W,Q행렬 이용 발견.

▼ 내용정리

- 병행성 종류
    - **다중프로그래밍** : 1개의 CPU 에 여러개의 프로세스 실행 ← **Interleaving**
    - **멀티프로세서** : 여러개의 CPU에 여러개의 프로세스 중첩 실행 ←- **Overlaping**
    - **분산처리** : 여러개의 코어에 여러개의 프로세스 실행
        - Clustering : 병렬 컴퓨팅
        - grid : 병렬 네크워크
- 용어정리
    - **임계영역** : 공유자원 코드영역, 프로세스가 점유할 때 다른 프로세스가 못 들어가게해야하는 코드 영역
    - **상호배제** : 공유자원 코드영역, 프로세스가 점유할 때 다른 프로세스가 못 들어가게해야함
    - **데드락** (교착상태) : 어느 한 프로세스가 공유자원에 lock을 풀어줄 때까지 둘 이상의 프로세스가 대기하고 있는 상태
    - **라이브락** : 프로세스의 상태는 바뀌는데 실제로 공유자원에 대한 접근이 안되고있는 상태
    - **기아상태** : 스케쥴러에의해 무한정 간과되고 있는 상태
    - **경쟁조건 : 둘 이상의 프로세스가 공유자원에 서로 접근하여 수행순서에 따라 결과가 달라질 수 있는 상태.**

        - ex) b=1, c=2 인 자원에 P1 은 b = b+c, P2는 c = b+ c 를 한다고 했을 때
            - P1→ P2 를 하면 **b=3, c=5** 됨
            - P2→P1 를 하면 **b=4, c=3** 됨
- **병행성 문제를 해결하는 방법 (아래부터 하나씩 서술)**
    - SW 해결법 : Dekker, 세마포어, 모니터, **메시지콜**
    - HW해결법 : Interrupt 불능, 특별한 명령어

▼ Dekker 해결방법 (TV → BW → BWM → BFA)

- Turn Variable

```
A프로세스
while (turn != 0) { do noting }

임계영역 접근
임계영역 수정
turn = 1;


B프로세스
while (turn != 1) { do nothing }

임계영역 접근
임계영역 수정
turn = 0;

>>> 상대방쪽에서 Turn 안바꿔주면 DEADLOCK 걸림
```

- Busy Waiting

```
A프로세스
while (flag[1]) {
  do Noting
}
  ------> 스케쥴링
flag[0] = true;
임계영역 접근
임계영역 수정
flag[0] = false;


B프로세스
while (flag[0]) {
  do Noting
}
------> 스케쥴링
flag[1] = true;
```

```
임계영역 접근
임계영역 수정
flag[1] = false;

>>> 임계영역에 둘 다 접근해버림 (Mutual Exclusion 보장 X)
```

- Busy Waiting Modified

```
A프로세스
flag[0] = true;
------> 스케쥴링
while (flag[1]) {
  do Noting
}
임계영역 접근
임계영역 수정
flag[0] = false;


B프로세스
flag[1] = true;
------> 스케쥴링
while (flag[0]) {
  do Noting
}
임계영역 접근
임계영역 수정
flag[1] = false;

>>> lockstep 단계에서 Deadlock 발생.
```

- Busy Flag Again

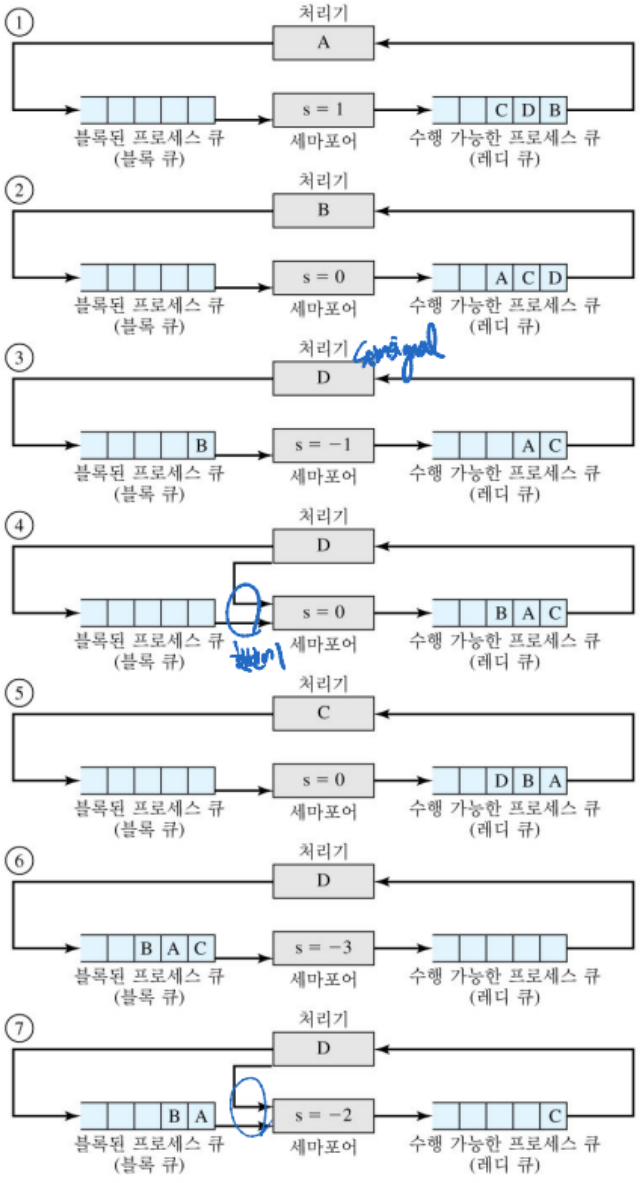
```
flag[0] = true;
------> 스케쥴링
while (flag[1]) {
  flag[0] = false;
  delay...
  flag[0] = true;
}
임계영역 접근
임계영역 수정
flag[0] = false;


B프로세스
flag[1] = true;
------> 스케쥴링
while (flag[0]) {
  flag[1] = false;
  delay...
  flag[1] = true;
}
```

```
임계영역 접근
임계영역 수정
flag[1] = false;


>>>  A,B 프로세스가 동시에 진입하면 LiveLock이 걸릴 순 있지만,
그럴일은 없음
```

▼ HW 해결방법

- Interrupt 불능화
  - 쉽지만, 부하가 크다.
    - 중요한 이벤트를 받을 수가 없다.
  - 멀티프로세서, 분산컴퓨팅에선 여전히 병행성 문제가 발생된다.
- **특별한 HW 명령어**
  - 원자적연산 (Atomic) 연산을 지원해주는 HW 명령어를 지원.
  - **TestAndSet**
    - i가 0이면 1로 바꿈, i가1이면 return;
  - **Exchange** (Intel에선 **ECHG**)
    - register변수와 memory변수를 교환한것
  - 문제점
    - **바쁜대기**를 사용 + 기아상태를 초래할 수 있음.
    - 우선순위가 낮은 프로세스가 HW명령어를 수행함 + 우선순위가 높은 프로세스가 스케쥴링됨 → 우선순위가 낮은 프로세스가 HW 를 놔줄 기회가 없어 **DeadLock**이발생.
-

▼ **세마포어 (중요)**



- 세마포어 구현 (범용 세마포어 기준)
  - **세마포어 : P (Pause:Wait), V(Signal)**
  - 초기화
  - wait
    - sem--;
    - sem줄인 값이 음수일 시 큐에서 대기
  - signal
    - sem++;
    - sem값이 음수였으면 증가시키면서 큐에서 프로세스 하나 뺌. (없으면 값만 증가)

- 이진 세마포어
  - **표현력이 떨어짐**
  - wait할 때 1이면 값줄임, 0이면 대기
  - signal일 때 큐 비어있으면 1로 변경, 아니면 sem=1 변경

```
wait
  if (sem == 1)
    sem = 0;
  else
    Queue에서 대기 (append)

signal
  if (queue is empty)
    sem = 1;
  else
    queue에서 하나 추출
```

- 범용 세마포어
  - **세마포어 변수값**
    - 0이상 : 해당 임계영역에 접근할 수 있는 프로세스 수.
    - 음수 : 큐에서 대기하는 프로세스의 수
- **생산자-소비자 문제**
  - 항상 in ≥ out 이어야함 (out이 in을 가로지를 수 없음.)
  - Queue는 **상호배제가 지원되어야함**
  - (나중에 유한버퍼를 지원해야함) == **in이 Queue 최대크기를 넘어설 수 없음.**

```
생산자
while ((in+1) % n == out) { 대기 }
N[in] = c;
in = (in+1) % n;

소비자
while (in == out) {대기}
c = N[out];
out = (out+1) % n;
```

  - 코드
    1. 무한버퍼 + 이진세마포어로 구현 (잘못된 구현 → 해결책)

```
전역
int n; //in-out을 가리키는 공유자원
semaphore s = 1; // 락(상호배제)를 위한 세마포어
semaphore d = 0; // 순서를 위한 세마포어

생산자
while(true) {
  produce(); //생산
  semwaitB(s);
  append(); //추가
  n++;
  if (n == 1)
    semsignalB(d); // 소비자가 실행할 수 있게
  semsignalB(s);
}

소비자
semwait(d); //순서를 위해 대기
while(true) {
  semwaitB(s); //상호배제
  take();
  n--;
//m = n;
  semsignalB(s);
-------> 스케쥴링이되면?
/*
  해당위치에서 스케쥴링이 되면 돌아왔을 때
아래 if(n==0) 부분이 실행될 수 없고 소비자가 연속으로 2번 실행됨.

생산자가 스케쥴링되어 돌아온 시점 n=1, d=1

그럼 while이 끝나는 지점이
n=1 d=1
n=0 d=0 //if에서 d하나 줄이고 n하나 소비
n=-1 d=0 상태로 들어가서 문제가 발생
(n과 d가 일치하지 않음)

=====> 해결책
1. 임시변수 m 으로 두어 signal(s) 전에 m=n설정
2. 아래 2번 범용세마포어로 변경

*/
  consume();
  if (n == 0) //if(m == 0)
    semwaitB(d);
}
```

2. 무한버퍼 + 범용 세마포어

```
전역
semaphore s = 1; // 락(상호배제)를 위한 세마포어
semaphore n = 0; // 순서를 위한 세마포어

생산자
```

```
while(true) {
  produce(); //생산
  semwait(s);
  append(); //추가
  semsignal(s);
  semsignal(n);
}

소비자
while(true) {
  //semwait n,s 위치바뀌면 deadlock 발생 (s를 가진채 hold-lock)
  semwait(n);
  semwait(s); //상호배제
  take();
  semsignal(s);
  consume();
}
```

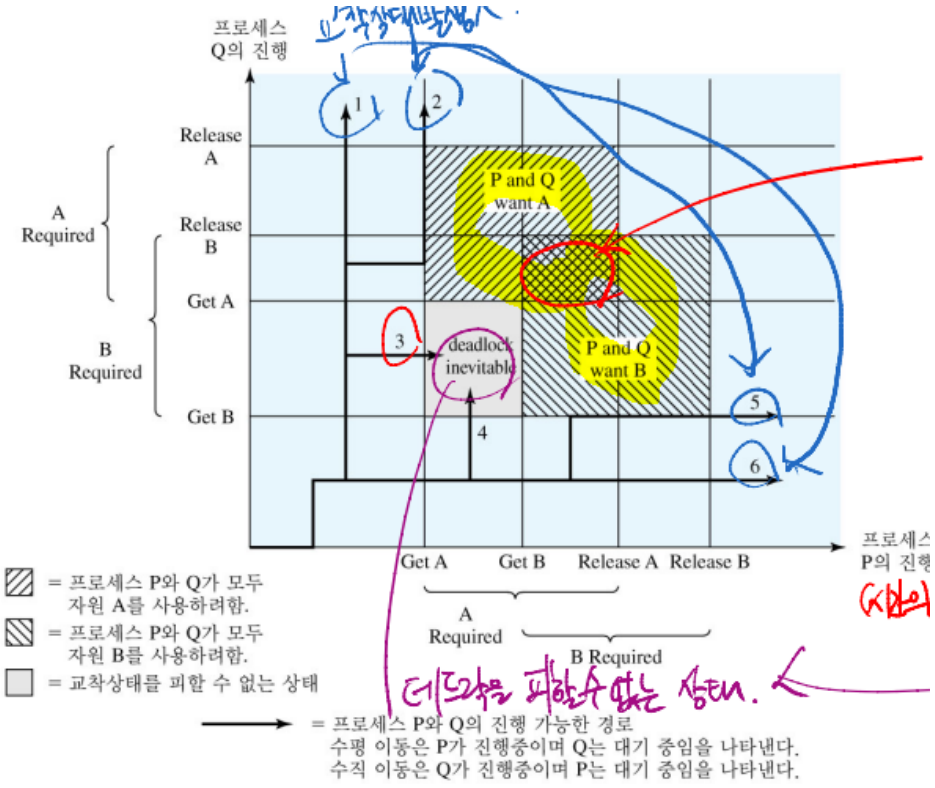
3. 유한버퍼 + 범용 세마포어
    - 유한버퍼 순서를 위한 세마포어 e 추가

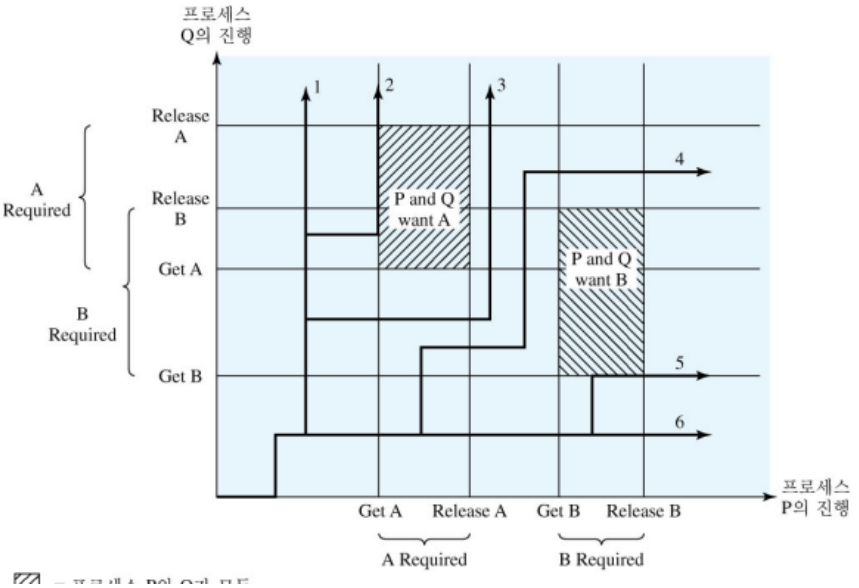
```
전역
semaphore s = 1; // 락(상호배제)를 위한 세마포어
semaphore n = 0; // 순서를 위한 세마포어
semaphore e = sizeof(buf); //생산한계를 조절하기 위한 세마포어

생산자
while(true) {
  produce(); //생산
  semwait(e);
  semwait(s);
  append(); //추가
  semsignal(s);
  semsignal(n);
}

소비자
while(true) {
  //semwait n,s 위치바뀌면 deadlock 발생 (s를 가진채 hold-lock)
  semwait(n);
  semwait(s); //상호배제
  take();
  semsignal(s);
  semsignal(e);
  consume();
}
```

- 모니터
    - Semaphore가 구현이어렵고 복잡해서 나온 대안
    - PL 수준에서 지원하는 병행성 해결 SW (자바, 파스칼 등에서 지원)

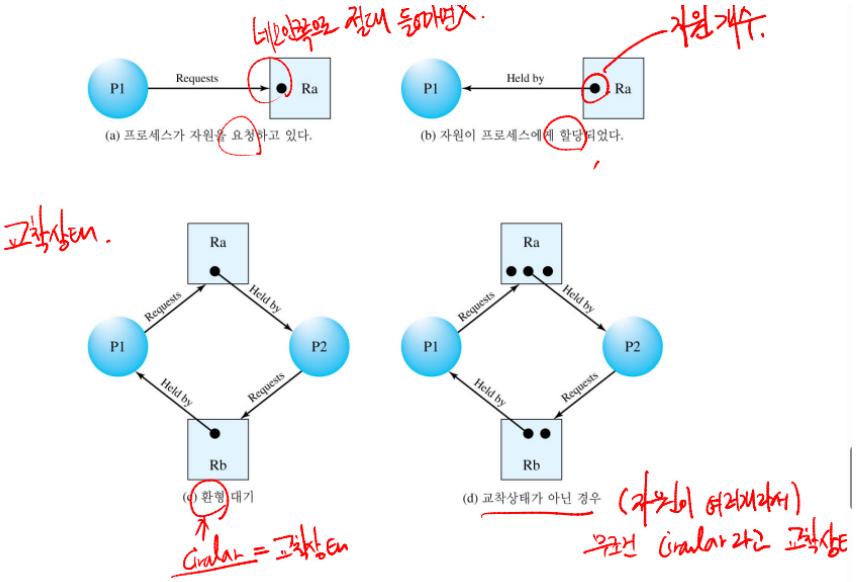
▼ 교착상태 (중요)

- 교착상태 (프로세스가 진행하지 못하고 영구적으로 블록되어있는 상태)
  - **결합 진행 다이어그램 (Joint Progress Diagram)**
    - 서로의 프로세의 getA - release A 꼭짓점을 그린 사각형을 그렸을 때 겹치는지점이 존재하면 **교착상태가 발생하는 지점**
    - 서로의 다이어그램이 겹치기전 지점은 **교착상태를 피할 수 없는 지점**





- **자원 할당 그래프**
  - 자원이 할당되었을 때는 자원내부에서 프로세스를 향하게 화살표
  - 요구할 때는 네모박스 안으로 선이 들어가면 안됨



- **교착 상태가 발생하는 조건**
  - 상호배제 (Mutual Exclusion)
  - 비선점 (No Preemption)
  - 점유및 대기 (Hold and Lock)

↓(필수조건)

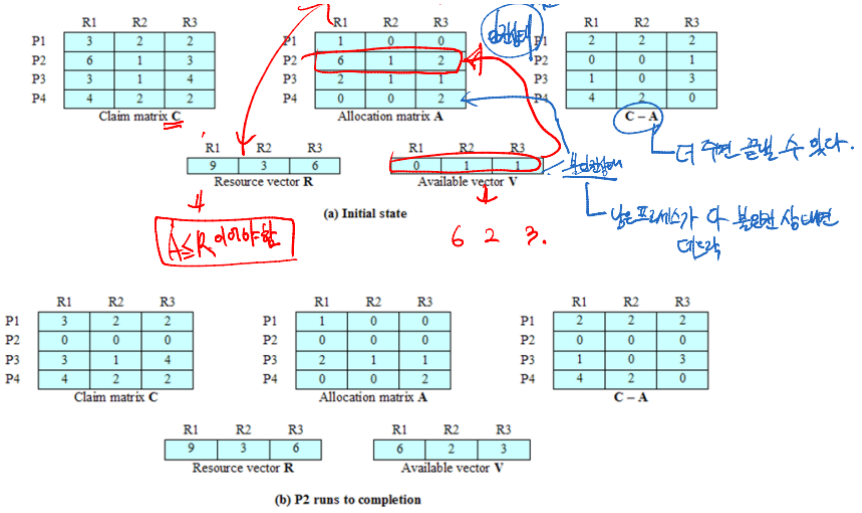
  - 순환대기 (Circular wait)

(필요충분 조건)

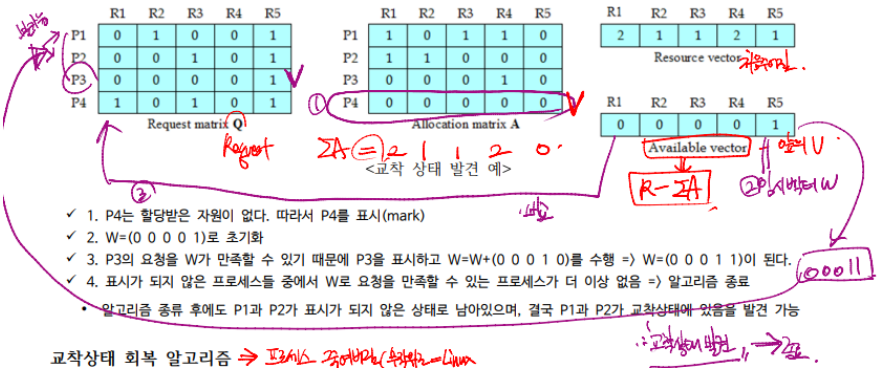
- 교착상태를 해결하는 방법
  - **예방**
    - 위의 3개의 필수조건을 막아보겠다. (실질적으론 불가능)
    - 선점이 가능하도록.. → 선점 부하가 생김
    - 자원할당 순서를 정해놓는다.(컴파일시점) → 점진적 자원할당이 안된다. (추가할당이 불가능)
  - 한 프로세스가 모든 자원을 처리한 후 다음프로세스 실행 → 자원할당 효율이 떨어짐.
  - **회피 (교착상태가 발생할 것 같은 경우는 피한다.)**
    - 프로세스 상태 벡터/행렬
      - R:자원 총 개수, V:남은 자원개수, A:할당된 자원 개수 : C:요구한 자원개수.
      - $R = R_1, R_2, ..., R_n$
      - $V = V_1, V_2, ..., V_n$
      - $A = [A_{i1}, A_{i2}, ..., A_{in}]$ for i = 1 ~ m
      - $C = [C_{i1}, C_{i2}, ..., C_{in}]$ for i = 1 ~ m
    - **규칙**
      - $R_j = \Sigma A_{ij} + V_j$
        - 자원의 총량은 **프로세스별 할당된 자원량 총합 + 가용자원량 이다.**
      - $R_j >= C_{ij}$ for all i,j
        - 자원의 총량이 프로세스별 요구량보다 커야한다.
      - $A_{ij} <= C_{ij}$ for all i,j
        - 할당된 자원량은 요구자원량보다 작아야한다.
    - **프로세스 시작거부 (RV(벡터) CA (행렬))**
      - 다음 실행될 프로세스가 시작이되려면 , 현재프로세스까지의 요구 자원량 + 다음 프로세스 요구자원량이 R보다 작아야한다.

- $R_j >= C_{(n+1)j} + \sum C_{ij}$

- **자원할당 거부 → 은행원 알고리즘 (Q = C-A (남은 요구량), W벡터 : 남은 자원량 = R-∑A)**
  - C-A행렬 (남은 요구량) 의 모든 요소가 V벡터 (남은 자원량) 보다 작은 프로세스 실행
    - 프로세스 실행 후 $V_j = V_j + A_{ij}$ 로 갱신 (할당된 자원 free)
  - 나온 V를 통해 남은 프로세스 C-A에서 가능한 프로세스 할당 (safe 프로세스)
  - 해당 과정에서 unsafe 한 프로세스는 자원을 할당하지 않음.



- **발견**
  - 알고리즘
    - 하나도 자원이 할당외 되어있지 프로세스는 체크
    - W = R-A (남은자원량) 으로 할당할 수 있는 프로세스 체크 (W와 Q행렬비교)
      - 체크 후 $W_j = W_j + A_{ij}$ 추가 (W벡터 갱신)
    - W에서 할당할 프로세스가 없으면 종료
    - 만약 종료하였는데 체크가 안된 프로세스가 있으면 데드락 발생한 것으로 간주



- **문제점 : 해당 알고리즘을 사용해도 이후 진행사항에서 교착상태가 발생할 수 있음.**

- 식사하는 철학자 문제 : 철학자 5명
  - 해결법 1 : 철학자보다 하나작은 수의 semaphore로 제한

```
semaphore fork[5] = {1};
semaphore room = 4;

void philosopher(int i) {
  while(true) {
    think();
    semwait(room);
    semwait(fork[i]);
    semwait(fork[(i+1) % 5]);
    eat();
    semsignal(fork[(i+1) % 5]);
    semsignal(fork[i%5]);
    semsignal(room);
  }
}
```

- Cigaratte-Smoker 문제
  - Smoker 프로세스는 담뱃잎,불,종이 3개 중 1개만 가지고 있고, Supplier 는 2개를 생산하는 프로세스다. 초기에 서로다른 종류 2개를 가진 쓰레드가 3개 모두 실행될 때 Supplier 는 초기에 가장 먼저 실행된 후 요청받을 때 마다 무한으로 2개의 종류를 생산해낸다. 병행성 문제를 해결하라.

```
Semaphore item=0; //락을 위한 세마포어
Semaphore more=0; // more 요청을 위한 세마포어
Semaphore tmp=0; // Smoker Queue 세마포어
int counter;
#Smoker i
wait(item); //락을 들고 들어감
if (flag[(i-1) % 3] and flag[(i+1) % 3]) {
    flag[(i-1) % 3] = flag[(i+1) % 3] = false;
    Smoking();
    while(count > 0) {
```

```
        count--;
        signal(tmp);
    }
    signal(more);
}
else { //재료부족..
    count++;
    signal(item);
    wait(tmp); //Queue에서 대기
}


#Supplier
flag[?] = flag[??] = true;
signal(item);
wait(tmp);
```

▼ 파일시스템

▼ 목차

- **파일** (Persistance : 영속성)
  - User Readable File → SystemCall Interface → Virtual FS (파일시스템종류..) → 버퍼 → Disk Driver
  - **+ 디렉토리.. (dirent 에는 inode, strName, [strlen, reclen]**
- 파일 시스템의 종류
- 시스템콜 (SystemCall Interface)
- 가상 파일시스템 구현내용
  - 연속할당 파일시스템
  - 불연속할당 파일시스템
    - 연결리스트
    - 인덱스 *Unix-Inode
    - 파일테이블 *DOS-MFT(Master File Table)
- **디스크 구조와 정리**
  - 디스크 스케쥴링 방식
    - 디스크 구현방식
      - 섹터분할방법 * Track Skew (왜곡) , 재배치…
        - + Track Buffer
      - 디스크 Read/Write 시간 (3개의시간)

        - Seek/Rotational/Transfer Time
    - FIFO
    - SSTF (Shortest SeekTime First)
    - SCAN
        - Scan (look)
        - CScan (look)
        - N-step SCAN
            - N=1 : FIFO다.
            - N=무한대 : SCAN과 유사
        - FSCan
- **파일시스템 심화**
    - UFS : UnixFS 의 한계
        - 느리다. (Fragmentation 때문)
    - FFS : Fast File System
        - 디스크-실린더 친화적인 배치 (블럭/실린더 그룹)
        - **name-based locality (이름기반 구역성)**
    - XFS시초 FSCK 와 저널링 그리고 LFS (log Structed File System)
        - FSCK
            - 방법론 (S1,B1,I2,D3)
        - **저널링 (TxB, D[v2], I[v2], B[v2], TxE)**
            - Journel Write → Commit → Checkpoint
            - **Data / Metadata Journaling**
            - **저널링 3단계 (Journaling Write → Commit → Checkpoint)**
            - 저널링 Recovary
            - 저널링 Update 주기
            - Ciruclar Log (순환로그)
        - LFS
            - 디스크를 순차적으로 사용 (왔다갔다하지않고 작성)

- CR 접근 → IMap 접근 → Inode 접근 → Data Block 접근 순으로 이루어짐
    - CR : Checkpoint Region블록

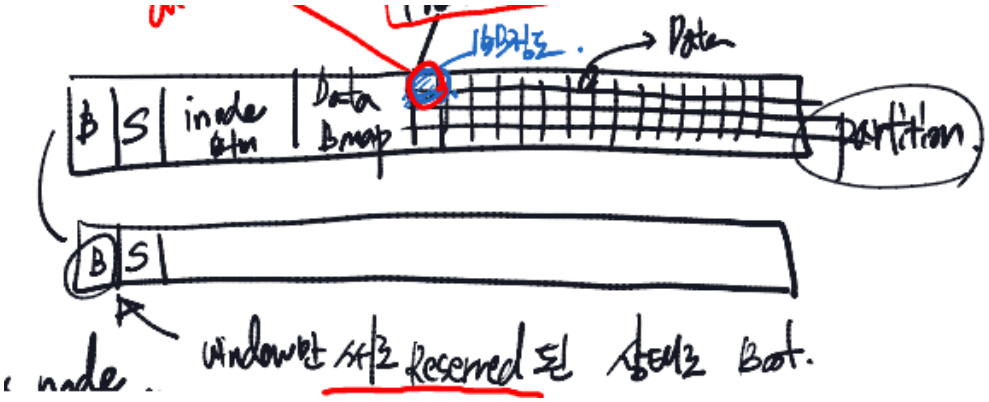
▼ 내용정리

LEC 10

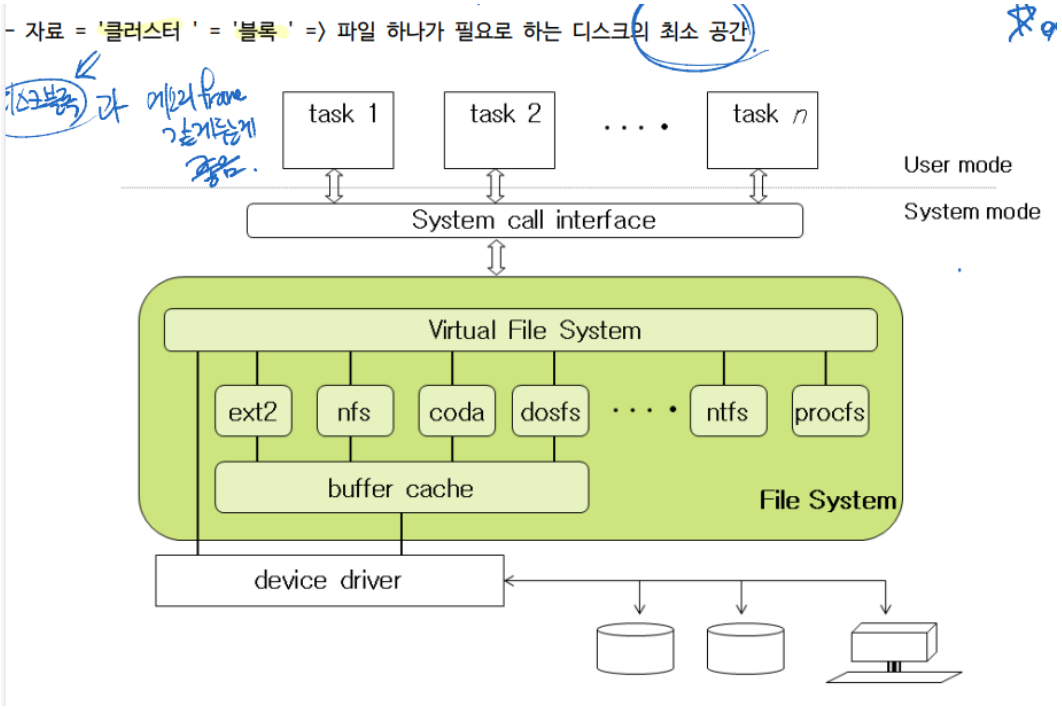
- **파일 (File)**
    - **파일은 바이트스트임의 구조를 띔.**
    - 데이터를 영속적으로 저장하기 위한 수단
    - 접근방법 : **(User Readable name → Low-level name → Metadata → File 접근)**
- **디렉토리 (디렉토리도 파일의 한 종류)**
    - 디렉토리 데이터블록에는 아래 2개는 필수적으로 가지고있다. (dirent)
        - UserReadable name (string) : d_name
        - low-level name (inode) : d_ino
        - [옵션] : strlen (공백문자포함)
        - [옵션] : reclen (4B 로 몇 공간?)
- **파일시스템의 종류**
    - Unix : UFS (Unix FS) → S III → S IV → S V (NFS (Network FS) / RFS (Remote FS))
    - Linux : **ext2** → **ext4** → **.. + XFS** (저널링기법)
    - Dos : FAT16 → FAT32 → exFAT (I-node개념을 약간추가) + NTFS (저널링 도입)
    - **BSD : FFS (Fast FS)**
    - Mac : HFS → APFS (Apple FS)
    - Coda FS (모바일 파일시스템에 유리) → AFS (Andrew FS)
    - Proc FS (프로세스정보를 가지는 파일시스템 : **메모리 내에 존재하는 FS**)
- **Sysacll Interface**

- 생성 (creat, open) | 쓰기 (write) | 읽기 (read) | file offset : (lseek)
- 지연쓰기 (Write back) → sync()
- 하드/소프트 링크 (symlink, link)
- inode의 축약판 stat

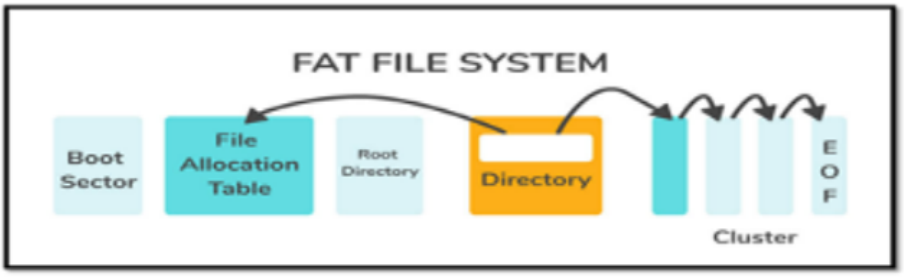
- **(중요) : File System 깊게 이해**
  - **디스크 내부 구조**

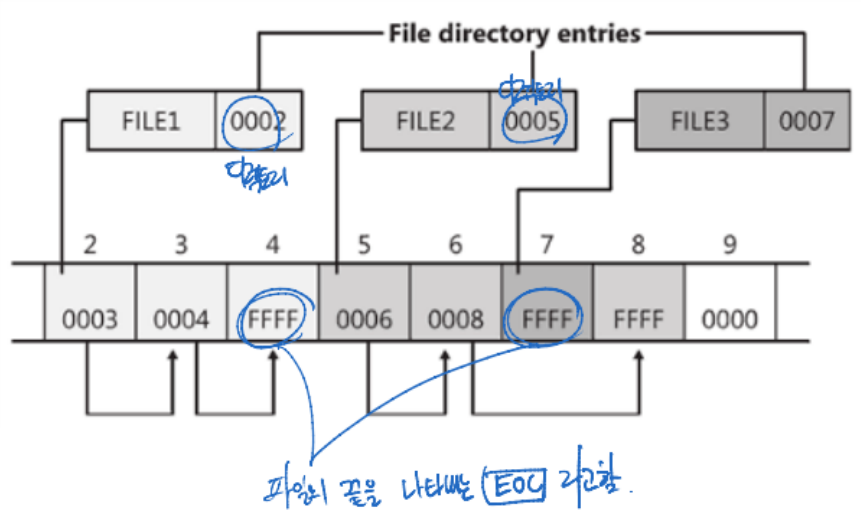


    - B : Boot Block ← 메인 파티션에는 MBR (Master Boot Record) 라는게 존재
    - S : Super Block
    - [i-node or FAT | Bitmap…]
    - 데이터블록 …
  - 파일시스템은 아래와같은 과정으로 흘러감
    - Task요청 → SystemCall Interface → Virtual FS (Ext2, FAT, …) → 버퍼 → 디스크드라이버



- **파일의 최소단위**
  - 자료 = 블록 = **클러스터 (Cluster)**
- 파일 할당방식에 따른 분류
  - 연속 할당 분류
    - 시작 위치 + 할당 블록 개수 만 알면 됨
    - **외부 단편화 (이빨 빠짐 현상) 이 커짐**
  - 불연속 할당 분류
    - 체이닝 : 블록마다 연결 리스트
      - 블록마다 체이닝으로 인한 비효율 발생
    - **인덱스 : Index node (Inode) ← Unix, BSD, Linux …**
    - **파일테이블 : FAT (File Allocation Table) ← DoS 가 주로..**
  - **DoS FS**



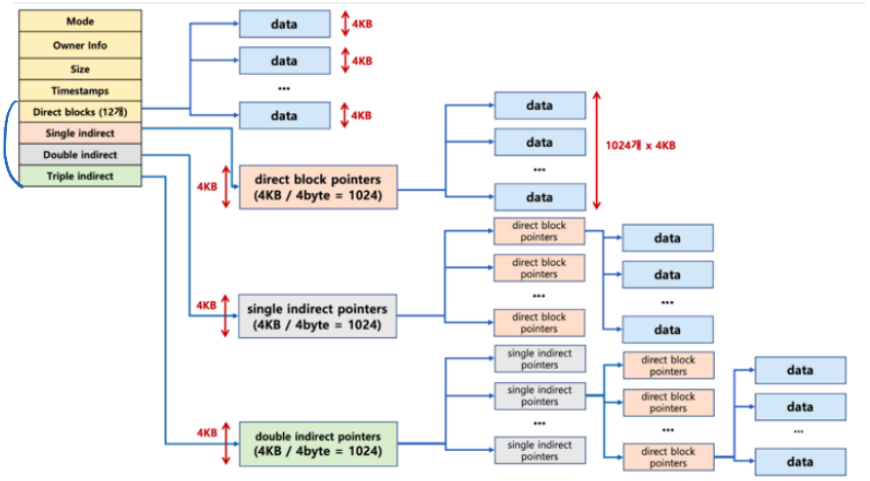


- 기본적으로 FCB(파일 제어블록) 이 MFT로 구성됨 (MFT = Master File Allocation Table) {MFT == FAT}
  - FAT의 인덱스는 할당된 정숫값으로 상태를 구분한다.
    - 0 : 할당 X
    - 1 : OS 에 Resrverd 된 비트
    - 1 ~ 0xFFFF 이전 : 할당된 블록 번호
    - 0xFFFF : EOC : end of chain

- Super 블록에 FAT안의 Root Directory Idxf를 가지고있음
  - SuperBlock → FAT Root 접근 → … → **체이닝으로 연결됨**
- **원하는 블록의 위치를 FAT에서 찾았음**
  - B1 → B2 → … → 0xFFFF (EOC : End Of Chain) 블록까지 체이닝으로 연결

- **Unix FS**
  - 파일의 종류

    - 일반파일 | 디렉토리파일 | 특수파일 | Pipe파일 | 링크 (hard,sym)
- FCB (파일제어블록) 으로 Inode 를 사용한다.
- Inode 내부에는 파일의 메타데이터 정보가 담겨있음
    - 파일크기, 파일블록 위치, 접근시간(time), …
- Inode에서 파일 크기를 확장하기 위해 **간접 블록을 사용할 수 있다.**



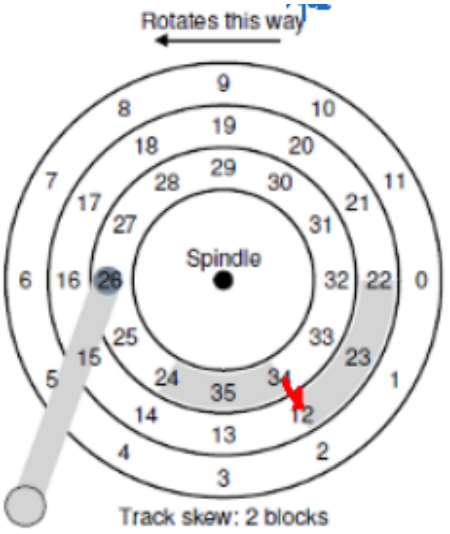
    - **Unix 기본 파일시스템은 기본적으로 12개의 직접 블록 + 1개의 간접 + 1개의 2중간접 + 1개의 3중간접포인터를 가지고 있다.**
    - 1중 간접 : 파일 포인터 → 간접블록 (4K / 4 = 1024개의 주소블록) → 실제블록
    - 2중간접
    - 3중간접…
- **I-node 블록 위치 계산법과 섹터번호 계산방법 (중요)**
    - **찾으려는 Inode 번호 (얜 따로 요청) : 32번 I-node**
    - 4개의 메타정보가 필요함 (SuperBlock에 위치함)
        - **I-node 구조의 크기 : 512B**
        - I-node 블록의 시작위치 : 12KB
        - 디스크 블록의 크기 : 4KB
        - 디스크 섹터의 크기 : 512B
            - 블록크기가 섹터크기의 배수이다.

- **계산 :**
  - I-node 주소 : 12KB (시작위치) + **4K** * 32(번호) * / (4K (디스크블록크기) / 256B (i-node 블록크기)) = 12KB + 4KB * 32/16 = **20KB**
  - 디스크 섹터 주소 : 20KB / 512B (섹터크기) = **40번**

LEC 12 : 디스크구조

디스크 **구조방식 (Track Skew)**과 **정책 (Track Buffer, Scheduler) 등에 대한 내용**

- **디스크 구조방식**
  - 디스크 섹터는 **고정적으로 분할된게 아닌 약간 틀려지면서 분할되어있다.**
  - **Track Skew (왜곡)**
    - 섹터마다 크기가 동일하게 맞춰주기 위함
    - **디스크가 멈춰서 다시 되돌리는 (재배치) 시간을 줄여주기 위함**



- 디스크 탐색시간
  - Seek Time : 탐색시간 (**제일 중요 (가장 오래걸림)**)
    - 디스크 헤드가 움직이는 시간 (10,000 RPM == 1 회전당 6ms)
  - **Rotational Time : 회전지연시간**

        - RPM으로 표시
            - RPM : 분당 회전수
            - 1회전당 시간 : 60,000 / RPM
                - 10,000RPM 은 회전 당 : (60,000 / 10,000 = 6ms) 의 회전지연시간이 걸린다.
    - **Transfer Time : 전송시간**
        - 헤드가 위치한 순간부터 데이터가 읽히는데 걸리는데 시간.

- **디스크 정책**
    - **물리메모리에 Track Buffer 캐시를 둔다.**
        - 동일 트랙 후속조치를 위해 **한 트랙을 읽을 때 트랙의 전 섹터 데이터를 물리메모리에 옮기는 것**
    - 디스크에 쓸 때
        - Write Back : 모아뒀다 Clustering 방식으로 씀
            - 긴 데이터의 경우
        - Write Through : 바로 씀
            - 짧은 데이터의 경우
- **디스크 스케쥴링 알고리즘**
    - **FIFO**
        - 먼저들어온 작업부터 먼저 실행
    - **SSTF (Short Seektime First)**
        - 현 위치에서 SeekTime이 짧은(거리가 짧은) 블록 부터 처리
        - 기아 문제 발생 (SeekTIme이 길면 스케쥴링이 안되는 기아 문제 발생)
    - **SCAN (look (바닥/천장을 안찍음))**
        - 진행방향으로 가까운 작업부터 시행하고 감 (반대방향은 큐에서 대기)
    - **C-SCAN (look (바닥/천장을 안찍음))**
        - 진행방향이 한방향으로만 설정 (위→아래에서는 IO를 함, 아래→위는 안함 등등..)

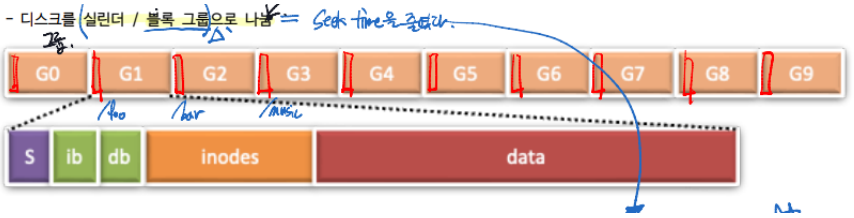
—> SCAN C-SCAN도 결국엔 기아문제가 발생할 수 있음.

암고정현상 (Arm Stickness) : 고밀도 다중표면 디스크에서는 특정트랙의 반복요청으로 암이 움직이지 않아 기아상태를 유발하는 것.

- **N-step Scan :**
  - Queue를 N개를 둔다.
  - N=1 : FIFO와 비슷
  - N = 무한대 : SCAN과 비슷한 효과를 보임.
- **FSCAN**
  - Queue를 2개를 두고 스캔중에 발생되는 IO Task는 두번째 큐에 넣는다.
  - 그렇게 번갈아가면서 시행한다.

Lec 13 : UFS, FFS, 저널링, Log Structered FS

- UFS : Unix FS 의 한계
  - UFS에서는 **성능이 최악**이다.
  - 데이터 블록이 **Random**하게 배치된다.
    - **Fragmentation** 하게 배치되어 디스크 대역폭이 너무 커진다.
- **FFS : Fast File System**
  - **이름기반 구역성 (Named-Based Locality) 을 이용한 블록/실린더 친화적인 파일시스템 (실린더,블록 마다 그룹으로 나누고 해당 그룹 내부에 B,S,Bitmap, I/D 블록이 존재)**



  - **파라미터 배치 (Parametered Placement) : 블록번호 위치를 순차읽기 (Serial Read) 로부터 재배치 시간을 줄이기 위해 블록번호를 번갈아가며 배치**

(FFS Standard vs. Parameterized Placement)

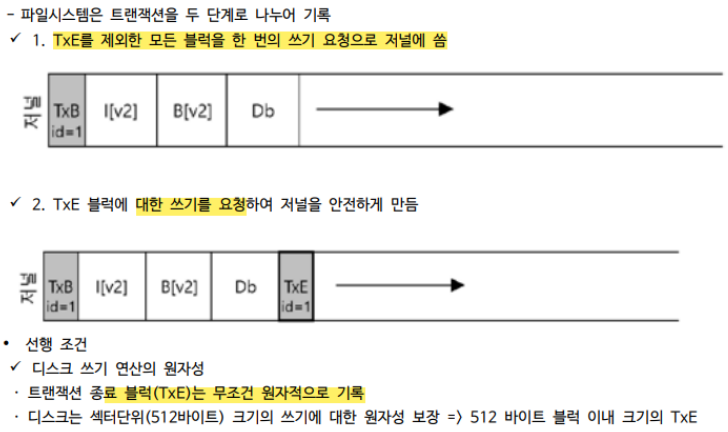
- **디스크 신뢰도**
  - 전원결함이나 시스템 결함에도 디스크는 어느정도 일관성을 유지해야함 → FSCK 나 Journaling 도입
  - **일관성이 없는 데이터 (중요)**

```
# 일관성 유무 : Data, Inode Bitmap, Inode

Data Inode InodeBitmap
  O     X     X     ==> Consistent (일관성O) : Data Loss 상태
  X     O     O     ==> Consistent (일관성O) : Read Garbage Data
------------------------ 아래부터는 다 InConsistent
  X     O     X     ==> inconsistent , Read Garbage Data
  X     X     O     ==> inconsistent , Space Leak (Bitmap만 On)
  O     X     O     ==> inconsistent , 고아상태
  O     O     X     ==> Inconsistent
```

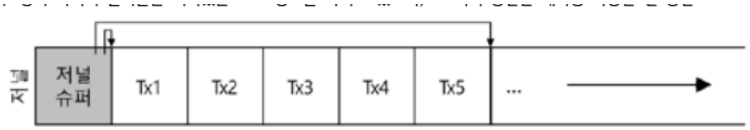
- FSCK
  - 일관성없는 상태여도 그대로 두었다가 Re_Boot를 할 때 작업.
  - Re-Boot시 시작하는 작업 (S1,Bitmap1, I2, D3)
    - [S**검사**] **S블록 검사 (슈퍼블록에서 가리키는 크기와 진짜 파일시스템 크기)**
    - [**Bitmap**] **free Block 검사**
      - 비트맵과 실제로 빈 블록인지 확인 ←비 일치시 I-node로 재접근
    - [**Inode**] **Inode 상태 검사 (I-node가 정상적으로 파일을 가리키는가?)**
    - [**Inode**] inode Link 파일 검사 (할당되어있는데 어떤 디렉토리도 없는 I-node는 lost+found 디렉토리로 이동)
    - [**DataBlock**] **Duplicate 중복**

        - 같은 데이터를 가리키는 2개이상의 블록이 있는지 검사

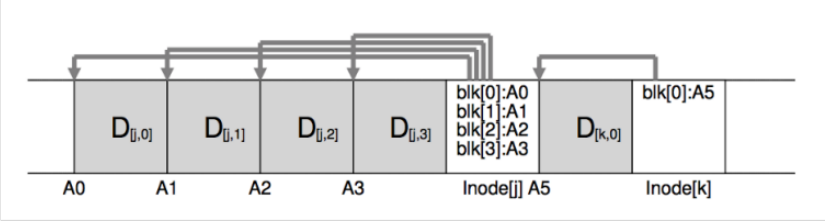
          ← Inode 검사로 초기화
      - [**DataBlock**] **Bad Block**
        - 파티션 영역을 넘어서는 블록 검사
      - [**DataBlock**] **Directory 검사**
        - 디렉토리 데이터블록에 ., .. 이 있는지, .. inode가 할당되어있는지..
- **Journaling (EXT2, JFS, XFS, NTFS)**
  - 쓰기 요청을 하면 벼변경될 부분만 저장 : “**Write-Ahead” + “Log”**
  - 종류
    - **Data Journaling**
      - TxB, D[v2], I[v2], B[v2], TxE
    - **Metadata Journaling** :데이터는 저널링블록에 저장하지 않음 (이중저장방지)
      - TxB, I[v2], B[v2], TxE
  - **방법**
    1. **Journal Write**
       - TxE를 제외한 TxB, I[v2], B[v2], (D[v2]) 등을 순서없이 작성
    2. **Commit**
       - 1번과정이 끝나면 **TxE**를 쓰면서 Commit 상태가됨
       - TxE 는 **원자적 연산**이 보장되어야함
    3. **Checkpoint**
       - **저널링 정보를 디스크로 업데이트**



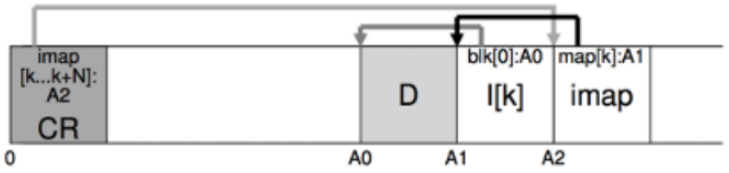
- **저널링 복구 방법**
  - Commit 전이라면?
    - **Undo :** 아무것도 할 필요 없음 (커밋 Log를 디스크가 확인만 할 뿐)
  - Commit 직후라면? (Commit ~ Checkpoint이전)
    - **Redu-Logging : 일관성 유지를 위해 Commit된 블록을 디스크에 저장 후 이 과정을 재생 (Replayed)**
- **저널링 업데이트 주기 : 저널링을 모아서 한 번에 하기때문..**
  - 저널링 데이터는 **Transaction Buffer (Journaling Buffer) 에 모아두고 일정 주기마다 업데이트를 함**
  - **일정 주기 5초마다 Journaling 기록**
  - **sync(), fsync() 호출 마다 저널링 기록**
- Circular log
  - 저널링 최대 공간을 넘었을 때 **Journaling Superblock**을 통해 끝(가장 최근), 시작(가장 오래된포인터) 를 둠



- LFS (log Structered FS) - 참고 (가볍게)
  - FFS에서 블록하나를 쓰기 위해 많은 접근을 방지하기 위해 나온 기법
  - 즉, 왔다갔다하지 않고 **Segment 내용을 차례대로 데이터블록→Inode블록→반복…→Imap블록을 저장 후 CR 블록 갱신**



- 쓰기 버퍼링 기법을 사용
  - Segment : 디스크에 한 번에 기록하는 단위로 얘가 충분히 커지면 디스크에 순차적으로 기록
- **Inode 를 찾는 시간**
  - **UFS < FFS < JFS 순으로 Inode 찾기 시간이 오래걸림**
    - JFS는 복잡함
  - JFS에서 I-node를 찾는 방법
    - **맨 앞 CR 블록 (Checkpoint Region) 에서 Imap 위치를찾아옴**
    - **Imap 위치에서 Inode 블록 위치를 찾음**
    - **Inode 블록에서 Data 블록들을 접근함**



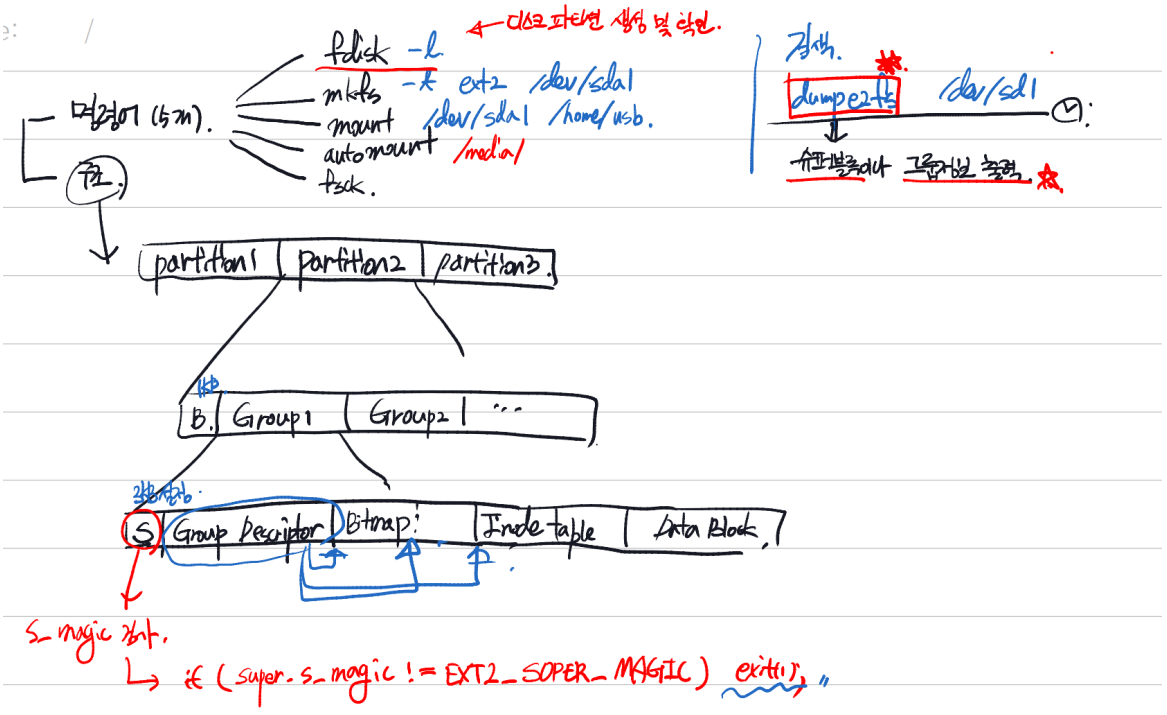
▼ RAID

- Strip : 디스크의 나눠진 블록 단위
- Stripe (스트라이프) : RAID 디스크의 한 행

- RAID0 : 스트라이핑
  - 큰 입출력 성능 : **읽기 쓰기 모두 매우 높음**
  - 작은 입출력 요구율 : **읽기 쓰기 모두 매우 높음**
- **RAID1 : 미러링**
  - 큰 입출력 성능 : 읽기 는 매우 빠름, 쓰기는 단일디스크와 비슷

    - 작은 입출력 요구율 : **읽기는 단일디스크의 2배, 쓰기는 비슷**
- **RAID2 : 해밍코드 중복**
    - 큰 입출력 성능 **: 모든 레벨에서 읽기 쓰기 가장 빠름**
    - 작은 입출력 요구율 : 단일디스크의 거의 2배임.
- **RAID3 : 비트 인터리브드 패리티**
    - 큰 입출력 성능 : **모든 레벨에서 읽기 쓰기 가장 빠름**
    - 작은 입출력 요구율 : 단일디스크의 거의 2배임.
- **RAID4 : 블록 인터리브드 패리티**
    - 큰 입출력 성능 : RAID0 과 읽기는 비슷, 쓰기는 상당히 낮음
    - 작은 입출력 요구율 : RAID0 과 읽기는 비슷, 쓰기는 상당히 낮음
- **RAID5 : 블록 인터리브드 패리티 분산**
    - 큰 입출력 성능 : RAID0 과 읽기는 비슷, 쓰기도 낮음
    - 작은 입출력 요구율 : RAID0 과 읽기는 비슷, 쓰기도 낮음
- **RAID6 : 블록 인터리브드 이중 분산 패리티**
    - 큰 입출력 성능 : RAID0 과 읽기는 비슷, RAID5보다 쓰기는 낮음
    - 작은 입출력 요구율 : RAID0 과 읽기는 비슷, RAID5보다 쓰기는 상당히 낮음

▼ EXT2 분석



- **기본 스킬**
  - **일단 32Bit 4개씩 묶어서 앞에 번호를 매긴다. (줄 수를 의미함.)**
    → 16비트는 두개를 묶어 크게 동그라미 하나를 친다.

  <S 블록 분석>

  - 0x0400 ~ 0x1000 이전까지의 비트를 32개를 4개씩 묶어서 분석한다.
  - **S블록에서 원하는 블록 위치를 위처럼 찾으면 쉽다.**

  <흐름>

  1. **B블록 → ext2_group_desc 구조체 에서 i_node_table 위치 찾기 (little endian + 0 3개 붙여주기)**
  2. **[Special inode 번호] Root idx 찾기 (EXT2_ROOT_INO 2) ←2일때 기준으로**
     - i_node_table 시작 위치 **+ 0x100 해주기.**
  3. [**i-node 데이터블록 위치로 Jump] :** 밑도 끝도 없이 **ext2_inode 구조체의 i_block[EXT2_N_BLOCKS] 를 찾기 ← (보통 3번째 3번째 32비트값임..)**
  4. **Root I-node에서 데이터블록 따라가기 (little Endian + 0x100)**
  5. 따라간 데이터블록에서부터는 **struct ext2_dir_entry_2 구조체 를 확인**
     - **(4,2,1,1 덩어리 + name덩어리)**
       - **즉, 덩어리 8개 이후에 값이 나옴**
     - 32 (I-node) | 16 (rec_len) | 8 (name_len) | 8 (file_type) | name(가변)
       - **name의 길이는 16비트 (덩어리 2개) 에서 확인하면됨.**
       - file_type (덩어리 1개) ⇒ **이 값이 1이면 일반파일 , 2이면 디렉토리이니까 해당 값 이용을 좀 하자.**
  - 3번째 ~ 5번째 과정을 일반파일이 나올 때 까지 계속 반복해주면 된다.
    - **(주의! : 파일 타입은 i-node 블록에서 찾을 수 있다. (16비트짜리) 즉, 데이터블록에서 찾으려고 하지마라, 결국 따라가야 나온다.) : ext2_inode 구조체의 첫 번째 2개덩어리가 File mode 비트이다.**

▼ 빈출 오답 및 지엽노트

강의자료

- **i-node 블록 위치 찾기 문제 (4KB 를 잘 빼먹음)**

    - I-node 블록 위치 : I-node 시작위치 (12KB) + **4KB** (블록크기) * 32 (inode 번호) / (4KB (블록크기) / 256B (inode 크기))
    - inode 섹터 위치 : i-node 블록 위치 / 섹터크기 (512B)
- **디스크 암 고정 현상**
    - 고밀도 트랙 디스크가 저밀도, 다층 디스크보다 트랙에서 오랫동안 암이 움직이지 않는 현상
- **Amortization (양도)**
    - 디스크 청크(Chunk) 크기를 충분히 해서 IO 시간을 상대적으로 줄이는 기법.
- **저널링 용어**
    - Journal Commit 이다. 그냥 Commit이라고 적지말자
- **Transaction Buffer = Journal Buffer**
- 병행성 문제에서 다중프로그래밍은 **인터리빙** 방식으로 멀티프로세싱과 분산처리는 **중첩(Overlapping) 방식으로** 진행된다.

기출문제

- **디스크 상에는 파일시스템에 존재하는 모든 파일들의 inode를 내포한 ( )이 있으며 파일이 오픈되면 그 파일의 inode가 주 기억장치로 반입되어 ( ) 에 저장됨**

  ⇒ i-node table
- **23번 : EXT2 67,000 번째 바이트 블록에 접근하기 위한 IO 개수? (1000B로 블록크기 가정)**

  ⇒ 이해 안감
- 24번 : 파일당 최대 하나의 inode 할당된 파일시스템에서 4KB 블록, 4B 파일포인터, 128B inode 를 가지고 있으면 그때 1GB 파일시스템을 지원하게 하는 블록크기?
    - **(4K-4) / 4 = 992, 1GB/992 = 1MB**
- **25번 : 디스크 트랙 문제**
    - **time stamp를 미리 찍어두지 말고 굴찍하게 25ms 단위로 찍고 사이사이 새로운 값 나올때마다 추가**
- (p10) **실수주의 10번**
    - 시스템은 **교착상태**가 발생할 수 있다.

      → **교착상태** 를 물었는데 나는 **경쟁상태**로 착각해서 풀었음

      교착상태는 Hold and Lock (무한루프) 가 아니라면 발생하지 않음

- 25번 **세마포어 변수값 0으로 페이크**
- 29번 : **세마포어 병행처리가 없을 때 어떻게 될 것인가 경우의수 문제**
  - **Proc 1 : wait(U) 만 있음**
  - Proc 3 : wait(V) 만 있음 을 이용해서 체크를 해두고 포인트를 잡아 풀면 쉽게 풀린다.
- **6번 Read-Write lock에서 복수 정답**
  - read의 read완료 (**release) 조건에** readcount ≤ 0 도 인정
- **0번 : 세마포어 경우의 수에 따른 변수값 (차근히 해보면서 풀면됨, 시간아낄필요없는 문제)**
  - **락 X : (A+B)! / (A! * B!) 만큼 경우의 수가 나옴**
  - **상호 락 (1) : 실행순서에 따라 AB냐 BA 냐의 2가지 경우의 수가 나옴**
  - **순서 락 (0) : 1가지 경우의 수가 나오는 경우가 많음**
- **주의!!!! (P16) : 블록 크기 8KB, inode structure 에 16개의 블록 포인터 (12직접, 1간접, 2이중간접, 1삼중간접(reserved)) 일 때 현 파일 시스템 크기 + 128TB까지 파일 시스템 지원하게 하는 방법**
  - 1) **현 파일시스템 크기는 reserved 는 고려하지 않아야함**
    - (2048^2 * 2 + 2048 + 12) * 8KB
  - 2) 128TB 까지 지원하게 하려면 일단 현재 블록크기로 최대 몇 개까지 지원되는지 확인해야함
    - d_inode에서 **파일 사이즈 표현 고려 : 32비트(4B) 는 2^32 인덱스를 가질 수 있음**
      - **2^32 인덱스 * 블록크기 = 지원되는 파일 최대 크기수**
      - 8KB * 2^32 = 2^45 B 까지 사이즈를 표현할 수 있음 (128TB = 2^47B)
        - 이를 방지하기 위해 블록 크기를 2^15 B 로 변경 = 32KB로 변경함
    - **블록크기를 키워도 128TB 는 삼중간접 하나로는 부족하기 때문에 삼중간접 블록을 하나 더 추가**

- (p17) : **2번 자원할당 그래프**
  - 자원할당 그래프에서 **데드락인지** 판단하는 방법
    - =⇒ 자원할당그래프가 → **O** → **라고 생겼으면 나가는 방향 '→' 의 자원을 대기하지 않고 얻을 수 있으면 해당 O 는 뒤에 딸린 → 자원들은 모두 해제 가능**

        - 위의 과정을 고려해서 **순환대기**가 발생하는지 체크해 볼 것.

- **P17 디스크 스케쥴링**
    - 빠른 계산방법 : SCAN 알고리즘에서는 방향이 꺾이는 지점과 시작지점을 빼서 쉽게 헤드 이동크기를 구할 수 있다.
    - **헤드 평균거리 이동거리에서는 탐색을 안하는 부분 192→1 부분도 계산해줘야함**
        - 평균거리이므로 **원소 총 개수**로 나누면됨

▼ 교재

▼ 병행성처리

- **역 어셈블러 (objdump) :** 실행 파일을 대상으로 역어셈블러를 실행하면 해당 프로그램이 어떤 어셈블리 명령어들로 구성되었는지 알 수 있다
- **프로그램 프로파일러 도구들 : valgrind, purify**
- **원자적 연산**
    - 예를 들어 파일 시스템은 **저널링 (journaling)**이나 **쓰기-시-복사 (Copy-On-Write)**와 같은 기법을 사용 하여 디스크 상태를 원자적으로 전이시킨다. 연속된 동작들을 ( ) 으로 만든다는 개념은 간단하게 “전부 아니면 전무”라고
      표현할 수 있다
- **락 소유자 :** 락을 획득한 쓰레드
- **mutex : POSIX 라이브러리의 락**
- **Lauer’s Law :** 더 짧고 간결한 코드가 더 좋은 코드이다.
- **컨디션 변수 :** 조건이 참이될 때까지 spinlock 방식이 아닌 sleep 방식으로 기다리게 해주는 변수
- **Mesa Sementic (메사) ↔ Hoare Sementic (상태유지 보장)**
    - 깨어난 쓰레드가 실제 실행되는 시점에도 그 **상태가 유지된다는 보장은 없다**. 이런 식으로 시그널을 정의하는 것
- **Hills’s Law : 간단한 개념이 복잡한 개념보다 더 좋다는 의미**
- UNIX의 **시그널**은 프로세스간의 통신 방법으로 사용
  특정 시그널을 받지 못하도록 하는 것을 시그널을 masking 했다 라고 한다.
- **차단 (=동기), 비차단(=비동기)**
    - 차단 (또는 동기(synchronous)) 인터페이스는 호출자에게 리턴하기 전에 자신의

작업을 모두 처리하는 반면 비차단 (또는 비동기(asynchronous)) 인터페이스는 작업을 시작하기는 하지만, 즉시 반환하기 때문에 처리되어야 하는 일이 백그라운드에서 완료가 된다.

    - **차단 호출**은 주로 I/O 때문에 발생한다. 예를 들어, 작업 완료를 위해서 디스크에서 읽어야 하는 자료가 있다면, 디스크에 요청한 I/O 요청을 대기해야 한다.
      비차단 인터페이스 (non-blocking interface) 는 모든 프로그래밍 (예, 멀티 쓰레드 프로그래밍) 스타일에서 사용될 수 있다. 하지만, 이벤트 기반의 프로그래밍 방식에서는 필수적이다. 차단 방식의 시스템 콜 (blocking call)이 전체 시스템을 멈출 수 있기 때문이다.

- **AIO 제어블록 :** Mac OS 에서는 struct aiocb 등을 활용해서 비동기 읽기 (asynchronous read) API를 제공한다.

▼ **파일시스템**

- **원자적 연산의 반대상태 : 찢긴상태 (torn write) :** 갑작스럽게 전력 손실이 발생한다면 대량의 쓰기 중에 일부만 완료될 수 있다
- 디스크헤드의 **안정화시간 :**
    - 탐색은 여러 단계로 되어 있다는 것에 유의해야 한다. 첫 번째는 가속 단계로 디스크의 암이 움직이기 시작한다. 디스크 암이 최고 속도로 움직이는 활주 단계를 지나고, 디스크 암의 속도가 줄어드는 감속 단계 이후에 안정화 단계에서 정확한 트랙 위에 헤드가 조심스럽게 위치하게 된다
- **차원 분석 (dimensional analysis)**
    - 단위를 잘 정리해놓고 상쇄시켜가며 문제를 푸는 것
- **I/O 속도 계산 : 디스크 전송시간 (Tseek + Trot + Ttransfer) / 전송된 메모리 크기**
- **I/O 병합**
    - **블록 33,34번 을 연속된 요청이 있으면 블록을 연합하여 처리**
- **작업 보전 ↔ 작업 비보전**
    - **작업 보전 : Write Through**
    - 작업 비보전 : Write Back (잠시 기다리고 IO개시)
- **RAID 평가방법**
    - 용량, 신뢰성, 성능
- **프로그램 시스템콜 추적 도구**

    - **Linux : strace**
    - Macs : dtruss
- fsync는 dirty 비트를 설정함으로써 진행됨
- rename은 원자적 연산으로 진행됨
- **메타데이터 :**
    - **파일 시스템**은 각 파일에 대한 정보를 관리한다. 그 정보가 **메타데이터** (metadata)의 핵심이다. 파일을 구성하는 데이터 블럭 (데이터 영역 내의) 들과 그 파일의 크기, 소유자, 접근 권한, 접근과 변경 시간 등과 같은 정보들이 이에 해당됨
- **익스텐트 (Extent)**
    - 포인터를 사용하는 대신 익스텐트(extent) 방식을 사용할 수도 있다. 익스텐트는 단순히 **디스크 포인터와 길이 (블럭 단위)**로 이루어진다
- **정적파티션과 동적 파티션**
    - **정적 방식**은 **고정된 크기로 자원을 한 번만 나눈다**. **동적 방식은 좀 더 유동적이라서 때에 따라 다른 양의 자원을 나누어 줄 수도 있다.** 예를 들어 한 사용자가 일정 시간 동안 높은 퍼센트로 디스크 대역폭을 쓰고 있다고 해보자. 잠시 후에 시스템은 사용 가능한 디스크 대역폭 중 많은 부분을 다른 사용자에게할당하기로 결정할 수도 있다
- **일원화된 페이지 캐시 (Unified Page Cache)**
    - 현대의 많은 운영체제는 **가상 메모리 페이지들과 파일 시스템 페이지**들을 통합하여 일원화된 페이지 캐시(uniied page cache)를 만들었다
- **쓰기 베리어 (write Barrier)**
    - 디스크 내에 있는 쓰기 캐시의 사용이 늘어나면서 상황이 좀 더 복잡해졌다. 쓰기 **버퍼링이 동작 중이면 (때로는 즉시 보고(immediate reporting)** 라고도 불림) 데이터는 디스크의 메모리 캐시에만 있고 디스크에는 다 쓰여지지 않았는데도 운영체제에게 쓰기가 완료했다고 알린다. 이때 두 번째 쓰기를 운영체제가 요청하면 첫 번째 쓰기 다음에 두 번째 쓰기가 디스크에 도달한다고 보장할 수 없기 때문에 쓰기 간의 **순서는 보존이 안 된다.** 한 가지 방법은 **쓰기 버퍼링을 끄는 잃이다.** 하지만, 좀 더 최신의 시스템은 좀 더 주의를 기울여서 명시적으로 **쓰기 배리어(write barrier)를 요청한다. 배리어 요청이 완료되었을 때는 어떤 쓰기 요청이든 배리어 이전에 요청된 잃들은 배리어 요청 이후에 쓰기 요청을 받은 잃들보다 먼저 디스크에 도달하도록 한다.**
- **환형 로그 (circular log)**

    - 저널 로그가 로그 끝까지 쓰면 다시 앞에서부터 쓰게해주는 것
- **철회 레코드 (revoke record)**
    - 블록 삭제 시 다시 해당 블록에 쓰여진 경우 **이전에 작성된 저널**이 문제가 되어 복구 시 **원데이터가 덮어 씌여저버리는 것을 방지**하기 위해 지워진 **블럭이 체크포인트될 때까지 절대로 재사용하지 않는 잃**이다. Linux ext3 은 저널에 철회 3코드 (revoke record)라는 새로운 항목을 추가
- Copy-On-Write (**COW)**
    - 파일의 일관성을 보장해줄 수 있는 3번째 방법으로 파일이나 디렉터리들을 절대로 원래 위치에 덮어쓰지 않는다. 대신에 **이제까지 디스크에서 사용 안 된 위치에 갱신 내용을 저장후 나중에 역으로 포인팅하는 LFS 와 비슷한 방식의 접근법**
- **BBC (Backpointer-bsed consistency) :**
    - Wisconsin에서 개발한 잃이다. **백포인터 기반 일관성 (backpointerbased consistency, BBC)**이라고 하는 기법으로 쓰기 사이에 어떤 순서도 강요하지 않는다. 대신 시스템의 모든 블럭에 가리키는 **inode에 대한 백포인터**를 추가