운영체제 기말 정리본

dldyou

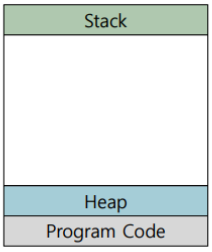
목 차

1. 14-Concurrency and Threads

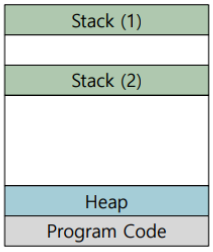
1.1. Threads

- Multi-threaded 프로그램
  - 스레드 하나의 상태는 프로세스의 상태와 매우 비슷하다.
    - 각 스레드는 그것의 PC(Program Counter)와 private한 레지스터를 가지고 있다.
    - 스레드 당 하나의 스택을 가지고 있다.
  - 같은 address space를 공유하므로 같은 데이터를 접근할 수 있다.
  - Context Switch
    - Thread Control Block (TCB)
    - 같은 address space에 남아있다. (switch를 하는데 page table이 필요하지 않음)

- Single-threaded address space



- Multi-threaded address space



- 사용하는 이유
  - 병렬성
    - Multiple CPUs
  - Blocking 회피
    - 느린 I/O
    - 프로그램에서 하나의 스레드가 기다리는 동안(I/O 작업을 위해 blocked 되어), CPU 스케줄러가 다른 스레드를 실행시킬 수 있다.
  - 많은 현대 서버 기반 어플리케이션은 멀티스레드를 사용하도록 구현되어 있다.
    - 웹 서버, 데이터베이스 관리 시스템, ...

1.1.1. Thread Create

```c
void *mythread(void *arg)
{
    printf("%s\n", (char *) arg);
    return NULL;
}
int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

- 실행 가능한 순서

main	Thread 1 (T1)	Thread 2 (T2)
prints "main: begin" creates Thread 1 creates Thread 2 waits for T1		
	prints "A"	
waits for T2		
		prints "B"
prints "main: end"		

main	Thread 1 (T1)	Thread 2 (T2)
prints "main: begin" creates Thread 1		
	prints "A"	
creates Thread 2		
		prints "B"
waits for T1 waits for T2 prints "main: end"		

- 공유 데이터

```c
static volatile int counter = 0;
void * mythread(void *arg)
{
    int i;
    printf("%s: begin\n", (char *) arg);
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

- 실행 결과
  - counter 값이 2e7이 아닌 다른 값이 나올 수 있다.

```
main: done with both (counter = 20000000)
main: done with both (counter = 19345221)
main: done with both (counter = 19221041)
```

1.1.2. Race Condition

```
counter = counter + 1;      100 mov 0x8049a1c, %eax
                            105 add $0x1, %eax
                            108 mov %eax, 0x8049a1c
```

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
			100	0	50
	mov		105	**50**	50
	add		108	**51**	50
Context switch					
			100	0	50
		mov	105	**50**	50
		add	108	**51**	50
		mov	113	51	**51**
Context switch					
			108	51	**51**
	mov		113	51	**51**

1.1.3. Critical Section

- Critical Section
  - 공유된 자원에 접근하는 코드 영역 (공유 변수)
  - 둘 이상의 스레드에 의해 동시에 실행되어서는 안 된다.
- Mutual Exclusion
  - 한 스레드가 critical section에 들어가면 다른 스레드는 들어갈 수 없다.

1.1.4. Atomicity

- Atomic
  - 한 번에 실행되어야 하는 연산

    - 하나의 명령이 시작되었다면 해당 명령이 종료될 때까지 다른 명령이 시작되어서는 안 된다.
- synchronizaion을 어떻게 보장하는지
  - 하드웨어 지원 (atomic instructions)
    - Atomic memory add → 있음
    - Atomic update of B-tree → 없음
  - OS는 이러한 명령어들에 따라 일반적인 동기화 primitive 집합을 구현한다.

1.1.5. Mutex

위의 Atomicity를 보장하기 위해 Mutex를 사용한다.
- Initialization
  - Static: `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`
  - Dynamic: `pthread_mutex_init(&lock, NULL);`
- Destory
  - `pthread_mutex_destroy();`
- Condition Variables
  - `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
    - 조건이 참이 될 때까지 대기하는 함수
    - `pthread_mutex_lock`으로 전달할 mutex을 잠근 후에 호출되어야 한다.
  - `int pthread_cond_signal(pthread_cond_t *cond);`
    - 대기 중인 스레드에게 signal을 보내는 함수
    - `pthread_cond_wait`로 대기 중인 스레드 중 하나를 깨운다.
  - 외부를 lock과 unlock으로 감싸줘야 한다.
- 두 스레드를 동기화

```
while (read == 0)
    ; // spin
```

```
ready = 1;
```

- 오랜 시간 spin하게 되어 CPU 자원을 낭비하게 된다.
- 오류가 발생하기 쉽다.
  - 현대 하드웨어의 메모리 consistency 모델 취약성
  - 컴파일러 최적화

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_lock(&lock);
while (ready == 0)
    pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock);
```

```
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

- `#include <pthread.h>` 컴파일 시 `gcc -o main main.c -Wall -pthread` 와 같이 진행

2. 15-Locks

2.1. Pthread Locks

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&lock);
counter = counter + 1; // critical section
pthread_mutex_unlock(&lock);
```

- Lock을 어떻게 설계해야 할까?
  - 하드웨어 / OS 차원에서의 지원이 필요한가?

2.1.1. Evaluting Locks

- 상호 배제(Mutual Exclution)
  - 둘 이상의 스레드가 동시에 critical section에 들어가는 것을 방지
- 공평(Fairness)
  - lock을 두고 경쟁할 때, lock이 free가 되었을 때, lock을 얻는 기회가 공평함
- 성능(Performance)
  - lock을 사용함으로써 생기는 오버헤드
    - 스레드의 수
    - CPU의 수

2.1.2. Controlling Interrupts

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

- 이러한 모델은 간단하지만 많은 단점이 있음
  - thread를 호출하는 것이 반드시 privileged operation으로 수행되어야 함
  - 멀티프로세서 환경에서 작동하지 않음
  - 인터럽트가 손실될 수 있음
- 한정된 contexts에서만 사용될 수 있음
  - 지저분한 인터럽트 처리 상황을 방지하기 위해

2.2. Support for Locks

2.2.1. Hardware Support

- 간단한 방법으로는 `yield()`(본인이 ready큐로, 즉 CPU자원을 포기한다고 함)를 사용할 수 있음
  - 그러나 여전히 비용이 높고 공평하지 않음
    - RR에 의해 스케줄 된 많은 스레드가 있는 상황을 고려해보자

```
void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        yield();
}
```

- 하드웨어 만으로는 상호 배제 및 공평성만 해결 할 수 있었음
  - 성능 문제는 여전히 존재 → OS의 도움이 필요

2.2.1.1. Spin Locks

2.2.1.1.1. Loads / Stores

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1) // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

- 상호 배제가 없음
  - thread 1에서 lock()을 호출하고 while(flag == 1)에서 1이 아니구나 하고 빠져나갈 때 context switch가 일어남
  - thread 2에서 lock()을 호출하고 while(flag == 1)에서 1이 아니구나 하고 빠져나가서 flag = 1로 만듦
  - context switch가 일어나 thread 1이 다시 돌아와서 flag = 1이 됨
  - 두 스레드 모두 lock을 얻게 됨
- 성능 문제
  - spin-wait으로 인한 CPU 사용량이 많아짐

2.2.1.1.2. Test-and-Set

- Test-and-Set atomic instruction

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store 'new' into old_ptr
    return old; // return the old value
}

typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *lock) {
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1);
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

- 공평하지 않음 (starvation이 발생할 수 있음)
- 단일 CPU에서 오버헤드가 굉장히 클 수 있음

2.2.1.1.3. Compare-and-Swap

- Compare-and-Swap atomic instruction

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}

void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1);
}
```

- Test-and-Set과 동일하게 동작함

2.2.1.2. Ticket Locks

- Fetch-and-Add atomic instruction
  - 번호표 발급으로 생각하면 됨

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}

typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn);
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

- fairness 하게 됨

2.2.2. OS Support

- spin을 하는 대신 sleep을 함
- Solaris
  - `park()`: 호출한 스레드를 sleep 상태로 만듦
  - `unpark(threadID)`: `threadID`의 스레드를 깨움
- Linux
  - `futex_wait(address, expected)`: address가 expected랑 같다면 sleep 상태로 만듦
  - `futex_wake(address)`: queue에서 스레드 하나를 깨움

2.2.2.1. Locks with Queues (Hardware + OS Support)

```
typedef struct __lock_t {
    int flag; // lock
    int guard; // spin-lock around the flag and
    // queue manipulations
    queue_t *q;
} lock_t;
void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    }
    else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park(); // wakeup/waiting race
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    if (queue_empty(m->q))
        m->flag = 0;
    else
        unpark(queue_remove(m->q));
    m->guard = 0;
}
```

setpark를 미리 불러주는 모습을 볼 수 있음

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    }
    else {
        queue_add(m->q, gettid());
        setpark(); // another thread calls unpark before
        m->guard = 0; // park is actually called, the
        park(); // subsequent park returns immediately
    }
}
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    if (queue_empty(m->q))
        m->flag = 0;
    else
        unpark(queue_remove(m->q));
    m->guard = 0;
}
```

3. 16-Lock-Based Concurrent Data Structures

- Correctness
  - 올바르게 작동하려면 lock을 어떻게 추가해야 할까? (어떻게 thread safe하게 만들 수 있을까?)
- Concurrency
  - 자료구조가 높은 성능을 발휘하고 많은 스레드가 동시에 접근할 수 있도록 하려면 lock을 어떻게 추가해야 할까?

3.1. Counter

3.1.1. Concurrent Counters

```c
typedef struct __counter_t {
    int value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    pthread_mutex_lock(&c->lock);
    c->value++;
    pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
    pthread_mutex_lock(&c->lock);
    c->value--;
    pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    pthread_mutex_lock(&c->lock);
    int rc = c->value;
    pthread_mutex_unlock(&c->lock);
    return rc;
}
```

- 간단하게 생각해보면 이렇게 구현할 수 있을 것이다. 그러나 매 count마다 lock을 걸어줘야 하므로 concurrency가 떨어진다.

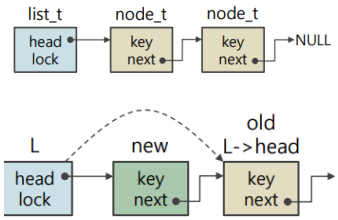
3.1.2. Sloppy Counters

- Logical counter
  - Local counter가 각 CPU 코어마다 존재
  - Global counter
  - Locks (각 local counter마다 하나, global counter에도 하나)
- 기본 아이디어
  - 각 CPU 코어마다 local counter를 가지고 있다가 global counter에 값을 옮기는 방식
    - 이는 일정 주기마다 이루어짐
  - global counter에 값을 옮기는 동안 lock을 걸어서 다른 코어가 접근하지 못하도록 함

```c
typedef struct __counter_t {
    int global;
    pthread_mutex_t glock;
    int local[NUMCPUS];
    pthread_mutex_t llock[NUMCPUS];
    int threshold; // update frequency
} counter_t;

void init(counter_t *c, int threshold) {
    c->threshold = threshold;
    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);
    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}

void update(counter_t *c, int threadID, int amt) {
    int cpu = threadID % NUMCPUS;
    pthread_mutex_lock(&c->llock[cpu]); // local lock
    c->local[cpu] += amt; // assumes amt>0
    if (c->local[cpu] >= c->threshold) {
        pthread_mutex_lock(&c->glock);// global lock
        c->global += c->local[cpu];
        pthread_mutex_unlock(&c->glock);
        c->local[cpu] = 0;
    }
    pthread_mutex_unlock(&c->llock[cpu]);
}

int get(counter_t *c) {
    pthread_mutex_lock(&c->glock); // global lock
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val; // only approximate!
}
```

3.2. Concurrent Data Structures

3.2.1. Linked Lists




```c
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
    pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; // fail
    }
    new->key = key;
    // mutex lock은 여기로 옮겨지는 것이 좋음 (critical section이 여기부터)
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}

int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; // success (그러나 ret = 0을 저장해놓고 break한 다음
에 마지막에 return ret을 하는 것이 좋음 -> 버그 찾기 쉬움)
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; // failure
}
```

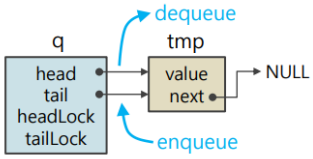
3.2.1.1. Scaling Linked Lists

- Hand-over-hand locking (lock coupling)
  - 각 노드에 대해 lock을 추가 (전체 list에 대한 하나의 lock을 갖는 대신)
  - list를 탐색할 때, 다음 노드의 lock을 얻고 현재 노드의 lock을 해제
  - 각 노드에 대해 lock을 얻고 해제하는 오버헤드 존재
- Non-blocking linked list
  - compare-and-swap(CAS) 이용

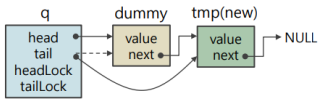
```c
void List_Insert(list_t *L, int key) {
    ...
RETRY: next = L->head;
    new->next = next;
    if (CAS(&L->head, next, new) == 0)
        goto RETRY;
}
```

3.2.2. Queues

```
typedef struct __node_t {
    int value;
    struct __node_t *next;
} node_t;

typedef struct __queue_t {
    node_t *head; // out
    node_t *tail; // in
    pthread_mutex_t headLock;
    pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t)); // dummy node (head와 tail 연산의 분리를 위해)
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

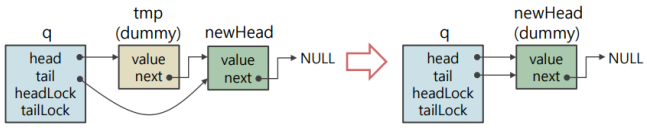



```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next = NULL;
    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}
```



- 길이가 제한된 큐에서는 제대로 작동하지 않음, 조건 변수에 대해서는 다음 장에서 다루게 될 예정

```
int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```



3.2.3. Hash Table

```
#define BUCKETS (101)
typedef struct __hash_t {
    list_t lists[BUCKETS]; // 앞에서 본 list_t를 사용
} hash_t;
void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++)
        List_Init(&H->lists[i]);
}
int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Insert(&H->lists[bucket], key);
}
int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Lookup(&H->lists[bucket], key);
}
```

4. 17-Condition Variables

스레드를 계속 진행하기 전에 특정 조건이 true가 될 때까지 기다리는 것이 유용한 경우가 많다. 그러나, condition이 true가 될 때까지 그냥 spin만 하는 것은 CPU cycle을 낭비하게 되고 이것은 부정확할 수 있다.

```
volatile int done = 0;
void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t c;
    printf("parent: begin\n");
    pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0); // spin
    printf("parent: end\n");
    return 0;
}
```

4.1. Condition Variable

- condition 변수는 명시적인 대기열과도 같다.
  - 스레드는 일부 상태(즉, 일부 condition)가 원하는 것과 다를 때 대기열에 들어갈 수 있다.
  - 몇몇 스레드는 상태가 변경되면, 대기열에 있는 스레드 중 하나(또는 그 이상)를 깨워 진행되도록 할 수 있다.
  - `pthread_cond_wait();`
    - 스레드가 자신을 sleep 상태로 만들려고 할 때 사용
      - lock을 해제하고 호출한 스레드를 sleep 상태로 만든다. (atomic하게)
      - 스레드가 깨어나면 반환하기 전에 lock을 다시 얻는다.
  - `pthread_cond_signal();`
    - 스레드가 프로그램에서 무언가를 변경하여 sleep 상태인 스레드를 깨우려고 할 때 사용

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

```c
void thr_exit() {
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void thr_join() {
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- 만약 여기서 상태 변수인 done이 없으면?
  - child가 바로 실행되고 thr_exit()을 호출하면?
    - child가 signal을 보내지만 그 상태에서 잠들어 있는 스레드가 없다.
- 만약 lock이 없다면?
  - child가 parent가 wait을 실행하기 직전에 signal을 보내면?
    - waiting 상태에 있는 스레드가 없으므로 깨어나는 스레드가 없다.

4.1.1. Producer / Consumer Problem

- Producers
  - 데이터를 생성하고 그들을 (제한된) 버퍼에 넣는다.
- Consumers
  - 버퍼에서 데이터를 가져와서 그것을 소비한다.
- 예시
  - Pipe
    - `grep foo file.txt | wc -l`
  - Web server
- 제한된 버퍼가 공유 자원이기에 당연히 이에 대한 동기화된 접근이 필요하다.

```c
int buffer; // single buffer
int count = 0; // initially, empty
void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}
int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

```c
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex); // p1
        if (count == 1) // p2
            pthread_cond_wait(&cond, &mutex); // p3
        put(i); // p4
        pthread_cond_signal(&cond); // p5
        pthread_mutex_unlock(&mutex); // p6
    }
}
```

```c
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex); // c1
        if (count == 0) // c2
            pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get(); // c4
        pthread_cond_signal(&cond); // c5
        pthread_mutex_unlock(&mutex); // c6
        printf("%d\n", tmp);
    }
}
```

- 단일 producer와 단일 consumer로 진행한다고 하자.

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

- There is no guarantee that when the woken thread (T_{c1}) runs, the state will still be as desired (count==1)

- 위 그림에서 알 수 있듯이 T_{c1}가 다시 깨어나 실행될 때 state가 여전히 원하던 값이라는 보장이 없다.
- count를 체크하는 부분을 if문에서 while문으로 바꾸어주면 아래와 같이 돌아간다.

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

- A consumer should not wake other consumers, only producers, and vice-versa

- consumer는 다른 consumer를 깨우면 안 되고, producer만 깨우면 되고, 반대의 경우도 마찬가지이다. 위의 경우는 그것이 안 지켜져서 모두가 잠들어버린 상황이다.
- 이는 condition 변수를 하나를 사용하기에 발생하는 문제이다. (같은 큐에 잠들기에 producer를 깨우고자 했으나 다른 결과를 야기할 수 있음)
  - p3의 cv를 `&empty`로 p5의 cv를 `&fill`로
  - c3의 cv를 `&fill`로 c5의 cv를 `&empty`로

```c
int buffer[MAX];
int fill_ptr = 0;
int use_ptr = 0;
int count = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

- 이와 같이 버퍼를 만들고 producer에서 `count == MAX`로 바꾸어주면 동시성과 효율성을 챙길 수 있다.
- Covering Conditions
  - `pthread_cond_broadcast()`
    - 대기 중인 모든 스레드를 깨운다.

5. 18-Semaphores

- 세마포어는 lock이나 condition 변수를 통해 사용할 수 있다.
- POSIX Semaphores
  - `int sem_init(sem_t *s, int pshared, unsigned int value);`

  - pshared가 0이면 프로세스 내에서만 사용 가능하고, 1이면 프로세스 간에도 사용 가능하지만, 공유 메모리에 있어야 한다.
- `int sem_wait(sem_t *s);`
  - 세마포어 값을 감소시키고, 값이 0보다 작으면 대기한다.
- `int sem_post(sem_t *s);`
  - 세마포어 값을 증가시킨다.
  - 만약 대기 중인 스레드가 있다면 하나를 깨운다.

- Binary Semaphores (lock이랑 비슷함)

```
sem_t m;
sem_init(&m, 0, 1);

sem_wait(&m);
// critical section here
sem_post(&m);
```

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call sem_wait()	Run		Ready
0	sem_wait() returns	Run		Ready
0	(crit sect begin)	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call sem_wait()	Run
-1		Ready	decr sem	Run
-1		Ready	(sem<0) →sleep	Sleep
-1		Run	*Switch→T0*	Sleep
-1	(crit sect end)	Run		Sleep
-1	call sem_post()	Run		Sleep
0	incr sem	Run		Sleep
0	wake(T1)	Run		Ready
0	sem_post() returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	sem_wait() returns	Run
0		Ready	(crit sect)	Run
0		Ready	call sem_post()	Run
1		Ready	sem_post() returns	Run

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

- Semaphores for Ordering

세마포어를 사용해 스레드간의 순서를 정할 수 있다.

```
sem_t s;
void * child(void *arg) {
    printf("child\n");
    sem_post(&s);
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t c;
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_create(&c, NULL, child, NULL);
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

- X는 0이어야 한다. 그래야 다음 `sem_wait`이 바로 실행되더라도 세마포어 값이 음수가 되며 잠들 수 있고, child가 먼저 실행되어 post를 실행하여 세마포어 값이 1이 되고 `sem_wait`가 실행되더라도 잠에 들지 않아 deadlock이 발생하지 않는다.

Val	Parent	State	Child	State
0	create(Child)	Run	*(Child exists, can run)*	Ready
0	call sem_wait()	Run		Ready
-1	decr sem	Run		Ready
-1	(sem<0) →sleep	Sleep		Ready
-1	*Switch→Child*	Sleep	child runs	Run
-1		Sleep	call sem_post()	Run
0		Sleep	inc sem	Run
0		Ready	wake(Parent)	Run
0		Ready	sem_post() returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	sem_wait() returns	Run		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

Val	Parent	State	Child	State
0	create(Child)	Run	*(Child exists; can run)*	Ready
0	*Interrupt→Child*	Ready	child runs	Run
0		Ready	call sem_post()	Run
1		Ready	inc sem	Run
1		Ready	wake(nobody)	Run
1		Ready	sem_post() returns	Run
1	parent runs	Run	*Interrupt→Parent*	Ready
1	call sem_wait()	Run		Ready
0	decrement sem	Run		Ready
0	(sem≥0) →awake	Run		Ready
0	sem_wait() returns	Run		Ready

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

5.1. Producer / Consumer Problem

```
int buffer[MAX]; // bounded buffer
int fill = 0;
int use = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    return tmp;
}

sem_t empty, sem_t full;
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);
        tmp = get();
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0); // 0 are full
    // ...
}
```

- Race Condition이 발생한다.
  - 생산자와 소비자가 여럿인 경우 `put()`과 `get()`에서 race condition이 발생한다.

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex); // 2
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full); // 1
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
    }
}
```

- 이렇게 mutex를 추가하면 deadlock이 발생한다.
  - 소비자가 먼저 실행되어 wait에 의해 mutex를 0으로 감소시키고 1까지 실행되어 sleep을 하게 된다.
  - 생산자가 실행되고, wait에 의해 mutex가 −1이 되어 잠들게 된다.
  - 둘 다 잠들어버리게 되어 deadlock이 발생한다.
- **mutex를 모두 안쪽으로 옮겨주면 해결된다.**

5.2. Reader / Writer Locks

- Reader
  - `rwlock_acquire_readlock()`
  - `rwlock_release_readlock()`
- Writer
  - `rwlock_acquire_writelock()`
  - `rwlock_release_writelock()`

```
typedef struct _rwlock_t {
    // binary semaphore (basic lock)
    sem_t lock;
    // used to allow ONE writer or MANY readers
    sem_t writelock;
    // count of readers reading in critical section
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers++;
    if (rw->readers == 1)
        // first reader acquires writelock
        sem_wait(&rw->writelock);
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        // last reader releases writelock
        sem_post(&rw->writelock);
    sem_post(&rw->lock);
}
```

- reader에게 유리함 (writer가 굶을 수 있음)

5.3. How To Implement Semaphores

```
typedef struct __Sem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Sem_t;

// only one thread can call this
void Sem_init(Sem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Sem_wait(Sem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Sem_post(Sem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

- 원래 구현: 값이 음수인 경우 대기 중인 스레드의 수를 반영
- Linux: 값은 0보다 낮아지지 않음

6. 19-Common Concurrency Problems

6.1. Concurrency Problems

- Non-deadlock 버그
  - Atomicity 위반
  - 순서 위반
- deadlock bug

6.1.1. Atomicity-Violation

- 메모리 영역에 대해 여러 개의 스레드가 동시에 접근할 때 serializable 해서 race condition이 발생하지 않을 것이라 예상하지만 그렇지 않은 경우가 있다.
- MySQL 버그

```
Thread 1:
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}

Thread 2:
thd->proc_info = NULL;
```

- Thread 1의 if문을 확인하고 들어왔으나 Thread 2가 값을 NULL로 바꾸어버리면서 fputs에서 비정상 종료가 된다.
- 해결 방법

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1:
pthread_mutex_lock(&proc_info_lock);
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}
pthread_mutex_unlock(&proc_info_lock);

Thread 2:
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```

6.1.2. Order-Violation

- A → B 스레드 순서로 실행되기를 바랬으나 다르게 실행되는 경우
- Mozilla 버그

```
Thread 1:
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    ...
}
Thread 2:
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

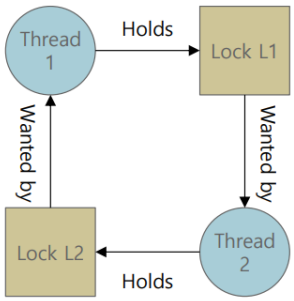
- Thread 2가 생성되자마자 mState를 읽어버리면서 mThread가 초기화되기 전에 읽어버리는 문제가 발생한다. (Null 포인터를 접근하게 됨)
- 해결 방법

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0;

Thread 1:
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);

    // signal that the thread has been created...
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    ...
}

Thread 2:
void mMain(...) {
    ...
    // wait for the thread to be initialized...
    pthread_mutex_lock(&mtLock);
    while (mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);
    mState = mThread->State;
    ...
}
```

6.1.3. Deadlock Bugs

- Circular Dependencies

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```



Resource-allocation graph

- Thread 1은 L1을 먼저 잡고, Thread 2는 L2를 먼저 잡은 상태에서 서로를 기다리게 되어 deadlock이 발생한다.
- 왜 deadlock이 발생할까?
  - 큰 코드 베이스에서는 컴포넌트 간의 의존성이 복잡함
  - 캡슐화의 특징
    - `Vector v1, v2`
    - `Thread 1: v1.addAll(v2)`
    - `Thread 2: v2.addAll(v1)`

6.1.4. Conditions for Deadlock

- Mutual Exclusion
  - 한 번에 하나의 스레드만이 자원을 사용할 수 있음
- Hold and Wait
  - 스레드가 자원을 가지고 있는 상태에서 다른 자원을 기다림
- No Preemption
  - 스레드가 자원을 강제로 뺏을 수 없음
- Circular Wait
  - 스레드 A가 스레드 B가 가지고 있는 자원을 기다리고, 스레드 B가 스레드 A가 가지고 있는 자원을 기다림

6.1.4.1. Deadlock Prevention

- Circular Wait
  - lock acqustition 순서를 정함
- Hold and Wait
  - 모든 자원을 한 번에 요청 (전체를 lock으로 한 번 감싸기)
  - critical section이 커지는 문제가 발생할 수 있었음
  - 미리 lock을 알아야 함

```
pthread_mutex_lock(prevention); // begin lock acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

- No Preemption
  - `pthread_mutex_trylock()`: lock을 얻을 수 없으면 바로 반환
  - 아래처럼 구현하면 livelock(deadlock처럼 모든 스레드가 lock을 얻지 못하고 멈췄는데, 코드는 돌아가고 있는 상태)이 발생할 수 있음
    - random delay를 추가해 누군가는 acquire에 성공하도록 할 수 있음
  - 획득한 자원이 있다면 반드시 해제해야 함
    - lock이나 메모리...

```
top:
pthread_mutex_lock(L1);
if (pthread_mutex_trylock(L2) != 0) {
    pthread_mutex_unlock(L1);
    goto top;
}
```

- Mutual Exclusion
  - race condition을 없애기 위해 mutual exclusion을 사용함
  - 그런데 이거를 없애야 하나? (X) → lock을 안 쓴다는 것으로 이해하면 됨
  - lock free 접근법 (atomic operation을 이용)

```
int CompareAndSwap(int *address, int expected, int new) {
    if (*address == expected) {
        *address = new;
        return 1; // success
    }
    return 0; // failure
}

void AtomicIncrement(int *value, int amount) {
    do {
        int old = *value;
    } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

```c
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    n->next = head;
    head = n;
}

void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    pthread_mutex_lock(listlock);
    n->next = head;
    head = n;
    pthread_mutex_unlock(listlock);
}

void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

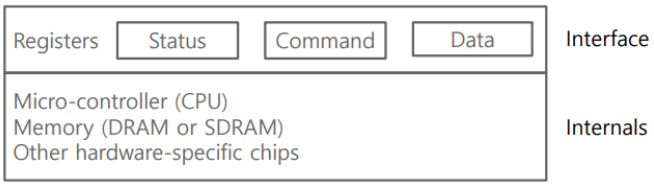
7. 20-I/O Devices and HDD

7.1. System Architecture

- CPU / Main Memory
- (Memory Bus)
- (General I/O Bus(PCI))
- Graphics
- (주변기기 I/O Bus(SCSI, SATA, USB))
- HDD

7.2. I/O Devices

- 인터페이스
  - 시스템 소프트웨어로 작동을 제어할 수 있도록 함
  - 모든 장치는 일반적인 상호작용을 위한 특정 인터페이스와 프로토콜이 있음
- 내부 구조
  - 시스템에 제공하는 추상화된 구현



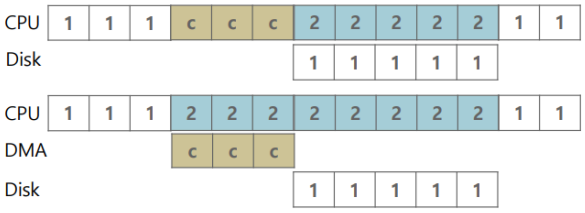
7.3. Interrupts

- Interrupt로 CPU 오버헤드를 낮춤
  - 장치를 반복적으로 polling 하는 대신 OS 요청을 날리고, 호출한 프로세스를 sleep 상태로 만들고 다른 작업으로 context switch를 함
  - 장치가 최종적으로 작업을 마치면 하드웨어 interrupt를 발생시켜 CPU가 미리 결정된 interrupt service routine(ISR)에서 OS로 넘어가게 함
- Interrupts는 I/O 연산을 하는 동안 CPU를 다른 작업에 사용할 수 있게 함

7.4. Direct Memory Access (DMA)

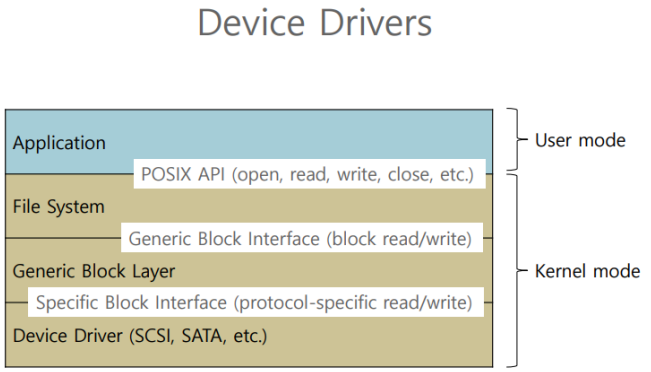
- DMA를 사용하면 더 효율적인 데이터 이동을 할 수 있다.
  - DMA 엔진은 CPU 개입 없이 장치와 주메모리 간의 전송을 조율할 수 있는 장치이다.
  - OS는 데이터가 메모리에 있는 위치와 복사할 위치를 알려주어 DMA 엔진을 프로그래밍한다.
  - DMA가 완료되면 DMA 컨트롤러는 interrupt를 발생시킨다.

- Because the CPU is free during that time, the OS can do something else



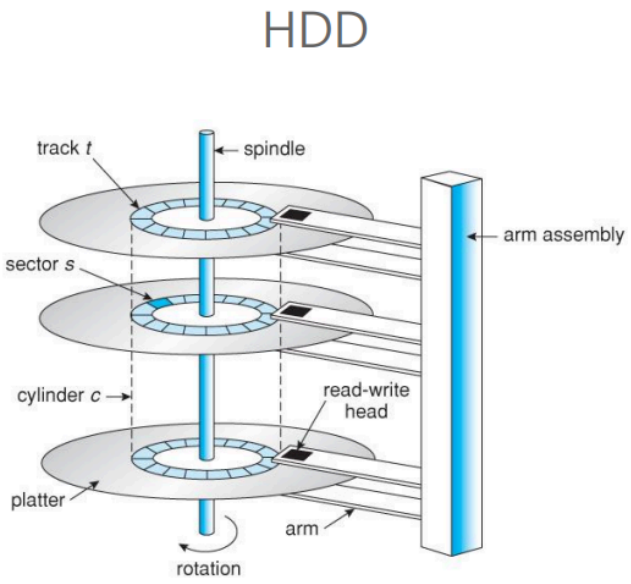
7.5. Methods of Device Interaction

- I/O instructions
  - `in` / `out` (x86)
  - 장치에 데이터를 보내기 위해 호출자는 데이터가 포함된 레지스터와 장치 이름을 지정하는 특정 포트를 지정한다.
  - 일반적으로 privileged instruction이다.
- Memory-mapped I/O
  - 하드웨어는 마치 메모리 위치인 것처럼 장치 레지스터를 사용할 수 있게 만든다.
  - 특정 레지스터에 접근하기 위해 OS는 주소를 읽거나 쓴다.



7.6. HDD

- 기본 요소
  - Platter
    - 데이터가 지속적으로 저장되는 원형의 단단한 표면
    - 디스크는 하나 또는 그 이상의 platters를 가진다. 각 platter는 `surface`라고 불리는 두 면을 가진다.
  - Spindle
    - platters를 일정한 속도로 회전시키는 모터를 연결
    - 회전 속도는 RPM으로 측정된다. (7200 ~ 15000 RPM)
  - Track
    - 데이터는 각 구역(sector)의 동심원으로 각 표면에 인코딩된다. (512-byte blocks)
  - Disk head and disk arm
    - 읽기 및 쓰기는 디스크 헤드에 의해 수행된다. 드라이브 표면 당 하나의 헤드가 있다.
    - 디스크 헤드는 단일 디스크 암에 부착되어 표면을 가로질러 이동하여 원하는 track 위에 헤드를 배치한다.



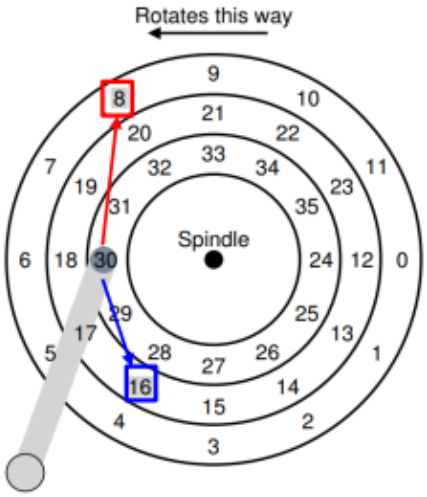
7.6.1. I/O Time

$$T_{I/O} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}}$$

- Seek time
  - 디스크 암을 올바른 트랙으로 옮기는데 걸리는 시간
- Rotational delay
  - 디스크가 올바른 섹터로 회전하는데 걸리는 시간

7.6.2. Disk Scheduling

- OS가 디스크로 날릴 I/O 요청들의 순서를 결정한다.
  - I/O 요청의 집합이 주어지면, 디스크 스케줄러는 요청을 검사하고 다음에 무엇을 실행해야 하는지 결정한다.
- 요청: 98, 183, 37, 122, 14, 124, 65, 67 (Head: 53)
  - **FCFS (First Come First Serve)**
    - 98 → 183 → 37 → 122 → 14 → 124 → 65 → 67
  - **Elevator (SCAN or C-SCAN)**
    - **SCAN**: 맨 앞으로 가면서 훑고 다시 순차로 가는 방식
      - 37 → 14 → 65 → 67 → 98 → 122 → 124 → 183
    - **C-SCAN**: 현 위치부터 뒤로 쭉 가서 앞으로 나오는 원형 방식
      - 65 → 67 → 98 → 122 → 124 → 183 → 14 → 37
  - **SPTF (Shortest Positioning Time First)**
    - track과 sector를 고려하여 가장 가까운 것을 먼저 처리
    - 현대 드라이브는 seek과 rotation 비용이 거의 동일하다.
    - 아래 그림에서 rotation이 중요하면 8을 먼저 접근함 (디스크의 하드웨어 특성에 따라 달라짐)

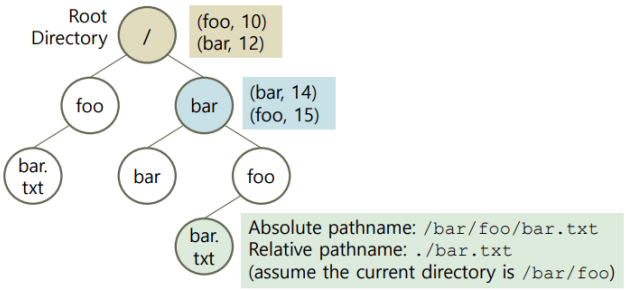


8. 21-Assignment 2: KURock

9. 22-Files and Directions

9.1. Abstractions for Storage

- 파일
  - bytes의 선형 배열
  - 각 파일은 low-level 이름을 가지고 있음 (`inode`)
  - OS는 파일의 구조에 대해 별로 알지 못함 (그 파일이 사진인지, 텍스트인지, C인지)
- 디렉토리
  - (user-readable name, low-level name)쌍의 리스트를 포함한다.
  - 디렉토리 또한 low-level 이름을 가지고 있음 (`inode`)



9.2. Interface

9.2.1. Creating

- `O_CREAT`를 같이 사용한 `open()` system call

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

  - `O_CREAT`: 파일이 없으면 생성
  - `O_WRONLY`: 쓰기 전용
  - `O_TRUNC`: 파일이 이미 존재하면 비우기
  - `S_IRUSR | S_IWUSR`: 파일 권한 (user에 대한 읽기, 쓰기 권한)
- File descriptor
  - An integer
    - 파일을 읽거나 쓰기 위해 file descriptor 사용(그 작업을 할 수 있는 권한이 있다고 가정)
    - 파일 형식 객체를 가리키는 포인터라고 생각할 수 있음
  - 각 프로세스끼리 독립적이다. (private하다)
    - 각 프로세스는 file descriptors의 리스트를 유지함 (각각은 system-wide하게 열린 파일 테이블에 있는 항목을 가리킨다)

9.2.2. Accessing

9.2.2.1. Sequential

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6) = 6
hello
read(3, "", 4096) = 0
close(3) = 0
...
prompt>
```

- `strace`는 프로그램이 실행되는 동안 만드는 모든 system call 을 추적한다. 그리고 그 결과를 화면에 보여준다.
- file descriptors 0, 1, 2는 각각 stdin, stdout, stderr를 가리킨다.

9.2.2.2. Random

- OS는 "현재" offset을 추적한다.
  - 다음 읽기 또는 쓰기가 어디서 시작할지는 파일을 읽고 있는 혹은 쓰고 있는 것이 결정한다.

- 암묵적인 업데이트
  - 해당 위치에서 N바이트를 읽거나 쓰면 현재 offset에 N만큼 추가된다.
- 명시적인 업데이트
  - `off_t lseek(int fd, off_t offset, int whence);`
    - whence
      - SEEK_SET: 파일의 시작부터
      - SEEK_CUR: 현재 위치부터
      - SEEK_END: 파일의 끝부터
  - 임의로 offset의 위치를 변경할 수 있다.

9.2.2.3. Open File Table

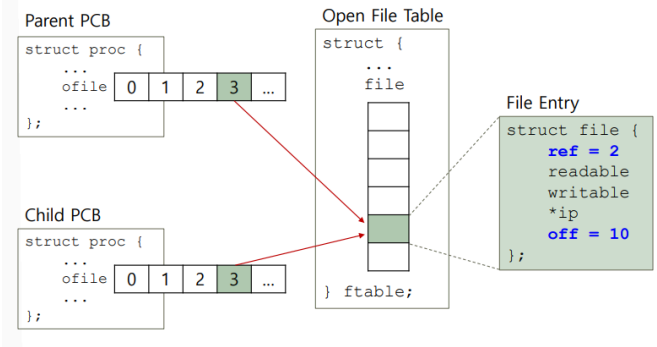
- 시스템에서 현재 열린 모든 파일을 보여준다.
  - 테이블의 각 항목은 descriptor가 참조하는 기본 파일, 현재 offset 및 파일 권한과 같은 기타 관련 정보를 추적한다.
- 파일은 기본적으로 open 파일 테이블에 고유한 항목을 가지고 있다.
  - 다른 프로세스가 동시에 동일한 파일을 읽는 경우에도 각 프로세스는 open 파일 테이블에 자체적인 항목을 갖는다.
  - 파일의 논리적 읽기 또는 쓰기는 각각 독립적이다.

9.2.2.4. Shared File Entries

- `fork()`로 file entry 공유

```c
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("C: offset % d\n", rc);
    }
    else if (rc > 0) {
        (void)wait(NULL);
        printf("P: offset % d\n", (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```



- `dup()`으로 file entry 공유
  - `dup()`은 프로세스가 기존 descriptor와 동일한 open file을 참조하는 새 file descriptor를 생성한다.
    - 새 file descriptor에 대해 가장 작은 사용되지 않는 file descriptor를 사용해 file descriptor의 복사본을 만든다.
  - output redirection에 유용함

```c
int fd = open("output.txt", O_APPEND|O_WRONLY);
close(1);
dup(fd); //duplicate fd to file descriptor 1
printf("My message\n");
```

  - `dup2()`, `dup3()`

9.2.2.5. Writing Immediately

- `write()`
  - 파일 시스템은 한동안 쓰기 작업을 하는 것을 버퍼에 집어넣고, 나중에 특정 시점에 쓰기가 디스크에 실제로 실행된다.
- `fsync()`
  - 파일 시스템이 모든 dirty 데이터(아직 쓰이지 않은)를 강제로 디스크에 쓴다.

9.2.3. Removing

- `unlink()`

9.2.4. Functions

- `mkdir()`
  - 디렉토리를 생성할 때, 빈 디렉토리를 생성한다.
  - 기본 항목
    - `.`: 현재 디렉토리
    - `..`: 상위 디렉토리
    - `ls -a`로 확인하면 위 2개가 나옴
- `opendir()`, `readdir()`, `closedir()`

```c
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long)d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

```c
struct dirent {
    char d_name[256]; // filename
    ino_t d_ino; // inode number
    off_t d_off; // offset to the next dirent
    unsigned short d_reclen; // length of this record
    unsigned char d_type; // type of file
};
```

- `rmdir()`
  - 빈 디렉토리를 삭제한다.
    - 빈 디렉토리가 아니면 삭제되지 않는다.
- `ln` command, `link()` system call (Hard Links)

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

  - 디렉토리에 다른 이름을 생성하고 그것이 원본 파일의 같은 `inode`를 가리키게 한다.
- `rm` command, `unlink()` system call

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

  - user-readable name와 inode number 사이의 link를 제거한다.
  - reference count를 감소시키고 0이 되면 파일이 삭제된다.

9.3. Mechanisms for Resource Sharing

- 프로세스의 추상화
  - CPU 가상화 → private CPU
  - 메모리 가상화 → private memory
- 파일 시스템
  - 디스크 가상화 → 파일과 디렉토리
  - 파일들은 일반적으로 다른 유저 및 프로세스와 공유되므로 private하지 않다.
  - Permission bits

9.3.1. Permission Bits

```
prompt> ls -l foo.txt
-rw-r--r-- 1 remzi wheel 0 Aug 24 16:29 foo.txt
```

- 파일의 타입
  - `-`: 일반 파일
  - `d`: 디렉토리
  - `l`: symbolic link
- Permission bits
  - owner, group, other 순서로 읽기, 쓰기, 실행 권한을 나타낸다.
  - `r`: 읽기
  - `w`: 쓰기
  - `x`: 실행
  - 디렉토리의 경우 `x` 권한을 주면 사용자가 디렉토리 변경(`cd`)으로 특정 디렉토리로 이동할 수 있다.

9.3.2. Making a File System

- `mkfs` command
  - 해당 디스크 파티션에 루트 디렉토리부터 시작하여 빈 파일 시스템을 만든다.

```
mkfs.ext4 /dev/sda1
```

  - 균일한 파일 시스템 트리 내에서 접근 가능해야 한다.

9.3.3. Mounting a File System

- `mount` command
  - 기존 디렉토리를 대상 마운트 지점으로 사용하고, 기본적으로 해당 지점의 디렉토리 트리에 새로운 파일 시스템을 연결한다.

```
mount -t ext4 /dev/sda1 /home/users
```

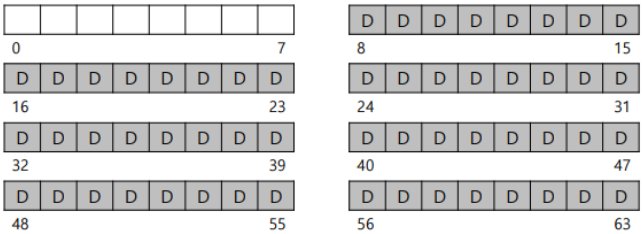
  - 경로 `/home/users`는 이제 새롭게 마운트된 파일 시스템의 루트를 가리킨다.

10. 23-File System Implementation

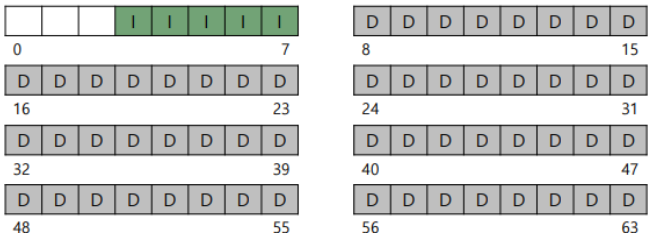
파일 시스템은 순수한 소프트웨어이다.
- CPU와 메모리 가상화와는 달리 파일 시스템이 더 좋은 성능을 내기 위한 측면으로 하드웨어 기능을 추가하지는 않는다.

10.1. Overall Organization

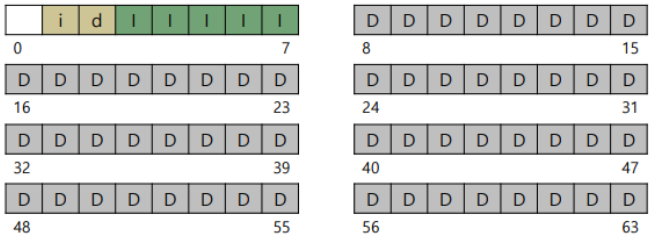
- Blocks
  - 디스크를 블록으로 나누어 관리한다.
- Data region
  - 이 블록들을 위해 디스크의 고정된 부분을 예약한다.



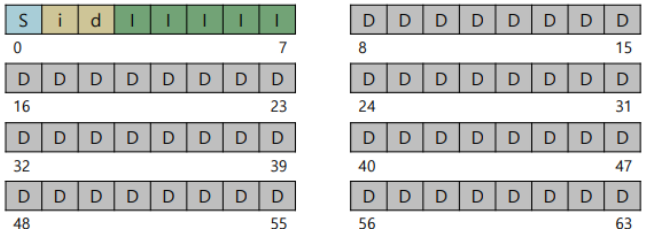
- Metadata
  - 각 파일에 대한 정보를 추적한다.
    - 파일을 구성하는 데이터 블록 (데이터 영역 내), 파일 크기, 소유자 및 접근 권한, 접근 및 수정 시간 등
  - `inode` (index node)
    - 메타데이터를 저장한다.
    - 디스크의 일부 공간에 `inode` 테이블을 예약한다.



- Allocation structures
  - `inode` 또는 데이터 블록이 free인지 할당되어 있는지를 추적한다.
  - free list, bitmap으로 구현



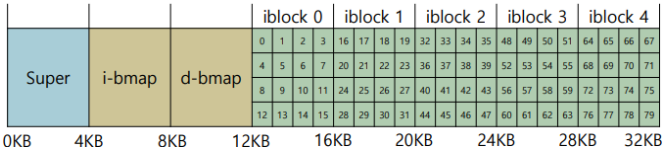
- Superblock
  - 이 특정 파일 시스템에 대한 정보를 포함한다.
  - `inode`와 데이터 블록이 파일 시스템에 얼마나 있는지, `inode` 테이블의 시작점이 어딘지와 같은 정보를 포함한다.
  - 파일 시스템이 마운트될 때, OS가 superblock을 먼저 읽고 다양한 파라미터를 초기화하고 나서 파일 시스템 트리에 볼륨을 할당한다.



- Block size: 4 KB
  - 256 KB partition (64-block partition)
- `inode` size: 256 B
  - 16 `inode`s per block
  - 총 80 `inode`s

10.2. `inode`

- i-number
  - 각 `inode`는 암묵적으로 숫자로 참조된다.
  - i-number가 주어지면 디스크에서 해당 `inode`가 어디에 있는지 바로 계산할 수 있다.



- `inode` number 32를 읽으려고 한다.
  - `inode` region에서의 offset: $32 \times \text{sizeof(inode)} = 8\text{KB}$
    - `inode` size = 256B
  - 디스크에서 `inode` 테이블의 시작 주소: 12KB
  - 12KB + 8KB = 20KB
- 디스크는 바이트 주소 지정이 가능하지 않지만, 주소 지정이 가능한 다수의 구간으로 구성되어 있다.
  - 일반적으로 512B
  - sector address: $\frac{20\times1024}{512} = 40$
- 그렇다면 `inode`가 데이터 블록이 어디에 있는지를 참조하는 방법은?
  - Multi-level index
    - `inode`는 고정된 수의 direct pointer와 하나의 indirect pointer를 가지고 있다.
    - indirect pointer
      - 사용자 데이터가 포함된 블록을 가리키는 대신, 사용자 데이터를 가리키는 포인터가 더 많이 포함된 블록을 가리킨다.
      - 파일이 충분히 커지면 indirect block이 할당된다.

  - double indirect pointer
    - 더 큰 파일들을 지원할 수 있다.
    - indirect block을 가리키는 포인터들을 포함한 블록을 참조한다.
- 예시
  - 12개의 direct pointer
  - 1개의 indirect pointer
  - Block size: 4KB
  - 4B disk address
  - $(12 + 1024) \times 4\text{KB} = 4144\text{KB}$ 크기의 파일 수용 가능

10.3. Directory Organization

- 디렉토리
  - 디렉토리는 파일의 특별한 형태이다.
    - `inode`의 type 필드는 “regular file” 대신 “directory"로 마킹한다.
  - 데이터 블록에 (entry, i-number)쌍들의 배열을 포함한다.

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_long_name

- Record length (reclen): total bytes for the name plus any left over space (e.g., must be a multiple of 4 in ext4)
- String length (strlen): actual length of the name

  - 파일을 제거하는 것은 디렉토리의 중간에 빈 공간을 남길 수 있다.
    - i-number (inum) → 0 (reserved)
    - reclen
      - 새로운 항목은 오래되거나 더 큰 항목을 덮어쓸 수 있으므로 안에 extra 공간을 가진다.

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
0	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_long_name

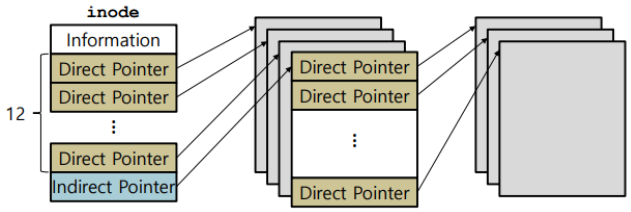
10.4. Free Space Management

파일을 생성할 때를 예로 들어보자.

- 파일 시스템은 비트맵을 탐색하며 `inode`가 free인 것을 찾고 그것을 파일에 할당한다. (1을 적으면 사용 중)
  - 데이터 블록에서와 유사함
- 몇몇 파일 시스템은 새로운 파일이 생성되어 데이터 블록을 필요로 할 때, 순차적으로 블록을 탐색하여 free한지를 본다.

10.5. System Calls (FILE)

- `open("/foo/bar", O_RDONLY)`
  - `/foo/bar` 파일을 찾기 위해 `inode`를 먼저 찾는다.
    - root 디렉토리의 `inode`를 읽는다.
      - `/`의 i-number는 일반적으로 2이다.
      - 0: `inode`가 없음
      - 1: `inode`가 올바르지 않은 블록에 있음
    - 하나 또는 그 이상의 디렉토리 데이터 블록을 읽으므로서 `foo` 항목을 찾을 수 있다. (`foo`의 i-number도)
    - `foo`의 `inode`를 포함한 블록을 읽고나서 그것의 디렉토리 데이터인 `bar`의 `inode` number를 찾는다.
  - `bar`의 `inode`를 메모리로 읽어온다.
  - 권한을 확인한다.
  - per-process open-file 테이블에 있는 이 프로세스에 file descriptor를 할당한다.
  - 유저에게 이것을 반환한다.
- `read()`
  - 파일의 첫 번째 블록을 읽고, 해당 블록의 위치를 찾기 위해 `inode`를 참조한다.
  - 마지막 접근 시간으로 `inode`가 새롭게 업데이트 될 수 있다.
  - file offset을 업데이트 한다.



- `close()`
  - file descriptor 할당을 해제한다.
  - disk I/O가 발생하지 않는다.

Reading a File from Disk

	d-bmap	i-bmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]
open() /foo/bar			read	read	read	read	read		
read() (4KB)					read write			read	
read() (4KB)					read write				read
...									

- `write()`
  - 새 파일을 쓸 때, 각 쓰기 작업은 디스크에 데이터를 써야 할 뿐만 아니라 먼저 파일에 할당할 블록을 결정하고 이에 따라 디스크의 다른 구조(data bitmap과 `inode`)를 업데이트 해야 한다.
  - 각 쓰기 작업은 논리적으로 5개의 I/O를 발생시킨다.
    - 데이터 비트맵을 읽는데 1개
    - 비트맵을 쓰는데 1개
    - `inode`를 읽고 쓰는데 2개 이상
    - 실제 블록에 쓰는데 1개

	d-bmap	i-bmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]
create /foo/bar		read write	read	read	(read) write write	read	read write		
write() (4KB)	read write				read write			write	
...									

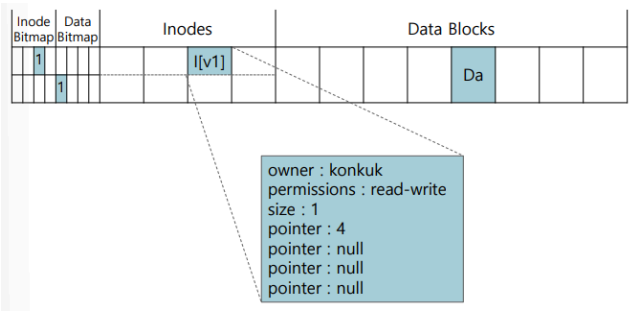
10.6. Caching and Buffering

- 디스크에 많은 I/O가 있으면 파일 입출력 비용이 클 수 있다.
  - 파일을 열 때마다 디렉토리 계층 구조의 모든 level에 대해 최소 2번의 읽기가 필요하다.
    - 하나는 쿼리를 한 디렉토리의 `inode`를 읽는 것에, 하나는 그것의 데이터를 최소 하나라도 읽어놓는 것에 필요하다.
- Page cache
  - 처음 open에는 디렉토리에 `inode`와 데이터를 읽는데 많은 I/O 트래픽을 생성할 수 있다.
  - 동일한 파일(또는 동일한 디렉토리에 있는 파일)의 후속 파일을 열 때에는 대부분 캐시에 hit된다.
- Write buffering
  - 쓰기 작업에 딜레이를 줘서 파일 시스템은 작은 집합의 I/O들에 대해 일괄적으로 업데이트 할 수 있다.
  - 동일한 I/O들을 스케줄링 할 수 있다.
  - 일부 쓰기 작업은 딜레이를 통해 완전히 방지될 수 있다.

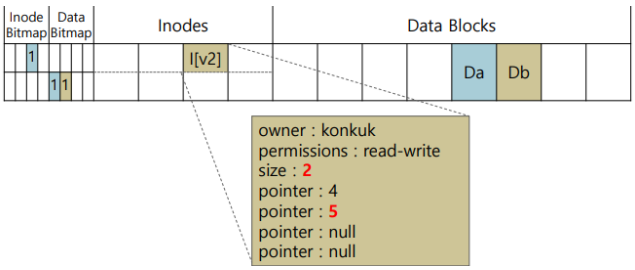
11. 24-FSCK and Journaling

11.1. How to Update the Disk despite Crashes

충돌이 일어나도 디스크를 업데이트 하는 방법
- 두 번의 쓰기 작업 사이에 시스템이 충돌하거나 전원이 꺼질 수 있다.
  - 충돌은 임의의 시점에 발생할 수 있다.
  - On-disk 상태는 일부분만 업데이트 되어 있을 수 있다.
  - 충돌 후 시스템이 부팅되고, 파일 시스템을 다시 마운트하려고 한다.
- 파일 시스템이 On-disk 이미지를 적절한 상태로 유지하는 것을 어떻게 보장할까?
  - File System Checker (FSCK)
  - Journaling
- 예시



  - 기존 파일에 하나의 데이터 블록을 추가하려고 한다.
    - 파일을 연다
    - `lseek()`으로 file offset을 파일의 끝으로 옮긴다.
    - 파일을 닫기 전에 파일에 4KB 쓰기 작업 하나를 요청한다.



  - Data bitmap, `inode`, data block을 쓴다.
  - **충돌 시나리오 (오직 한 번의 쓰기만 성공한 경우)**
    - 데이터 블록(Db)만 쓰인 경우
      - 파일 시스템 충돌 일관성의 관점에서 문제가 없다.
    - `inode`(I[v2])만 쓰인 경우
      - 디스크로부터 쓰레기 값을 읽어온다.
      - File-system inconsistency
    - bitmap(B[v2])만 쓰인 경우
      - 공간 누수를 발생시킬 수 있다.
      - File-system inconsistency
  - **충돌 시나리오 (2번의 쓰기가 성공, 하나는 실패한 경우)**
    - `inode`와 bitmap이 쓰인 경우
      - 파일 시스템의 메타데이터 관점에서는 괜찮아 보이지만, 데이터 블록은 쓰레기 값이다.
    - `inode`와 데이터 블록이 쓰인 경우
      - File-system inconsistency
    - bitmap과 데이터 블록이 쓰인 경우
      - File-system inconsistency

11.2. File System Checker (FSCK)

- fsck
  - 파일 시스템 불일치를 찾아 복구하기 위한 UNIX 도구
  - 불일치가 발생하도록 두고 나중에(재부팅 시) 수정한다.
    - 모든 문제를 고칠 순 없다.
    - 파일 시스템 메타데이터가 내부적으로 일관성을 유지하는지 확인하는 것이 목표이다.
  - Superblock
    - 먼저 superblock이 reasonable하게 보이는지 확인한다.
  - Free blocks
    - 다음으로 `inode`, indirect blocks, double indirect blocks 등을 스캔하여 파일 시스템 내에 현재 할당된 블록을 읽는다.
    - 비트맵과 `inode` 사이에 불일치가 있는 경우 `inode`내의 정보를 신뢰하여 문제를 해결한다.
  - `inode` state
    - 할당된 각 `inode`에 유효한 유형의 필드가 있는지 확인한다.
      - 예: 일반 파일, 디렉토리, symbolic link 등
    - `inode`필드에 쉽게 해결되지 않는 문제가 있는 경우 해당 `inode`는 의심스러운 것으로 간주되어 삭제된다.
      - 이에 따라 `inode` 비트맵이 업데이트 된다.
  - `inode` link
    - 각 `inode`에 대한 참조 수를 계산한다.
  - Duplicates
    - duplicate pointer를 확인하다. 즉, 두 개의 서로 다른 `inode`가 동일한 블록을 참조하는 경우이다.
      - 올바르지 않은 `inode`를 제거하고 블록을 복사한다.
  - Bad block pointers
    - 포인터가 유효 범위 밖의 무언가를 가리키는 것이 확실하면 포인터는 "올바르지 않은 것"으로 간주된다.
    - 포인터를 지운다.
  - Directory checks
    - 각 디렉토리의 내용에 대해 무결성 검사를 수행한다.
      - `/.`과 `/..`이 첫 번째 항목이다.
      - 디렉토리 항목에서 참조되는 각 `inode`가 할당된다.
  - 단점
    - 너무 느리다.
      - 매우 큰 용량의 디스크에서 모든 디스크를 읽는 것은 몇 분에서 몇 시간까지 걸린다.
    - 작동하지만 낭비가 많다.
      - 몇 개의 블록을 업데이트하는 동안 발생한 문제를 해결하기 위해 전체 디스크를 읽는 비용이 크다.

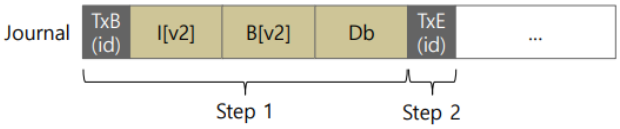
11.3. Journaling (Write-ahead Logging)

- Journaling
  - 디스크를 업데이트 할 때, 구조를 제자리에 덮어쓰기 전에 먼저 수행할 작업을 설명하는 작은 메모(디스크의 다른위치에)를 적어 둔다.
- Checkpointing
  - pending 메타데이터 및 데이터 업데이트를 파일 시스템의 최종 위치에 쓴다.
- Ext3
  - On-disk 구조
    - 디스크는 블록 그룹으로 나누어진다.
    - 각 블록 그룹은 `inode` 비트맵, 데이터 비트맵, `inode`와 데이터 블록을 포함한다.
    - Journal

11.3.1. Data Journaling

- journal 쓰기
  - 한 가지 간단한 방법은 한 번에 하나씩 요청을 하고 각각이 완료될 때까지 기다린 후 다음을 요청하는 것이다.
    - 느리다.
  - 한 번에 5개의 블록 쓰기를 모두 실행한다.
    - 디스크 스케줄링 → 재정렬 필요
    - journal을 쓰는 동안의 충돌

  - 2-step으로 트랜잭션 쓰기를 요청한다.
    - Step 1: TxE 블록을 제외한 모든 블록을 쓴다.
    - Step 2: Step 1이 완료되면, TxE 블록의 쓰기를 요청한다.



- TxE 쓰기가 atomic하게 이루어지도록 하려면 이를 단일 512-byte 구간으로 만들어야 한다.
  - 디스크는 512-byte 쓰기 작업 발생 여부를 보장한다.

11.3.2. Recovery

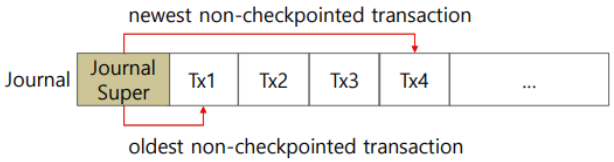
- 만약 충돌이 트랜잭션이 로그에 안전하게 쓰이기 전에 일어나면?
  - pending 업데이트는 무시된다.
- 만약 충돌이 트랜잭션이 log에 커밋된 후 체크포인트가 완성되기 전에 일어나면?
  - log를 읽어서 디스크에 커밋된 트랜잭션을 찾는다.
  - 트랜잭션이 (순서대로) 재생되어 트랜잭션의 블록을 디스크의 최종 위치에 쓰려고 시도한다.
    - 일부 업데이트는 복구 중에 다시 중복 수행된다.

11.3.3. Batching Log Updates

- 문제점: 추가적인 디스크 트래픽을 많이 발생시킬 수 있다.
  - 예: 같은 디렉토리에 두 개의 파일을 생성할 때
    - 만약 이 파일들의 `inode`들이 같은 `inode` 블록에 있으면 같은 블록을 계속해서 쓰게 된다.
- 해결법: 버퍼링 업데이트
  - 파일 시스템은 일정 시간 동안 메모리에 업데이트를 버퍼링한다. (고의적인 딜레이를 주는 것)
  - 디스크에 대한 과도한 쓰기 트래픽을 방지할 수 있다.

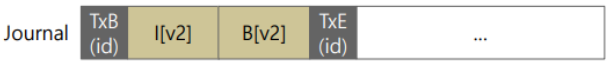
11.3.4. Making the Log Finite

- 문제점: 로그가 가득 찼을 때
  - 로그를 더 키우면, 복구에 더 많은 시간이 걸린다.
  - 더 이상 트랜잭션이 커밋될 수 없다.
- 해결법: Cirular log
  - 트랜잭션이 체크포인트가 되면, 파일 시스템은 공간을 free시킬 수 있다.



11.3.5. Ordered Journaling (=Metadata Journaling)

- 문제점: Data journaling
  - 디스크에 각 쓰기 작업에서 journal에 먼저 쓰게 되므로 쓰기 트래픽이 2배가 된다.
- 해결법: Metadata journaling
  - 유저 데이터는 journal에 쓰지 않는다.



  - 데이터 블록을 언제 디스크에 쓸까?
    - 일부 파일 시스템(예: Linux ext3)은 관련 메타데이터가 디스크에 쓰이기 전에 먼저 데이터 블록을 디스크에 기록한다.
- Basic Protocol
  1. Data write
  2. Journal metadata write
  3. Journal commit
  4. Checkpoint metadata
  5. Free

12. 25-Log-Structured File Systems

- 시스템 메모리가 늘어나고 있다.
  - 더 많은 데이터가 캐싱 됨에 따라 디스크 트래픽은 점점 더 쓰기 작업으로 구성되고 읽기 작업은 캐시에 의해 처리된다.
- 랜덤 I/O와 순차 I/O는 큰 성능적 차이가 있다.
  - 그러나 탐색 및 회전 지연 비용은 천천히 감소해왔다.
- 기존 파일 시스템은 여러 일반적인 워크로드에서 제대로 작동하지 않는다.
  - 예를 들어, 파일 시스템은 한 블록 크기의 새 파일을 생성하기 위해 많은 수의 쓰기 작업을 수행한다.
- **LFS**
  - 디스크에 쓸 때, LFS는 먼저 모든 업데이트(메타데이터 포함)를 in-memory 세그먼트에 버퍼링한다.
  - 세그먼트가 가득 차면, 디스크의 사용되지 않은 부분에 하나의 긴 순차적 전송으로 디스크에 쓴다.
    - 기존 데이터를 덮어쓰지 않고 항상 빈 위치에 세그먼트를 쓴다.
  - 최근 연구에 따르면 플래시 기반 SSD의 고성능을 위해서는 대규모 I/O가 필요하다.
    - LFS 스타일의 파일 시스템은 다음 수업에서 플래시 기반 SSD에도 탁월한 선택이 될 수 있다는 것을 보일 것이다.

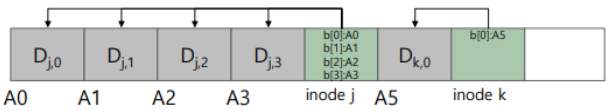
12.1. Writing to Disk Sequentially

- 파일 시스템 상태에 대한 모든 업데이트를 디스크에 순차적인 쓰기 작업의 일환으로 어떻게 변환할까?

  - 유저가 데이터 블록을 쓸 때, 디스크에 기록되는 것은 데이터 뿐만이 아니고 업데이트 해야 하는 다른 메타데이터도 있다.

12.2. Writing Sequentially and Effevtively

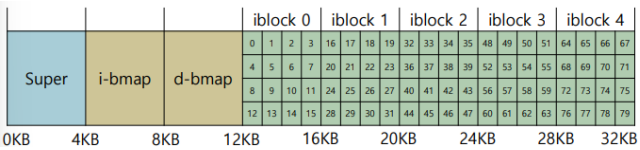
- 단순히 순차적으로 디스크에 쓰는 것만으로는 최고 성능을 달성하기에 충분하지 않다.
  - 시간 T: 단일 블록을 주소 A에 쓰는 것
  - 시간 $T + \delta$: 디스크 주소 A + 1에 쓰는 것
  - 디스크는 두 번째 쓰기를 디스크 표면에 커밋할 수 있기 전까지 $T_{\text{rotation}} - \delta$ 만큼 기다릴 것이다.
- 오히려, 드라이브에 다수에 연속 쓰기(또는 하나의 대규모 쓰기)를 실행해야 한다.
- 쓰기 버퍼링
  - 디스크에 쓰기 전에, LFS는 메모리에 업데이트의 track을 유지한다.
  - 충분한 수의 업데이트를 받으면 디스크에 한 번에 쓴다.
- 세그먼트
  - LFS가 한 번에 쓰는 큰 청크 (예: 몇 MB)



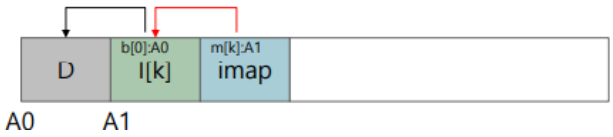
- 얼마나 많은 업데이트를 LFS가 디스크에 쓰기 전에 버퍼링해야 할까?
  - 디스크 자체에 의존한다. (뒤에 내용을 보자)

12.3. `inode` Map (imap)

- 단순한 파일 시스템

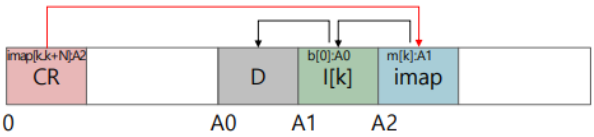


- LFS
  - `inode`는 디스크 전체에 흩어져 있다.
  - 절대 덮어쓰지 않으므로 최신 버전의 `inode`는 계속해서 이동한다. (어떻게 찾지..?)
- imap
  - `inode` 번호를 입력으로 사용하고 `inode`의 최신 버전의 디스크 주소를 생성하는 구조이다.
  - `inode`가 디스크에 쓰일 때마다 imap은 새로운 위치로 업데이트 된다.
- persistent를 유지해야 한다.
  - imap은 디스크의 어디에 위치해야 할까?
    - 디스크의 고정된 위치 → 성능이 저하됨 (다시 이거 찾으려고 회전해야 함)
    - 다른 모든 새로운 정보가 기록되는 바로 옆에 위치



- 근데 imap은 그러면 또 어떻게 찾을까?

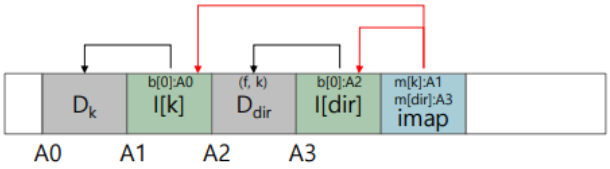
- Checkpoint Region (CR)
  - `inode` 맵의 최신 버전에 대한 포인터를 포함하므로 CR을 읽어 `inode` 맵 위치를 찾을 수 있다.



  - 주기적으로 업데이트 됨 (일반적으로 매 30초 정도에 1번)
    - 이렇게 안 하면 저렇게 뒤에 순차적으로 붙이는 의미가 없음..
- 파일을 디스크로부터 읽어올 때
  - checkpoint region을 먼저 읽는다.
  - 그리고 나서 `inode` 맵 전체를 읽고 메모리에 캐싱한다.
  - 파일로부터 block을 읽으려면 여기서 LFS는 일반적인 파일 시스템과 동일하게 진행된다.
    - 전체 imap이 캐싱되므로 LFS가 읽는 동안 수행하는 추가 작업은 imap에서 `inode`의 주소를 찾는 것이다.

12.4. What About Directories?

- 파일 `dir/f`를 생성할 때
  - imap에는 디렉토리 파일 `dir`과 새로 생성된 파일 `f`의 위치에 대한 정보가 포함되어 있다.

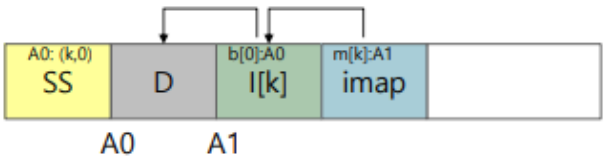


- 파일 `f`를 접근할 때
  - 먼저 `inode` 맵(보통 메모리에 캐싱됨)을 보고 디렉토리 `dir`(A3)의 `inode` 위치를 찾는다.
  - 그런 다음 디렉토리 데이터의 위치를 제공하는 디렉토리 `inode`를 읽는다. (A2)
  - 이 데이터 블록을 읽으면 (f, k)의 (파일 이름, `inode` 번호)쌍을 알 수 있다.
  - 그런 다음 `inode` 맵을 다시 참조하여 `inode` 번호 k(A1)의 위치를 찾는다.
  - 마지막으로 주소 A0에서 원하는 데이터 블록을 읽는다.

12.5. Garbage Collection

- 파일 데이터, `inode` 및 기타 구조의 오래된 버전을 주기적으로 찾아서 정리한다.
  - 이어지는 쓰기에 사용할 수 있도록 디스크의 블록을 다시 사용 가능하게 만든다.
- 이어지는 쓰기를 위해 큰 청크를 정리한다.
  - LFS 클리너는 주기적으로 M개의 오래된(부분적으로 사용된) 세그먼트를 읽는다.
  - 해당 세그먼트 내에서 어떤 블록이 활성화되어 있는지 결정한다.
  - 해당 내용을 N개의 새 세그먼트(N < M)로 압축한 다음, N개의 세그먼트를 디스크의 새 위치에 쓴다.
  - 이전 M개의 세그먼트는 해제되어 이어지는 쓰기를 위해 파일 시스템에서 사용할 수 있다.
- 블록의 생명 결정
  - LFS는 세그먼트 내의 어떤 블록이 활성 상태이고 어떤 블록이 죽은 상태인지 어떻게 알 수 있을까?
  - Segment summary block
    - 디스크 주소 A에 있는 각 데이터 블록 D에 대해 해당 `indoe` 번호 N과 오프셋 T를 포함한다.

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```



- 어떤 블록을 지워야하고 언제 지워야 할까?
  - cleaner가 얼마나 자주 실행되어야 하고, 어떤 세그먼트를 골라서 지워야 할까?
  - 언제 지울지 결정
    - 주기적으로, 유휴 시간 동안, 디스크가 가득 차면
  - 어떤 블록을 지울지 결정
    - Hot Segments
      - 내용이 자주 덮어쓰이는 것
    - Cold Segments
      - 죽은 블록이 몇 개 있을 수 있지만 나머지 내용은 비교적 안정적인 것
    - cold segment 먼저, hot segment 나중에

12.6. Crash Recovery

- LFS가 디스크에 쓰는 동안 시스템 충돌이 일어나면 journaling을 다시 호출한다.
- CR에 쓰는 동안 충돌이 일어나면
  - LFS는 두 개의 CR을 유지하고 교대로 쓴다.
  1. 먼저 헤더(타임스탬프 포함)를 작성한다.
  2. 그런 다음 CR의 body를 작성한다.
  3. 마지막으로 마지막 블록 하나(타임스탬프 포함)를 작성한다.
  - 항상 일관된 타임스탬프가 있는 가장 최근 CR을 사용하도록 선택한다.
- 세그먼트에 쓰는 동안 충돌이 일어나면
  - LFS는 약 30초마다 CR을 작성하므로 파일 시스템의 마지막 일관된 스냅샷은 꽤 오래되었을 수 있다.
  - Roll Forward
    - 로그(CR에 포함된)의 끝을 찾은 후 이를 사요하여 다음 세그먼트를 읽고 그 안에 유효한 업데이트가 있는지 확인한다.
    - 있는 경우 LFS는 그에 따라 파일 시스템을 업데이트하여 마지막 체크포인트 이후에 기록된 많은 데이터와 메타데이터를 복구한다.

13. 26-Flash-based SSDs

- Solid-State Storage (SSD)
  - 기계적으로 움직이는 부분들이 없음 (arm이 돌아가는 것)
  - 메모리 같은 형태로 동작하지만, 파워가 없어진 후에도 정보가 남아 있음
- Flash (NAND-based flash)
  - 플래시 칩은 단일 트랜지스터에 하나 이상의 비트를 저장하도록 설계되었다.
  - 트랜지스터 내에 비트 수준은 이진 값으로 매핑된다.
    - single-level cell(SLC) 플래시에서는 트랜지스터 내에 단일 비트(예: 1 또는 0)만 저장된다.
    - multi-level cell(MLC) 플래시를 사용하면 서로 다른 비트가 다른 level(예: 00, 01, 10, 11)로 인코딩된다.
    - 셀당 3비트를 인코딩하는 triple-level cell(TLC) 플래시도 있다.

13.1. Basic Flash Operations

- 단순한 예제

		`iiii`	Initial: pages in block are invalid (i)
`Erase()`	→	`EEEE`	State of pages in block set to erased (E)
`Program(0)`	→	`VEEE`	Program page 0; state set to valid (V)
`Program(0)`	→	`error`	Cannot re-program page after programming
`Program(1)`	→	`VVEE`	Program page 1
`Erase()`	→	`EEEE`	Contents erased; all pages programmable

- 디테일한 예제
  - 8-bit 페이지, 4-page 블록

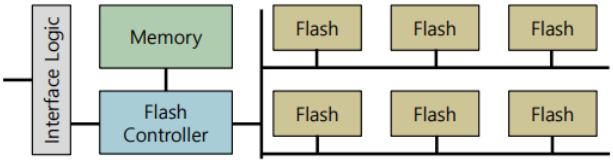
Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VAILD	VAILD	VAILD	VAILD

– Wish to write to page 0 with 00000011

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

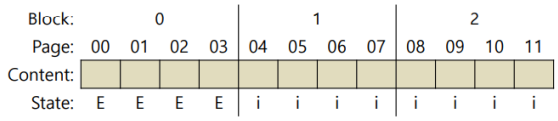
  - 블록에 있는 어떠한 페이지를 덮어쓰기 전에, 먼저 중요한 데이터는 다른 블록에 복사해놓아야 한다. (덮어쓰려면 블록 단위로 erase를 진행해야 하기 때문)
- 기본 플래시 칩을 일반적인 저장 장치처럼 보이는 것으로 바꾸는 방법은 무엇일까?
- SSD
  - 몇 개의 플래시 칩이 있다.
  - 일정량의 휘발성 메모리(예: SRAM)가 있다.
    - 캐싱 및 버퍼링에 유용하다.
  - control logic
    - Flash Translation Layer(FTL): 논리 블록에 대한 읽기 및 쓰기 요청을 받아 이를 low-level의 읽기, 삭제 및 program(쓰기) 명령으로 변환한다.

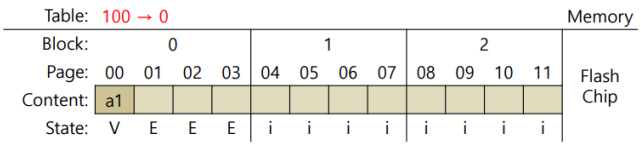


- Direct mapped (가장 간단하나 성능은 좋지 않은 방식 - bad approach)
- 논리적 페이지 N개 읽기는 물리적 페이지 N개 읽기에 직접 매핑된다.
- 논리적 페이지 N개에 쓰기는 더 복잡하다.
  - 먼저 페이지 N개가 포함된 전체 블록을 먼저 읽는다.
  - 그런 다음 블록을 지운다.
  - 마지막으로 이전 페이지와 새 페이지를 program(쓰기)한다.
- 성능 문제
  - 쓰기 증폭: FTL이 플래시 칩에 날린 총 쓰기 트래픽(byte)을 클라이언트가 날린 총 쓰기 트래픽(byte)으로 나눈 것
- 신뢰성 문제
  - 마모: 단일 블록을 너무 자주 지우고 program(쓰기)하면 더 이상 사용이 불가능하다. (소자의 특성 상 블록을 지우는 횟수에 제한이 있다.)

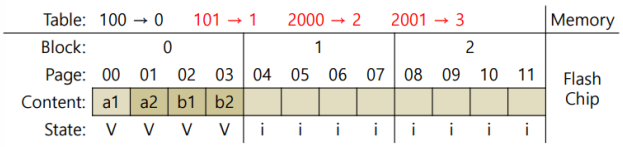
13.2. A Log-Structured FTL

- 오늘날 대부분의 FTL들은 log structured를 사용하고 있다.
  - 논리적 블록 N개를 쓰려고 하면, 장치는 현재 쓰고 있는 블록에서 다음 free 공간에 쓴다.
- 예시
  - 가정
    - 클라이언트는 4-KB 사이즈에 대해 읽기 또는 쓰기를 요청한다.
    - SSD는 4개의 4-KB 페이지로 구성된 16-KB 크기의 블록이 있다.
  - 100번 논리 주소에 a1을 쓰려고 한다.

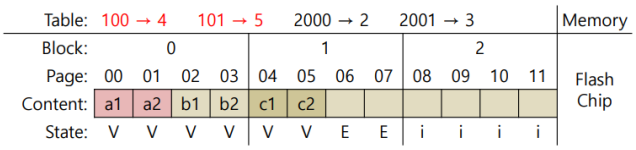




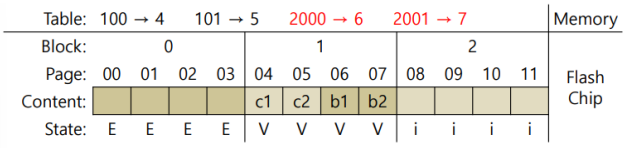
  - 101번 논리 주소에 a2를 쓰려고 한다.
  - 2000번 논리 주소에 b1을 쓰려고 한다.
  - 2001번 논리 주소에 b2를 쓰려고 한다.



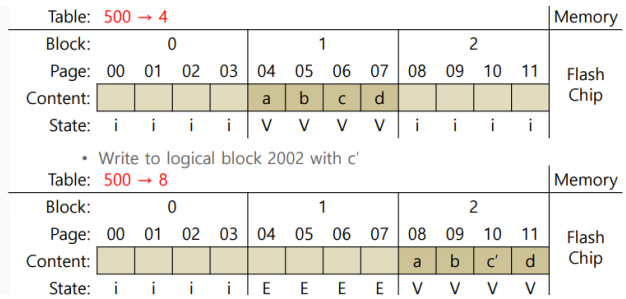
  - 장점
    - 로그 기반 접근법은 특성상 성능을 향상시킨다.(삭제를 덜 함)
    - 모든 페이지에 쓰기를 분산시켜(wear leveling) 장치의 수명을 늘린다.
  - 단점
    - Garbage collection
      - 논리 블록의 덮어쓰기는 garbage을 만든다.
    - in-memory mapping table의 높은 비용
      - 매핑 테이블(Out-Of-Band(OOB) area)의 지속성 처리 (전원이 꺼졌을 때 날라가지 않는 곳에 저장해놓아야 함)
      - 장치가 클수록 해당 테이블에 더 많은 메모리가 필요하다.
- 예시 (Garbage Collection)
  - 블록 100, 101에 다시 쓰기를 하려고 한다.



    - 하나 이상의 garbage page를 포함하는 블록 찾기
    - 해당 블록의 live(non-garbage) 페이지를 읽는다.
    - 해당 live 페이지를 로그에 기록하고 마지막으로 쓰기에 사용하기 위해 전체 블록을 회수한다.



  - 매핑 테이블의 크기
    - 1TB SSD에서 page가 4KB이고 각 엔트리가 4B인 경우
      - 매핑을 위해 1GB 메모리가 필요하다. (1TB / 4KB x 4B = 1GB)
      - 페이지 단위 FTL 체계는 실용적이지 않다.
    - Block-based 매핑
      - 페이지 단위가 아닌 장치의 블록 단위로만 포인터를 유지한다.
      - 논리 블록 주소: chunk number + offset (4개의 페이지로 구성되어 있으므로 offset bit는 2개)
        - 논리 블록 2000: 0111 1101 0000
        - 논리 블록 2001: 0111 1101 0001
      - 가장 큰 문제는 작은 쓰기가 발생할 때 생긴다.
        - 기존 블록에서 대량의 실시간 데이터를 읽어서 새 블록에 복사해야 함
  - 예시 (Block-based mapping)
    - 기존 방식: 2000 → 4, 2001 → 5, 2002 → 6, 2003 → 7에 매핑되어 있음



    - 블록 전체를 옮겨야 할 수 있는 단점이 있음
  - 하이브리드 매핑
    - 로그 테이블: 작은 단위의 페이지별 매핑
    - 데이터 테이블: 더 큰 블록별 매핑
    - 특정 논리 블록을 찾을 때, 먼저 로그 테이블을 참조한 다음 데이터 테이블을 참조
    - 로그 블록의 수를 작게 유지하기 위해 FTL은 주기적으로 로그 블록을 검사하여 블록으로 전환해야 한다.

- Wear Leveling
  - 여러 번의 삭제 / program 주기로 인해 플래시 블록이 마모된다. FTL은 해당 작업을 장치의 모든 블록에 균등하게 분산시키기 위해 최선을 다해야 한다.
    - 모든 블록은 거의 동시에 마모된다.
    - 기본 로그 구조 접근법은 쓰기 부하를 분산시키는 초기 작업을 잘 수행하며 garbage collection에도 도움이 된다.
    - 덮어쓰지 않는 수명이 긴 데이터로 블록이 채워지면 garbage collection은 블록을 회수하지 않는다.
    - 쓰기 부하를 공평하게 분배 받지 못한다.
    - 이 문제를 해결하려면 FTL은 주기적으로 해당 블록에서 모든 live 데이터를 읽고 다른 곳에 써야 한다.