운영체제 기말 정리본

dldyou

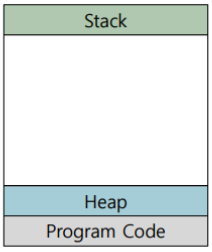
목 차

1. 14-Concurrency and Threads

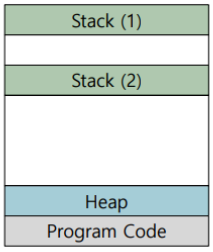
1.1. Threads

- 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

```c
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

```c
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

```c
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;
}
```

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)

```c
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

```c
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.

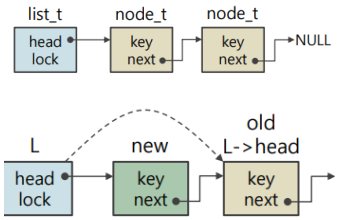
3.1.2. Sloppy Counters

- 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
}
```

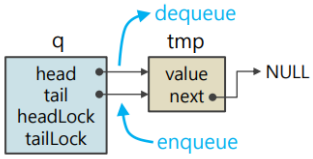
3.2.1.1. Scaling Linked Lists

- 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

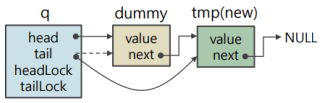
```c
void List_Insert(list_t *L, int key) {
    ...
RETRY: next = L->head;
    new->next = next;
    if (CAS(&L->head, next, new) == 0)
        goto RETRY;
}
```

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);
}
```

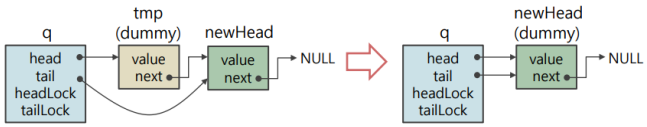



```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next = NULL;
    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}
```



- 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);
}
```