Races

경주처럼 여러 곳에서 접근하여 실행하는 것.

- Race condition
  공유자원 바꾸는 함수 실행 중에 또 다른 스레드에서 공유자원 바꿀수도 있다.
  언제 도착할지 예측하지 못한다.
  --> 한번에 하나만 작업 가능하도록 순서대로 동작하도록 동기화 작업 필요

Synchronization

동기화 작업을 통해 Race Condition 피하기

Data Races

발생 시점

- 두개 이상의 concurrent flow가 공유 자원에 접근한 경우
- 하나 이상의 흐름이 상태를 변경하는 경우
- 접근/수정 순서가 중요하다

data read는 상관 없지만 write가 중요하다

Race Example

```
char* s[4];
int strnum;

void setstring(char *str){
    int index = strnum;
    s[index]=str;
    strnum++;
}
```

thread 접근이 랜덤이기 때문에 strnum이 순차적으로 증가하지 않는다.
-> s에 덮어써지는 이슈 등 발생..

Critical Section

공유자원에 영향을 미치는 곳
해당 부분을 정의하여 single thread만 돌도록 한다.

single-thread로 작동하기 때문에 cirtical 부분이 넓을수록 성능 저하

- shared data에 write하는 대부분의 코드가 critical section...
- 해당 부분이 변경되지 않는다면 read는 critical section아님

Race Condition 해결을 위한 primitive

1. Atomic - HW support / atomic한 연산이 필요하다.
2. Mutex - 한 thread 독점
3. Semaphore - 한정된 수만 접근하는 공유자원

Atomic Operation

하드웨어적으로 접근한다.
가장 단순한 동기화 메카니즘이다.

- cannot be interrupted
- 동시에 실행되지 않는 것처럼 보인다 (sigle-thread만 수행하도록 되어있기에)
- 모두 성공 아니면 모두 실패다 (부분은 없음 atomic: 쪼개질 수 없는, 원자의)

모든 atomic operation은 하드웨어의 서포트가 필요하다.

모든 기계어가 atomic 한 것은 아니다

ex) 더하기

- add -> atomic_add

--> c에서는 라이브러리 형태로 사용한다

Mutex

Mutual Exclusion
atomic operation은 하드웨어의 도움이 필요하기에.. 해당 방법 이용

- 공유 자원에 한번에 하나의 thread만 접근할 수 있도록 한다. (only one logical control flow)
- 상호 배타적이다 - lock을 건 스레드가 ownership을 가진다.

lock

thread가 mutex에 진입하여 lock걸기
lock 성공하면 return 0

- mutex
  locked 상태
- other thread
  - lock
    locked된 상태에서 다시 lock걸리지 못하기 때문에 계속 lock 시도함
  - trylock
    locked 된 뮤텍스에 lock걸면 안걸림
    -> 자기 자리로 돌아감..
- deadlock
  locked된 상태에서 계속 주인 스레드가 lock 걸면 발생

unlock

locked 된 mutex를 unlock하여 다른 스레드가 lock을 걸 수 있도록 함.

- mutex
  unlocked 상태
- ohter thread
    - lock/trylock
      lock을 걸려고 한 스레드 중 랜덤으로 ownership 가짐

      T1이 locked했을 때, T2가 lock 계속 걸고있고, mutex unlock된 상태에서 T3이 lock/trylock 하였어도 T2가 될지 T3이 될지 모름.

      lock 거는 순서에 따라 정해지는 것이 아니다

- deadlock
  unlocked된 mutex에 unlock을 걸면 이러한 상황이 발생할 수 있다.

trylock

mutex에 lock을 한 번 시도하고, 성공시 locked / 실패시 계속 시도하지 않고 호출부로 돌아감
always return immediately

lock 성공하면 return 0

Using Mutex around Critical Section

- 뮤텍스를 통해 critical section을 보호한다

1. lock mutex
2. critical section 실행
3. unlock mutex

--> critical section에서 하나의 스레드만 돌아감 (only one flow)

mutex ensure mutual exclusion in the critical section

pthread_mutex_t

- pthread_mutex_init()
  뮤텍스 초기화 및 생성
- pthread_mutex_lock(pthread_mutex_t *mutex)
- pthread_mutex_trylock(pthread_mutex_t *mutex)
- pthread_mutex_unlock(pthread_mutex_t *mutex)

▶ lock/trylock.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

pthread_mutex_t mutex;
```

```
8
9  void* routine_lock(void* arg) {
10     pthread_mutex_lock(&mutex);
11     printf("Got lock\n");
12     sleep(1);
13     pthread_mutex_unlock(&mutex);
14 }
15
16 void* routine_trylock(void* arg) {
17     if (pthread_mutex_trylock(&mutex) == 0) {
18         printf("Got lock\n");
19         sleep(1);
20         pthread_mutex_unlock(&mutex);
21     } else {
22         printf("Didn't get lock\n");
23     }
24 }
25
26 int main(int argc, char* argv[]) {
27     pthread_t th[4];
28
29     pthread_mutex_init(&mutex, NULL);
30     for (int i = 0; i < 4; i++) {
31         //lock 성공시 return 0 -> 처음 스레드가 lock건 1초 동안 나머지 3개의 thread도 l
32         //1초마다 unlock, lock 반복되어 4개 모두 lock가능해진다
33
34         //trylock 성공시 return 0 -> 1초 안에 나머지 3개 thread는 lock하지 못함
35         if (pthread_create(&th[i], NULL, &routine_lock, NULL) != 0) { // change the
36             perror("Error at creating thread");
37         }
38     }
39     for (int i = 0; i < 4; i++) {
40         if (pthread_join(th[i], NULL) != 0) {
41             perror("Error at joining thread");
42         }
43     }
44     pthread_mutex_destroy(&mutex);
45     return 0;
46 }
47
48
```

lock -> 4개의 thread 모두 걸 수 있음

trylock -> lock 걸린 상태면 1초 내에 나머지 thread는 lock걸지 못함

destroying mutex

뮤텍스 사용 후 폐기
만약 locked 상태에서 destroy하는 경우 에러 발생함

deadlock

```
void deadlock(){
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&mutex);
    pthread_mutex_lock(&mutex);
}
```

recursive lock may cause deadlock

Condition Variable

1. cond_wait() -> 일시적 unlock 상태
2. 다른 스레드의 lock 허용
3. 다른 스레드에서 unlock 후 cond_signal()
4. re-lock 후 조건 맞으면 정상 실행, 맞지 않으면 다시 cond_wait()

cond 기다리기..

1. cond_wait() -> 일시적 unlock mutex
2. put the thread to sleep until the condition is signaled

cond - 특정 하나 or broadcast

lock된 상태에서 여러개의 스레드와 함께 뮤텍스를 공유하는 방식

T1 -> lock -> signal T2 -> wait -> signal from T2 -> 실행 -> unlock..
T2 -> lock 실패 -> signal from T1 -> lock -> 실행 -> signal T1 -> unlock..

1. cond는 logical flow를 block한다(wait)
2. 다른 스레드가 실행 후 signal -> cond 만족시 다시 실행

Mutex Interaction

- waiting flow must hold a mutex to wait
- mutex muest protect data used in th condition check
- wait을 통해 mutex일시적인 unlock상태
- 조건 만족하고, wait 끝나면 mutex re-lock
- wait하는 동안 공유자원이 변경될수도 있다는 걸 의미한다..

Code

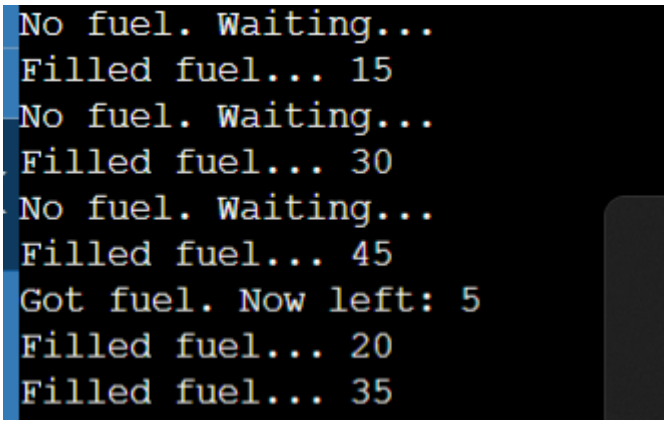
```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init();
int pthread_cond_wait();
int pthread_cond_destroy();
```

▶ carfuelling.c

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

```c
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

pthread_mutex_t mutexFuel;
pthread_cond_t condFuel;
int fuel = 0;

void* fuel_filling(void* arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutexFuel);
        fuel += 15;
        printf("Filled fuel... %d\n", fuel);
        pthread_mutex_unlock(&mutexFuel);
        //unlock 후 signal
        pthread_cond_signal(&condFuel);
        sleep(1); //sleep 하지 않으면 signal 받아서 lock하는 것보다 빨리 lock됨
    }
}

void* car(void* arg) {
    //signal 받아서 lock 가능
    pthread_mutex_lock(&mutexFuel);
    while (fuel < 40) {
        printf("No fuel. Waiting...\n");
        //fuel <40이면 fuel_fulling한테 signal 후 일시적 unlock상태태
        pthread_cond_wait(&condFuel, &mutexFuel);
        // Equivalent to:
        // pthread_mutex_unlock(&mutexFuel);
        // wait for signal on condFuel
        // pthread_mutex_lock(&mutexFuel);
    }
    fuel -= 40;
    printf("Got fuel. Now left: %d\n", fuel);
    pthread_mutex_unlock(&mutexFuel);
}

int main(int argc, char* argv[]) {
    pthread_t th[2];

    pthread_mutex_init(&mutexFuel, NULL);
    pthread_cond_init(&condFuel, NULL);
    for (int i = 0; i < 2; i++) {
        if (i == 1) {
            //fuel_filling 시작작
            if (pthread_create(&th[i], NULL, &fuel_filling, NULL) != 0) {
                perror("Failed to create thread");
            }
        } else {
            //car 달려서 fuel--시작작
            if (pthread_create(&th[i], NULL, &car, NULL) != 0) {
                perror("Failed to create thread");
            }
        }
    }

    for (int i = 0; i < 2; i++) {
```

```
60          if (pthread_join(th[i], NULL) != 0) {
61              perror("Failed to join thread");
62          }
63      }
64      pthread_mutex_destroy(&mutexFuel);
65      pthread_cond_destroy(&condFuel);
66      return 0;
67  }
68
69
70
```

- without cond

1. car 함수가 영원히 mutex를 잡고있었을 거임 (처음 fule 15에서 unlock되기에 while 무한루프에 빠짐)
2. fuel_filling func에서 5번 반복 중 하나만 하고 끝남 (signal 주고받기가 없기 때문에)

Semaphore

2개 이상의 mutex개념,, 다중 화장실
최대 count 정하고 thread 접근할때마다 하나씩 줄여나가는 형식

- First Term: P, Try, Wait = mutex lock
- Second Term: V, increment(증가), Post/Signal = mutex unlock

thread or process 사이에서 사용할 수 있다.

Mutex와 Semaphore(maxnum=1) 차이점

원리는 동일하지만 Mutex는 상호배제를 갖고있음. (ownership)

Semaphore Creation

```
#include <semaphore.h>
int sem_init(sem, pshared, value);
//pshared: true -> 프로세스간 사용 가능
//value: max thread num
```

▶ semaphore.c

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>

#define THREAD_NUM 4

sem_t semaphore;

void* routine(void* args) {
    sem_wait(&semaphore);
    //랜덤으로 2개 나온 후 1초 쉬고 또 2개 나오기기
    sleep(1);
    printf("Hello from thread %d\n", *(int*)args);
    sem_post(&semaphore);
    free(args);
}

int main(int argc, char *argv[]) {
    pthread_t th[THREAD_NUM];
    sem_init(&semaphore, 0, 2);
    int i;
    for (i = 0; i < THREAD_NUM; i++) {
        int* a = malloc(sizeof(int));
        *a = i;
        if (pthread_create(&th[i], NULL, &routine, a) != 0) {
            perror("Failed to create thread");
        }
    }

    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    sem_destroy(&semaphore);
    return 0;
}


```


▶ loginqueue.c

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>

#define THREAD_NUM 10

sem_t semaphore;
```

```c

void* routine(void* args) {
    printf("(User %d) Waiting in the login queue\n", *(int*)args);
    sem_wait(&semaphore);
    printf("(User %d) Logged in\n", *(int*)args);
    //세마포어에 들어온 스레드는 랜덤시간동안 sleep
    sleep(rand() % 5 + 1);
    printf("(User %d) Logged out\n", *(int*)args);
    //세마포어 나가기기
    sem_post(&semaphore);
    free(args);
}

int main(int argc, char *argv[]) {
    pthread_t th[THREAD_NUM];
    sem_init(&semaphore, 0, 4);
    int i;
    for (i = 0; i < THREAD_NUM; i++) {
        int* a = malloc(sizeof(int));
        *a = i;
        if (pthread_create(&th[i], NULL, &routine, a) != 0) {
            perror("Failed to create thread");
        }
    }

    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    sem_destroy(&semaphore);
    return 0;
}


```

```
(User 1) Waiting in the login queue
(User 1) Logged in
(User 4) Waiting in the login queue
(User 4) Logged in
(User 5) Waiting in the login queue
(User 5) Logged in
(User 6) Waiting in the login queue
(User 6) Logged in
(User 7) Waiting in the login queue
(User 9) Waiting in the login queue
(User 8) Waiting in the login queue
(User 3) Waiting in the login queue
(User 2) Waiting in the login queue
(User 0) Waiting in the login queue
(User 6) Logged out
(User 7) Logged in
(User 4) Logged out
(User 9) Logged in
(User 5) Logged out
(User 8) Logged in
```

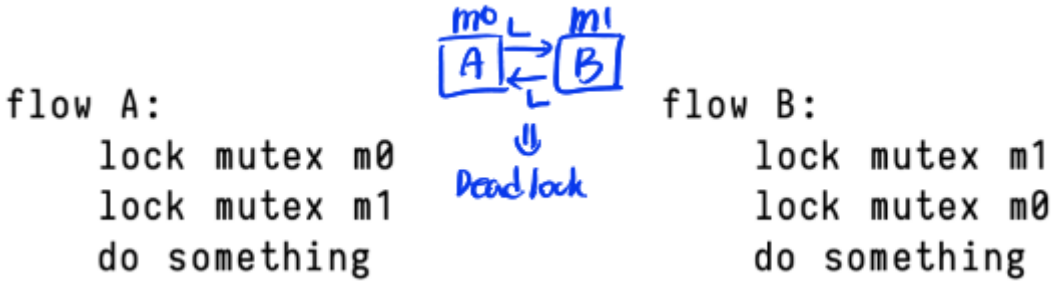
처음 세마포어 크기만큼 들어가고, 기다림
스레드가 세마포어 나갈 때마다 다른 랜덤 스레드가 세마포어에 들어감

- 생산자 - 소비자 문제
- Reader - Writer 문제
- Dining-철학자 문제

세마포어 max를 1로 설정하면 뮤텍스처럼 사용할 수 있다
-> 하지만 이러한 Binary 세마포어에는 ownership 존재 X

Deadlock Condition

1. mutex 하나 이상 -> m1 locked 건 상태에서 다른 사람이 locked한 m2에 lock 걸기



2. mutex 하나일때 owner thread abort

- 동기화시 발생
- 뮤텍스의 상호배제, 복수 lock..등에 의해 발생

Avoiding Deadlock (mutex 한정)

- ordering하기 -> 성능저하

Summary

- race condition
  여러 thread/process 상황에서 공유자원 사용/접근이 명확하지 않을 때 발생함
  동시에 여러 logical flow가 공유자원에 접근하려고 할 때 발생
- data race 공유자원을 수정할때 발생한다.
- Synchronization event ordering -> critical section 설정
- Synchronization primitives
  1. Atomic Operation - HW Support 필요
  2. Mutex
     2.1. Condition variable
  3. Semaphore
- Deadlock 동기화때문에 일어난다...