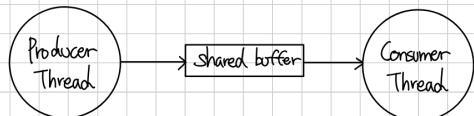


Synchronization: Advanced

Using Semaphores to Coordinate Access to Shared Resources

- classic examples
 - The Producer-Consumer Problem
 - The Readers-Writers Problem

Producer Consumer Problem



- Producer는 empty slot 대기, buffer에 item, consumer에게 알리기.
- Consumer는 item 대기, buffer에서 item, producer에게 알리기.

Producer-Consumer on n-element buffer

→ mutex & 2 counting semaphores

- mutex → buffer all at once exclusive access → binary semaphore 티어 컨트롤
- slots → buffer all slot 티어 → counting semaphore
- items → " items 티어 → counting semaphore

```
#include "csapp.h"
```

```
typedef struct {
    int *buf;      /* Buffer array */
    int n;         /* Maximum number of slots */
    int front;    /* buf[(front+1)%n] is first item */
    int rear;     /* buf[rear%n] is last item */
    sem_t mutex;   /* Protects accesses to buf */
    sem_t slots;   /* Counts available slots */
    sem_t items;   /* Counts available items */
} sbuf_t;
```

```
void sbuf_init(sbuf_t *sp, int n);
void sbuf_deinit(sbuf_t *sp);
void sbuf_insert(sbuf_t *sp, int item);
int sbuf_remove(sbuf_t *sp);
```

→ empty slot
→ first item
→ last item

```
/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```

```
void sbuf_init(sbuf_t *sp, int n)
```

```
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;          /* Buffer holds max of n items */
    sp->front = sp->rear = 0; /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1); /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n); /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0); /* Initially, buf has 0 items */
}
```

```
/* Insert item onto the rear of shared buffer sp */
```

```
void sbuf_insert(sbuf_t *sp, int item) producer
{
    P(&sp->slots);           /* Wait for available slot */
    P(&sp->mutex);           /* Lock the buffer */
    sp->buf[(++sp->rear)%sp->n] = item; /* Insert the item */
    V(&sp->mutex);           /* Unlock the buffer */
    V(&sp->items);           /* Announce available item */
}
```

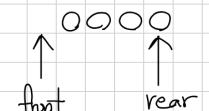
```
int sbuf_remove(sbuf_t *sp)
```

```
{
    int item;
    P(&sp->items);           /* Wait for available item */
    P(&sp->mutex);           /* Lock the buffer */
    item = sp->buf[(++sp->front)%sp->n]; /* Remove the item */
    V(&sp->mutex);           /* Unlock the buffer */
    V(&sp->slots);           /* Announce available slot */
    return item;
}
```

Producer	Consumer
P(&slots)	P(&item)
P(&mutex)	P(&mutex)
Critical Section	Critical Section
V(&mutex)	V(&mutex)
V(&item)	V(&slots)

→ empty slot
n empty slot
0 item

05/27



Readers-Writers Problem

Problem statement

- Reader's object read
- Writer's object modify
- writer's exclusive access
- reader's shared

Solution

- Online airline reservation system
- Multithreaded caching Web proxy

Variants of Readers-Writers

First readers-writers problem (favors readers)

- reader can wait.
- No reader should be kept waiting unless a writer has already been granted permission to use the object!
- A reader that arrives after a waiting writer gets priority over the writer

∴ starvation

Solution to First Readers-Writers Problem

Readers:

```
while(1) {
    P(&mutex);
    read_cnt++;
    if(read_cnt == 1)
        P(&w);
    V(&mutex);
    /* critical section */
    P(&mutex);
    read_cnt--;
    if(read_cnt == 0)
        V(&w);
    V(&mutex);
}
```

Writers:

```
while(1) {
    P(&w);
    /* critical section */
    V(&w);
}
```

- Second readers-writers problem (favors writers)
- Writer can wait.
- Once a writer is ready to write, it performs its write as soon as possible
- A reader that arrives after a writer must wait even if the writer is also waiting

```
sbuf_t sbuf; /* Shared buffer of connected descriptors */
```

```
int main(int argc, char **argv)
{
    int i, listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    sbuf_init(&sbuf, SBUFSIZE);
    for (i = 0; i < NTHREADS; i++) /* Create worker threads */
        Pthread_create(&tid, NULL, thread, NULL);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
    }
    void *thread(void *vargp)
    {
        Pthread_detach(pthread_self());
        while (1) {
            int connfd = sbuf_remove(&sbuf); /* Remove connfd from buf */
            echo_cnt(connfd); /* Service client */
            Close(connfd);
        }
    }
}
```

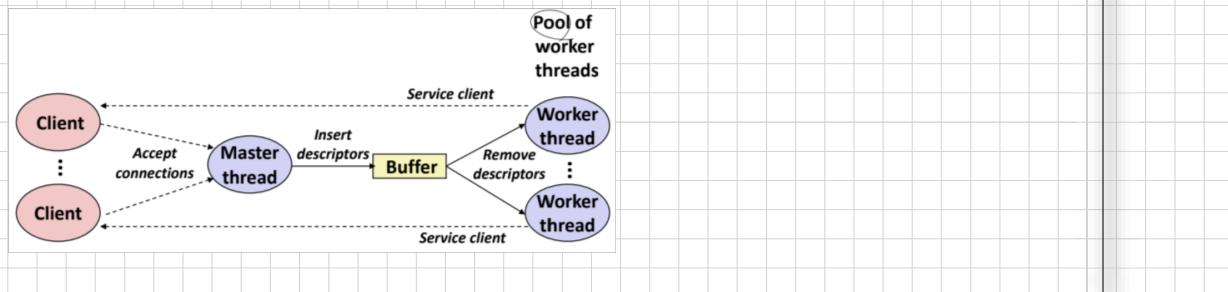
```
static int byte_cnt; /* Byte counter */
static sem_t mutex; /* and the mutex that prot
void echo_cnt(int connfd)

static void init_echo_cnt(void)
{
    Sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}
```

제일 처음
Thread at 490px도,

```
Pthread_once(&once, init_echo_cnt);
Rio_readinitb(&rio, connfd);
while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    P(&mutex);
    byte_cnt += n;
    printf("thread %d received %d (%d total) bytes on fd %d\n",
        (int) pthread_self(), n, byte_cnt, connfd);
    V(&mutex);
    Rio_writen(connfd, buf, n);
}
```

Putting It All Together: Prethreaded Concurrent Server



Crucial concept: Thread Safety

→ thread에서 호출하는 함수는 thread-safe여야 함.

↳ "Thread Safe": concurrent하게 특정 함수를 반복적으로 호출할 때 항상 correct한 결과가 나오는 함수

→ Classes of thread-unsafe functions

Class 1: Functions that do not protect shared variables

Class 2: Functions that keep state across multiple invocations

Class 3: Functions that return a pointer to a static variable

Class 4: Functions that call thread-unsafe functions

→ Class 1 : Failing to protect shared variables (공유 변수를 protect하지 X)

- 대기 P와 V 함수 사용

ex) goodt.c ⇒ binary semaphore을 사용해 mutex lock을 걸어서 concurrency issue 해결
Issue: synchronization으로 인해 code 느려짐.

→ Class 2 : State across multiple function invocations

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

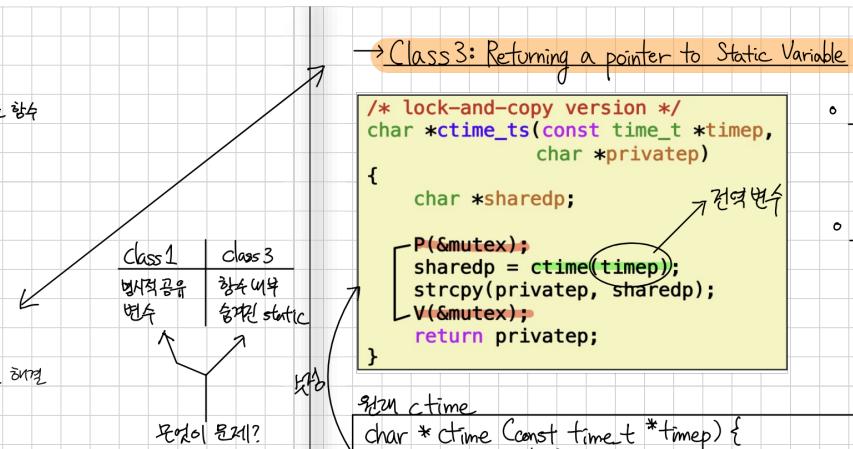
/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

```
/* rand_r - return pseudo-random integer on 0..32767 */
int rand_r(int *nextp)
{
    *nextp = *nextp * 1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

rand 함수 연속 호출 중간에, 다른 Thread가 srand() 호출 ⇒ seed(=next)가 바뀌어 버림.

해결: Eliminate global state

대신 인자값에 계속 신경써야 함.



Class 3: Returning a pointer to Static Variable

```
/* lock-and-copy version */
char *ctime_ts(const time_t *timep,
               char *privatep)
{
    char *sharedp;
    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

문제? ctime

```
char * ctime (const time_t *timep) {
    static char buf[26]; <----- 애매문제?
    strftime (buf,26,"%a %b %Y %H:%M:%S",localtime (timep));
    return buf;
}
```

Class 4: Calling thread-unsafe functions

- 함수 내부에서 Thread-Unsafe Function을 호출하는 함수

- 기존의 ctime 함수
시간을 나타내는 특정 구조체 (static variable)에
현재 시간 (pointer)을 가져와 가족
- 왜 문제?
여러 thread가 ctime 호출 가능.
우연히 않게 ctime 함수가 값을 return하기 직전에
interrupt가 걸리면 (A일 당시)
A에게 보여지는 시간은 B가 ctime을 호출한 시간이 될 것.

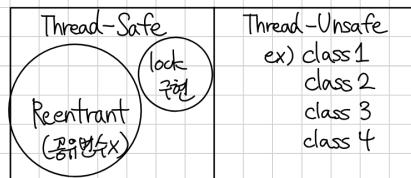
Fix 1) 함수 다시 쓰기.

caller와 callee 모두 변경 필요.

Fix 2) Lock-and-copy
caller에서 간단한 변수.
단 caller가 memory를 free해줘야 한다?

Reentrant functions

→ multiple threads에서 특정한 함수를 호출할 때,
해당 함수가 shared variable에 대한 접근을 하지 않으면 'Reentrant Function'



(Subset)

- Reentrant Function은 Thread-Safe function의 부분집합이다.
- synchronization operation은 필요로 하지 X
- Class 2 function은 Thread-safe 하지 만든 법은 오직 re-entrant하게 만드는 법뿐이다. (ex rand_r)

+ 인터넷 자료 참조 : Re-entrant의 실제 의미

→ 함수 내부에서 'Shared Variable'을 함수 결과(return 값)에 영향을 주도록 사용하면 reentrant 하지 않을 확률↑

→ Reentrant의 다른 정의

- 자신의 Thread에서 특정 함수를 호출 했을 때
호출 순서와 상관 없이, timer interrupt에 의해
Interleaving과 상관없이, 해당 함수가 마치 독립적(동시에)
흐른 것처럼 각 Thread에서 모두 correct한 결과를 나타내는 것
※ 동시에 흐른다는 뉘앙스

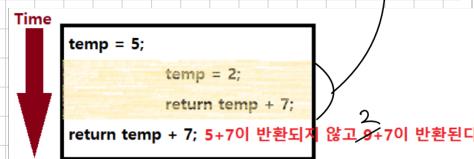
→ Thread-Safe의 정의

- 여러 Thread가 concurrent하게
특정 함수를 반복적으로 호출할 때 항상
correct 한 결과가 나오는 함수

Case 1: Thread-Unsafe, NOT Reentrant

```
int temp;
int add(int a) {
    temp = a;
    return temp + 7;
}
```

- 전역 변수인 temp가 return 직전에 다른 thread가 봐서 값이 바뀔 수 있기 때문에 Thread-Unsafe
- 전역변수를 직접 이용해서 return하기에 NOT Reentrant



+ Thread-Unsafe는 '다른 Thread'에 의해 오염되는 것

NOT Reentrant는 Signal Handler에 의해 '함수의 Result'가 오염되는 것.

Case 2: Thread-Safe, NOT Reentrant

```
thread_local int tmp;
int add(int a) {
    tmp = a;
    return tmp + 10;
}
```

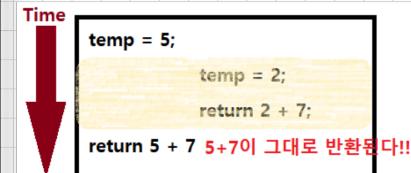
- thread_local은 nonlocal data에 대비
Lock & Copy를 시스템으로 제공한다는 의미. (synchronization 보장)
- Lock이 적용되고 있으므로 thread-safe
 - ↳ 여러 Thread끼리 중첩되지 않음
 - 함수 호출에도, tmp가锁이 제공되거나.
'다른 Thread'에 의해 오염이 일어나지 X.

→ 그러나! 전역변수 사용해서 직접 return하고 NOT Reentrant.
ex) add를 사용하는 signal handler.

Case 3: Thread-Unsafe, Reentrant

```
int temp;
int add(int a) {
    temp = a;
    return a + 7;
}
```

→ 전역변수를 사용하고 thread-unsafe
 ↳ Return 직전에 temp값이 다른 thread에 의해 바뀔 수도
→ 전역변수를 이용하지 않기로 reentrant
전역변수가 각 명령의 right side all global reentrant.



signal handler에서 add 함수 사용.
↳ result는 유지됨.

- ∴ Reentrant 판단) signal handler 중에서 오염이 발생하는지
(각 명령의 right value나 return 값에서 shared variable을 사용하는지)
- ∴ Thread-Safe 판단) 여러 Thread가 동시에 수행시 오염이 발생할 것인지.
(shared variable을 특정조차 없이 그대로 사용하는지)

Case 4: Thread-Safe, Reentrant

```
int add(int a) {
    return a + 7;
}
```

Thread-Safe Library Functions

ex) malloc, free, printf, scanf ⇒ async-safe 하지는 않다, thread-safe는 한다.

Thread-unsafe function	Class	Reentrant Version
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_aton	3	(none)
localtime	3	localtime_r
rand	2	rand_r

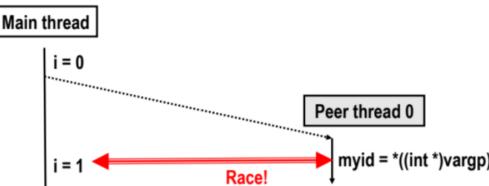
One worry: Races

Race 정의) Program의 correctness가 Thread의 경쟁 결과에 의존하는 현상

```
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;

    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```



Vargp 파라미터의 dereferencing이 main thread의 i++ 보다 늦게 일어난 것.
→ multicore 환경에서 더 많이 발생.

Solution for Race

• 의도치 않은 state 공유를 피해야 한다. → peer-thread는 이미 값이 확정된 서로 다른 메모리 영역 이용

```
/* Threaded program without the race */
int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

Another worry : Deadlock

→ 절대 true가 되지 않을 상황을 기다리는 것

- Thread 1은 A를 가지고 있고, B가 있어야 진행 가능
 - Thread 2는 B를 가지고 있고, A가 있어야 진행 가능
- ex) 이전의 Process와 signal handler 사이의 printf lock

Deadlocking With Semaphores

```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

Tid[0]	Tid[1]
P(S ₀)	P(S ₁)
P(S ₁)	P(S ₀)
cnt++	cnt++
V(S ₀)	V(S ₁)
V(S ₁)	V(S ₀)

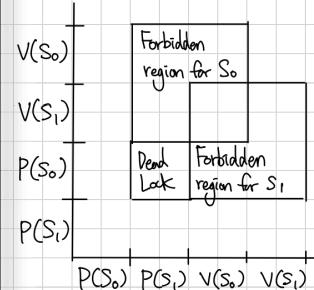
Tid[0]	Tid[1]
P(S ₀)	P(S ₀)
P(S ₁)	P(S ₁)
cnt++	cnt++
V(S ₀)	V(S ₁)
V(S ₁)	V(S ₀)

```
void *count(void *vargp)
{
    int i;
    int id = (int)vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}

↓ P(&mutex[0]); P(&mutex[1])
```

Dead Lock Visualized in Progress Graph

Thread 1



Avoid Deadlock in Process Graph

Thread 1

