chapter13. Thread Synchronization

objectives

1. Learn the basics of thread synchronization
2. Experiment with mutex locks and condition variables
3. Explore classic synchronization problems
4. Use threads with signals

Mutex

상호배제(mutual exclusion)의 약자, critical section 진입을 순서대로 허가해줌

→ 충돌 문제 : 동시에 resource에 access하여 resource를 update할 때 발생

→ 충돌 문제 발생 시 access하고 있는 다른 thread가 있다면 기다리도록 요구 사항 제시

⇒ 이 요구 사항을 만족하는 영역 = critical section

⇒ critical section마다 mutex variable 가지고 있음. → acts like lock

⇒ 해당 critical section에 대한 mutex variable에 대해 lock을 가진 thread는 접근 가능

(보통 전역 변수를 updating 할때 많이 사용됨 → 전역 변수에 대해 critical section 이어야 함)

Mutex information

- **mutex variable이 하는 일**
  1. implementing thread synchoronization
  2. protecting shared data
  3. preventing 'race' conditions(충돌 문제 발생하는 상황)
     a. ex) 다른 사이트에서 같은 계좌에 대해 입금을 수행하려는 경우
        ⇒ 여러 thread의 같은 resource에 대한 access

thread 1이 끝나고 thread 2가 시작해야 함.

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

400이 되어야함

b. ex) count ++ , count - -

```
count++ could be implemented as

    register1 = count
    register1 = register1 + 1
    count = register1
count-- could be implemented as

    register2 = count
    register2 = register2 - 1
    count = register2
Consider this execution interleaving with "count = 5" initially:
S0: producer execute register1 = count    {register1 = 5}
S1: producer execute register1 = register1 + 1    {register1 = 6}
S2: consumer execute register2 = count    {register2 = 5}
S3: consumer execute register2 = register2 - 1    {register2 = 4}
S4: producer execute count = register1    {count = 6 }
S5: consumer execute count = register2    {count = 4}
```

여러가지 경우의 수

→ 순서에 상관없이 순차적으로

⇒ 'lock'을 가진 thread는 resource에 접근 가능 / 'lock'이 없는 thread는 waiting queue에서 대기

- **the basic concept**

→ 해당 Mutex 변수에 대해 lock을 가진 thread는 오직 하나
⇒ Critical section이 여러개 필요하다면 여러 Mutex 선언

1. 오직 한 thread만 그 시점에 mutex variable을 'lock' 가능

   ↔ lock이 없는 thread : waiting queue에서 대기 / lock을 받을 수 있는 조건 만족해야함

   → 여러 thread가 lock을 시도하면 오직 한 thread만 가능

   → lock을 가진 thread가 unlock할 때까지 다른 thread는 mutex를 가질 수 없음

   ↳ 같은 mutex

2. thread : 반드시 보호된 data에만 접근하며 실행되어야 함

⇒ Mutex : "race" condition 방지 → OS : system call 함수 제공
(좋은 문제로 발생하는 상황)

- **Mutex 사용 순서**

1. create and initialize a mutex variable
2. Several threads : mutex lock 요청
3. 오직 하나만 성공하고 그 thread가 mutex 가짐
4. the owner thread : 실행 완료
5. the owner thread → unlock
6. 다른 thread가 위의 과정을 반복
7. mutex → destroyed

(다른 locking mechanism에도 해당 sequence 적용 가능)

⇒ 다른 사항들은 .. programer에게 책임 전가!!

Mutex system call function → thread synchronization

- **initialization/creation**
  - 주의 : 초기화했던 걸 다시 초기화하는 것은 define 되어 있지 x

```
#include <pthread.h>

//1. static 변수로 초기화
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//pthread_mutex_t : mutex lock에 대한 variable type, 구조체 형태로 구현
//PTHREAD_MUTEX_INITIALIZER : default 속성인 static variable로 초기화

//2. 동적 할당 변수를 이용하여 초기화
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
//pthread_mutex_init : 초기화 함수
//mutex : 초기화하고자 하는 mutex 변수
//attr : mutex attribute object
//   -> NULL : default value

//return 0 -> successful
//return nonzero error code -> unsuccessful
```

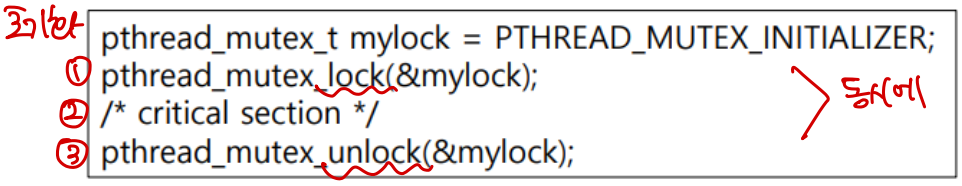
- **Destroy**
  - 주의 : destroy한 것을 사용하려고 하면 ~~error~~ → destroy하기 전에 다 사용했는지 확인 필요

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
//mutex 해제 작업을 OS에게 요청하는 함수

//mutex : 해제하고자 하는 mutex 변수

//return 0 -> successful
//return nonzero error code -> unsuccessful
```

- **Locking/unlocking → critical section 구현**

```
#include <pthread.h>

//mutex : 모두 initialize 완료된 mutex 변수만 가능

int pthread_mutex_lock(pthread_mutex_t *mutex);
//mutex에 대해 lock을 요청하는 함수 -> blocking ver. lock
//=> wating queue에 들어간 thread는 모두 block
//=> 확인하지 않고도 바로 lock 실행된다면 해당 함수 사용
```

```
② int pthread_mutex_trylock(pthread_mutex_t *mutex);
//mutex가 lock을 요청할 수 있는지 check하는 함수 -> nonblocking ver. lock
//=> lock이 다른 thread에게 할당되었는지 확인함.
//=> 항상 즉시 return

③ int pthread_mutex_unlock(pthread_mutex_t *mutex);
//mutex에 대해 unlock을 요청하는 함수

//return 0 -> successful(lock을 받음)
//return nonzero error code -> unsuccessful(lock이 이미 다른 thread에게 할당됨)
```



At-Most-Once execution

- parameter의 함수를 기껏해야 한 번 실행 혹은 아예 안 함
  - init과 같은 func() → 한 번만 호출해야 함
    → 실행 시멘틱이 At-Most-Once로 요구됨
    → but, 여러 번 호출되어야 하는 프로그램이 작성될 수 있음
    → at-most-once execution으로 하면 최초의 한 번만 실행됨

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
//pthread_once : at=most-once sematic을 보장하는 POSIX 함수
//once_control : static하게 PTHREAD_ONCE_INIT 초기화
pthread_once_t once_control = PTHREAD_ONCE_INIT;
//init_routine : pthread_once에 의해 한 번만 실행될 함수
//               void type 제약 사항 -> pthread_mutex_init은 호출이 불가함 → 어떻게?
```

- example - program 13.10 → init 함수를 이용하여 mutex 초기화

```
//printinitonce.c
#include <pthread.h>
#include <stdio.h>

static pthread_once_t initonce = PTHREAD_ONCE_INIT; → 한번만 초기화 ⇒ 다른데서 critical section으로 보호할 이유X
int var; ← mutex 보호는X

static void initialization(void) {
  var = 1;                    ↳ 초기화 수행 (간접함수 호출) → 다시 실행되지는 X
  printf("The variable was initialized to %d\n", var);
```

```
}
int printinitonce(void) { /* call initialization at most once */
  return pthread_once(&initonce, initialization);
}
```

한번만 실행

```
//printinitoncetext.c
#include <stdio.h>

int printinitonce(void);
extern int var;

int main(void){
  printinitonce();
  printf("var is %d\n", var);
  printinitonce();
  printf("var is %d\n", var);
  printinitonce();
  printf("var is %d\n", var);
  return 0;
}
```

실행 0 ⇒ message

실행x

실행x

```
ccslab@ccslab-linux:~/programs/usp_a
The variable was initialized to 1
var is 1
var is 1
var is 1
```

- example - alternative example, program 13.11 → static 변수를 활용하여 mutex 초기화
  - static : 초기화된 정적 변수 저장 공간 영역이 따로 존재 → 변경되지 x

```
//printinitmutex.c
#include <pthread.h>
#include <stdio.h>

int printinitmutex(int *var, int value) {
  static int done = 0;
  static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
  int error;
  if (error = pthread_mutex_lock(&lock))
    return error;
  if (!done) {
    *var = value;
    printf("The variable was initialized to %d\n", value);
    done = 1;
  }
  return pthread_mutex_unlock(&lock);
}
```

초기값을 alternative하게 받음

⇒ 초기화 x=0 , 초기화 완료 =1 (flag 역할)

함수가 다시 호출되더라도 첫 실행에서 지정한 값 유지 ⇒ 공유 상태

초기화 작업

critical section

```
#include <stdio.h>

int printinitmutex(int *var, int value)

static void print_once_test(int *var){
  int error;

  error = printinitmutex(var, 1);
  if(error)
    printf("Error initializing variable\n");
  else
```

```
    printf("OK\n");
}

int main(void){
  int var;

  print_once_test(&var);
  print_once_test(&var);
  print_once_text(&var);
  return 0;
}
```

변수

```
ccslab@ccslab-linux:~/programs/usp_a
The variable was initialized to 1
OK
OK
OK
```

At-Least-Once execution

- parameter의 함수를 적어도 한 번은 수행함
  - pthread_mutex_init
    → at-least-once sementic (①) + at-most-once sementic (②) 둘 다 사용! → 정확히 한 번 실행
  - 혹은 main thread가 어떤 thread를 생성하기 이전에 모두 init이 필요할 때 sementic 사용

Condition Variables

process의 condition이 true가 될 때까지 기다리는 waitng queue에 관련된 data type

→ ciritcal section 내에서의 동기화 구현

↔ mutex : critical section 구현(lock, unlock)

- mutex : condition이 만족될 때까지 기다리는데 문제가 발생할 수 있음(ex. x==y)
  1. busy waiting solution : while(x ≠ y) → 비효율적
  2. non-busy waiting solution : mutex 사용
     a. lock a mutex → test the condition
     b. if ture → unlock the mutex, exit the loop
     c. if false → suspend the thread, unlock the mutex
        ⇒ 이 사이에 다른 thread 진입하여 condition 변경 시 문제(x,y 값 변경)
        ⇒ mutex가 아닌 다른 data type 필요

⇒ mutex가 호출한 thread가 data에 접근하는 동안

☆ → condition value : thread가 data의 실제 값에 따라 synchronize하도록! 낭비 없이!

→ condition value : mutex lock과 항상 함께 사용

1. pthread_cond_wait(condition variable, mutex)
   - 호출한 thread suspend와 mutex unlock을 동시에 수행하는 func
   - thread가 notification을 받았을 때 mutex와 함께 return
   - mutex를 가진 thread에 의해서만 호출 되어야 함
2. pthread_cond signal(condition vaiables)
   - coreesponding queue에서 대기하는 thread 중 적어도 하나의 thread wake하는 func
   - thread를condition variable queue에서 mutex queue로 옮기는 효과!

```
//example
//thread1 : x==y condition wating
//m : mutex / v : condition variable

pthread_mutex_lock(&m);
while( x != y )
  pthread_cond_wait(&v, &m);
/* modify x or y if necessary */
pthread_mutex_unlock(&m);

//thread2 : 같은 변수 사용 -> x를 증가하면서 thread waiting
pthread_mutex_lock(&m);
x++;
pthread_cond_signal(&v);
pthread_mutex_unlock(&m);
```

Creating condition values

```
#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

//pthread_cond_t : condition variable type -> 항상 사용 전에 초기화 필요
//     -> PTHREAD_COND_INITIALIZER : static variable로 초기화
//     -> init 함수 호출로 초기화 ⇒동적할당 사용
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
//condition value 초기화 함수
//attr : 특성 지정
//   -> NULL : default value

//return 0 -> successful
//return nonzero -> unsuccessful

//중복 초기화 -> define X
```

Destroying condition values

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
//condition value destroy 함수

//return 0 -> successful
//return nonzero -> unsuccessful

//더 이상 사용하지 않는 condition variable인지 사용할 것
//만약 destroy한 condition variable 사용하면 block
```

Waiting condition values

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);
//정해진 시간만큼 wait or 해당 thread를 누군가 깨워주면 return


//cond : condition variable pointer
//mutex : call 이전에 thread가 얻은 mutex pointer
//abstime : cond signal이 도착하지 않으면 return할 time 지정

//return 0 -> successful
//return nonzero -> unsuccessful
//return ETIMEDOUT -> abstime 만료되어서 return된 경우

①//signal이랑 condition variable 같이 사용하는 경우 의도치 않게 wait 종료되는 경우 가능
②//POSIX : thread들이 같은 condition variable에 대한 councurrent한 wait operation을
//        다른 mutex lock으로 사용하는 경우 -> define x
```

Signaling on condition variables

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
//'cond'로 block되어 있는 모든 thread unblock
int pthread_cond_signal(pthread_cond_t *cond);
//'cond'로 block되어 있는 thread 중 하나라도 unblock

//return 0 -> successful
//return nonzero -> unsuccessful
```

Signal handling and threads

Signal delivery in threads

- process 내에 있는 모든 thread : process signal handler 공유
- thread : 각자의 signal mask 가짐

- signal delivery methods
  1. Asynchronous : unblock 되어 있는 thread에게 deliver
  2. Synchronous : 이 signal을 호출한 thread에게 deliver
  3. Directed :특정된 thread에게 deliver → pthread_kill

Directing a signal

```
#include <signal.h>
#include <pthread.h>
int pthread_kill(pthread_t thread, int sig); ⇒ 생성+전달

//'sig'의 signal number를 가진 signal 생성 -> 'thread'에게 deliver

//return 0 -> successful
//return nonzero -> unsuccessful
```

Masking signals for threads

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *restrict set, sigset_t *restrict oset)

//thread의 signal mask 설정

//set : 설정하고자 하는 signal mask
//how : 어떻게 설정?
//  1. SIG_SETMASK : 'set'으로 signal mask 설정
//  2. SIG_BLOCK : 'set'에 있는 걸 현재 signal mask에 추가
//  3. SIG_UNBLOCK : 'set'에 있는 것만 제외
//oset : NULL이 아니면 원래 signal mask값 저장

//return 0 -> successful
//return nonzero -> unsuccessful
```

Dedicating threads for signal handling

다중 thread process에서 signal를 다룰 때 어떻게?

1. main thread : 모든 signal block
2. dedicated thread 생성 ← signal마다 전담 thread 생성
3. dedicated thread
   a. 지정된 signal에 대해 sigwait() 실행
   b. signal을 unblock하기 위해 pthread_sigmask 사용 가능

- **program 13.14**
  1. signalthreadinit() → 'signo'에 대해 block 후 이 siganl을 wait할 전담 thread 생성
  2. signo가 pending → sigwait은 return → 전담 thread : setdone() 실행
  3. pending에서 이 signal을 sigwait가 없애면 더 이상 signal handler가 필요 x ⇒ target process에게 전달 X

```
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include "doneflag.h"
#include "globalerror.h"

static int signalnum = 0;

/* ARGSUSED */
static void *signalthread(void *arg) {    /* dedicated to handling signalnum */
   int error;
   sigset_t intmask;
   struct sched_param param;
   int policy;
   int sig;

   if (error = pthread_getschedparam(pthread_self(), &policy, &param)) {
      seterror(error);
      return NULL;
   }
   fprintf(stderr, "Signal thread entered with policy %d and priority %d\n",
             policy,  param.sched_priority);
   if ((sigemptyset(&intmask) == -1) ||
       (sigaddset(&intmask, signalnum) == -1) ||
       (sigwait(&intmask, &sig) == -1))
      seterror(errno);
   else
      seterror(setdone());
   return NULL;
}

int signalthreadinit(int signo) {
   int error;
   pthread_attr_t highprio;
   struct sched_param param;
   int policy;
   sigset_t set;
   pthread_t sighandid;

   signalnum = signo;                                   /* block the signal */
   if ((sigemptyset(&set) == -1) || (sigaddset(&set, signalnum) == -1) ||
       (sigprocmask(SIG_BLOCK, &set, NULL) == -1))
      return errno;
   if ( (error = pthread_attr_init(&highprio)) ||    /* with higher priority */
        (error = pthread_attr_getschedparam(&highprio, &param)) ||
        (error = pthread_attr_getschedpolicy(&highprio, &policy)) )
      return error;
   if (param.sched_priority < sched_get_priority_max(policy)) {
      param.sched_priority++;
      if (error = pthread_attr_setschedparam(&highprio, &param))
```

```
        return error;
    } else
      fprintf(stderr, "Warning, cannot increase priority of signal thread.\n");
    if (error = pthread_create(&sighandid, &highprio, signalthread, NULL))
       return error;
    return 0;
}
```

② 전담 thread 생성

실행할 함수

Readers and writers ⇒ classic한 동기화 문제

Reader-writer problem

resource가 read와 write 두 가지의 access 가능할 경우

1. writing : 독립적으로 수행되어야 함
2. reading : 공유될 수 있음

⇒ operation에 따라 lock (read만 한다면 굳이 lock X)

→ 어떻게?

1. Strong reader synchronization : reader thread에게 우선권
   → write thread가 writing하지 않고 있다면 reader에게 우선권
2. Strong writer synchronization : writer thread에게 우선권(reader가 기다림)
   → waiting하고 있는 writer thread, 혹은
   writing하고 있는 thread 모두 끝날 때까지 reader가 기다림

만약 writer가 lock을 가지고 있다면 reader가 lock을 얻을 수 있는지에 따라 달려있음.

Initialization of read-write locks

⇒ read-write lock ① 선언 ② 초기화 ③ 사용

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
//초기화 함수
//pthread_rwlock_t : a read_write lock을 나타내는 variable type
//attr : 초기화 할 때 지정할 속성

//return 0 -> successful
//return nonzero -> unsuccessful

//중복 초기화 -> define X
```

→ read/write lock type

pthread와 동일

+) 반드시 초기화하고 사용해야 함.

Destroying read-write locks

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

```
//destroy 함수
//pthread_rwlock_t -> destroy한 걸 재사용하려면 init 함수로 다시 초기화해서 사용

//return 0 -> successful
//return nonzero -> unsuccessful

//destroy된 걸 사용하면 x -> define X
```

Locking/unlocking

```
#include <pthread.h>

//1. read용
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

//2. write용
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

//3. unlock
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

//return 0 -> successful
//return nonzero -> unsuccessful
//try '' 함수가 return EBUSY하는 경우 -> 이미 lock을 가지고 있는데 요청되었을 경우

//rdlock을 가지고 있는데 wrlock을 기다리는 경우 -> deadlock
```

⇒ thread들이 read용 lock을 요청했을 때만 multi thread 허용

non-blocking mode

∵ 두 lock은 exclusive하게 할당해야 함

- **program 13.16** → 함수별로 하는지만 간단하게 정리
  - list에 접근할 때 다중 thread가 read/write lock 사용해서 access → 안전하도록 작성 (thread safe)
  - initialize_r() : rw_lock을 초기화 → pthread_once sementic(한번만 실행)

↔ **program 13.9** : mutex lock을 사용한 같은 예제 → 무조건 lock 요청이라 시간이 오래 걸림

```
#include <errno.h>
#include <pthread.h>

static pthread_rwlock_t listlock;
static int lockiniterror = 0;
static pthread_once_t lockisinitialized = PTHREAD_ONCE_INIT;

static void ilock(void) {
   lockiniterror = pthread_rwlock_init(&listlock, NULL);
}

int initialize_r(void) {    /* must be called at least once before using list */
   if (pthread_once(&lockisinitialized, ilock))
      lockiniterror = EINVAL;
   return lockiniterror;
}
```

→ 초기화 함수

```
int accessdata_r(void) {               /* get a nonnegative key if successful */
   int error;
   int errorkey = 0;
   int key;
   if (error = pthread_rwlock_wrlock(&listlock)) {  /* no write lock, give up */
      errno = error;
      return -1;
   }
   key = accessdata();
   if (key == -1) {
      errorkey = errno;
      pthread_rwlock_unlock(&listlock);
      errno = errorkey;
      return -1;
   }
   if (error = pthread_rwlock_unlock(&listlock)) {
      errno = error;
      return -1;
   }
   return key;
}

int adddata_r(data_t data) {          /* allocate a node on list to hold data */
   int error;
   if (error = pthread_rwlock_wrlock(&listlock)) { /* no writer lock, give up */
      errno = error;
      return -1;
   }
   if (adddata(data) == -1) {
      error = errno;
      pthread_rwlock_unlock(&listlock);
      errno = error;
      return -1;
   }
   if (error = pthread_rwlock_unlock(&listlock)) {
      errno = error;
      return -1;
   }
   return 0;
}

int getdata_r(int key, data_t *datap) {               /* retrieve node by key */
   int error;
   if (error = pthread_rwlock_rdlock(&listlock)) { /* no reader lock, give up */
      errno = error;
      return -1;
   }
   if (getdata(key, datap) == -1) {
      error = errno;
      pthread_rwlock_unlock(&listlock);
      errno = error;
      return -1;
   }
   if (error = pthread_rwlock_unlock(&listlock)) {
      errno = error;
      return -1;
   }
   return 0;
}
```

```
int freekey_r(int key) {                                    /* free the key */
   int error;
   if (error = pthread_rwlock_wrlock(&listlock)) {
      errno = error;
      return -1;
   }
   if (freekey(key) == -1) {
      error = errno;
      pthread_rwlock_unlock(&listlock);
      errno = error;
      return -1;
   }
   if (error = pthread_rwlock_unlock(&listlock)) {
      errno = error;
      return -1;
   }
   return 0;
}
```

A strerror_r implementation → error 출력 thread safe version

- strerror()의 problem
  1. not thread-safe한 함수 중 하나
  2. 만약 strerror()이 concurrent하게 실행 → mutex lock으로 protect 해야 함
  3. perror(), strerror() → 모두 async-signal safe하지 않음
     - mutex 사용 → strerror에 의해 사용된 static buf에 대한 concurrent access 불가
     - perror() : 같은 mutex에 의해 concurrent execution도 불가
     - 모든 signal이 mutex가 lock을 가지기 전에 block 됨

       → deadlock

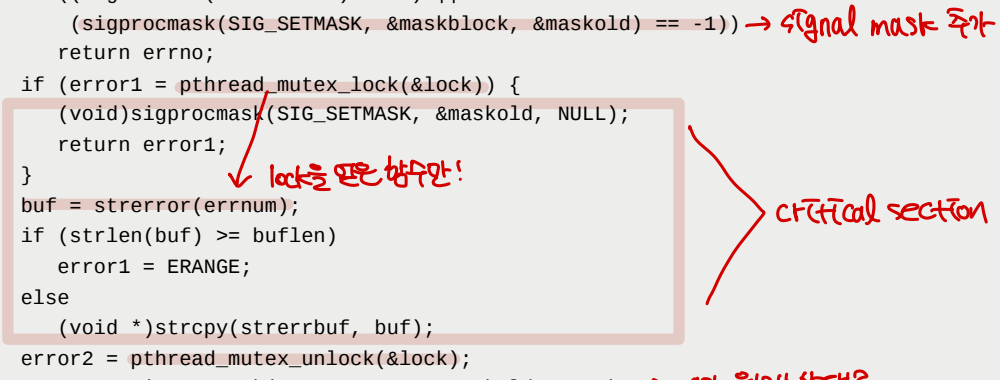
- solution(like Program 13.17)
  - perror_r(), strerror_r() → thread safe + async-signal safe

⇒ ① mutex로 strerror가 사용하는 static buf에 대해 concurrent한 access X
② perror : 같은 mutex에 대해 concurrent한 실행 막음
③ 모든 signal : mutex가 lock되기 전에 모든 signal block (signal handler 안에서 한 경우 deadlock)

- **program 13.17**

```
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>

static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int strerror_r(int errnum, char *strerrbuf, size_t buflen) {
```

```
    char *buf;
    int error1;
    int error2;
    int error3;
    sigset_t maskblock;
    sigset_t maskold;

    if ((sigfillset(&maskblock)== -1) ||
        (sigprocmask(SIG_SETMASK, &maskblock, &maskold) == -1))
       return errno;
    if (error1 = pthread_mutex_lock(&lock)) {
       (void)sigprocmask(SIG_SETMASK, &maskold, NULL);
       return error1;
    }
    buf = strerror(errnum);
    if (strlen(buf) >= buflen)
       error1 = ERANGE;
    else
       (void *)strcpy(strerrbuf, buf);
    error2 = pthread_mutex_unlock(&lock);
    error3 = sigprocmask(SIG_SETMASK, &maskold, NULL);
    return error1 ? error1 : (error2 ? error2 : error3);
}
```



Deadlocks

- Deadlocks and other pesky problems
  - synchronization의 programs : POSIX implementation에 의해 deadlock의 가능성 존재
    1. thread가 이미 mutex lock을 가지고 있는데 pthread_mutex_lock을 호출할 경우
       a. pthread_mutex_lock : fail 혹은 EDEADLK return → standard가 아님
    2. lock을 가지고 있는 thread가 error에 진입할 경우
       a. return 하기 전에 lock을 반드시 release 해야 함
    3. priority를 가지고 있는 thread
       a. 중간 우선순위의 thread가 종료되면 낮은 우선순위의 thread lock 우선순위가 높아짐
       b. 높은 우선순위의 thread는 낮은 우선순위의 thread가 unlock 해야 가능

       → 어마어마한 delay 발생