제 9 장
쓰레드 동기화

ACS30021 고급 프로그래밍

산기대학교
컴퓨터공학부
뇌리공학 연구소

학습 목표

- Process vs. Thread
- Definition of Thread
- Motivation for Threads
- Thread Usage
- Thread Synchronization
  - ✓ Mutex variable functions
  - ✓ Condition variable functions
  - ✓ Reader/writer exclusion
  - ✓ Semaphore
- Deadlock Problem

Pthread API

Process

- IEEE POSIX Section 1003.1c
  - ✓ IEEE (Institute of Electric and Electronic Engineering)
  - ✓ POSIX (Portable Operating System Interface)
  - ✓ Pthread is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel
    - Specifics are different for each implementation
    - Mach Threads and NT Threads

- Pthread of Linux is kernel level thread

The Pthreads API

- The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 – 1995 standard.
- The subroutines which comprise the Pthreads API can be informally grouped into three major classes:
  1. Thread management: The first class of functions work directly on threads – creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)
  2. Mutexes: The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
  3. Condition variables: The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Basic Pthreads API

- Pthread_create: thread를 생성
- Pthread_exit : process를 종료하지 않고, Thread만 종료
- Pthread_join : thread 수행이 종료되기를 기다림
- Pthread_kill : thread에게 signal을 보냄
- Pthread_detach : thread가 자원을 해제하도록 설정
- Pthread_equal : 두 thread ID가 동일한 지 검사
- Pthread_self : 자신의 thread ID를 얻는다
- Pthread_cancel : 다른 thread의 수행을 취소

Creating Threads

- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

  pthread_create (thread, attr, start_routine, arg)

  - ✓ Creates a new thread and makes it executable. Once created, threads are peers, and may create other threads.
  - ✓ Returns:
    - the new thread ID via the *thread* argument. This ID should be checked to ensure that the thread was successfully created.
  - ✓ *attr* :
    - is used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
  - ✓ *start_routine:*
    - is the C routine that the thread will execute once it is created.
  - ✓ arg :
    - An argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void.
- The maximum number of threads that may be created by a process is implementation dependent.

Terminating Thread Execution

- Pthread 종료 방법:
  - ✓ 모든 스레드 루틴 실행 후 종료
  - ✓ pthread_exit () 함수 호출
  - ✓ 다른 스레드가 pthread_cancel () 함수 호출
  - ✓ 프로세스 종료 – exec () or exit ()
- ■
- 스레드와 표준 출력
  - ✓ printf () 의 스레드 불안전
  - ✓ fprintf () 스레드 안전

Terminating Thread Execution

- Routines: pthread_exit (status)
  - ✓ If main() finishes before the threads it has created, and exits with pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.
  - ✓ Status: The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread.
  - ✓ Cleanup: the pthread_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
  - ✓ Recommendation: Use pthread_exit() to exit from all threads... especially main().

스레드 생성과 종료

```
#include <pthread.h>
#define NUM_THREADS 5


void *PrintHello (void *threadid)
{
      fprintf (stderr, "₩n %d: Hello World! ₩n", threadid);
      pthread_exit (NULL);
}


int main (int argc, char *argv[])
{
      pthread_t threads[NUM_THREADS];
      int rc, i;


      for (i=0; i < NUM_THREADS; i++) {
                fprintf (stderr, "Creating thread %d₩n", i);
                rc = pthread_create (&threads[i], NULL, PrintHello, (void *)i);
                if (rc) {
                   fprintf (stderr, "ERROR; return code from pthread_create() is %d₩n", rc);
                  pthread_exit (-1);
                }
      }
      pthread_exit(NULL);
}
```

Passing Arguments to Threads

- The pthread_create() permits the programmer to pass one argument to the thread start routine.
- For cases of multiple arguments:
  - ✓ creating a **structure** which contains all of the arguments,
  - ✓ then passing a pointer to that structure in the pthread_create() routine.
- All arguments must be passed by reference and cast to (void *).

Designing Threaded Programs

- In order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently.

- Splitting
  - ✓ CPU-based (e.g., computation)
  - ✓ I/O-based (e.g., read data block from a file on disk)

- Potential parallelism
  - ✓ The property that statements can be executed in any order without changing the result

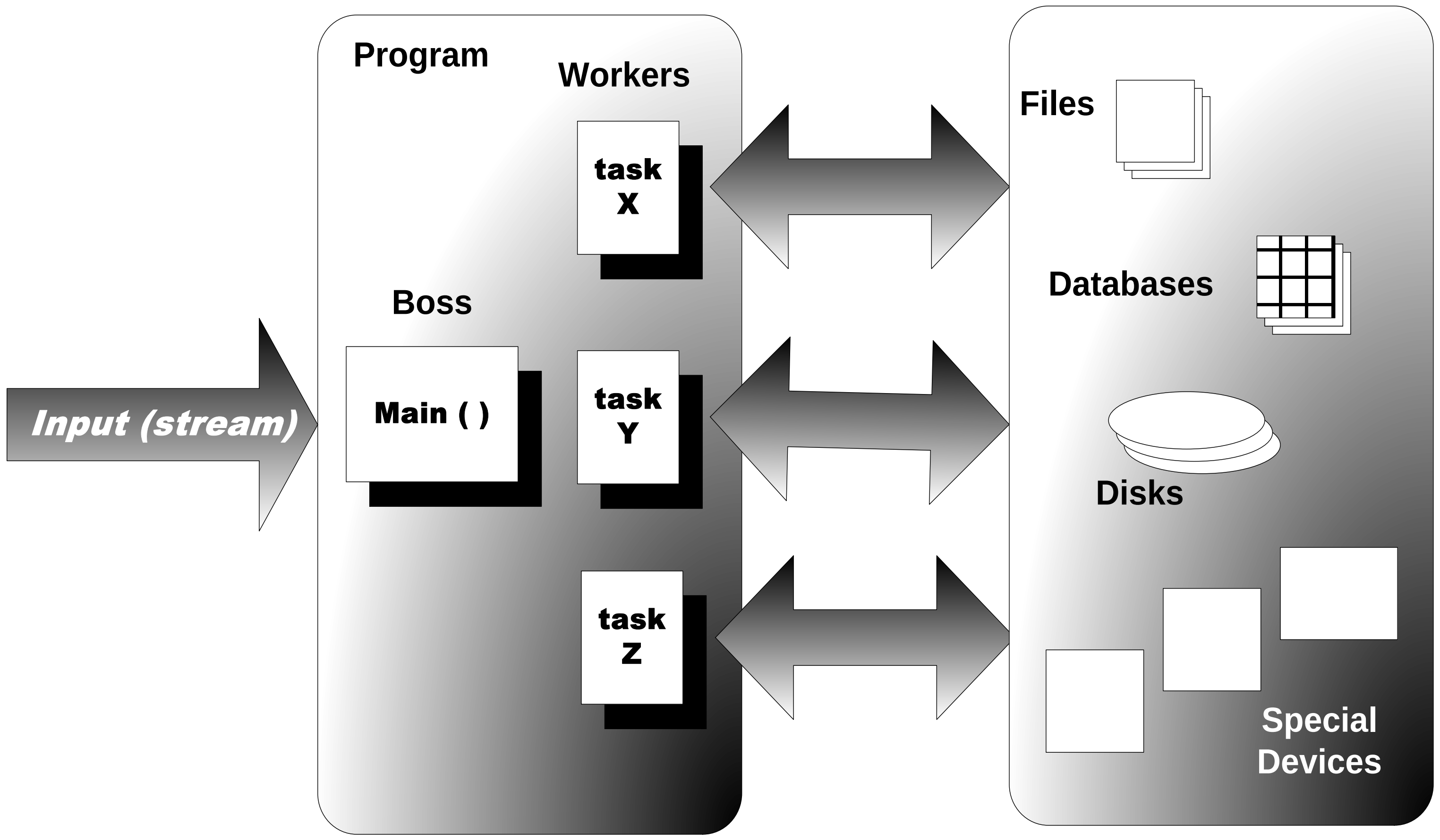
Potential Parallelism

- Reasons for exploiting potential parallelism
  - Obvious : make a program run faster on a multiprocessor
  - Additional reasons
    - Overlapping I/O
      - Parallel executing of I/O-bound and CPU-bound jobs
      - E.g., word processor -> printing & editing
    - Asynchronous events
      - If one more tasks is subject to the indeterminate occurrence of events of unknown duration and unknown frequency, it may be more efficient to allow other tasks to proceed while the task subject to asynchronous events is in some unknown state of completion.
      - E.g., network-based server

Designing Threaded Programs

- Several common models for threaded programs
  - Manager/worker:
    - a single thread, the *manager* assigns works to other threads, the *workers*. Typically, the manager handles all input and parcels out works to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.
  - Pipeline:
    - a task is broken into a series of sub-operations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line describes well this model.
  - Peer:
    - similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

Manager/Worker Model



Manager/Worker Model

- 네트워크 매니저 – video, audio, control 데이터 패킷 분류
- Example 1 (manager/worker model program)

```
main ( void )  /* the manager */
{
   forever  {
      get a request;
      switch request
         case X : pthread_create ( … taskX); break;
         case Y : pthread_create ( … taskY); break;
                  ……
  }
}

taskX ()  /* Workers processing requests of type X  */
{
   perform the task, synchronize as needed if accessing shared resources;
   done;
}
```

Manager/Worker Model

- Thread pool
  - ✓ A variant of the Manager/worker model
  - ✓ The manager could save some run-time overhead by creating all worker threads up front
  - ✓ example 2 (manager/worker model with a thread pool)

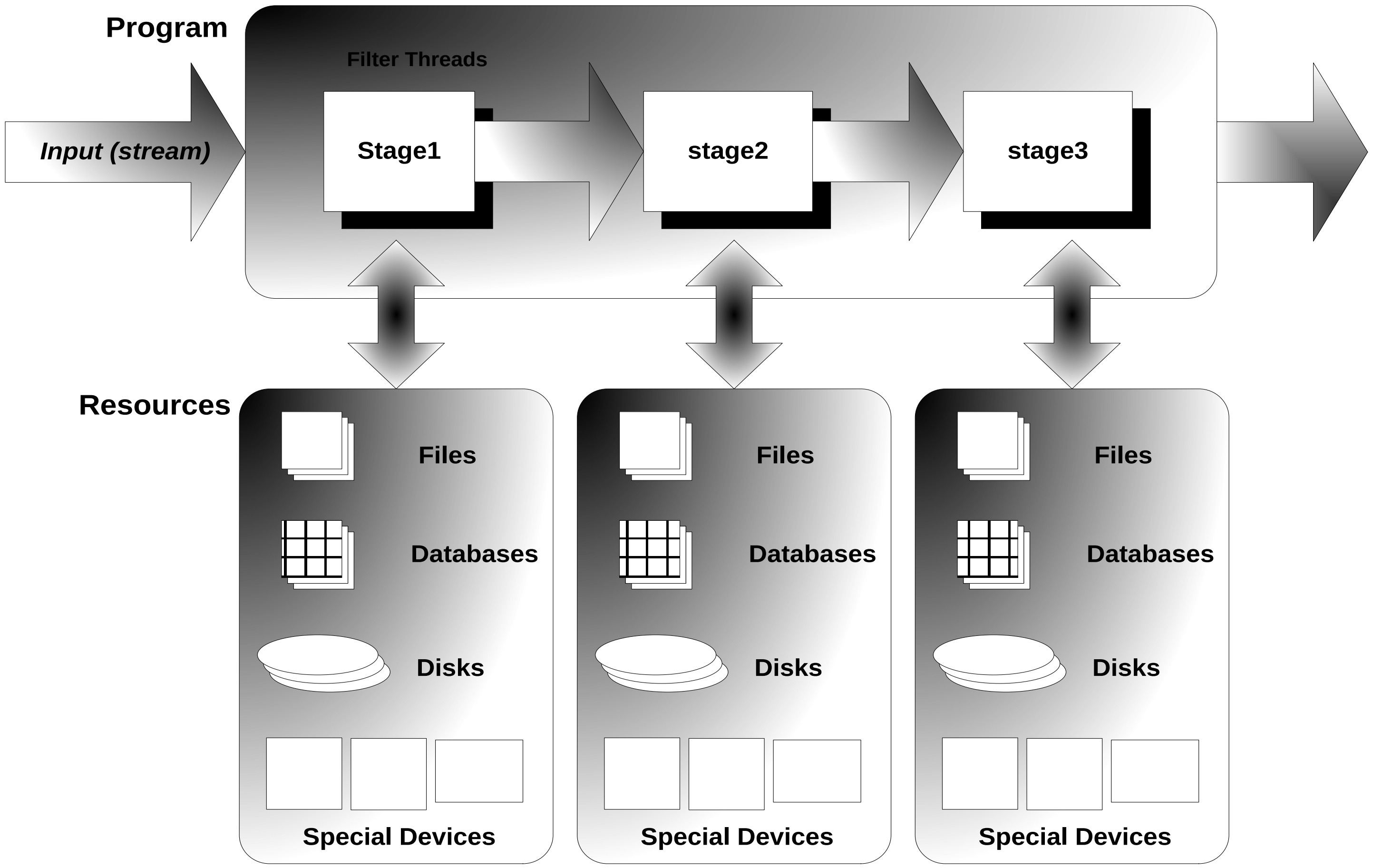
```
main(void) /* manager */
{
   for the number of workers
      pthread_create( ….pool_base );
         forever {
              get a request;
              place request in work queue;
              signal sleeping threads that work is available;
         }
}
```

Manager/Worker Model

- Example 2 (cont'd)

```
pool_base()  /* All workers */
{
   forever {
      sleep until awoken by boss;
      dequeue a work request;
      switch {
         case request X : taskX();
         case request Y : taskY();
              …
      }
   }
}
```

Pipeline Model



Pipeline Model

- 네트워크로부터 패킷 수신 쓰레드; 데이터 디코딩 쓰레드; 화면 출력 쓰레드
- Example (pipeline model program)

```
main (void)
{
        pthread_create( …stage1 );
        pthread_create( …stage2);
        …
        wait for all pipeline threads to finish;
        do any clean up;
}

Stage1 ()
{
        forever {
            get next input for the program;
            do stage1 processing of the input;
            pass result to next thread in pipeline;
        }
}
```

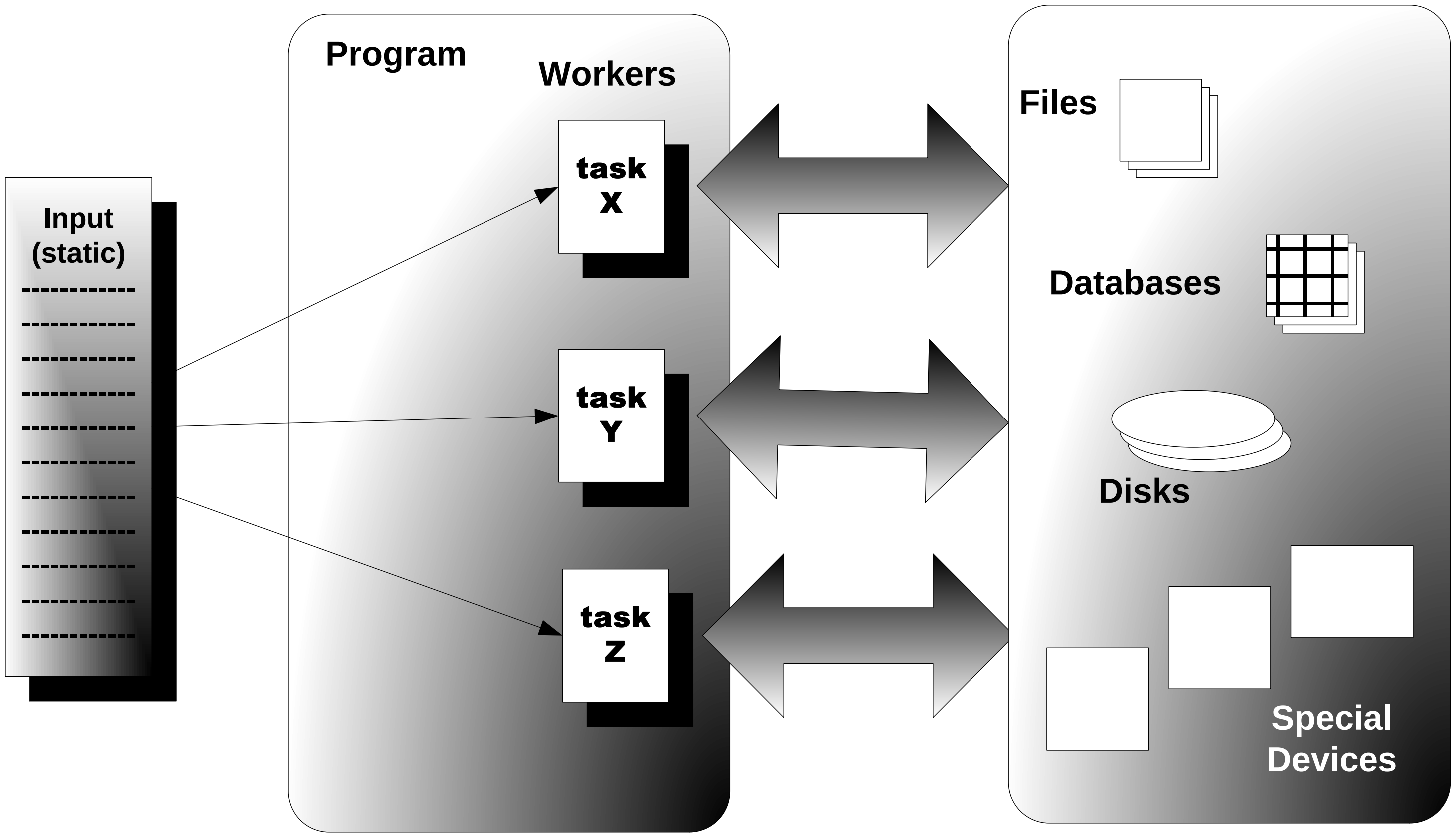
Pipeline Model

- Example (cont'd)

```
stage2 ()
{
   forever {
      get input from previous thread in pipeline;
      do stage2 processing of the input;
      pass result to next thread in pipeline;
   }
}


stageN ()
{
   forever {
      get input from previous thread in pipeline;
      do stageN processing of the input;
      pass result to program output;
   }
}
```

Peer Model


Input
(static)
Program
Workers
task X
task Y
task Z
Files
Databases
Disks
Special Devices

Peer Model

- Example (peer model program)

```
main (void)
{
   pthread_create ( …thread1 … task1 );
   pthread_create ( …thread2 … task2 );
              …
   signal all workers to start;
   wait for all workers to finish;
   do any clean up;
}

task1 ()
{
   wait for start;
   perform task, synchronize as needed if accessing shared resources;
   done;
}
```

사례: 스레드 생성과 종료

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

typedef struct net_t_tag {
....
} net_t;

void *Thread_odo_send (void *arg);

int main(int argc, char **argv)
{
  int i;
  pthread_t tid;
  pthread_attr_t attr;

  net_t tx;

  pthread_attr_init (&attr);
  pthread_attr_setscope (&attr,
      PTHREAD_SCOPE_SYSTEM);
```

```
  sleep (1);
  if(pthread_create(&tid, &attr, Thread_odo_send, &tx)) {
    fprintf(stderr, "Error: Thread_odo_send() Creation!\n");
    pthread_exit(0);
  }

   ...

  if (pthread_join (tid, NULL) != 0)
    fprintf (stderr, "ERROR: ptherad_join()");

  return 0;
}

void *Thread_odo_send (void *arg)
{
  net_t *tx = (net_t *) arg;
  int i;

  ...

  pthread_exit (0);
}
```

void형 포인터

- 자료형을 지정하지 않은 포인터 변수
- 어떤 자료형의 주소라도 저장할 수 있는 포인터 변수
- * 연산자로 값을 접근하려면 반드시 강제 형변환 필요

```
#include <stdio.h>

void main ( ) {
  char c=3;
  double d=3.1;

  void *vx=NULL;

  vx=&c;
  printf("vx가 저장한 값 : %x \n", vx);
  printf("*vx의 값 : %d \n", *(char*)vx);      // 강제 형변환

  vx=&d;
  printf("vx가 저장한 값 : %x \n", vx);
  printf("*vx의 값 : %lf \n", *(double*)vx);   // 강제 형변환
}
```

구조체 멤버 접근

```
typedef struct std_t {
   char name[32];
   int id;
} std;

std *st, *q;
void *p;

p = std;
q = (std *) p;
```

실습:

- 3개의 쓰레드 생성
  - 화면에 각각 출력
    - "한국 산업기술대학교"
    - "컴퓨터 공학과"
    - "홍길동"

실습: 쓰레드 간 데이터 전달

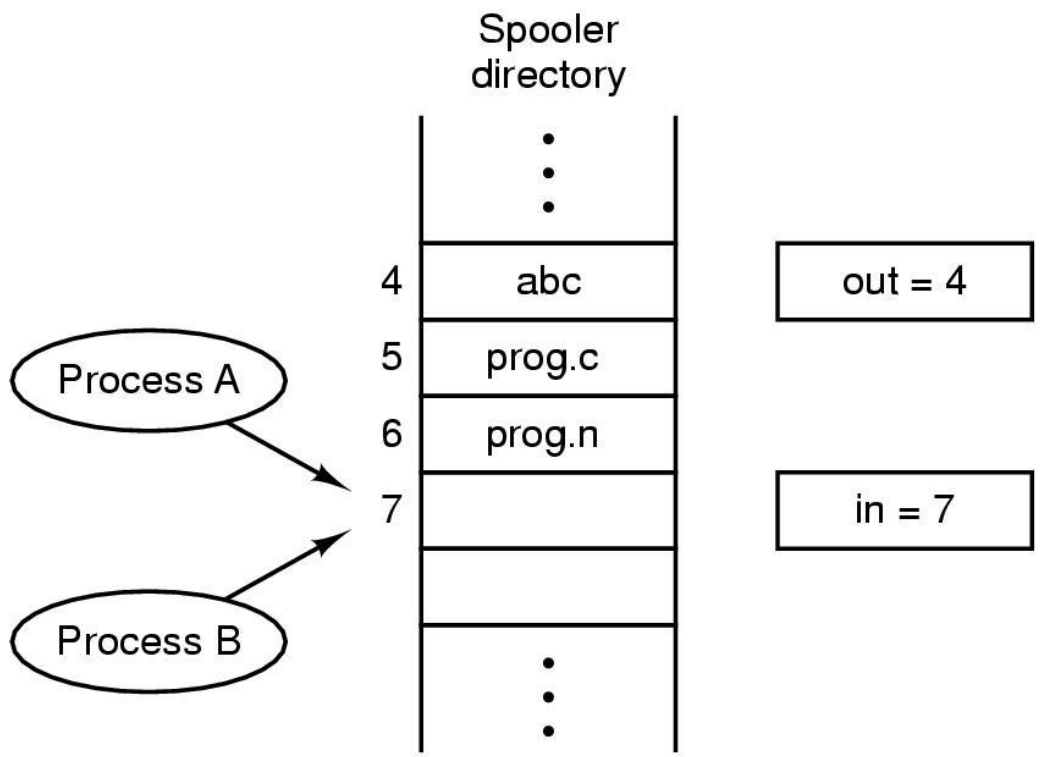
- 쓰레드 1:
  - ✓ 파일 이름 입력
  - ✓ 파일 읽기
  - ✓ 쓰레드 2에게 데이터 전달

- 쓰레드 2:
  - ✓ 전달 받은 데이터 화면 출력

Thread Synchronization

- Creating thread is the easy part. It's harder to get them to share data properly
- **Sharing global variables is dangerous** – two threads may attempt to modify the same variable at the same time.
- *Just because you don't see a problem when running your code doesn't mean it can't and won't happen!!!!*

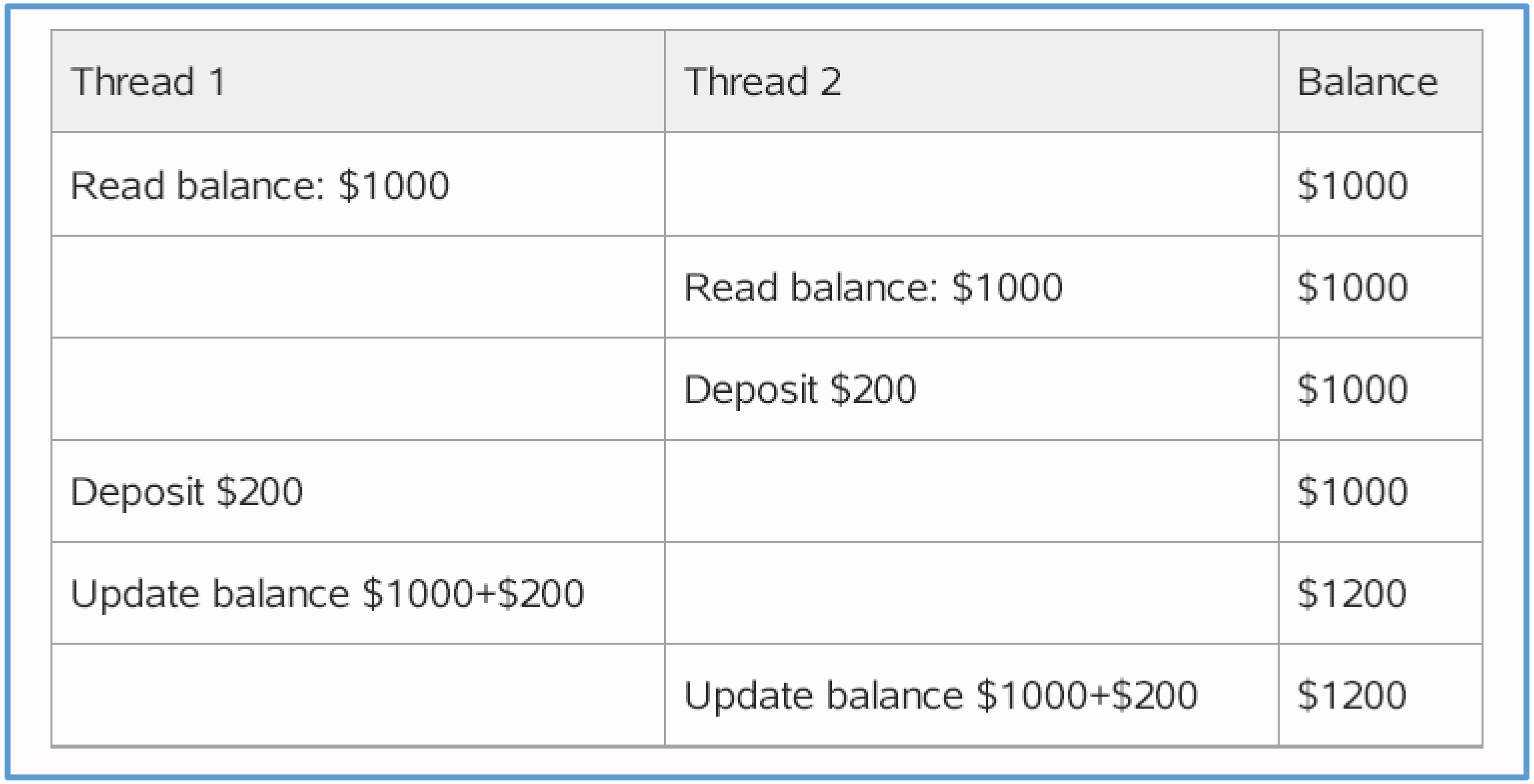
Race Conditions



- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

Race Conditions

- Two or more threads access the same resource at the same time



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

Synchronization Tool

- *pthread_join* function
  - ✓ Allows one thread to suspend execution until another has terminated
- *Mutex* variable functions
  - ✓ Mutually exclusive lock
  - ✓ Only one thread at a time can hold the lock and access the data it protects
- *Condition variable* functions
  - ✓ An event in which threads have a general interest
  - ✓ Pthread library provides ways for threads both to express their interest in a condition and to signal that an awaited condition has been met

Synchronization Tool

- Reader/writer exclusion
  - ✓ Allow multiple threads to read data concurrently but any thread writing to the data has exclusive access
- Semaphores
  - ✓ POSIX real-time extensions(POSIX.1b)

Mutex Variables

- Critical section
  - Exclusive access to the code paths or routines that access data
  - How large does a critical section have to be?
    - Even a single statement
    - Single statement is no longer atomic at the hardware level
  - Should always use mutexes to ensure that a thread's shared data operations are atomic with respect to other threads

- Mutual exclusion (mutex)
  - Exclusive access to data
    - When one thread has exclusive access to data, other threads can not simultaneously be accessing the same data

Mutex Variables

- Using mutex variables in pthread
  - ✓ Create and initialize a mutex for each resource you want to protect, like a record in a database
  - ✓ When a thread must access the resource, use pthread_mutex_lock to lock the resource's mutex.
    - ➢ Only one thread at a time can lock the mutex, and others must wait
  - ✓ When the thread is finished with the resource, unlock the mutex by calling pthread_mutex_unlock

Creating/Destroying Mutexes

- Mutex variable 은 declare 후, 반드시 초기화 필요
- Mutex variable 초기화의 두 가지 방법
  - ✓ Statically, declare 시:
    - pthread_mutex_t mymutex = THREAD_MUTEX_INITIALIZER;
  - ✓ Thread 실행 중,
    - pthread_mutex_init (mymutex, attr) 에의해,
- The mutex is initially unlocked.
- Mutex Destroy
    - int pthread_mutex_destroy (mymutex)

Using Mutex

- The pthread_mutex_lock()
  - used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked

- pthread_mutex_unlock()
  - Unlock a mutex by the owning thread.
  - An error will be returned if:
    - If the mutex was already unlocked
    - If the mutex is owned by another thread

Using Mutex

- pthread_mutex_trylock()
  - Does not suspend its caller if another thread already holds the mutex, instead, returns immediately
  - 블록되지 않고 다른 핸들러 함수 호출 가능
  - Practically, trying and backtracking may cause
    - Polling overhead
    - Starvation
  - Acceptable situation
    - Real-time programmers to poll for state changes
    - Detecting and avoiding deadlock and priority inversion

mutual exclusion의 사용

- critical section을 보호

```
. . .
pthread_mutex_lock (&lock);


// critical section


pthread_mutex_unlock (&lock);
. . .
```

- Thread-safe하지 못한 함수를 thread-safe하도록 사용 가능하게 함

```
. . .
pthread_mutex_lock (&lock);
printf (“this function is thread-safe now”);
pthread_mutex_unlock (&lock);
. . .
```

Example: Using Mutexes

- Example
  - This example program illustrates the use of mutex variables in a threads program that performs a dot product.
  - The main data is made available to all threads through a globally accessible structure.
  - Each thread works on a different part of the data.
  - The main thread waits for all the threads to complete their computations, and then it prints the resulting sum.

Example: Using Mutexes

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

typedef struct {
   double *a;     // first vector
   double *b;     // second vector
   double sum; // dot product of two vectors
   int veclen;    // dimension
} DOTDATA;

#define NUMTHRDS 4
#define VECLEN 100

DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;

void *dotprod (void *arg)
{
      int i, start, end, offset, len ;
      double mysum, *x, *y;


      offset = (int)arg;
      len = dotstr.veclen;
      start = offset*len;
      end = start + len;
      x = dotstr.a;
      y = dotstr.b;

      /* Perform the dot product */
      mysum = 0;
      for (i=start; i < end ; i++) {
         mysum += (x[i] * y[i]);
      }

     /* Lock a mutex prior to updating the value in the
      * shared structure, and unlock it upon updating.
      */
      pthread_mutex_lock (&mutexsum);
      dotstr.sum += mysum;
      pthread_mutex_unlock (&mutexsum);
      pthread_exit ((void*) 0);
}

int main (int argc, char *argv[])
{
      int i;
      double *a, *b;
      int status;
      pthread_attr_t attr;

      a = malloc(NUMTHRDS*VECLEN*sizeof(double));
      b = malloc (NUMTHRDS*VECLEN*sizeof(double));
```

Example: Using Mutexes

```
    for (i=0; i < VECLEN*NUMTHRDS; i++) { a[i]=1;  b[i]=a[i]; }

    dotstr.veclen = VECLEN;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum=0;

    pthread_mutex_init(&mutexsum, NULL);

    /* Create threads to perform the dotproduct */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);

    for(i=0; i < NUMTHRDS; i++) { pthread_create( &callThd[i], &attr, dotprod, (void *)i); }
    pthread_attr_destroy(&attr);

    /* Wait on the other threads */
    for(i=0; i<NUMTHRDS; i++) { pthread_join (callThd[i], (void **)&status); }

    /* After joining, print out the results and cleanup */
    printf ("Sum = %f ₩n", dotstr.sum);
    free (a);
    free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

실습:

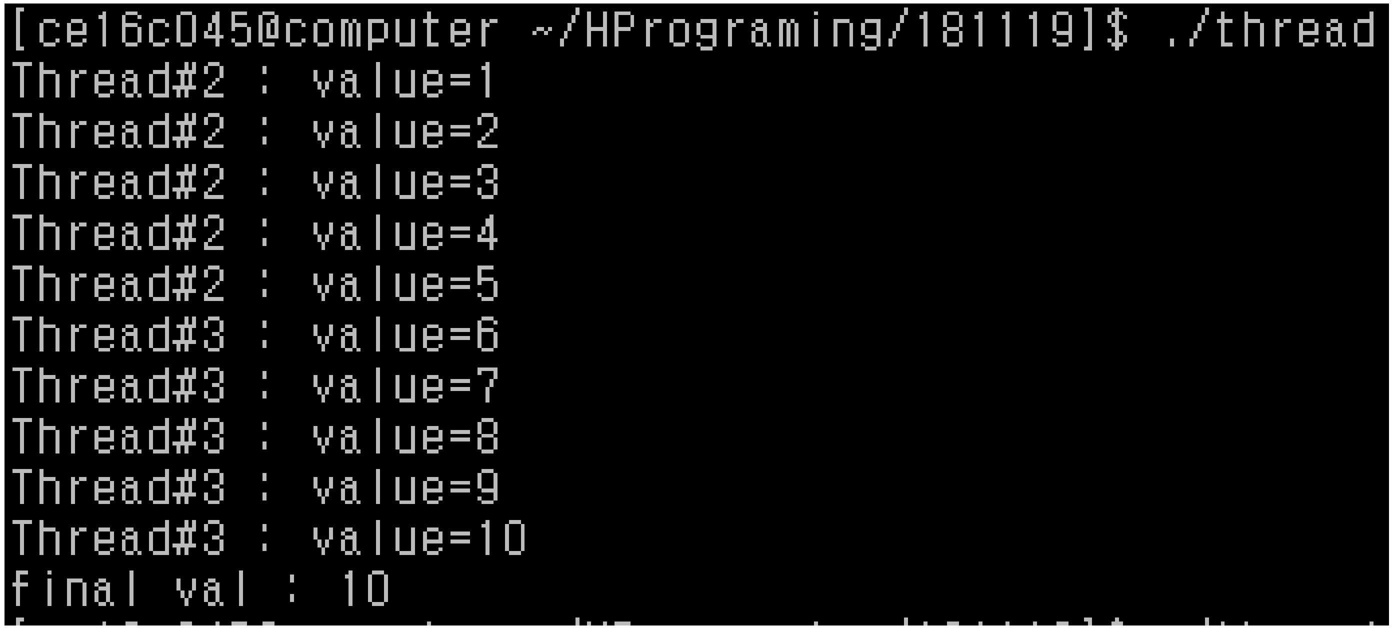
```
int main ()
{
    int critical_var=0;
```

- 각 쓰레드에서
  - ✓ 변수 critical_var ++
  - ✓ 표준 출력
    - fprintf (stderr, “Thread#%d: value=%d₩n”, pthread_self (), critical_var);
  - ✓ pthread_mutex_lock () 를 이용한 동기화

- 각각 쓰레드의 순서가 항상 일정하게 실행되는가?

실습 결과:

- 쓰레드 스케줄링 시 각 쓰레드에게 주어진 시간은?
- 한 쓰레드에서 critical_val++ 와 fprintf () 실행에 걸린 시간은?



```
[ce16c045@computer ~/HPrograming/181119]$ ./thread
Thread#2 : value=1
Thread#2 : value=2
Thread#2 : value=3
Thread#2 : value=4
Thread#2 : value=5
Thread#3 : value=6
Thread#3 : value=7
Thread#3 : value=8
Thread#3 : value=9
Thread#3 : value=10
final val : 10
```

Condition Variables

- Thread 간의 synchronization 을 위해 사용
- 어떤 기다리는 조건이 만족되었음을 다른 thread 에게 알림으로써 동기화
- Condition variable은 항상 mutex lock과 연계해서 사용

Creating/Destroying Condition Variables

- Condition variable 은
  1. pthread_cond_t type 으로 선언
  2. 사용 전에 초기화 필수.

- 초기화의 방법:
  - ✓ Static 한 방법:
    - ➢ pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
  - ✓ 동적인 방법
    - ➢ pthread_cond_init (condition, attr);

- Condition Variable Destroy
    - ➢ pthread_cond_destroy(condition)

Waiting/Signalling on Conditional Variables

- Routines:
  - ✓ pthread_cond_wait (condition,mutex)
  - ✓ Pthread_cond_timedwait(condition,mutex)
  - ✓ pthread_cond_signal (condition)
  - ✓ pthread_cond_broadcast (condition)

- pthread_cond_wait()
  - ✓ specified condition이 signal 될 때까지 block되는 함수
  - ✓ 반드시, mutex 가 locked된 상태에서 호출되어야 하며, 호출되면 자동적으로 mutex 를 unlock하고 대기한다.
  - ✓ Signal에 의해 깨어날 때에는 mutex 는 다시 lock이 걸린 상태

Waiting/Signalling on Conditional Variables

- pthread_cond_signal()
  - ✓ pthread_cond_wait() 로 대기하는 thread를 wakeup하는 함수
  - ✓ mutex 가 lock된 상태에서 호출되어야 한다.
- pthread_cond_broadcast()
  - ✓ Condition에서 대기하는 **여러** thread 모두를 깨울 때 사용

- pthread_cond_wait()에 의한 대기 thread가 없는 상태에서의 pthread_cond_signal() 호출은 no action
  - ✓ cond_signal은 stack되지 않는다.

Conditional Variables

- When many threads are waiting
  - If multiple threads are waiting on a condition variable, who gets awakened first when another thread issues a pthread_cond_signal call?
    - Scheduling priority
    - First-in first-out order
  - 모두에게 신호를 전달하고 싶으면?
    - pthread_cond_broadcast ()

Conditional Variables

- 2개의 쓰레드간 동기화
  - ✓ 한 개의 쓰레드가 다른 하나의 쓰레드에게 동기화 전달
  - ✓ 생산자/소비자 동기화 문제

- 3개 또는 그 이상 쓰레드간 동기화
  - ✓ 한 개의 쓰레드가 여러 개의 쓰레드에게 동기화 전달
  - ✓ 조건 집단 동기화 문제

Example: Using Conditional Variables

- Example:
  - This simple example code demonstrates the use of several Pthread condition variable routines.
  - The main routine creates three threads. Two of the threads perform work and update a "count" variable.
  - The third thread waits until the count variable reaches a specified value.

Example: Using Conditional Variables

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 3
#define TCOUNT 10
#define COUNT_LIMIT 12

int count = 0;
int thread_ids[3] = {0,1,2};

pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp) {
   int i, j;
   double result=0.0;
   int *my_id = idp;
   for (i=0; i < TCOUNT; i++) {
      pthread_mutex_lock (&count_mutex);
      count++;
      /*
         Check the value of count and signal
         waiting thread when condition is
         reached. Note that this occurs while
         mutex is locked.
       */
      if (count == COUNT_LIMIT) {
         pthread_cond_signal
      (&count_threshold_cv);
         fprintf(stderr, "inc_count(): thread %d, count = %d
         Threshold reached.₩n", *my_id, count);
      }
      fprintf (stderr, "inc_count(): thread %d, count = %d,
               unlocking mutex₩n", *my_id, count);
      pthread_mutex_unlock (&count_mutex);
      /* Do some for threads to alternate on mutex lock */
      for (j=0; j<1000; j++)  result += (double)random();
   }
   pthread_exit (NULL);
}
void *watch_count(void *idp) {
   int *my_id = idp;
   fprintf (stderr, "Starting watch_count(): thread %d₩n",
      *my_id);
   pthread_mutex_lock (&count_mutex);
   while (count < COUNT_LIMIT) {
      pthread_cond_wait (&count_threshold_cv,
      &count_mutex);
      fprintf(stderr, "watch_count(): thread %d Condition₩
 signal received.₩n", *my_id);
   }
   pthread_mutex_unlock (&count_mutex);
   pthread_exit (NULL);
}
```

Example: Using Conditional Variables

```
int main (int argc, char *argv[])
{
      int i, rc;
      pthread_t threads[3];
      pthread_attr_t attr; // Initialize mutex and condition variable objects
      pthread_mutex_init (&count_mutex, NULL);
      pthread_cond_init (&count_threshold_cv, NULL);

      /* For portability, explicitly create threads in a joinable state so that they can be joined later. */
      pthread_attr_init (&attr);
      pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE);
      pthread_create (&threads[0], &attr, inc_count, (void *) &thread_ids[0]);
      pthread_create (&threads[1], &attr, inc_count, (void *) &thread_ids[1]);
      pthread_create (&threads[2], &attr, watch_count, (void*) &thread_ids[2]);

      for (i = 0; i < NUM_THREADS; i++) pthread_join (threads[i], NULL);
      fprintf (stderr, "Main(): Waited on %d threads. Done.₩n",  NUM_THREADS);
      /* Clean up and exit */
      pthread_attr_destroy (&attr);
      pthread_mutex_destroy (&count_mutex);
      pthread_cond_destroy (&count_threshold_cv);
      pthread_exit (NULL);
}
```

생산자/소비자 동기화 예

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

typedef struct param_t_tag {
  int x;
  int y;
} param_t;

pthread_mutex_t cons_lock =
   PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cons_cond =
   PTHREAD_COND_INITIALIZER;

void *Thread_cons (void *arg);

int main (int argc, char **argv)
{
  int i;
  pthread_t tid;
  pthread_attr_t attr;
  param_t param;

  param.x = 0;
  param.y = 0;

  pthread_attr_init (&attr);
  pthread_attr_setscope (&attr, PTHREAD_SCOPE_SYSTEM);
  sleep (1);
  if(pthread_create(&tid, &attr, Thread_cons, &param)) {
    fprintf(stderr, "Error: Thread_cons () Creation!₩n");
    pthread_exit(0);
  }
  while (1) {
    pthread_mutex_lock (&cons_lock);
    param.y = 5;
    param.x = (param.x ++) % 3;
    pthread_mutex_unlock (&cons_lock);
    pthread_cond_signal (&cons_cond);
  }
  return 0;
}
```

생산자/소비자 동기화 예

```
void *Thread_cons (void *arg)
{
  param_t *param = (param_t *) arg;
  int i;

  while (1) {
    pthread_mutex_lock (&cons_lock);
    while (param->x != 1)
       pthread_cond_wait (&cons_cond, &cons_lock);


    pthread_mutex_unlock (&cons_lock);
  }
  pthread_exit (0);
}
```

과제:

- 3개의 쓰레드를 생성
  - ✓ thread_1 쓰레드
    - "한국산업기술대학교"를 출력
  - ✓ thread_2
    - "컴퓨터 공학부" 출력
  - ✓ thread_3
    - "홍길동"을 출력
- 항상 다음과 같이 출력되도록 동기화 하라.

```
한국산업기술대학교

컴퓨터 공학과

홍길동
```

```
void *Thread_one (void *arg) {
  int *myTurn = (param_t *) arg;

  for (;;) {
    pthread_mutex_lock (&cons_lock);
    while (myTurn != 1) pthread_cond_wait (&cons_cond, &cons_lock);
    . . .
    myTurn = 2;
    pthread_mutex_unlock (&cons_lock);
    pthread_cond_broadcast (&cons_cond);
  }
  pthread_exit (0);
}
```

Reader/Writer Lock

- Reasons and rules
  - ✓ If a thread tries to get a **read lock** on a resource, it will succeed if none of threads hold a lock on the resource or if all of them that hold a lock are readers.
  - ✓ If a thread holds a **write lock** on the resource, the would-be reader must wait.
  - ✓ Conversely, if a thread tries to get a **write lock** on the resource, it must wait to get if any other thread holds a read or write lock

Reader/Writer Lock

- Define reader/writer variable of type pthread_rdwr_t
  - ✓ pthread_rdwr_init_np : initialize reader/writer lock
    - pthread_rwlock_init
  - ✓ Pthread_rdwr_rlock_np : obtain read lock
    - pthread_rwlock_rdlock, pthread_rwlock_tryrdlock, pthread_rwlock_timedrdlock
  - ✓ pthread_rdwr_wlock_np : obtain write lock
    - pthread_rwlock_wrlock, pthread_rwlock_trywrlock, pthread_rwlock_timedwrlock
  - ✓ pthread_rdwr_runlock_np : release read lock
    - pthread_rwlock_unlock
  - ✓ pthread_rdwr_wunlock_np : release write lock
    - pthread_rwlock_unlock
  - ✓ pthread_rwlock_destroy: destroy reader/writer lock

스레드(Thread) 동기화 - RW Lock

- Reader-Writer Lock(이후 RW lock)은 Mutex와 유사하나, **더 병렬화** 제공
  - ✓ Mutex는 두 가지 상태(lock과 unlock)만을 가지면서 동시에 한 스레드만 락을 걸 수 있지만, RW lock은 세 가지 상태(read-lock, write-lock, unlock) 가능
- write-lock은 동시에 하나의 스레드만 락을 걸 수 있지만, read-lock은 동시에 여러 스레드에서 락 가능
  - ✓ write-lock이 걸린 상태에서 락을 시도하는 스레드들은 락이 풀릴때까지 모두 블록
- read-lock이 걸린 상태에서 read-lock을 시도하는 스레드들은 락을 걸 수 있지만, write-lock을 시도하는 스레드는 락을 걸고 있는 모든 스레드가 락을 풀 때까지 블록
  - ✓ 일반적으로 write-lock을 걸기위해 블록되어 있는 스레드가 있다면, 이후에 read-lock을 시도하는 스레드는 블록됩니다. 그렇지 않으면 write-lock을 걸기위해 대기하고 있는 스레드는 영원히 블록되어 있을 수도 있기 때문
  - ✓ write-lock이 걸린 상태에서 데이터에 대한 접근은 하나의 스레드만이 가능, read-lock이 걸린 상태에서는 여러 스레드가 데이터에 접근이 가능
  - ✓ RW lock은 락에 의해 보호되는 데이터가 **변경(modify)보다는 읽기(read)가 자주** 일어나는 경우에 효과적

스레드(Thread) 동기화 - RW Lock

```
int pthread_rwlock_init (pthread_rwlock_t *restrict rwlock, const
pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
```

RW lock은 사용하기 전에 pthread_rwlock_init로 초기화하고, 더 이상 사용하지 않게 되면 pthread_rwlock_destroy로 삭제

pthread_rwlock_init의 attr은 RW lock의 attribute를 지정하기 위한 것인데, NULL을 사용하면 default attribute로 초기화

스레드(Thread) 동기화 - RW Lock

- RW lock에서 락을 걸고 풀기 위해 제공되는 함수

```
int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

- pthread_rwlock_rdlock은 read-lock을 걸고, pthread_rwlock_wrlock은 write-lock을 설정
- read-lock이든 write-lock이든 락 해제는 pthread_rwlock_unlock을 사용
- 동시에 걸 수 있는 read-lock의 수는 제한적
  - ✓ 따라서, pthread_rwlock_rdlock를 사용할 때는 리턴값을 체크
- pthread_rwlock_rdlock()는 심지어 간단한 deadlock 체크(같은 스레드에서 같은 락을 두번 연속 거는 상황)까지도 수행하여 실패 코드를 반환

표준 유닉스(Single UNIX)에서 위의 락을 수행하는 함수들의 조건부 수행 버전을 정의

```
int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);
```

- 락을 시도하여 성공하면 0을 반환하고,
- 이미 다른 락이 걸려있는 상태라면 EBUSY를 리턴,
- 따라서, 구조적으로 deadlock을 피하기 어려운 상황에서 이들을 이용 가능

Reader/Writer Lock

```
int llist_init(llist_t *llistp)
{
    int rtn;
    llistp->first=NULL;
    if ((rtn = pthread_rdwr_init_np (&(llistp->rwlock), NULL))!=0) {
        fprintf (stderr, "pthread_rdwr_init_np error %d", rtn);
        exit(1);
    }
    return 0;
}


int llist_insert_data(int index, void* datap, llist_t *llistp)
{
    llist_node_t *cur, *prev, *new;
    int found = FALSE;
    pthread_rdwr_wlock_np (&(llistp->rwlock));
```

Reader/Writer Lock

```
    for (cur=prev=llistp->first;cur!=NULL;prev=cur,cur=cur->nextp) {
        if (cur->index == index) {
                free (cur->datap); cur->datap=datap;
                found=TRUE; break;
        } else if (cur->index>index) {
                break;
        }
    }
    if (!found) {
    new = (llist_node_t *)malloc(sizeof(llist_node_t));
    new->index=index; new->datap=datap; new->nextp=cur;
    if (cur==llistp->first)
        llistp->first=new;
    else
        prev->nextp= new;
    }
    pthread_rdwr_wunlock_np (&(llistp->rwlock));
    return 0;
}
```

Reader/Writer Lock

```
int llist_find_data(int index, void **datapp, llist_t *llistp)
{
    llist_node_t *cur, *prev;
    /* Initialize to "not found" */
    *datapp = NULL;
    pthread_rdwr_rlock_np (&(llistp->rwlock));
    /* Look through index for our entry */
    for (cur=prev=llistp->first;cur!=NULL;prev=cur, cur=cur->nestp) {
            if (cur->index == index) {
                *datapp=cur->datap; break;
            } else if (cur->index>index) {
                break;
            }
    }
    pthread_rdwr_runlock_np (&(llistp->rwlock));
    return 0;
}
```

Semaphore

- Review of mutex
  - most popular primitives
  - easy to use
    - easy to understand what it is
  - prone to errors
    - programmers forget to unlock
    - what if another thread forgets to use lock
    - very difficult to understand programs that contain it

Semaphore

- Why semaphore
  - mutex may result in busy-waiting
  - mutex is only for "mutual exclusion" – no sharing
  - no guarantee of fairness
- semaphore
  - a shared variable with two attributes
    - integer value: number of threads that can share this semaphore
      - allows n threads to share
    - thread list: list of threads waiting for this semaphore
      - guarantees FIFO order
- operations
  - cc [flag …] file … -lposix4 [library …] /* lib for real time extension */

```
#include <semaphore.h>

int sem_init (sem_t *sem, int pshared, unsigned int value );
```

  - pshared: if non-zero, it is shared between processes
    - i.e., zero means that it will be used between threads

Semaphore

int sem_wait (sem_t *sem);
int sem_trywait (sem_t *sem);

if the integer value > 0, decrement it and proceeds

else block (or fail for trywait)

int sem_post (sem_t *sem);

- ✓ if there is a thread waiting,
  - ➢ wake up a thread according to its schedparam
    - ▪ ptread_attr_setschedparm();
- ✓ else increment the integer value

int sem_destroy (sem_t *sem);

- ✓ other combination
  - ➢ sem_open (), sem_close ()

Semaphore

```
void producer_function(void)
 {
   while(1)    {
       pthread_mutex_lock (&mutex);
       if ( buffer_has_item == 0 )   {
            buffer =
             make_new_item ();
            buffer_has_item = 1;
       }
       pthread_mutex_unlock (&mutex );
       pthread_delay_np (&delay);
   }
 }
```

```
void consumer_function(void)
  {
    while(1)  {
         pthread_mutex_lock (&mutex);
         if (buffer_has_item == 1)  {
              consume_item (buffer);
              buffer_has_item = 0;
         }
         pthread_mutex_unlock (&mutex);
         pthread_delay_np (&delay);
    }
  }
```

Semaphore

```
void producer_function(void)
 {
   while(1)   {
        semaphore_down( &writers_turn );
        buffer = make_new_item();
        semaphore_up( &readers_turn );
   }
 }
```

```
 void consumer_function(void)
 {
   while(1)  {
        semaphore_down( &readers_turn );
        consume_item( buffer );
        semaphore_up( &writers_turn );
   }
 }
```

Memory lock

- 리눅스 시스템에서 free memory 영역 중 프로그래머가 원하는 만큼 할당하고 할당한 부분을 락킹하는(페이징이 일어나지 않도록 하는) 기법
- mlock() 함수를 이용
- ! 단, 아래 소스에는 프로그래머가 원하는 만큼 메모리를 할당하는 부분은 생략되어있다.

즉, addr 변수는 char *addr = malloc(bytes_size) 형식으로 미리 받아서 넘겨주어야 한다.

```
// Example 4-1:  Aligning and Locking a Memory Segment

#include <unistd.h>  /* Support all standards */
#include <sys/mman.h> /* Memory locking functions */

#define DATA_SIZE 2048


 lock_memory (char *addr, size_t size)  {
   unsigned long page_offset, page_size;

   page_size = sysconf(_SC_PAGE_SIZE);
   page_offset = (unsigned long) addr % page_size;
   addr -= page_offset;    /* Adjust addr to page boundary */
   size += page_offset;   /* Adjust size with page_offset */

   return ( mlock(addr, size) ); /* Lock the memory */
}

unlock_memory (char *addr, size_t size)  {
   unsigned long page_offset, page_size;

   page_size = sysconf(_SC_PAGE_SIZE);
   page_offset = (unsigned long) addr % page_size;
   addr -= page_offset; /* Adjust addr to page boundary */
   size += page_offset; /* Adjust size with page_offset */

   return ( munlock(addr, size) ); /* Unlock the memory */
}

int main()  {
   char data[DATA_SIZE];

   if ( lock_memory(data, DATA_SIZE) == -1 )
      perror("lock_memory");

     /* Do work here */

   if ( unlock_memory(data, DATA_SIZE) == -1 ) perror("unlock_memory");
}
```

Memory Locking

- Linux는 demand paging(필요할 때 swap in 하고, 안 쓰면 디스크로 다시 swap out)
  - ✓ 실제 가지고 있는 물리적 메모리 공간 보다 더 큰 공간을 가상으로 제공
  - ✓ 하지만, 경우에 따라서 자신의 메모리가 디스크로 swap out 되면 안 되는 경우도 있다.

- **Determinism**
  - ✓ 특정 메모리에 올라가 있는 루틴이 일정 시간 내에 반드시 종료
  - ✓ 디스크에 swap out 되어 있으면 swap in 하는데 걸리는 시간이 상당하므로 정해진 시간에 미 응답 발생 가능

- **Security**
  - ✓ 암호화된 비밀 정보를 디스크에서 읽어 복호화 한 다음 메모리에 가지고 있었는데, 이 영역이 swap out되면 디스크에 복호화된 비밀정보가 남아 보안상의 취약점
  - ✓ 메모리의 일정 부분은 swap out 되지 않도록 잠글 수 있도록 사용되는 함수가 mlock()
  - ✓
  - ✓ `#include <sys/mman.h>`
  - ✓ `int mlock (const void *addr, size_t len);`

Memory Locking

- 특정 프로세스의 모든 부분을 locking 하고 싶은 경우에는 mlockall()을 사용

```
#include <sys/mman.h>
int mlockall (int flags);
```

  - ✓ mlockall()에서 사용되는 flag으로는 아래 두가지가 있는데 일반적으로 두개를 다 OR해서 사용
    - **MCL_CURRENT** 현재 프로세스가 사용하는 모든 메모리를 lock
    - **MCL_FUTURE** 앞으로 사용할 메모리까지도 lock
- lock된 메모리의 해제
  - ✓ munlock(), munlockall(void)
  - ✓ CAP_IPC_LOCK이 설정된 경우에는 무한정으로 메모리를 잠글
  - ✓ 그 외는 RLIMIT_MEMLOCK 에 설정된 바이트 까지만 locking 가능 (default로 32KB)
- 특정 페이지가 디스크에 있는지, 메모리에 있는지를 확인하는 함수

```
#include <sys/mman.h>
#include <unistd.h>
int mincore (void *start, size_t length, unsigned char *vec);
```

  - ✓ bit vector 형식으로 페이지의 메모리/디스크 상주 여부를 리턴
  - ✓ MAP_SHARED 옵션으로 mmap 맵핑된(실제 파일에) 영역에 대해서만 정상 동작

Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - ✓ System has 2 tape drives.
  - ✓ P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - ✓ semaphores *A* and *B*, initialized to 1

```
        P0                  P1

    wait (A);           wait(B)

    wait (B);           wait(A)
```

핵심 정리

Q & A