객체지향프로그래밍과 자료구조

9. C++11 환경의 다중 스레드 (Multi-thread)와 mutex

교수 김 영 탁

영남대학교 기계IT대학 정보통신공학과

(Tel : +82-53-810-2497; E-mail : ytkim@yu.ac.kr)

Outline

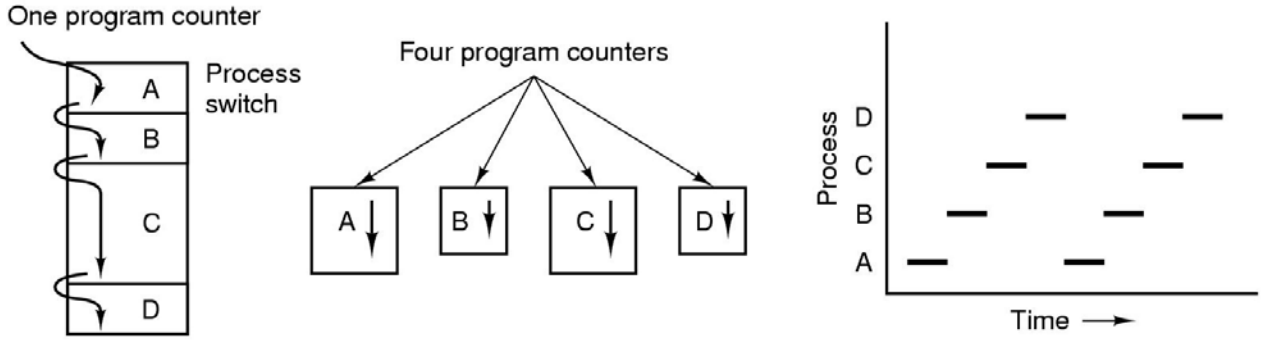
- Process vs. Thread
- Simple Three Threads: main(), thread_A, thread_B
- Simple Three Threads with mutex (critical section)
- Simple Three Threads with mutex and turn
- Simple Three Threads with Circular Queue for Message Passing
- Simple Three Threads with Priority Queue for Event Handling with Priority

멀티 스레드,
공유 자원의 임계구역

프로세스 (Process)

◆ Process

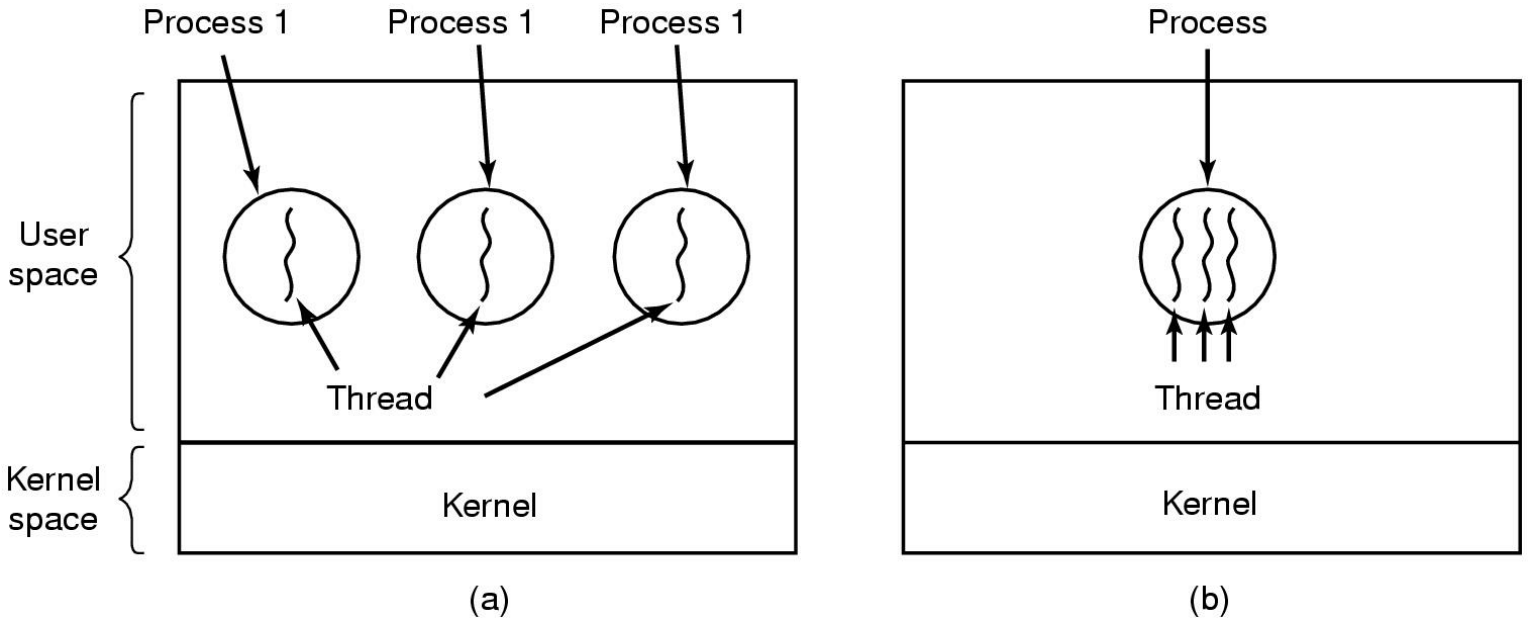
- 프로세스 (process)란 프로그램이 수행중인 상태 (***program in execution***)
  - 각 프로세스 마다 개별적으로 메모리가 할당 됨 (text core, initialized data (BSS), non-initialized data, heap (dynamically allocated memory), stack)
- 일반적인 PC나 대부분의 컴퓨터 환경에서 하나의 물리적인 CPU 상에 다수의 프로세스가 실행되는 ***Multi-tasking*** 이 지원되며, 운영체제가 다수의 프로세스를 일정 시간 마다 실행 기회를 가지게 하는 테스크 스케쥴링 (task scheduling)을 지원
- 하나의 프로세스가 실행을 중단하고, 다른 프로세스가 실행될 수 있게 하는 것을 컨텍스트 스위칭 (***Context switching***) 이라 하며, 운영체제의 process scheduling & switching이 프로세스간의 교체를 수행함
- 하나의 물리적인 CPU가 사용되는 시스템에서는 임의의 순간에는 하나의 프로세스만 실행되나, 일정 시간 (예: 100ms)마다 프로세스가 교체되며 실행되기 때문에 전체적으로 다수의 프로그램 들이 동시에 실행되는 것과 같이 보이게 됨



스레드 (Thread)

◆ 스레드 (Thread)

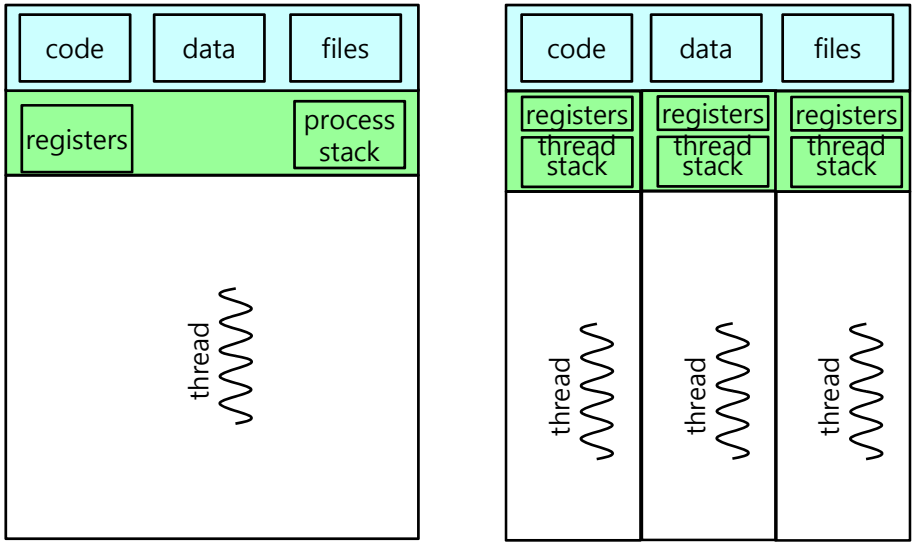
- 스레드는 하나의 프로세스 내부에 포함되는 함수들이 동시에 실행될 수 있게 한 작은 단위 프로세서 (lightweight process)
- 기본적으로 CPU를 사용하는 기본 단위
- 하나의 프로세스에 포함된 다수의 스레드 들은 프로세스의 메모리 자원들 (code section, data section, Heap 등)과 운영체제의 자원들 (예: 파일 입출력 등)을 공유함



프로세스 (Process)와 스레드 (Thread)의 차이점

◆ Multi-thread 란?

- 어떠한 프로그램 내에서, 특히 프로세스(process) 내에서 실행되는 흐름의 단위.
- 일반적으로 한 프로그램은 하나의 thread를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 thread를 동시에 실행할 수 있다.
  이를 **멀티스레드(multi-thread)**라 함.
- 프로세스는 각각 개별적인 code, data, file을 가지나, 스레드는 자신들이 포함된 프로세스의 code, data, file들을 공유함

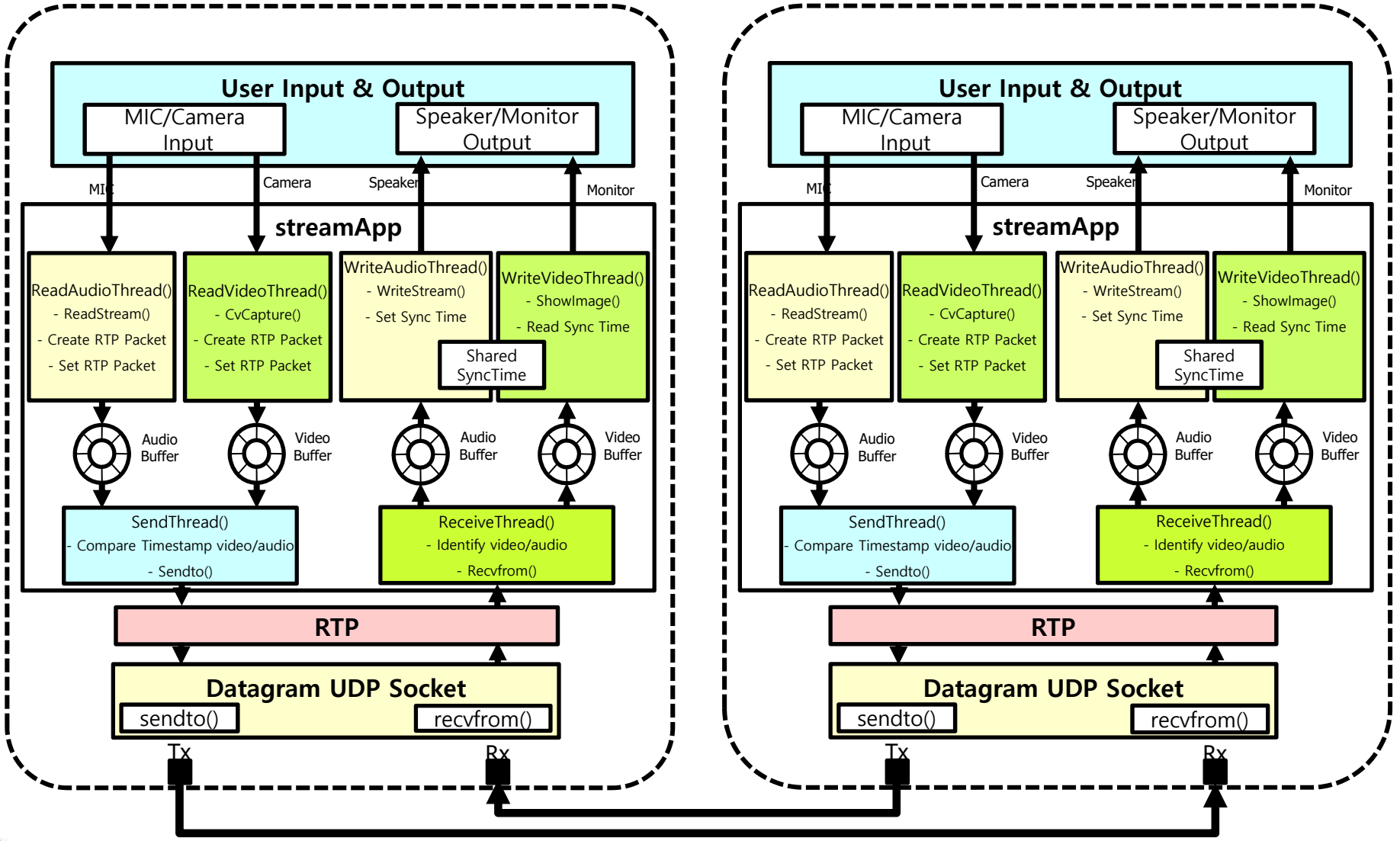


(a) single-thread process (b) multi-thread process

Task 수행이 병렬로 처리되어야 하는 경우

- **양방향 동시 전송이 지원되는 멀티미디어 정보통신 응용 프로그램 (application)**
  - full-duplex 실시간 전화서비스: 상대편의 음성 정보를 수신하면서, 동시에 나의 음성정보를 전송하여야 함
  - 음성정보의 입력과 출력이 동시에 처리될 수 있어야 함
  - 영상정보의 입력과 출력이 동시에 처리될 수 있어야 함
- **다수의 사건 발생을 등록하며, 우선 순위에 따라 처리하여야 하는 Event Handling/Management**
  - Event가 발생하면 이를 즉시 접수/등록
  - 접수/등록된 event를 우선 순위에 따라 처리
  - Event 처리 프로세서가 다수 사용될 수 있음
  - Event를 처리하고 있는 중에도 더 시급한 event가 발생되면 이를 처리할 수 있도록 운영

◆ 실시간 영상/화상 전화기의 기능 블럭도



C++11의 멀티스레드 관련 클래스 및 멤버함수

◆C++11의 스레드 관련 클래스 및 멤버 함수

스레드 관련 클래스 및 멤버 함수	설명
std::thread	스레드 클래스, 스레드 생성 thread myThread(func, &thread_param);
join()	스레드의 실행이 종료될 때까지 대기 myThread.join();
get_id()	스레드의 identifier을 반환 thread_id = myThread.get_id();
sleep_for(sleep_duration)	지정된 시간 만큼 스레드 실행을 중지 (sleep) sleep_duration 설정 예 (설정 시간 단위): - std::chrono::seconds/milliseconds/microseconds/nanoseconds /* 초/밀리초/마이크로초/나노초 */ - std::chrono::minutes/hours /* 분/시 */ - std::chrono::/days/weeks/months/years /* 일/주/월/년, since C++20 */
_sleep(sleep_duration_ms)	Windows 운영체제에서 제공하는 API 함수 (#include <Windows.h> 필요) sleep_duration_ms은 milli-second 단위

스레드의 함수의 구현

◆ 스레드 함수의 구현

- 프로그램에 포함되는 함수 중, 병렬로 실행되어야 하는 함수를 스레드로 지정
- 스레드 파라메터 구조체 포인터 (pParam)를 통하여, 스레드 생성 및 실행에 관련된 정보를 main() 함수로 부터 전달 받으며, 파라메터 구조체는 필요에 따라 정의
- 스레드는 보통 지정된 회수 만큼 실행을 하거나, 무한 루프로 실행함

```
/* Multi_Thread.h */
#include <thread>
#include <mutex>  // mutual exclusive semaphore
#include <string>

typedef struct
{
      mutex *pCS;
      string name;
      char myMark;
      char *pFlag_Terminate;  // controlled by main thread
} ThreadParam;
void simpleThread(ThreadParam *pParam);
```

Thread 예제

```
/* Simple_Thread.cpp */
#include <stdio.h>
#include <thread>
#include <mutex>
#include "Multi_Thread.h"
void Simple_Thread(ThreadParam *pThrdParam)
{
   string myName = pThrdParam->name;

   FILE *fout = pThrdParam->fout;
   char myMark = pThrdParam->myMark;
   int counter;
   char *pFlag_Terminate = pThrdParam->pFlag_Terminate;

   // Simple_Thread procedure
   while (*pFlag_Terminate == 0)
   {
        //fprintf(fout, "%s : ", myName);
        for (int j = 0; j < 100; j++)
            fprintf(fout, "%c", myMark);
        fprintf(fout, "\n");
        Sleep(1);
   }
   //fprintf(fout, "%s is terminating ...\n", myName);
}
```

스레드 함수로의 파라메터 전달

◆ 스레드 함수로의 파라메터 전달을 위한 구조체 정의 (예)

- 필요에 따라 파라메터 항목들을 포함하는 구조체 정의
- 기본적으로 mutex에 관련된 정보, 공유되는 큐의 정보, 파일 입출력에 관련된 정보를 포함

```
typedef struct
{
    CircularQueue *cirQ;
    mutex* pCS;
    int role;
} ThreadParam;
```

```
typedef struct
{
    mutex *pCS;
    Queue *cirQ;
    ROLE role; //
    unsigned int  addr;
    int max_queue;
    int duration;
    FILE *fout;
} ThreadParam;
```

```
typedef struct
{
        FILE *fout; // for log file
        int id;
        mutex *pCS_main;
        ThreadStatus *pThreadStatus;
            // status of packet generator and
            // packet forwarder
        Packet *pPacketStatusTbl;
        CirQ *pCirQ; // pointer array for circular queue
        int num_pkt_gen_procs;
        int num_links;
        ROLE role;
        UINT_32 myAddr;
        int max_Q_capa;
        int num_packets_to_generate;
        int *pThread_Pkt_Gen_Terminate_Flag;
        int *pThread_Link_Terminate_Flag;
        int max_rounds;
        HANDLE consoleHandler;
} ThreadParam;
```

Critical Section (임계구역)과 mutex (1)

◆ Critical Section (임계구역)

- 다중 스레드 사용을 지원하는 운영체제는 프로그램 실행 중에 스레드 또는 프로세스간에 교체가 일어 날 수 있게 하여, 다수의 스레드/프로세스가 병렬로 처리될 수 있도록 관리
- Context switching이 일어나면, 현재 실행 중이던 스레드/프로세스의 중간 상태가임시 저장되고, 다른 스레드/프로세스가 실행됨
- 프로그램 실행 중에 특정 구역은 실행이 종료될 때 까지 스레드/프로세서 교체가 일어 나지 않도록 관리하여야 하는 경우가 있음
- 아래의 인터넷 은행 입금 및 출금 스레드 예에서 critical section으로 보호하여야 할 구역은 ?

```
1.  Thread_Deposit (int deposit)
2.  {
3.      // account is shared variable
4.      l_account = g_account;

5.      l_account =
            l_account + deposit;

6.      g_account = l_account;

7.      print(l_account);
8.      . . . .
9.  }
```

shared resource


```
1.  Thread_Withdraw (int withdraw)
2.  {
3.      // account is shared variable
4.      l_account = g_account;

5.      l_account =
            l_account - withdraw;

6.      g_account = l_account;

7.      print(l_account);
8.      . . . .
9.  }
```

정상적인 실행

실행 순서	Thread_Deposit (deposit = 70)	account (g_acct = 100)	Thread_Withdraw (widthdraw = 80)
0	Thread Switching		
1	l_acct = g_acct	100	
2	l_acct = l_acct + deposit;		
3	g_acct = l_acct;	170	
4	print(l_acct);		
5	Thread Switching		
6			l_acct = g_acct;
7			l_acct = l_acct - widthdraw;
8		90	g_acct = l_acct;
9			print(l_acct);
10	Thread Switching		

문제발생 경우

실행 순서	Thread_Deposit (deposit = 70)	account (g_acct = 100)	Thread_Withdraw (widthdraw = 80)
0	Thread Switching		
1	l_acct = g_acct	100	
2	Thread Switching		
3			l_acct = g_acct;
4			l_acct = l_acct - widthdraw;
5		20	g_acct = l_acct;
6			print(l_acct);
7	Thread Switching		
8	l_acct = l_acct + deposit;		
9	g_acct = l_acct;	170	
10	print(l_acct);		
11	Thread Switching		

Critical Section (임계구역)과 mutex (2)

◆ mutex를 사용한 임계구역 (critical section) 설정

```
1.   Thread_Deposit (deposit, pCS)
2.   {
3.      // account is shared variable
4.      pCS->lock();
5.      l_account = g_account;

6.      l_account =
           l_account + deposit;

7.      g_account = l_account;
8.      print(l_account);
9.      pCS->unlock();

10.     . . . .
11.  }
```

shared resource

은행잔고
g_account

mutex cs_acct

```
1.   Thread_Withdraw (withdraw, pCS)
2.   {
3.      // account is shared variable
4.      pCS->lock();
5.      l_account = g_account;

6.      l_account =
           l_account - withdraw;

7.      g_account = l_account;
8.      print(l_account);
9.      pCS->unlock();

10.     . . . .
11.  }
```

mutex 관련 라이브러리 함수	설명
mutex CS	임계구역 (critical section) 설정을 위한 세마포 (semaphore) 생성
CS.lock()	임계구역 설정 시작, mutex를 획득
CS.unlock()	임계구역 설정 종료, mutex를 반환

임계구역 (Critical Section)과 mutex (3)

- **mutex의 설정: 현재 어떤 스레드/프로세스가 실행 중에 있다는 상태를 mutex을 표시하는 변수로 표시**
  - semaphore 라고 부르기도 함
- **mutex 변수의 설정**
  - mutex mtx
    - mutex 생성
    - mutex의 lock() 및 unlock() 실행 이전에 생성되어 있어야 함
- **mutex를 사용한 critical section 영역 지정**
  - mtx.lock()
  - mtx.unlock()

C++11환경에서의
스레드 생성 및 실행 제어

스레드의 생성 및 종료 (1)

◆ 스레드 생성, 소멸 및 관리

- thread 클래스를 사용하여 생성
- thread의 join() 함수를 사용하여 생성된 스레드가 스스로 함수 실행을 종료 할 때 까지 대기

```
/* Sample multi_threads.c (1) */
#include <thread>
#include <mutex>
#include "Multi_Thread.h“  // contains ThreadParam
void simpleThread(ThreadParam *pParam);

void main()
{
   ThreadParam thrdParam;
   mutex cs_console;
   unsigned int thread_id;
```

```
/* Sample multi_threads.c (2) */

    thrdParam.name = string("Thread_A");
    thrdParam.pCS = &cs_console;
    thread simThrd(simpleThread, &thrdParam); // create & activate thread
    thread_id = simThrd.get_id();
    cs_console.lock();
    printf("main() : Thread (id: %d) is successfully created !\n", thread_id);
    cs_console.unlock();

    // .... execution of thread
    cs_console.lock();
    printf("main() : Waiting the thread (%d ) to terminate by itself ...\n", thread_id);
    cs_console.unlock();

    simThrd.join(); // wait for thread termination
    cs_console.lock();
    printf("main() : Thread (%d) is terminated now.\n",  thread_id);
    cs_console.unlock();
} // end main()
```

스레드의 생성 및 종료 (2)

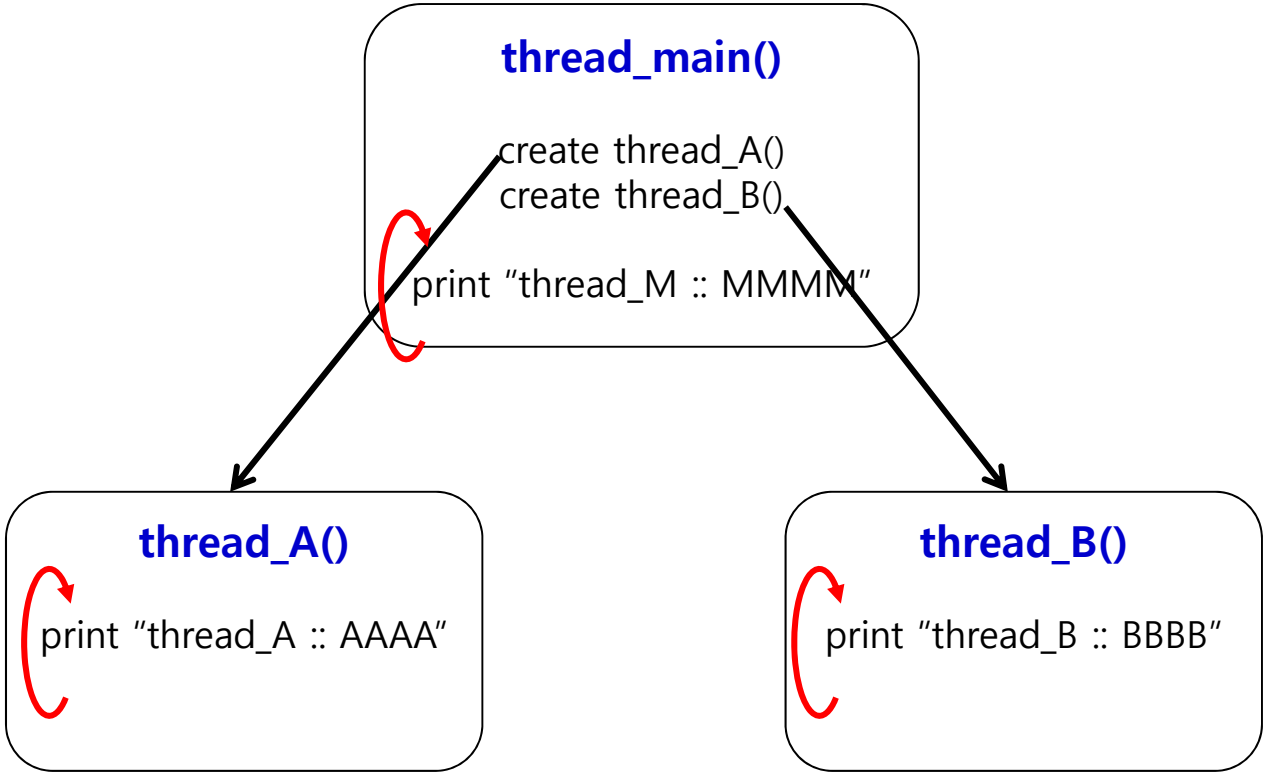
◆ 스레드가 스스로 종료할 때 까지 기다리는 경우

- main() 함수에서는 스레드가 종료할 때 까지 기다림

```
myThread.join(); // wait for terminate thread
```

Three Simple Threads – Ver. 1

◆ **Three simple threads printing mark 'A', 'B', and 'M'**



```
/* SimpleThreadsVer1.cpp  (1) */

#include<stdio.h>
#include <thread>
#include <mutex>
#include<time.h>
enum ROLE { PRODUCER, CONSUMER };
typedef struct
{
    char mark;
} ThreadParam;
#define THREAD_EXIT_CODE 7
void Thread_A(ThreadParam *pParam);
void Thread_B(ThreadParam *pParam);

void main()
{
    /*   변수 선언 */
    ThreadParam thrParam_A, thrParam_B; /* 각 스레드로 전달될 파라미터 구조체*/

    thrParam_A.mark = 'A'; /* thread_A에 전달 될 파라미터값 초기화 */
    thread  thrd_A(Thread_A, &thrParam_A); /*  스레스 생성, 활성화 */

    thrParam_B.mark = 'B'; /* thread_B에 전달 될 파라미터값 초기화 */
    thread thrd_B(Thread_B, &thrParam_B);
```

```
/* SimpleThreadsVer1.cpp  (2) */

   /* main() thread 실행 */
   char mark = 'M';
   for (int i = 0; i < 100; i++)
   {
      printf("Thread_main … : ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);
      }
      printf("\n");
   }
   thrd_A.join();  /* thrd_A가 종료할 때 까지 대기 */
   thrd_B.join();  /* thrd_B가 종료할 때 까지 대기 */
}
```

```
/* SimpleThreadsVer1.cpp  (3) */

void Thread_A(ThreadParam *pThrParam)
{
   char mark = pThrParam->mark;

   for (int i = 0; i < 100; i++)
   {
      printf("Thread_A ... ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);  // print 'A'
      }
      printf("\n");
   }
}
```

```
/* SimpleThreadsVer1.cpp  (4) */

void Thread_B(ThreadParam *pThrParam)
{
   char mark = pThrParam->mark;

   for (int i = 0; i < 100; i++)
   {
      printf("Thread_B ... ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark); // print 'B'
      }
      printf("\n");
    }
}
```

◆ 실행결과

- thread_A, thread_B, thread_main이 일정 시간마다 번갈아 가며 실행
- 한 줄이 다 출력되기 전에 thread간의 교체가 발생되어, 출력이 섞임

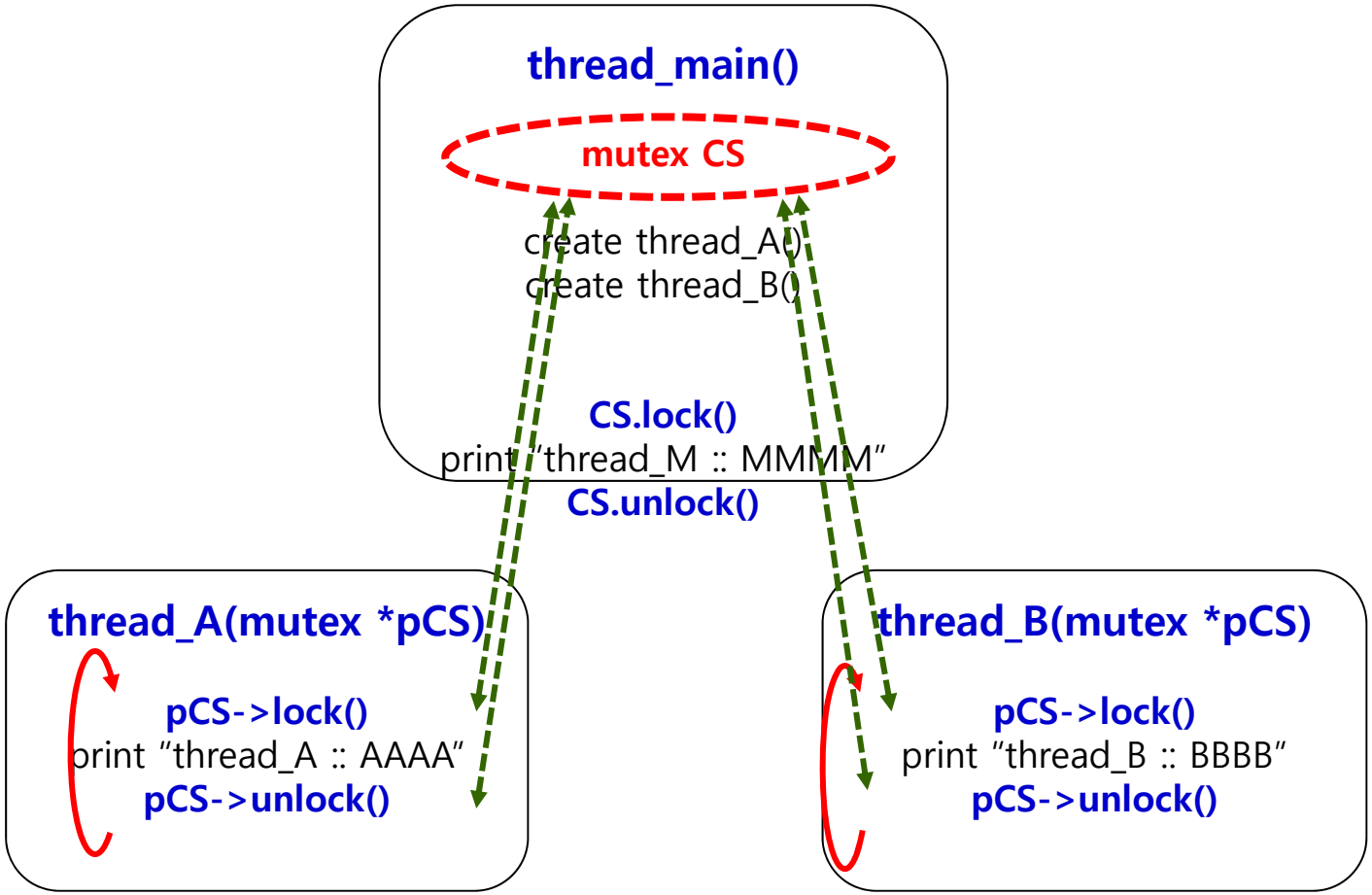
◆ 만약, 한번에 thread하나가 한 줄씩만 출력하고, 번갈아 가며 출력하기 위해서는 어떻게 수정하여야 하나?

- **mutex (mutual exclusion)**사용: 한 줄을 다 출력 할 때까지, 다른 스레드가 실행되지 못하도록 보호
- **"turn" semaphore** 사용: 한 줄을 다 출력한 후에는 반드시 다른 스레드가 실행되도록 실행 순서(turn) 제어

```
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBThread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBThread_M ... BMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBMBM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBThread_A ... BABABABABABABA
AAAAAAAAAAAAThread_M ... AMAMAMAMAMAMAMAMAMAMAMAMAMAM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... Thread_A ... BAThread_M ... BMAMABMAMBMABABMBABMABMBMBA
MBMBMBMBMBMBMBMBMBMBM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMThread_B ... MBMBMBMBMBMBMBMBM
MMMMMMMMMM
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBThread_M ... BMBMBMBMBMBMBMBThread_A ...
MAMAMAMAM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBThread_A ... BABABABABABABABThread_M ... BABABABA
AMAMAMAMAMAM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBThread_M ... BMBMBMBMBMBMBMBMBMBM
MBMBMBMBMBMBMBBMBMBMBMBMBM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... Thread_A ... BABABABABABABABABABABABABABABABABABABABABABABA
```

Three Simple Threads – Version 2

◆ Three simple threads using mutex



```
/* SimpleThreadsVer2.cpp  (1) */
#include<stdio.h>
#include <thread>
#include <mutex>
#include<time.h>
enum ROLE { PRODUCER, CONSUMER };
typedef struct
{
   mutex* pCS; // pSemaphore
   char mark;
}ThreadParam;
#define THREAD_EXIT_CODE 7
void Thread_A(ThreadParam *pParam);
void Thread_B(ThreadParam *pParam);

void main()
{
   /*    변수 선언 */
   mutex crit; // semaphore
   ThreadParam  thrParam_A, thrParam_B;

 /* Thread_A에 전달 될 파라미터값 초기화 */
   thrParam_A.pCS = &crit;
   thrParam_A.mark = 'A';
   thread thrd_A(Thread_A, &thrParam_A);
```

```
/* SimpleThreadsVer2.cpp  (2) */

   /* Thread_B에 전달 될 파라미터값 초기화 */
   thrParam_B.pCS = &crit;
   thrParam_B.mark = 'B';
   thread thrd_B(Thread_B, &thrParam_B);

   /* main() thread 실행 */
   char mark = 'M';
   for (int i = 0; i < 100; i++)
   {
      crit.lock();
      printf("Thread_main ... ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);
      }
      printf("\n");
      crit.unlock();
   }
   thrd_A.join();
   thrd_B.join();
}
```

```
/* SimpleThreadsVer2.cpp  (3) */

void Thread_A(ThreadParam *pThrParam)
{
   char mark = pThrParam->mark;
   thread *pCS = pThrParam->pCS;

   for (int i = 0; i < 20; i++)
   {
      pCS->lock();
      printf("Thread_A ... ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);
      }
      printf("\n");
      pCS->unlock();
      Sleep(100);
   }
   pCS->lock();
   printf("Thread_A finished ...\n");
   pCS->unlock();
}
```

```
/* SimpleThreadsVer2.cpp  (4) */

void Thread_B(ThreadParam *pThrParam)
{
   char mark = pThrParam->mark;
   thread *pCS = pThrParam->pCS;

   for (int i = 0; i < 20; i++)
   {
      pCS->lock();
      printf("Thread_B ... ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);
      }
      printf("\n");
      pCS->unlock();
      Sleep(100);
   }
   pCS->lock();
   printf("Thread_B finished ...\n");
   pCS->unlock();
}
```

◆ 실행결과 (ver 2)

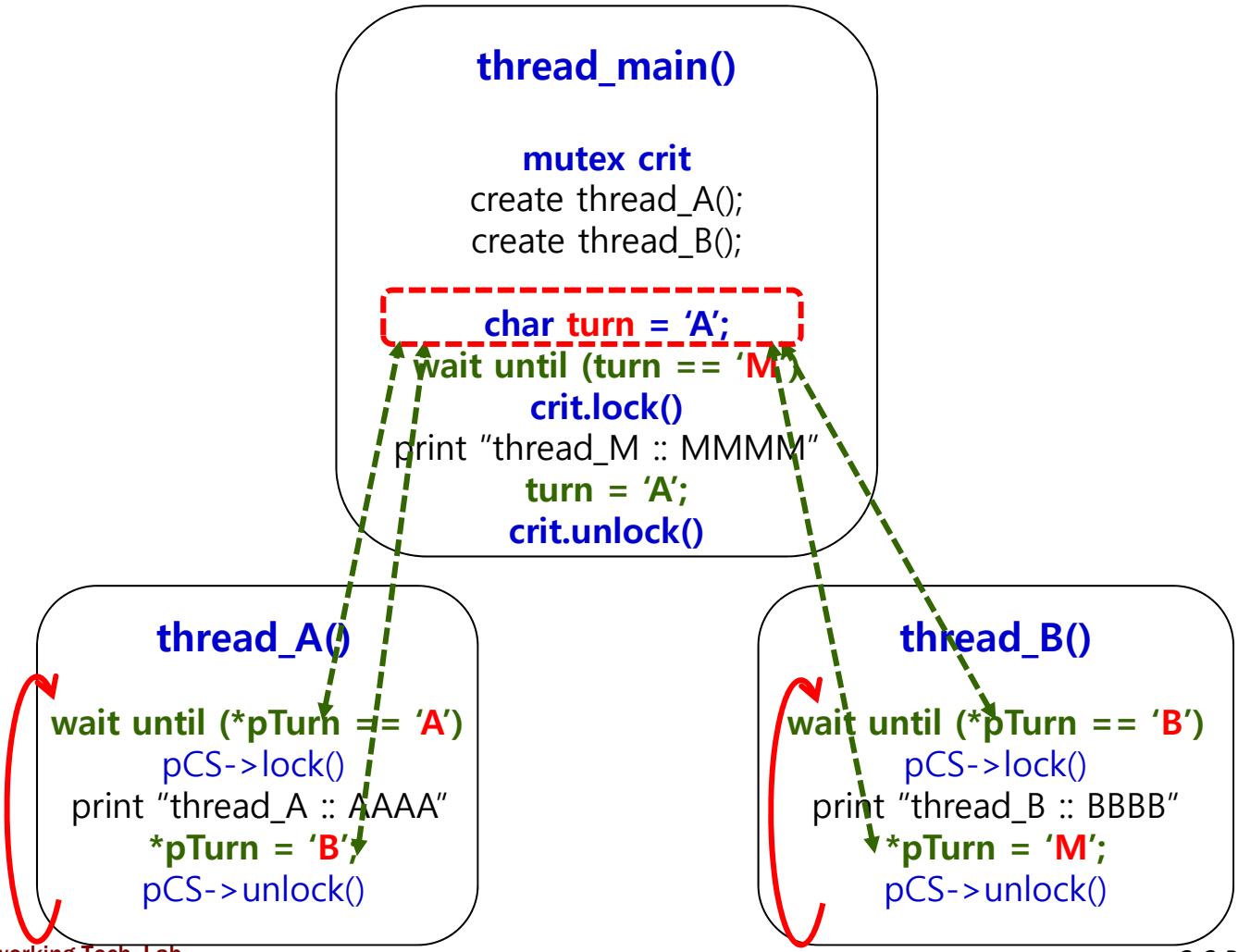
- critical section을 사용하여, 하나의 스레드가 한 줄의 출력을 완전히 완료할 때까지 다른 스레드가 실행되지 못하게 보호
- thread_A 또는 thread_B가 두 줄씩 출력하는 경우가 있음

```
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_M ... MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

스레드 M, A, B가 한번에 각각 한 줄씩만 출력하게 하는 방법은 ?

Three Simple Threads – Version 3

◆ **Three threads printing "A', 'B', and 'M' with turn**



```
/* SimpleThreadsVer3.cpp  (1) */
#include<stdio.h>
#include <thread>
#include <mutex>
#include<time.h>
enum ROLE { PRODUCER, CONSUMER };

typedef struct
{
    mutex* pCS;
    char mark;
    char *pTurn;
} ThreadParam;

void Thread_A(ThreadParam *pParam);
void Thread_B(ThreadParam *pParam);

void main()
{
    /*     변수 선언 */
    mutex crit;
    ThreadParam thrParam_A, thrParam_B;
    char turn = 'A';

    /* Thread_A에 전달 될 파라미터값 초기화 */
    thrParam_A.pCS = &crit;
    thrParam_A.mark = 'A';
    thrParam_A.pTurn = &turn;
    thread thrd_A(Thread_A, &thrParam_A);
```

```
/* SimpleThreadsVer3.cpp  (2) */

/* Thread_B에 전달 될 파라미터값 초기화 */
    thrParam_B.pCS = &crit;
    thrParam_B.mark = 'B';
    thrParam_B.pTurn = &turn;
    thread thrd_B (Thread_B,  &thrParam_B);
    /* main() thread 실행 */
    char mark = 'M';
    for (int round = 0; round < 10; round++)
    {
        while (turn != 'M')
            Sleep(10);
        crit.lock();
        printf("Thread_main : ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("\n");
        turn = 'A';
        crit.unlock();
    }
}
```

```
/* SimpleThreadsVer3.cpp  (3) */

void Thread_A(ThreadParam * pThrParam)
{
   char mark = pThrParam->mark;
   mutex *pCS = pThrParam->pCS;

   for (int i = 0; i < 10; i++)
   {
      while ('A' != *pThrParam->pTurn)
         Sleep(10);
      pCS->lock();
      printf("Thread_A : ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);
      }
      printf("\n");
      *pThrParam->pTurn = 'B';
      pCS->unlock();
      Sleep(10);
   }
   pCS->lock();
   printf("Thread_A finished ...\n");
   pCS->unlock();
}
```

```
/* SimpleThreadsVer3.cpp  (4) */

void Thread_B(ThreadParam * pThrParam)
{
   char mark = pThrParam->mark;
   mutex *pCS = pThrParam->pCS;

   for (int i = 0; i < 10; i++)
   {
      while ('B' != *pThrParam->pTurn)
         Sleep(10);
      pCS->lock();
      printf("Thread_B : ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);
      }
      printf("\n");
      *pThrParam->pTurn = 'M';
      pCS->unlock();
      Sleep(10);
   }
   pCS->lock();
   printf("Thread_B finished ...\n");
   pCS->unlock();
}
```

◆ 실행결과 (ver 3)

- mutex과 함께 turn semaphore를 사용
- 하나의 스레드가 한 줄의 출력을 완료한 후, 다른 스레드가 출력을 완료할 때 까지 계속 기다리게 함
- thread_A 또는 thread_B가 한번에 한 줄씩만 출력하며, 번갈아 가며 출력

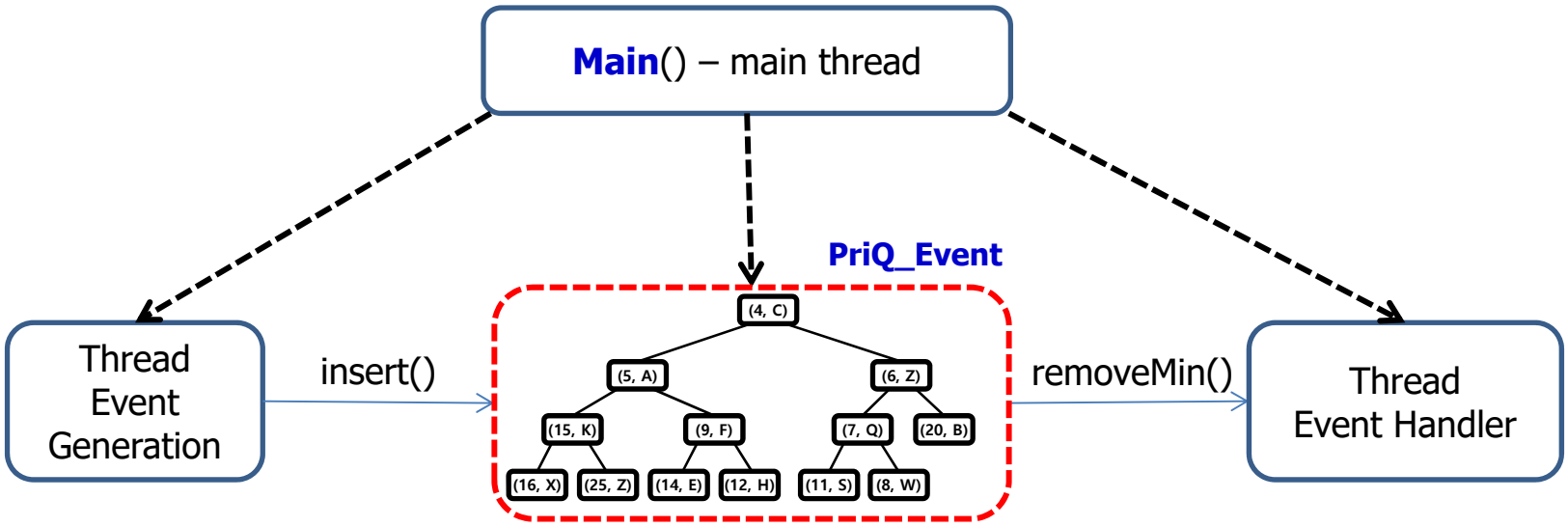
```
main() : Thread_A (id:202444) is successfully created !
Thread_A is activated now !
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B is activated now !
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
main() : Thread_B (id:199500) is successfully created !
main() : Current exit code of Thread_A (202444) is 259 that means this Thread_A is STILL_ACTIVE.
main() : Current exit code of Thread_B (199500) is 259 that means this Thread_B is STILL_ACTIVE.
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_M : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
main() : Waiting the Thread_A (202444) to terminate by itself ...
Thread_A : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
main() : Thread_A (202444) is terminated now with ExitCode (65).
main() : Waiting the Thread_B (199500) to terminate by itself ...
Thread_B : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
main() : Thread_A (199500) is terminated now with ExitCode (66).
```

멀티스레드와 우선순위큐 구조의 이벤트 생성 및 처리과정 시뮬레이션 (1)

Three Simple Threads – Version 4

◆ Two Event Handling Threads with Priority Queue

- Two Threads
  - Event Generator
  - Event Handler
- Shared Priority Queue
  - PriQ for Events



class Event

```
/* Event.h (1) */

#include <iostream>
#include <string>
#include <fstream>
#include <Windows.h> // for LARGE_INTEGER used in QueryPerformanceCounter()
#include <iomanip>
using namespace std;

enum EventStatus { GENERATED, ENQUEUED, PROCESSED, UNDEFINED };
#define MAX_EVENT_PRIORITY 100
#define NUM_EVENT_GENERATORS 10

class Event
{
    friend ostream& operator<<(ostream& fout, const Event& e);
public:
    Event(); // default constructor
    Event(int event_id, int event_pri, int srcAddr); //constructor
    void printEvent_proc();
    void setEventHandlerAddr(int evtHndlerAddr) { event_handler_addr = evtHndlerAddr; }
    void setEventGenAddr(int genAddr) { event_gen_addr = genAddr; }
    void setEventNo(int evtNo) { event_no = evtNo; }
    void setEventPri(int pri) { event_pri = pri; }
    void setEventStatus(EventStatus evtStatus) { eventStatus = evtStatus; }
    void setEventGenTime(LARGE_INTEGER t_gen) { t_event_gen = t_gen; }
    void setEventProcTime(LARGE_INTEGER t_proc) { t_event_proc = t_proc; }
```

```
/* Event.h (2) */

    LARGE_INTEGER getEventGenTime() { return t_event_gen; }
    LARGE_INTEGER getEventProcTime() { return t_event_proc; }
    void setEventElaspsedTime(double t_elapsed_ms) { t_elapsed_time_ms = t_elapsed_ms; }
    double getEventElapsedTime() { return t_elapsed_time_ms; }
    int getEventPri() { return event_pri; }
    int getEventNo() { return event_no; }
    bool operator>(Event& e) { return (event_pri > e.event_pri); }
    bool operator<(Event& e) { return (event_pri < e.event_pri); }
private:
    int event_no;
    int event_gen_addr;
    int event_handler_addr;
    int event_pri;  // event_priority
    LARGE_INTEGER t_event_gen;
    LARGE_INTEGER t_event_proc;
    double t_elapsed_time_ms;
    EventStatus eventStatus;
};

Event* genRandEvent(int evt_no);
```

```
/* Event.cpp (1) */
#include "Event.h"

Event::Event(int evt_no, int evt_pri, int evtGenAddr)
{
    event_no = evt_no;
    event_gen_addr = evtGenAddr;
    event_handler_addr = -1; // event handler is not defined at this moment
    event_pri = evt_pri;  // event_priority
    eventStatus = GENERATED;
}

Event* genRandEvent(int evt_no)
{
    Event *pEv;
    int evt_prio;
    int evt_generator_id;

    evt_prio = rand() % MAX_EVENT_PRIORITY;
    evt_generator_id = rand() % NUM_EVENT_GENERATORS;

    pEv = (Event *) new Event(evt_no, evt_prio, evt_generator_id);

    return pEv;
}
```

```
/* Event.cpp (2) */
#include "Event.h"

void Event::printEvent_proc()
{
    cout << "Ev(no:" << setw(2) << event_no << ", pri:" << setw(2) << event_pri;
    cout.precision(2);
    cout.setf(ios::fixed);
    cout << ", t_elapsed:" << setw(8) << t_elapsed_time_ms << ") ";
}

ostream& operator<<(ostream& fout, const Event& evt)
{
    fout << "Event(pri:" << setw(3) << evt.event_pri << ", gen:" << setw(3) << evt.event_gen_addr;
    fout << ", no:" << setw(3) << evt.event_no << ")";

    return fout;
}
```

class CompleteBinaryTree<K, V>

```
/* CompleteBinaryTree.h (1) */
#ifndef COMPLETE_BINARY_TREE_H
#define COMPLETE_BINARY_TREE_H
#include "TA_Entry.h"
#include "T_Entry.h"
#define CBT_ROOT 1

template<typename K, typename V>
class CompleteBinaryTree : public TA_Entry<K, V>
{
public:
    CompleteBinaryTree(int capa, string nm);
    int add_at_end(const T_Entry<K, V>& elem);
    T_Entry<K, V>& getEndElement() { return t_array[end]; }
    T_Entry<K, V>& getRootElement() { return t_array[CBT_ROOT]; }
    int getEndIndex() { return end; }
    void removeCBTEnd();
    void fprintCBT(ofstream &fout);
    void fprintCBT_byLevel(ofstream &fout);
protected:
    void _fprintCBT_byLevel(ofstream &fout, int p, int level);
    int parentIndex(int index) { return index / 2; }
    int leftChildIndex(int index) { return index * 2; }
    int rightChildIndex(int index) { return (index * 2 + 1); }
    bool hasLeftChild(int index) { return ((index * 2) <= end); }
    bool hasRightChild(int index) { return ((index * 2 + 1) <= end); }
    int end;
};
```

```
/* CompleteBinaryTree.h (2) */

template<typename K, typename V>
CompleteBinaryTree<K, V>::CompleteBinaryTree(int capa, string nm)
:TA_Entry<K, V>(capa+1, nm)
{
    end = 0; // reset to empty
}

template<typename K, typename V>
void CompleteBinaryTree<K, V>::fprintCBT(ofstream &fout)
{
    if (end <= 0)
    {
       fout << this->getName() << " is empty now !!" << endl;
       return;
    }
    int count = 0;
    for (int i = 1; i <= end; i++)
    {
        fout << setw(3) << t_array[i] << endl;
        //if ((((count + 1) % 10) == 0) && (i != end))
        //fout << endl;
        count++;
    }
}
```

```
/* CompleteBinaryTree.h (3) */

template<typename K, typename V>
void CompleteBinaryTree<K, V>::_fprintCBT_byLevel(ofstream &fout, int index, int level)
{
    int index_child;
    if (hasRightChild(index))
    {
        index_child = rightChildIndex(index);
        _printCBT_byLevel(fout, index_child, level + 1);
    }

    for (int i = 0; i<level; i++)
       fout << "     ";
    t_array[index].fprint(fout);
    fout << endl;

    if (hasLeftChild(index))
    {
        index_child = leftChildIndex(index);
        _printCBT_byLevel(fout, index_child, level + 1);
    }
}

template<typename K, typename V>
void CompleteBinaryTree<K, V>::fprintCBT_byLevel(ofstream &fout)
{
    if (end <= 0)
    {
        fout << "CBT is EMPTY now !!" << endl;
        return;
    }
    _printCBT_byLevel(fout, CBT_ROOT, 0);
}
```

```
Final status of insertions :
                3
          2
                10
     1
                9
          4
                13
0
                7
          6
                12
     5
                11
          8
                14
```

```
/* CompleteBinaryTree.h (4) */

template<typename K, typename V>
int CompleteBinaryTree<K, V>::add_at_end(const T_Entry<K, V>& elem)
{
    if (end >= capacity)
    {
       cout << this->getName() << " is FULL now !!" << endl;
       return -1;
    }
    end++;
    t_array[end] = elem;

    return end;
}

template<typename K, typename V>
void CompleteBinaryTree<K, V>::removeCBTEnd()
{
    end--;
    num_elements--;
}
#endif
```

class HeapPrioQ<K, V>

```
/* HeapPrioQ.h (1) */

#ifndef HEAP_PRIO_QUEUE_H
#define HEAP_PRIO_QUEUE_H
#include <mutex>
#include "CompleteBinaryTree.h"
using namespace std;

template<typename K, typename V>
class HeapPrioQueue : public CompleteBinaryTree<K, V>
{
public:
      HeapPrioQueue(int capa, string nm);
      ~HeapPrioQueue();
      bool isEmpty() { return (this->end <= 0); }
      bool isFull() { return (this->end >= this->heapPriQ_capa); }
      T_Entry<K, V>* insert(const T_Entry<K, V>& elem);
      T_Entry<K, V>* removeHeapMin();
      T_Entry<K, V>* getHeapMin();
      void fprint(ofstream &fout);
      int size() {return this->end; }
private:
     int heapPriQ_capa;
     mutex cs_priQ;
};
```

```
/* HeapPrioQ.h (2) */

template<typename K, typename V>
HeapPrioQueue<K, V>::HeapPrioQueue(int capa, string nm)
:CompleteBinaryTree(capa, nm), heapPriQ_capa(capa)
{ }

template<typename K, typename V>
HeapPrioQueue<K, V>::~HeapPrioQueue()
{ }

template<typename K, typename V>
void HeapPrioQueue<K, V>::fprint(ofstream &fout)
{
    if (size() <= 0)
    {
       fout << "HeapPriorityQueue is Empty !!" << endl;
       return;
    }
    else
       CompleteBinaryTree::printCBT(fout);
}
```

```
/* HeapPrioQ.h (3) */

template<typename K, typename V>
T_Entry<K, V>* HeapPrioQueue<K, V>::insert(const T_Entry<K, V>& elem)
{
    int index, parent_index;
    T_Entry<K, V> temp;
    if (isFull())
    {
        cout << "HeapPrioQ is Full !!" << endl;
        return NULL;
    }
    cs_priQ.lock()
    index = add_at_end(elem);

    /* up-heap bubbling */
    while (index != CBT_ROOT)
    {
        parent_index = parentIndex(index);
        if (t_array[index].getKey() >= t_array[parent_index].getKey())
            break;
        else
        {
            temp = t_array[index];
            t_array[index] = t_array[parent_index];
            t_array[parent_index] = temp;
            index = parent_index;
        }
    }
    cs_priQ.unlock();
    T_Entry<K, V>* pRoot = &(this->t_array[CBT_ROOT]);
    return pRoot;
}
```

```
/* HeapPrioQ.h (4) */

template<typename K, typename V>
T_Entry<K, V>* HeapPrioQueue<K, V>::getHeapMin()
{
    T_Entry<K, V>* pMinElem;
    if (this->end <= 0)
    {
       return NULL;
    }
    pMinElem = (T_Entry<K, V>*) new T_Entry<K, V>;
    *pMinElem = getRootElement();
    return pMinElem;

}
```

```
/* HeapPrioQ.h (4) */

template<typename K, typename V>
T_Entry<K, V>* HeapPrioQueue<K, V>::removeHeapMin()
{
    int index_p, index_c, index_rc;
    T_Entry<K, V> *pMinElem;
    T_Entry<K, V> temp, t_p, t_c;
    int HPQ_size = this->size();

    if (HPQ_size <= 0)
    {
       return NULL;
    }
    cs_priQ.lock();
    pMinElem = (T_Entry<K, V>*) new T_Entry<K, V>;
    *pMinElem = getRootElement();
    if (HPQ_size == 1)
    {
       this->removeCBTEnd();
    }
    else
    {
        index_p = CBT_ROOT;
        t_array[CBT_ROOT] = t_array[end];
        end--;
```

```
/* HeapPrioQ.h (5) */

        /* down-heap bubbling */
        while (hasLeftChild(index_p))
        {
            index_c = leftChildIndex(index_p);
            index_rc = rightChildIndex(index_p);
            if (hasRightChild(index_p) && (t_array[index_c] > t_array[index_rc]))
                index_c = index_rc;
            t_p = t_array[index_p];
            t_c = t_array[index_c];
            if (t_p > t_c)
            {
                //swap(index_u, index_c);
                temp = t_array[index_p];
                t_array[index_p] = t_array[index_c];
                t_array[index_c] = temp;
                index_p = index_c;
            }
            else
                 break;

        } // end while
    }
    cs_priQ.unlock();
    return pMinElem;
}
#endif
```

멀티스레드와 우선순위큐 구조의 이벤트 생성 및 처리과정 시뮬레이션 (2)

Thread_EventGen() and Thread_EventProc() with PriorityQueue

◆ Thread_EventGen() with PriorityQueue

- ThreadParam_Event에 PriQ_Event 주소 전달
- Event 생성 후 insertPriQ_Event() 함수를 사용하여 PriQ_Event에 생성된 Event 삽입

◆ Thread_EventProc() with PriorityQueue

- ThreadParam_Event에 PriQ_Event 주소 전달
- removeMinPriQ_Event() 함수를 사용하여 PriQ_Event로 부터 Event 하나를 추출하고 이를 처리

ThreadParam_Event, ThreadStatusMonitor

```
/* Multi_thread.h (1) */

#ifndef MULTI_THREAD_H
#define MULTI_THREAD_H
#include <iostream>
#include <fstream>
#include <Windows.h>
#include <thread>
#include <mutex>
#include <process.h>
#include <string>
#include "HeapPrioQ_CS.h"
#include "Event.h"
#include "SimParams.h"
using namespace std;

enum ROLE { EVENT_GENERATOR,
    EVENT_HANDLER };
enum THREAD_FLAG { INITIALIZE,
    RUN, TERMINATE };
```

```
/* Multi_thread.h (2) */

typedef struct ThreadParam
{
    mutex *pCS_main;
    mutex *pCS_thrd_mon;
    HeapPrioQ_CS<int, Event> *pPriQ_Event;
    FILE *fout;
    ROLE role;
    int myAddr;
    int maxRound;
    int targetEventGen;
    LARGE_INTEGER QP_freq; // used in measurements
    ThreadStatusMonitor *pThrdMon;
} ThreadParam_Event;

typedef struct ThreadStatusMonitor
{
     int numEventGenerated;
     int numEventProcessed;
     int totalEventGenerated;
     int totalEventProcessed;
     // used for monitoring only
     Event eventGenerated[TOTAL_NUM_EVENTS];
     Event eventProcessed[TOTAL_NUM_EVENTS];
     THREAD_FLAG *pFlagThreadTerminate;
} ThreadStatusMonitor;

#endif
```

Thread Event Generator

```
/* Thread_EventGenenerator.cpp (1) */

#include <Windows.h>
#include "Multi_Thread.h"
#include "HeapPrioQ_CS.h"
#include "Event.h"
#include "SimParams.h"
#define myExitCode 0
using std::this_thread::sleep_for;

void EventGen(ThreadParam_Event* pParam)
{
    ThreadParam_Event* pThrdParam;
    HeapPrioQ_CS<int, Event>* pPriQ_Event;
    int myRole;
    THREAD_FLAG* pFlagThreadTerminate;
    int maxRound;
    T_Entry<int, Event>* pEntry, entry_event;
    Event event, * pEvent;
    int event_no = 0;
    int event_priority = 0;
    int event_gen_count = 0;
    int targetEventGen;
    int myAddr = -1;
    int event_handler_addr;
    LARGE_INTEGER QP_freq, t_gen;
    ThreadStatusMonitor* pThrdMon;
```

```
/* Thread_EventGenenerator.cpp (2) */

    pThrdParam = (ThreadParam_Event*)pParam;
    myRole = pThrdParam->role;
    myAddr = pThrdParam->myAddr;
    pPriQ_Event = pThrdParam->pPriQ_Event;
    pThrdMon = pThrdParam->pThrdMon;
    maxRound = pThrdParam->maxRound;
    targetEventGen = pThrdParam->targetEventGen;


    for (int round = 0; round < maxRound; round++)
    {
        if (event_gen_count >= targetEventGen)
        {
            if (*pThrdMon->pFlagThreadTerminate == TERMINATE)
                break;
            else {
                sleep_for(std::chrono::milliseconds(500));
                continue;
            }
        }
        event_no = event_gen_count + NUM_EVENTS_PER_GEN * myAddr;
        event_priority = targetEventGen - event_gen_count - 1;
        event.setEventNo(event_no);
        event.setEventPri(event_priority);
        event.setEventGenAddr(myAddr);
        event.setEventHandlerAddr(-1); // event handler is not defined yet !!
        QueryPerformanceCounter(&t_gen);
        event.setEventGenTime(t_gen);
        event.setEventStatus(GENERATED);
        entry_event.setKey(event.getEventPri());
        entry_event.setValue(event);
```

```
/* Thread_EventGenenerator.cpp (3) */

    pThrdParam = (ThreadParam_Event*)pParam;
    myRole = pThrdParam->role;
    myAddr = pThrdParam->myAddr;
    pPriQ_Event = pThrdParam->pPriQ_Event;
    pThrdMon = pThrdParam->pThrdMon;
    maxRound = pThrdParam->maxRound;
    targetEventGen = pThrdParam->targetEventGen;


    for (int round = 0; round < maxRound; round++)
    {
        if (event_gen_count >= targetEventGen)
        {
            if (*pThrdMon->pFlagThreadTerminate == TERMINATE)
                break;
            else {
                sleep_for(std::chrono::milliseconds(500));
                continue;
            }
        }
        event_no = event_gen_count + NUM_EVENTS_PER_GEN * myAddr;
        event_priority = targetEventGen - event_gen_count - 1;
        event.setEventNo(event_no);
        event.setEventPri(event_priority);
        event.setEventGenAddr(myAddr);
        event.setEventHandlerAddr(-1); // event handler is not defined yet !!
        QueryPerformanceCounter(&t_gen);
        event.setEventGenTime(t_gen);
        event.setEventStatus(GENERATED);
        entry_event.setKey(event.getEventPri());
        entry_event.setValue(event);
```

```
/* Thread_EventGenenerator.cpp (4) */

        while (pPriQ_Event->insert(entry_event) == NULL)
        {
            pThrdParam->pCS_main->lock();
            cout << "PriQ_Event is Full, waiting ..." << endl;
            pThrdParam->pCS_main->unlock();
            sleep_for(std::chrono::milliseconds(100));
            /*
            pThrdParam->pCS_main->lock();
            cout << "Trying to insert an event into PriQ_Event " << endl;
            pThrdParam->pCS_main->unlock();
            sleep_for(std::chrono::milliseconds(100));
            */
        }
        pThrdParam->pCS_main->lock();
        cout << "Successfully inserted into PriQ_Event " << endl;
        pThrdParam->pCS_main->unlock();

        pThrdParam->pCS_thrd_mon->lock();
        pThrdMon->eventGenerated[pThrdMon->totalEventGenerated] = event;
        pThrdMon->numEventGenerated++;
        pThrdMon->totalEventGenerated++;
        pThrdParam->pCS_thrd_mon->unlock();
        event_gen_count++;
        //Sleep(100 + rand() % 300);
        sleep_for(std::chrono::milliseconds(10));
    }
}
```

Thread Event Handler

```
/* Thread_EventHandler.cpp (1) */
#include <Windows.h>
#include "Multi_Thread.h"
#include "HeapPrioQ_CS.h"
#include "Event.h"
using namespace std;
using std::this_thread::sleep_for;
void EventProc(ThreadParam_Event* pParam)
{
    ThreadParam_Event* pThrdParam;
    HeapPrioQ_CS<int, Event>* pPriQ_Event;
    int myRole;
    int myAddr;
    THREAD_FLAG* pFlagThreadTerminate;
    int maxRound;
    T_Entry<int, Event>* pEntry;
    Event event, * pEvent, * pEventProc;
    int event_no = 0;
    int eventPriority = 0;
    int event_gen_count = 0;
    int num_pkt_processed = 0;
    int targetEventGen;
    LARGE_INTEGER QP_freq, t_gen, t_proc;
    LONGLONG t_diff;
    double elapsed_time;
    ThreadStatusMonitor* pThrdMon;

    pThrdParam = (ThreadParam_Event*)pParam;
    myRole = pThrdParam->role;
    myAddr = pThrdParam->myAddr;
    pPriQ_Event = pThrdParam->pPriQ_Event;
    pThrdMon = pThrdParam->pThrdMon;
    maxRound = pThrdParam->maxRound;
    QP_freq = pThrdParam->QP_freq;
    targetEventGen = pThrdParam->targetEventGen;
```

```
/* Thread_EventHandler.cpp (2) */

    for (int round = 0; round < maxRound; round++)
    {
        if (*pThrdMon->pFlagThreadTerminate == TERMINATE)
            break;
        if (!pPriQ_Event->isEmpty())
        {
            pEntry = pPriQ_Event->removeHeapMin();
            event = pEntry->getValue();
            pThrdParam->pCS_thrd_mon->lock();
            //pThrdMon->ppEventsg[pThrdMon->numEventProcs] = pEvent;
            event.setEventHandlerAddr(myAddr);
            QueryPerformanceCounter(&t_proc);
            event.setEventProcTime(t_proc);
            t_gen = event.getEventGenTime();
            t_diff = t_proc.QuadPart - t_gen.QuadPart;
            elapsed_time = ((double)t_diff / QP_freq.QuadPart); // in second
            event.setEventElaspsedTime(elapsed_time * 1000); // in milli-second
            pThrdMon->eventProcessed[pThrdMon->totalEventProcessed] = event;
            pThrdMon->numEventProcessed++;
            pThrdMon->totalEventProcessed++;
            pThrdParam->pCS_thrd_mon->unlock();
        } // end if
        sleep_for(std::chrono::milliseconds(100 + rand() % 100));
    } // end for
}
```

Console Display for Thread Monitoring

```
/* ConsoleDisplay.h */
#ifndef CONSOLE_DISPLAY_H
#define CONSOLE_DISPLAY_H
#include <Windows.h>

HANDLE initConsoleHandler();
void cls(HANDLE hConsole);
void closeConsoleHandler(HANDLE hndlr);
int gotoxy(HANDLE consoleHandler, int x,
   int y);
#endif
```

```
/* ConsoleDisplay.cpp (1) */
#include <stdio.h>
#include "ConsoleDisplay.h"

HANDLE consoleHandler;
HANDLE initConsoleHandler()
{
    HANDLE stdCnslHndlr;
    stdCnslHndlr =
        GetStdHandle(STD_OUTPUT_HANDLE);
    consoleHandler = stdCnslHndlr;
    return consoleHandler;
}

void closeConsoleHandler(HANDLE hndlr)
{
    CloseHandle(hndlr);
}

int gotoxy(HANDLE consHndlr, int x, int y)
{
    if (consHndlr == INVALID_HANDLE_VALUE)
        return 0;
    COORD coords = { static_cast<short>(x),
     static_cast<short>(y) };
    SetConsoleCursorPosition(consHndlr, coords);
}
```

```
/* ConsoleDisplay.cpp (2) */

void cls(HANDLE hConsole)
{
        CONSOLE_SCREEN_BUFFER_INFO csbi;
        SMALL_RECT scrollRect;
        COORD scrollTarget;
        CHAR_INFO fill;

        // Get the number of character cells in the current buffer.
        if (!GetConsoleScreenBufferInfo(hConsole, &csbi))
        {
                return;
        }

        // Scroll the rectangle of the entire buffer.
        scrollRect.Left = 0;
        scrollRect.Top = 0;
        scrollRect.Right = csbi.dwSize.X;
        scrollRect.Bottom = csbi.dwSize.Y;

        // Scroll it upwards off the top of the buffer with a magnitude of the entire height.
        scrollTarget.X = 0;
        scrollTarget.Y = (SHORT)(0 - csbi.dwSize.Y);

        // Fill with empty spaces with the buffer's default text attribute.
        fill.Char.UnicodeChar = TEXT(' ');
        fill.Attributes = csbi.wAttributes;
```

```
/* ConsoleDisplay.cpp (3) */

        // Do the scroll
        ScrollConsoleScreenBuffer(hConsole, &scrollRect, NULL, scrollTarget, &fill);

        // Move the cursor to the top left corner too.
        csbi.dwCursorPosition.X = 0;
        csbi.dwCursorPosition.Y = 0;

        SetConsoleCursorPosition(hConsole, csbi.dwCursorPosition);
}
```

main() 함수에서 멀티스레드 진행 상황 파악

- **스레드 함수의 진행상황 파악을 위한 구조체 정의 및 변수 생성**
  - 스레드의 진행 상황을 파악하기 위한 ThreadStatusMonitor 구조체를 정의
  - main() 함수에서 ThreadStatusMonitor 구조체 변수를 생성하고, 그 구조체 변수의 주소를 스레드 생성의 인수로 전달
- **스레드의 진행 상황 기록**
  - 각 스레드는 진행상황 (예: 이벤트 생성, 이벤트 처리 등)을 ThreadStatusMonitor 구조체 변수에 기록
- **main() 함수에서의 주기적인 확인 및 콘솔 출력**
  - main() 함수에서 주기적으로 ThreadStatusMonitor 구조체 변수의 내용을 파악 및 분석
  - 전체 스레드의 진행 상황을 한 눈에 쉽게 볼 수 있도록 현황판 형식으로 콘솔에 출력

main() - Event Handling with PriQ

```
/* main_EventGen_PriQ_EventHandler.cpp  (1)  */

#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <thread>
#include <mutex>
#include "Multi_Thread.h"
#include "HeapPrioQ_CS.h"
#include "Event.h"
#include "ConsoleDisplay.h"
#include "SimParams.h"
#include <time.h>
#include <conio.h>
using namespace std;
```

```
/* main_EventGen_PriQ_EventHandler.cpp  (2)  */

    void main()
    {
        ofstream fout;
        LARGE_INTEGER QP_freq;
        double elapsed_time, min_elapsed_time, max_elapsed_time;
        double avg_elapsed_time, total_elapsed_time;
        HeapPrioQ_CS<int, Event> heapPriQ_Event(30, string("HeapPriorityQueue_Event"));
        Event *pEvent, *pEv_min_elasped_time, *pEv_max_elapsed_time;
        int myAddr = 0;
        int event_handler_addr, eventPriority;

        ThreadParam_Event thrdParam_EventGen, thrdParam_EventHndlr;
        HANDLE hThrd_EventGenerator, hThrd_EventHandler;
        mutex cs_main;
        mutex cs_thrd_mon;
        ThreadStatusMonitor thrdMon;
        HANDLE consHndlr;
        THREAD_FLAG eventThreadFlag = RUN;
        int count, numEventGenerated, numEventProcessed;
        int num_events_in_PrioQ;
        Event eventProcessed[TOTAL_NUM_EVENTS];

        fout.open("output.txt");
        if (fout.fail())
        {
            cout << "Fail to open output.txt file for results !!" << endl;
            exit;
        }
```

```
/* main_EventGen_PriQ_EventHandler.cpp  (3)  */

        consHndlr = initConsoleHandler();
        QueryPerformanceFrequency(&QP_freq);
        srand(time(NULL));

        thrdMon.pFlagThreadTerminate = &eventThreadFlag;
        thrdMon.totalEventGenerated = 0;
        thrdMon.totalEventProcessed = 0;
        for (int ev = 0; ev < TOTAL_NUM_EVENTS; ev++)
        {
            thrdMon.eventProcessed[ev].setEventNo(-1); // mark as not-processed
            thrdMon.eventProcessed[ev].setEventPri(-1);
        }

        /* Create and Activate Thread_EventHandler */

        thrdMon.numEventProcessed = 0;
        thrdParam_EventHndlr.role = EVENT_HANDLER;
        thrdParam_EventHndlr.myAddr = 1; // link address
        thrdParam_EventHndlr.pCS_main = &cs_main;
        thrdParam_EventHndlr.pCS_thrd_mon = &cs_thrd_mon;
        thrdParam_EventHndlr.pPriQ_Event = &heapPriQ_Event;
        thrdParam_EventHndlr.maxRound = MAX_ROUND;
        thrdParam_EventHndlr.QP_freq = QP_freq;
        thrdParam_EventHndlr.pThrdMon = &thrdMon;

        thread thrd_EvProc(EventProc, &thrdParam_EventHndlr);
        cs_main.lock();
        printf("Thread_EventProc is created and activated ...\n");
        cs_main.unlock();
```

```
/* main_EventGen_PriQ_EventHandler.cpp  (4)  */

        /* Create and Activate Thread_EventGen */
        thrdMon.numEventGenerated = 0;
        thrdParam_EventGen.role = EVENT_GENERATOR;
        thrdParam_EventGen.myAddr = 0; // my Address
        thrdParam_EventGen.pCS_main = &cs_main;
        thrdParam_EventGen.pCS_thrd_mon = &cs_thrd_mon;
        thrdParam_EventGen.pPriQ_Event = &heapPriQ_Event;
        thrdParam_EventGen.targetEventGen = NUM_EVENTS_PER_GEN;
        thrdParam_EventGen.maxRound = MAX_ROUND;
        thrdParam_EventGen.QP_freq = QP_freq;
        thrdParam_EventGen.pThrdMon = &thrdMon;

        thread thrd_EvGen(EventGen, &thrdParam_EventGen);
        cs_main.lock();
        printf("Thread_EventGen is created and activated ...\n");
        cs_main.unlock();

        /* periodic monitoring in main() */
        for (int round = 0; round < MAX_ROUND; round++)
        {
            cs_main.lock();
            cls(consHndlr);  // system("cls");
            gotoxy(consHndlr, 0, 0);

            printf("Thread monitoring by main() ::\n");
            printf("  round(%2d): current total_event_gen (%2d), total_event_proc(%2d)\n",
                round, thrdMon.totalEventGenerated, thrdMon.totalEventProcessed);

            printf("\n*****************************\n");
            numEventGenerated = thrdMon.numEventGenerated;
            printf("Events generated (current total = %2d)\n  ", numEventGenerated);
```

```
/* main_EventGen_PriQ_EventHandler.cpp  (5)  */

            count = 0;
            for (int ev = 0; ev < numEventGenerated; ev++)
            {
               pEvent = &thrdMon.eventGenerated[ev];
               if (pEvent != NULL)
               {
                  cout << *pEvent << " ";
                  if (((ev + 1) % EVENTS_PER_LINE) == 0)
                     printf("\n  ");
               }
            } //end for
            printf("\n");

            printf("\n*****************************\n");
            num_events_in_PrioQ = heapPriQ_Event.size();
            printf("Events currently in Priority_Queue (%d) : \n  ", num_events_in_PrioQ);
            heapPriQ_Event.fprint(cout);

            printf("\n\n*****************************\n");
            numEventProcessed = thrdMon.totalEventProcessed;
            printf("Events processed (current total = %d): \n  ", numEventProcessed);
            count = 0;
            total_elapsed_time = 0.0;
            for (int ev = 0; ev < numEventProcessed; ev++)
            {
               pEvent = &thrdMon.eventProcessed[ev];
               if (pEvent != NULL)
               {
                  pEvent->printEvent_proc();
                  if (((ev + 1) % EVENTS_PER_LINE) == 0)
                     printf("\n  ");
               }
```

```
/* main_EventGen_PriQ_EventHandler.cpp  (6)  */

                if (ev == 0)
                {
                    min_elapsed_time = max_elapsed_time = total_elapsed_time =
                        pEvent->getEventElapsedTime(); // in milli-second
                    pEv_min_elasped_time = pEv_max_elapsed_time = pEvent;
                }
                else
                {
                    if (min_elapsed_time > pEvent->getEventElapsedTime())
                    {
                        min_elapsed_time = pEvent->getEventElapsedTime(); // in milli-second
                        pEv_min_elasped_time = pEvent;
                    }
                    if (max_elapsed_time < pEvent->getEventElapsedTime())
                    {
                        max_elapsed_time = pEvent->getEventElapsedTime(); // in milli-second
                        pEv_max_elapsed_time = pEvent;
                    }
                    total_elapsed_time += pEvent->getEventElapsedTime();
                }
            } //end for showing eventProcessed
            printf("\n");

            if (numEventProcessed > 0)
            {
                printf("numEventProcessed = %d\n", numEventProcessed);
                printf("min_elapsed_time = %8.2lf[ms]; ", min_elapsed_time);
                cout << *pEv_min_elasped_time << endl;
                printf("max_elapsed_time = %8.2lf[ms]; ", max_elapsed_time);
                cout << *pEv_max_elapsed_time << endl;
                avg_elapsed_time = total_elapsed_time / numEventProcessed;
                printf("avg_elapsed_time = %8.2lf[ms]; \n", avg_elapsed_time);
            }
```

```
/* main_EventGen_PriQ_EventHandler.cpp  (7)  */

            if (numEventProcessed >= TOTAL_NUM_EVENTS)
            {
                eventThreadFlag = TERMINATE; // set 1 to terminate threads
                cs_main.unlock();
                break;
            } //end if
            cs_main.unlock();
            Sleep(100);
        } //end for (int round = 0; ...)

        thrd_EvProc.join();
        thrd_EvGen.join();

        fout.close();

        printf("Hit any key to terminate : ");
        _getch();
    }
```

실행 결과 (1)

```
Thread monitoring by main() ::
  round(13): current total_event_gen (41), total_event_proc(11)

*****************************
Events generated (current total = 41)
  Ev(no:  0, pri: 49) Ev(no:  1, pri: 48) Ev(no:  2, pri: 47) Ev(no:  3, pri: 46) Ev(no:  4, pri: 45)
  Ev(no:  5, pri: 44) Ev(no:  6, pri: 43) Ev(no:  7, pri: 42) Ev(no:  8, pri: 41) Ev(no:  9, pri: 40)
  Ev(no: 10, pri: 39) Ev(no: 11, pri: 38) Ev(no: 12, pri: 37) Ev(no: 13, pri: 36) Ev(no: 14, pri: 35)
  Ev(no: 15, pri: 34) Ev(no: 16, pri: 33) Ev(no: 17, pri: 32) Ev(no: 18, pri: 31) Ev(no: 19, pri: 30)
  Ev(no: 20, pri: 29) Ev(no: 21, pri: 28) Ev(no: 22, pri: 27) Ev(no: 23, pri: 26) Ev(no: 24, pri: 25)
  Ev(no: 25, pri: 24) Ev(no: 26, pri: 23) Ev(no: 27, pri: 22) Ev(no: 28, pri: 21) Ev(no: 29, pri: 20)
  Ev(no: 30, pri: 19) Ev(no: 31, pri: 18) Ev(no: 32, pri: 17) Ev(no: 33, pri: 16) Ev(no: 34, pri: 15)
  Ev(no: 35, pri: 14) Ev(no: 36, pri: 13) Ev(no: 37, pri: 12) Ev(no: 38, pri: 11) Ev(no: 39, pri: 10)
  Ev(no: 40, pri:  9)

*****************************
Events currently in Priority_Queue (31) :
  [Key: 8] [Key:26] [Key:17] [Key:32] [Key:27]
  [Key:23] [Key:18] [Key:39] [Key:33] [Key:31]
  [Key:28] [Key:35] [Key:24] [Key:22] [Key:19]
  [Key:49] [Key:43] [Key:46] [Key:34] [Key:47]
  [Key:40] [Key:42] [Key:29] [Key:48] [Key:38]
  [Key:44] [Key:25] [Key:45] [Key:36] [Key:37]
  [Key:20]

*****************************
Events processed (current total = 11):
  Ev(no: 8, pri:41, t_elapsed:   14.93) Ev(no:19, pri:30, t_elapsed:   14.87) Ev(no:28, pri:21, t_elapsed:    0.24) Ev(no:33, pri:16, t_elapsed:   14.96) Ev(no:34, pri:15, t_elapsed:  179.45)
  Ev(no:35, pri:14, t_elapsed:  298.95) Ev(no:36, pri:13, t_elapsed:  255.40) Ev(no:37, pri:12, t_elapsed:  285.24) Ev(no:38, pri:11, t_elapsed:  164.57) Ev(no:39, pri:10, t_elapsed:  314.14)
  Ev(no:40, pri: 9, t_elapsed:  165.06)
numEventProcessed = 11
min_elapsed_time =    0.24[ms]; Ev(no: 28, pri: 21)
max_elapsed_time =  314.14[ms]; Ev(no: 39, pri: 10)
avg_elapsed_time =  155.26[ms];
Successfully inserted into PriQ_Event
```

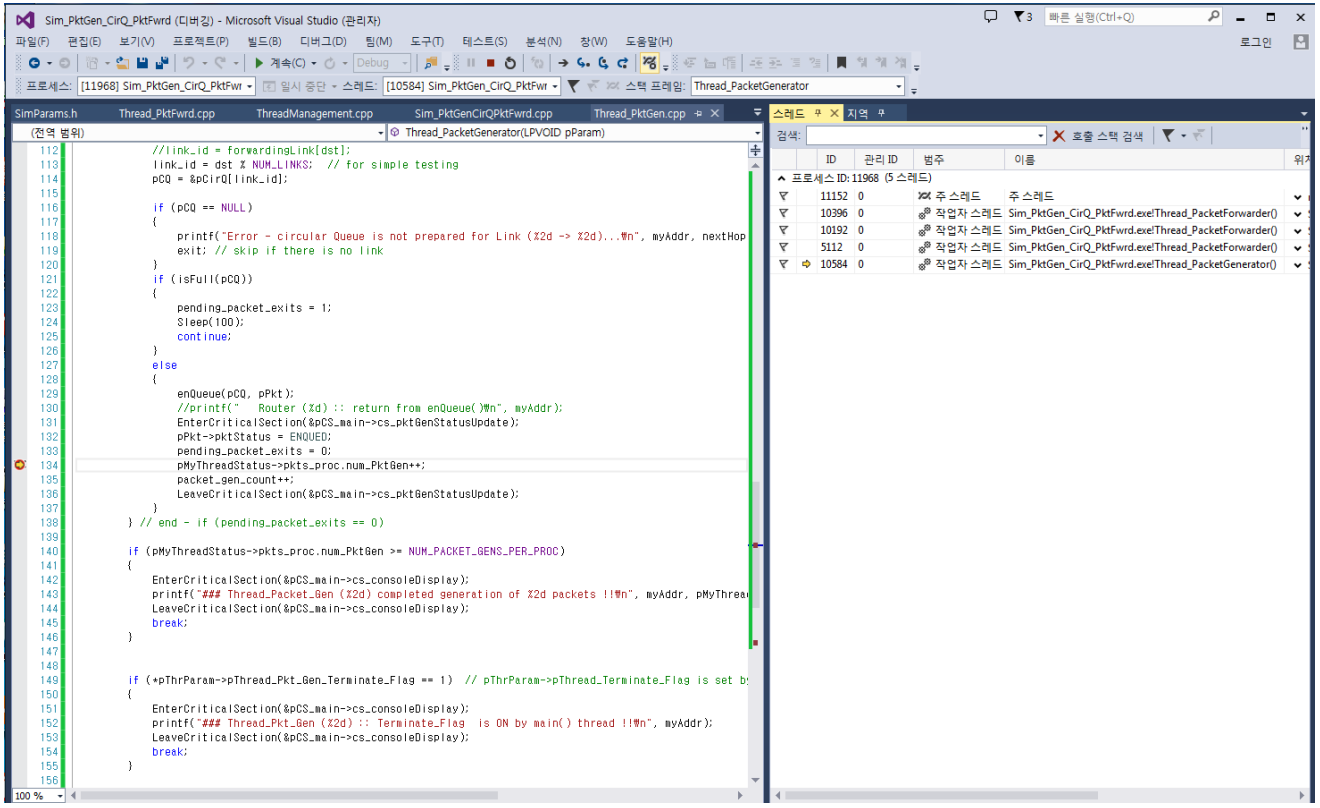
실행 결과 (2)

```
Thread monitoring by main() ::
  round(59): current total_event_gen (50), total_event_proc(50)

*****************************
Events generated (current total = 50)
  Ev(no:  0, pri: 49) Ev(no:  1, pri: 48) Ev(no:  2, pri: 47) Ev(no:  3, pri: 46) Ev(no:  4, pri: 45)
  Ev(no:  5, pri: 44) Ev(no:  6, pri: 43) Ev(no:  7, pri: 42) Ev(no:  8, pri: 41) Ev(no:  9, pri: 40)
  Ev(no: 10, pri: 39) Ev(no: 11, pri: 38) Ev(no: 12, pri: 37) Ev(no: 13, pri: 36) Ev(no: 14, pri: 35)
  Ev(no: 15, pri: 34) Ev(no: 16, pri: 33) Ev(no: 17, pri: 32) Ev(no: 18, pri: 31) Ev(no: 19, pri: 30)
  Ev(no: 20, pri: 29) Ev(no: 21, pri: 28) Ev(no: 22, pri: 27) Ev(no: 23, pri: 26) Ev(no: 24, pri: 25)
  Ev(no: 25, pri: 24) Ev(no: 26, pri: 23) Ev(no: 27, pri: 22) Ev(no: 28, pri: 21) Ev(no: 29, pri: 20)
  Ev(no: 30, pri: 19) Ev(no: 31, pri: 18) Ev(no: 32, pri: 17) Ev(no: 33, pri: 16) Ev(no: 34, pri: 15)
  Ev(no: 35, pri: 14) Ev(no: 36, pri: 13) Ev(no: 37, pri: 12) Ev(no: 38, pri: 11) Ev(no: 39, pri: 10)
  Ev(no: 40, pri:  9) Ev(no: 41, pri:  8) Ev(no: 42, pri:  7) Ev(no: 43, pri:  6) Ev(no: 44, pri:  5)
  Ev(no: 45, pri:  4) Ev(no: 46, pri:  3) Ev(no: 47, pri:  2) Ev(no: 48, pri:  1) Ev(no: 49, pri:  0)


*****************************
Events currently in Priority_Queue (0) :
  HeapPriorityQueue is Empty !!


*****************************
Events processed (current total = 50):
  Ev(no: 9, pri:40, t_elapsed:    0.02) Ev(no:19, pri:30, t_elapsed:   15.56) Ev(no:27, pri:22, t_elapsed:   15.56) Ev(no:33, pri:16, t_elapsed:   15.30) Ev(no:34, pri:15, t_elapsed:  181.14)
  Ev(no:35, pri:14, t_elapsed:  195.51) Ev(no:36, pri:13, t_elapsed:  256.52) Ev(no:37, pri:12, t_elapsed:  300.83) Ev(no:38, pri:11, t_elapsed:  242.72) Ev(no:39, pri:10, t_elapsed:  287.08)
  Ev(no:40, pri: 9, t_elapsed:  166.21) Ev(no:41, pri: 8, t_elapsed:  197.23) Ev(no:42, pri: 7, t_elapsed:  256.54) Ev(no:43, pri: 6, t_elapsed:  257.69) Ev(no:44, pri: 5, t_elapsed:  152.14)
  Ev(no:45, pri: 4, t_elapsed:  333.34) Ev(no:46, pri: 3, t_elapsed:  272.30) Ev(no:47, pri: 2, t_elapsed:  287.76) Ev(no:48, pri: 1, t_elapsed:  181.66) Ev(no:49, pri: 0, t_elapsed:  151.08)
  Ev(no:32, pri:17, t_elapsed: 2797.46) Ev(no:31, pri:18, t_elapsed: 2919.16) Ev(no:30, pri:19, t_elapsed: 3039.69) Ev(no:29, pri:20, t_elapsed: 3237.35) Ev(no:28, pri:21, t_elapsed: 3448.09)
  Ev(no:26, pri:23, t_elapsed: 3676.02) Ev(no:25, pri:24, t_elapsed: 3827.05) Ev(no:24, pri:25, t_elapsed: 3962.10) Ev(no:23, pri:26, t_elapsed: 4096.92) Ev(no:22, pri:27, t_elapsed: 4323.02)
  Ev(no:21, pri:28, t_elapsed: 4488.90) Ev(no:20, pri:29, t_elapsed: 4639.90) Ev(no:18, pri:31, t_elapsed: 4851.13) Ev(no:17, pri:32, t_elapsed: 5017.19) Ev(no:16, pri:33, t_elapsed: 5212.96)
  Ev(no:15, pri:34, t_elapsed: 5348.90) Ev(no:14, pri:35, t_elapsed: 5531.42) Ev(no:13, pri:36, t_elapsed: 5758.35) Ev(no:12, pri:37, t_elapsed: 5909.41) Ev(no:11, pri:38, t_elapsed: 6121.19)
  Ev(no:10, pri:39, t_elapsed: 6241.40) Ev(no: 8, pri:41, t_elapsed: 6392.39) Ev(no: 7, pri:42, t_elapsed: 6544.42) Ev(no: 6, pri:43, t_elapsed: 6694.82) Ev(no: 5, pri:44, t_elapsed: 6889.63)
  Ev(no: 4, pri:45, t_elapsed: 7071.83) Ev(no: 3, pri:46, t_elapsed: 7237.39) Ev(no: 2, pri:47, t_elapsed: 7373.02) Ev(no: 1, pri:48, t_elapsed: 7552.56) Ev(no: 0, pri:49, t_elapsed: 7758.17)

numEventProcessed = 50
min_elapsed_time =    0.02[ms]; Ev(no:  9, pri: 40)
max_elapsed_time =  7758.17[ms]; Ev(no:  0, pri: 49)
avg_elapsed_time =  3234.56[ms];
```

Debugging of Multi-Thread Operations

◆ Visual Studio Multi-thread Information

- Debug tab -> Window -> Thread(H)
- "Cntl+ALT+H"



Homework 9

Homework 9

9.1 Critical section (임계구역) 설정이 필요한 이유에 대하여 구체적으로 설명하고, mutex 사용하여 어떻게 임계 구역을 관리하는지에 대하여 예를 들어 설명하라.

9.2 Multi-thread에 파라메터를 전달하는 방법에 대하여 구체적으로 설명하라. 특히, Event의 생성에서 처리까지 경과된 시간을 측정하기 위하여 각 스레드에 어떤 파라메터를 전달하고, 각 스레드가 어떤 처리를 하여야 하는지에 대하여 설명하라.

9.3 Multi-thread의 동작 상태를 monitoring하여, 주기적으로 상태를 출력하는 함수를 구상하고, 필요한 파라메터 전달, 출력 포맷에 따른 주기적인 출력 방법에 대하여 설명하라.

9.4 동적으로 할당된 배열을 기반으로 구현된 Circular Queue의 기본 기능인 enQueue()와 deQueue() 함수의 기본 구조 및 동작 원리에 대하여 그림으로 나타내어 설명하라.

9.5 우선 순위를 고려한 Event처리를 위하여 사용되는 Priority Queue에서 우선 순위가 높은 event가 우선적으로 처리될 수 있는 구조와 동작 원리에 대하여 설명하라.