프로그래밍언어

13. 다중 스레드 (Multi-thread)

교수 김 영 탁

영남대학교 정보통신공학과

(Tel : +82-53-810-2497; Fax : +82-53-810-4742

http://antl.yu.ac.kr/; E-mail : ytkim@yu.ac.kr)

Outline

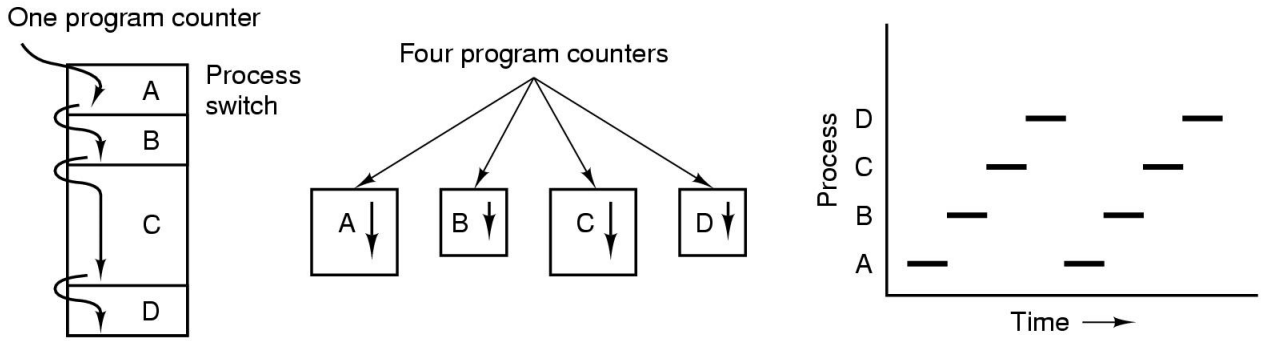
- Process vs. Thread
- Simple Three Threads
  - main (thread_M), thread_A, thread_B
- Simple Three Threads with Critical Section controlled by mutex
- Simple Three Threads with Critical Section and Turn
- Simple Two Threads with Circular Queue / mutex for Event Handling
- Simple Two Threads with Priority Queue / mutex for Event Handling with Priority

프로세스 (process)와 스레드 (thread)

프로세스 (Process)

◆ Process

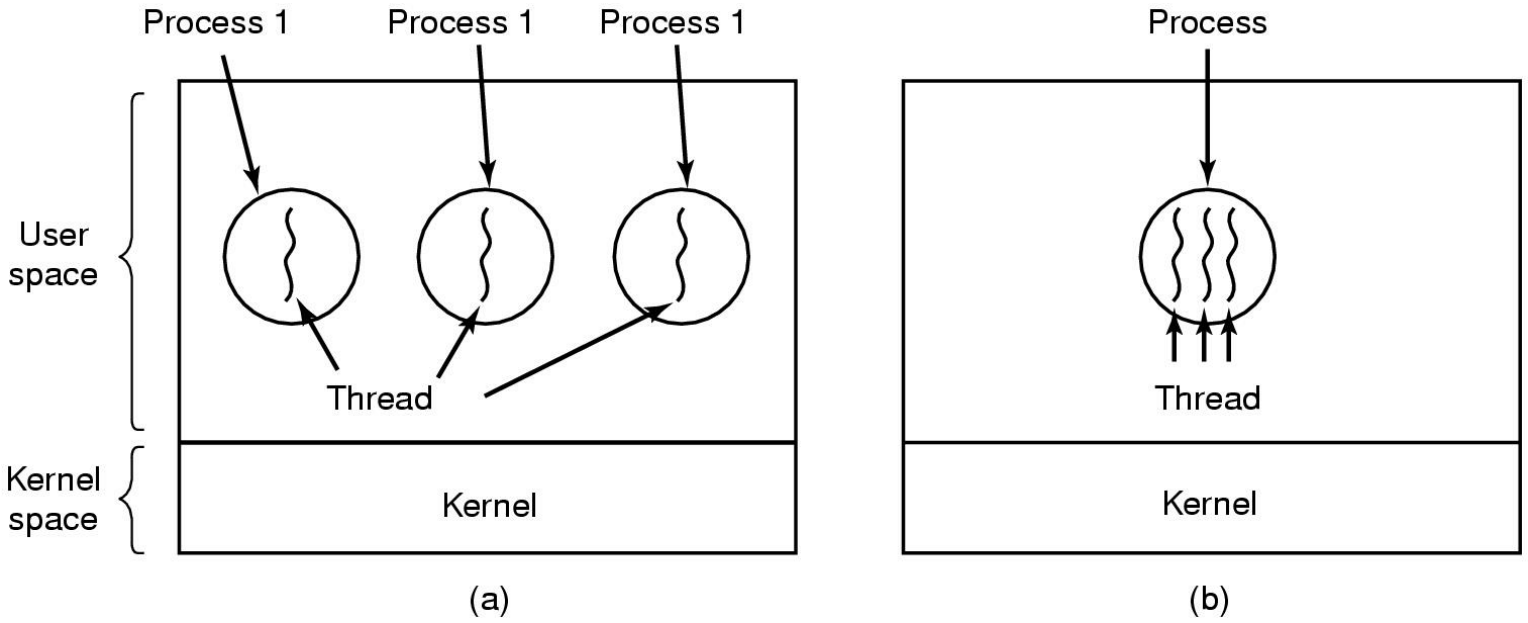
- 프로세스 (process)란 프로그램이 수행중인 상태 (***program in execution***)
  - 각 프로세스 마다 개별적으로 메모리가 할당 됨 (text core, initialized data (BSS), non-initialized data, heap (dynamically allocated memory), stack)
- 일반적인 PC나 대부분의 컴퓨터 환경에서 하나의 물리적인 CPU 상에 다수의 프로세스가 실행되는 ***Multi-tasking*** 이 지원되며, 운영체제가 다수의 프로세스를 일정 시간 마다 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)

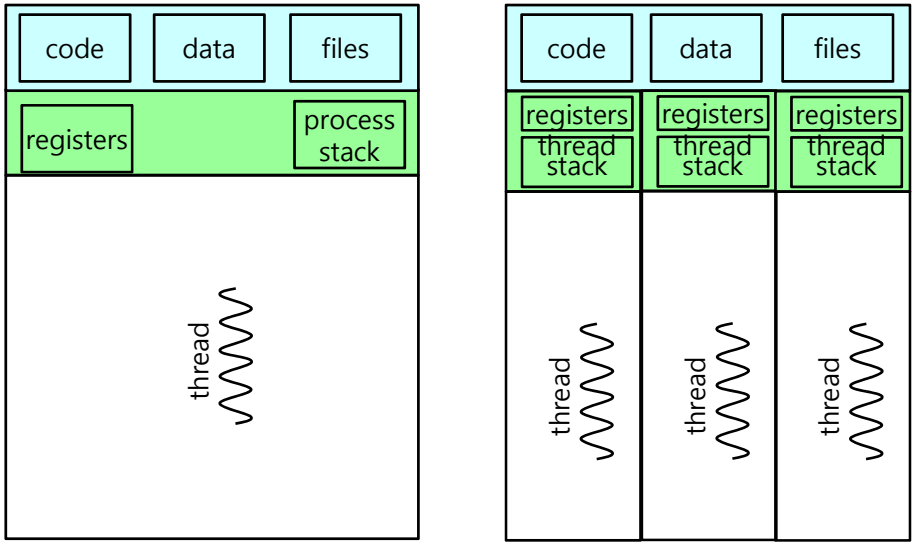
- 스레드는 하나의 프로세스 내부에 포함되는 함수들이 동시에 실행될 수 있게 한 작은 단위 경량 프로세스 (lightweight process)
- 기본적으로 CPU를 사용하는 기본 단위
- 하나의 프로세스에 포함된 다수의 스레드 들은 프로세스의 메모리 자원들 (code section, data section, Heap 등)과 운영체제의 자원들 (예: 파일 입출력 등)을 공유함



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

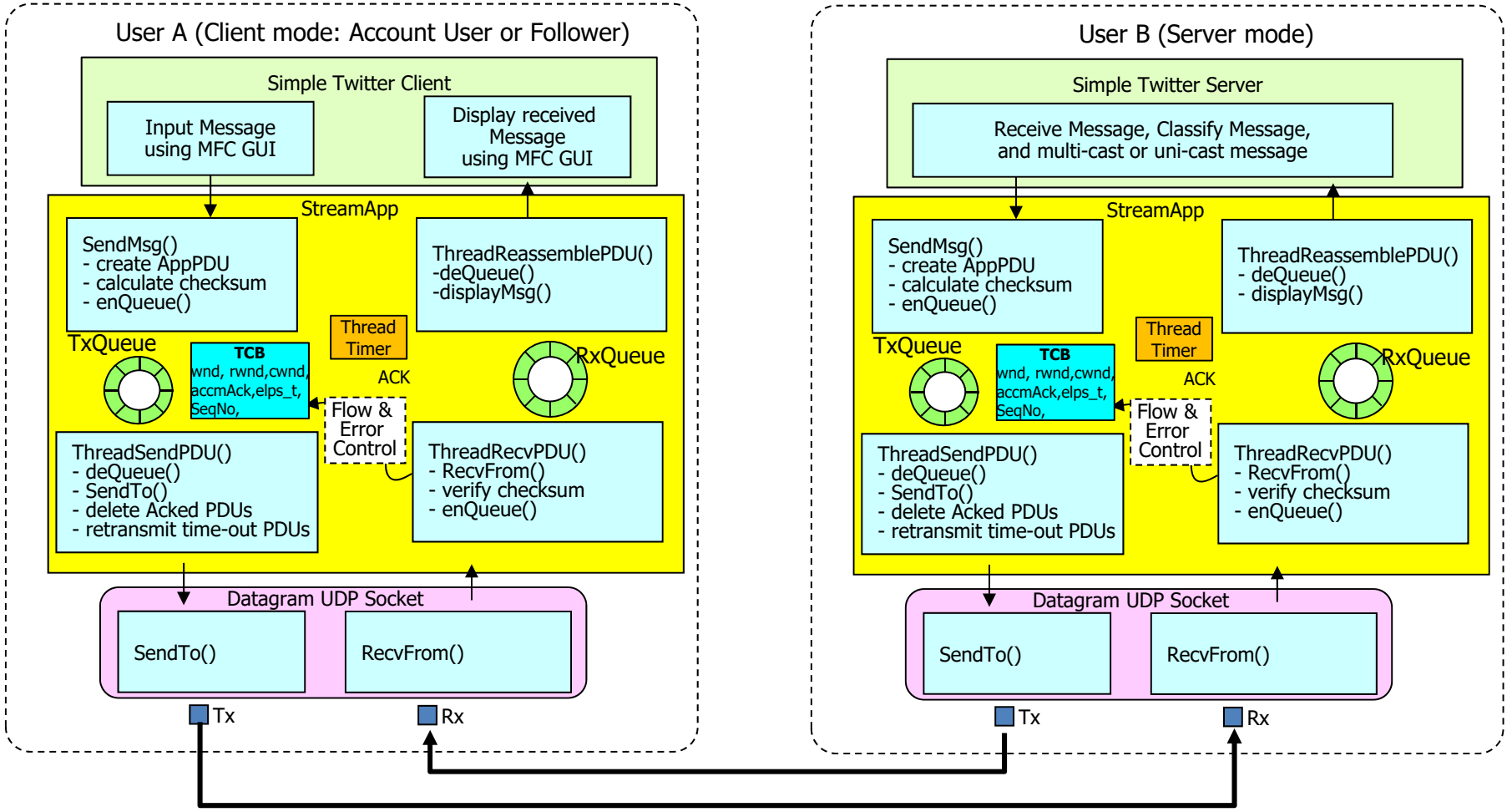


(a) single-thread process (b) multi-thread process

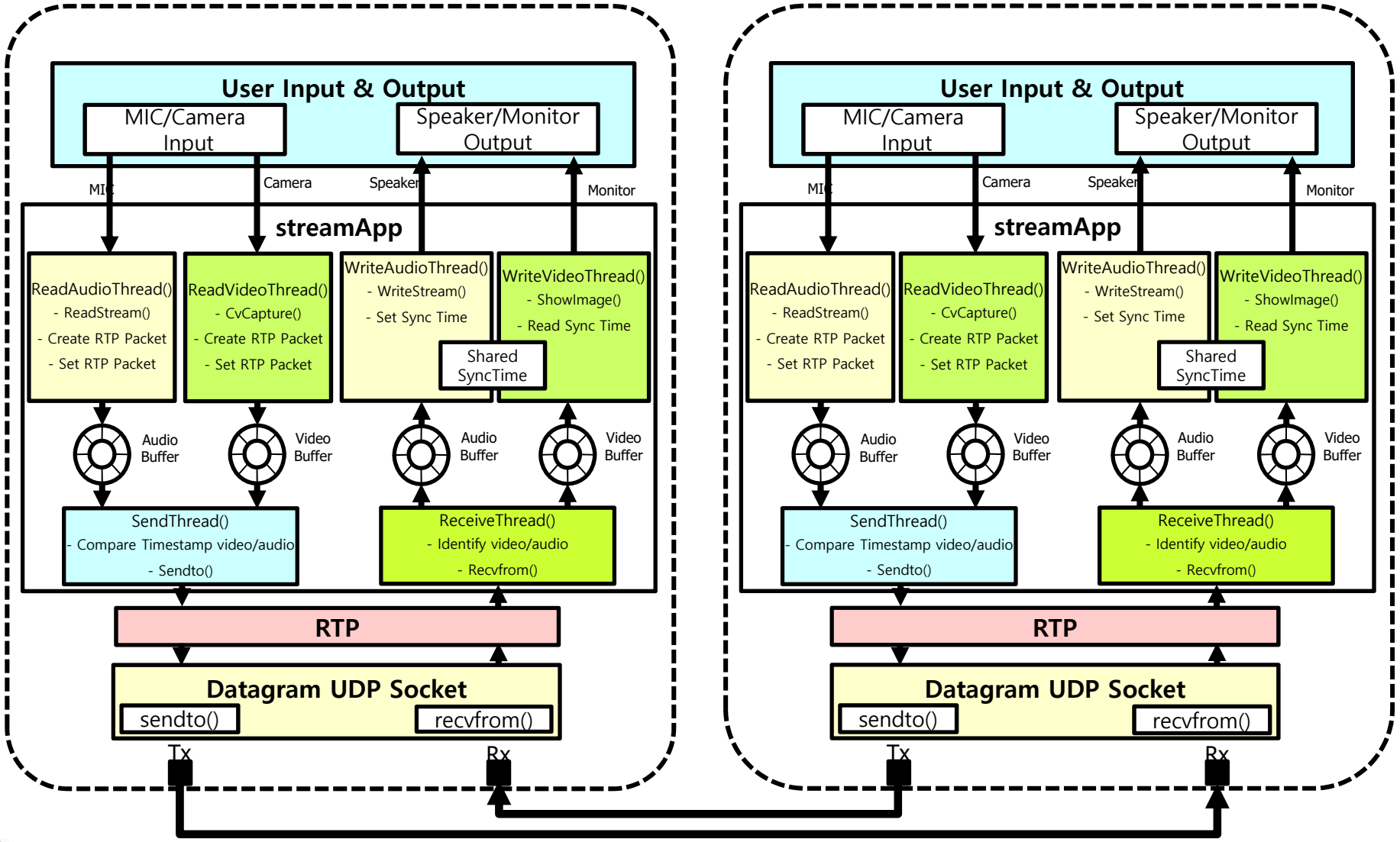
Task 수행이 병렬로 처리되어야 하는 경우

- **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

◆ 양방향 동시 전송 (full-duplex) Text Twitter



◆ 실시간 영상/화상 전화기의 기능 블럭도



멀티스레드 관련 라이브러리 함수

멀티스레드 관련 Windows 라이브러리 함수

◆ 스레드 관련 Windows 라이브러리 함수

스레드 관련 라이브러리 함수	설명
CreateThread()	스레드를 생성시킴. 개선된 함수로 _beginthreadex() 함수를 사용하는 것을 권고함.
_beginthreadex()	스레드를 생성시킴. CreateThread()의 보안 문제를 개선하였으며, 내부적으로 CreateThread()를 호출.
TerminateThread()	스레드를 강제 종료시킴. 특별한 상황이 아니면 스레드에 할당된 자원들이 정상적으로 반환될 수 있도록 TerminateThread()를 사용하지 않아야 함. _endthreadex()를 사용하는 것을 권고함.
_endthreadex()	할당된 자원을 반환한 후, ExitThread()를 호출하여 스레드를 종료시킴.
GetExitCodeThread()	스레드의 종료하면서 반환하는 ExitCode를 받아오며, 이를 사용하여 스레드의 종료 상태를 확인할 수 있음.
WaitForSingleObject()	스레드의 종료를 기다림.
ExitThread()	특정 위치에서 스레드를 종료시키고자 할 때 사용.
SuspendThread()	스레드를 일시 정지 시켜 Blocked 상태로 둠.
ResumeThread()	일시정지되어 있는 스레드를 다시 실행하도록 Ready 상태로 변경함.
Sleep()	스레드를 일정한 기간 동안 (millisecond로 지정) Blocked 상태로 정지시킴.
CloseHandle()	스레드 관리에 사용되었던 핸들을 소멸시킴.

C++11의 멀티스레드 관련 클래스 및 멤버함수

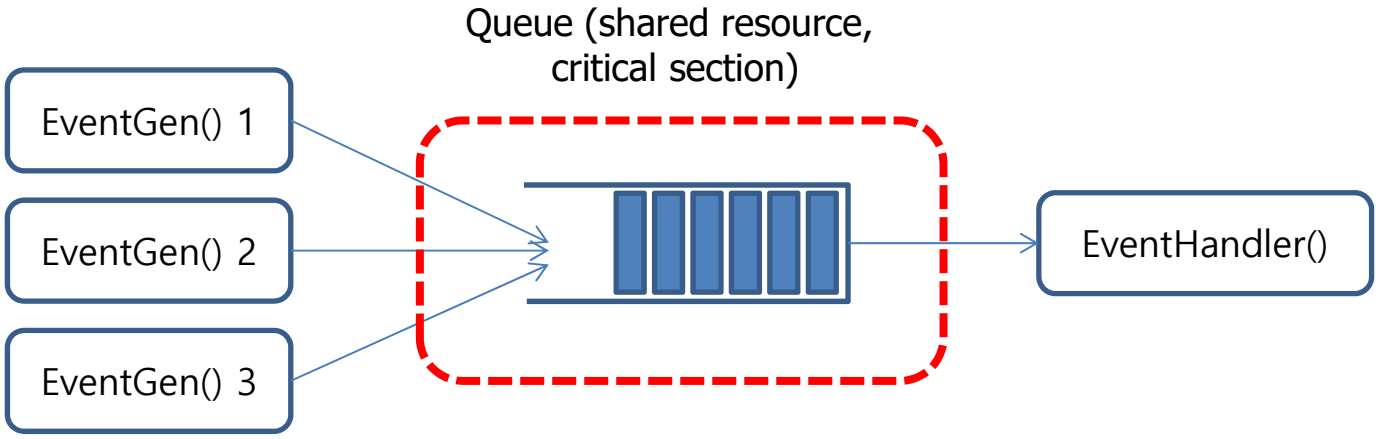
◆C++11의 스레드 관련 클래스 및 멤버 함수

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(); (Note: thread_id is not integer typ, but std::thread::id which is to be used for comparisons only, not for arithmetic).
sleep_for(sleep_duration)	지정된 시간 만큼 스레드 실행을 중지 (sleep)
_sleep(sleep_duration_ms)	Windows 운영체제에서 제공하는 API 함수 (#include <Windows.h> 필요) sleep_duration_ms은 milli-second 단위

멀티스레드 프로그래밍의 고려사항
- 스레드 간의 공유자원과 임계구역

◆ 공유자원의 예: 스레드간 정보 전달을 위한 queue

- 스레드 간에 정보/메시지/신호를 전달하기 위하여 큐 (예: Circular Queue, Priority Queue)를 사용
- Queue의 end에 정보를 추가하는 enqueue() 실행 동안 queue의 내부 변수 변경
- Queue의 front에 있는 정보를 추출하는 dequeue() 실행 동안 queue의 내부 변수 변경
- Queue는 다수의 스레드가 공유하는 자원 (shared resource) 이며, 임계구역 (critical section)으로 보호되어야 함



임계구역 (Critical Section)과 Mutex

◆ 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

◆ Mutex (mutual exclusion)

- 현재 어떤 스레드/프로세스가 임계구역 내에서 실행 중에 있다는 상태를 표시하기 위하여 mutex 변수를 사용
- semaphore라고도 함
- C++ 프로그래밍에서는 mutex를 사용

철도의 구식 semaphore

철도의 현대 semaphore

semaphore가 부착된 우편함

C++의 mutex 변수

◆ C++ 환경에서 mutex 변수의 설정

- **#include <mutex>**
  using namespace std;
  mutex mtx; // mutex (semaphore) 변수 설정/생성
  // mutex의 lock() 및 unlock() 실행 이전에 생성되어 있어야 함

◆ mutex를 사용한 critical section 영역 지정

- mtx.lock();
  - 공유 자원의 임계구역 (critical section) 진입을 표시
  - 이미 다른 thread가 공유자원에 먼저 진입되어 있는 경우, 그 thread가 임계구역을 벗어날때까지 대기하게 됨
- mtx.unlock()
  - 공유 자원의 임계구역 (critical section)을 벗어나는 것을 표시

임계구역 (Critical Section)

◆ Critical Section (임계구역)의 설정 필요성 예

- 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.  }
```


```
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		

mutex를 사용한 Critical Section 설정 및 관리

◆ mutex를 사용한 임계구역 (critical section) 설정 및 관리

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

6.      l_account =
            l_account + deposit;

7.      g_account = l_account;
8.      pMTX->unlock();
9.      print(l_account);
10.     . . . .
11. }
```

shared resource

은행잔고
g_account

mutex
mtx_acct

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

6.      l_account =
            l_account - withdraw;

7.      g_account = l_account;
8.      pMTX->unlock();
9.      print(l_account);
10.     . . . .
11. }
```

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

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

using namespace std;

typedef struct
{
        mutex *pMTX;
        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"
using namespace std;

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
{
    CirQ *pCirQ;
    mutex* pMTX;
    int role;
} ThreadParam;
```

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

```
typedef struct
{
        FILE *fout; // for log file
        int id;
        mutex *pMTX_main;
        ThreadStatus *pThreadStatus;
        Event *pGeneratedEvents;
        Event *pProcessedEvents;
        CirQ *pCirQ; // pointer array for circular queue
        int target_num_events;
        ROLE role; // generator or handler
        UINT_32 myAddr;
        int max_CirQ_capa;
        int *pThread_EventGen_Terminate_Flag;
        int *pThread_EventProc_Terminate_Flag;
        int max_rounds;
        HANDLE consoleHandler;
} ThreadParam;
```

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
using namespace std; // for mutex

void simpleThread(ThreadParam *pParam);

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

스레드의 생성 및 종료 (2)

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

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

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

    simThrd.join(); // wait for thread termination

    mtx_console.lock();
    printf("main() : Thread (%d) is terminated now.\n",  thread_id);
    mtx_console.unlock();
} // end main()
```

스레드의 생성 및 종료 (3)

◆ 스레드가 스스로 종료할 때 까지 기다리는 경우

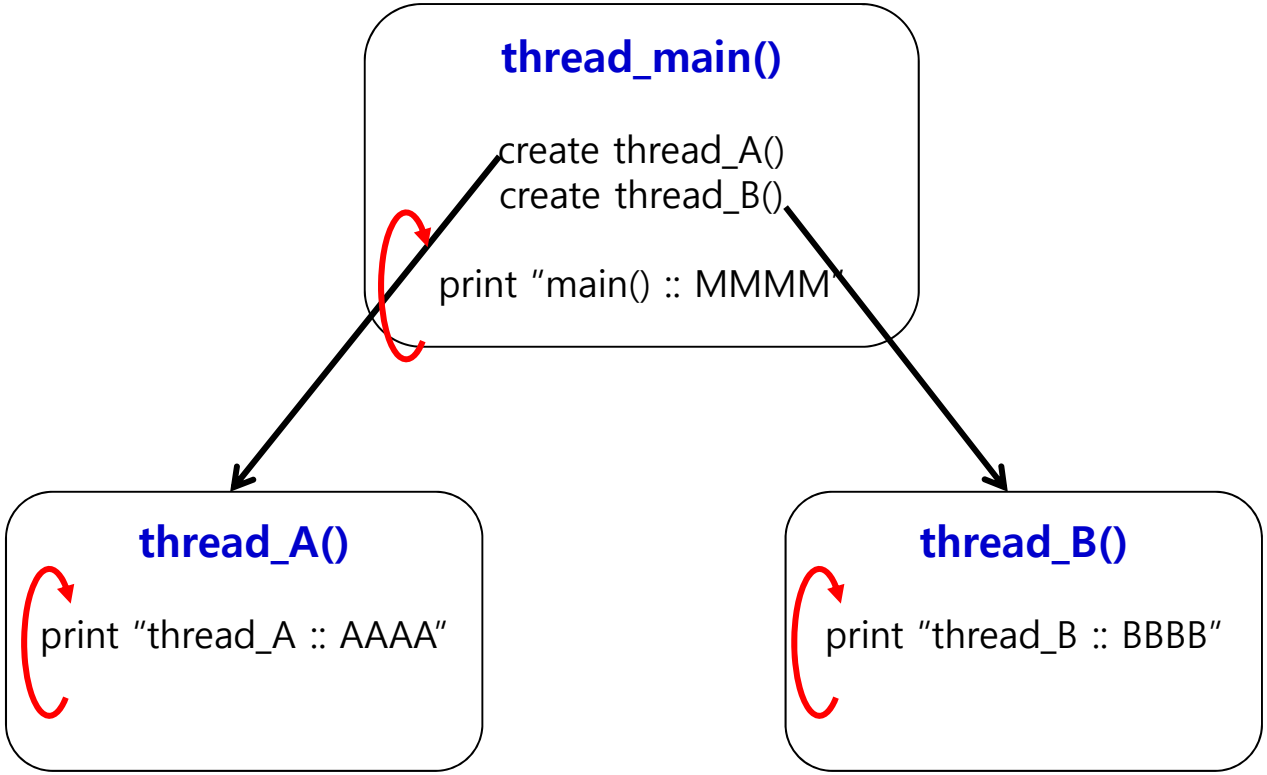
- main() 함수에서는 스레드가 종료할 때 까지 기다림

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

Simple Three Threads 예제

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>
using namespace std; // for mutex
typedef struct
{
    char mark;
} ThreadParam;
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 < 10; i++)
   {
      printf("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 < 10; 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 < 10; 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

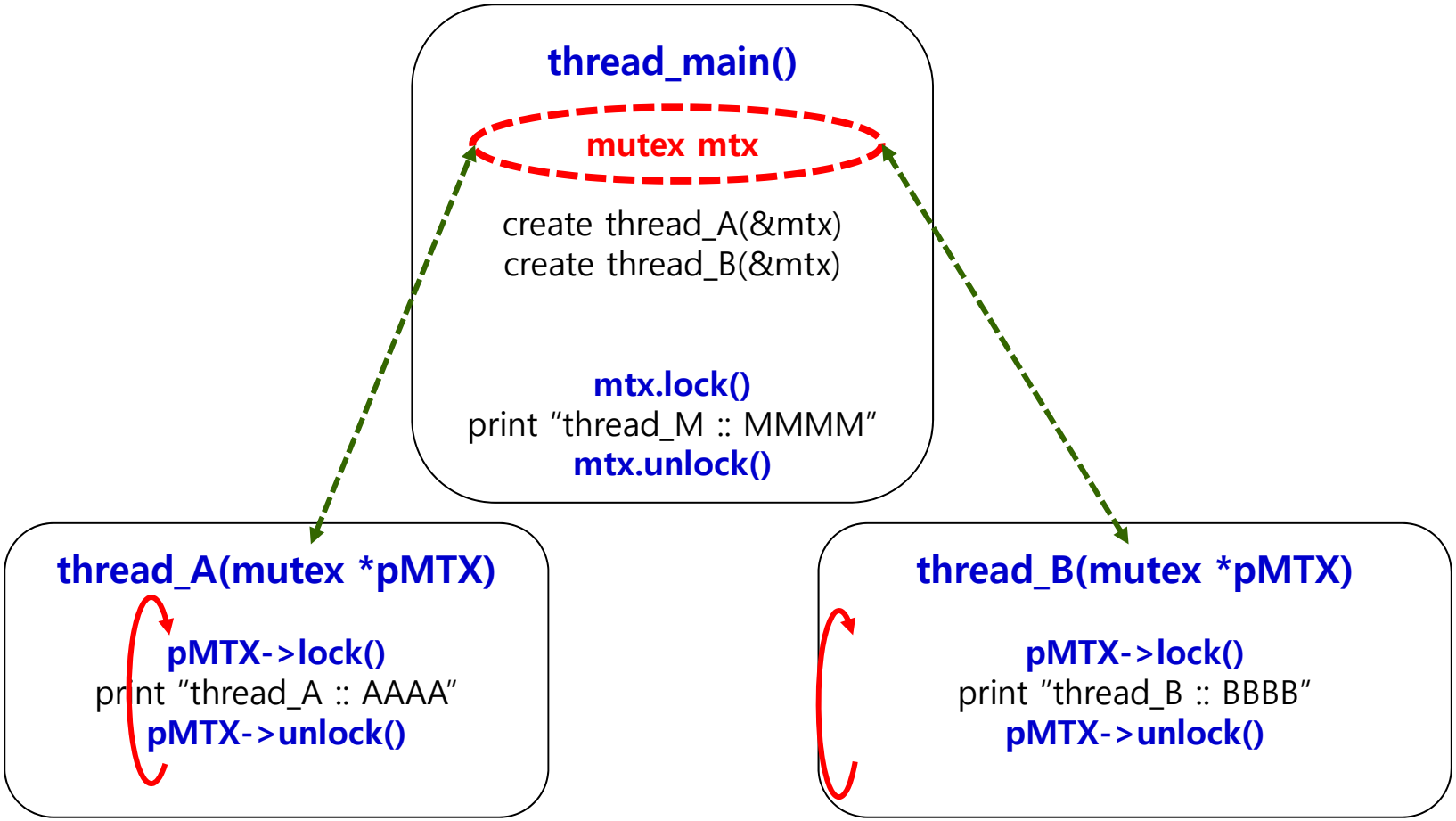
```
main() : Thread_A() is successfully created !
main() : Thread_B() is successfully created !
Thread_B() is activated !
Thread_B() : BBBBBBBBBBBBBround (  0)
main() : MMMMMMMMMMMMMMMMMMMBBBBBBBBThread_A() is activated !
Thread_A() : AAAAAAAABBBBAAAABBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
MMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAThread_B() : BBBBBBBBAAABBAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A() : AThread_B() : BAABBBBAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A() : Thread_B() : BAABBBBABBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A() : AAAAAAAAAAAAThread_B() : BBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B() : BBBBThread_A() : BBBABBBBBABAAAAABBAAABBABAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B() : BBThread_A() : AAABBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A() : AThread_B() : BABBBAAAAAABBABAAABBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B is terminating ...
Thread_A is terminating ...
```

◆ 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

Three Simple Threads – Version 2

◆ Three simple threads using mutex



```
/* SimpleThreadsVer2.cpp  (1) */
#include<stdio.h>
#include <thread>
#include <mutex>
#include<time.h>
using namespace std; // for mutex
typedef struct
{
    mutex* pMTX; // pSemaphore
    char mark;
}ThreadParam;
void Thread_A(ThreadParam *pParam);
void Thread_B(ThreadParam *pParam);

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

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

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

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

    /* main() thread 실행 */
    char mark = 'M';
    for (int i = 0; i < 10; i++)
    {
        mtx.lock();
        printf("main() : ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("\n");
        mtx.unlock();
    }
    thrd_A.join();
    thrd_B.join();
}
```

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

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

   for (int i = 0; i < 10; i++)
   {
      pMTX->lock();
      printf("Thread_A() : ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);
      }
      printf("\n");
      pMTX->unlock();
      Sleep(100);
   }
   pMTX->lock();
   printf("Thread_A finished ...\n");
   pMTX->unlock();
}
```

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

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

   for (int i = 0; i < 10; i++)
   {
      pMTX->lock();
      printf("Thread_B() : ");
      for (int j = 0; j < 50; j++)
      {
         printf("%c", mark);
      }
      printf("\n");
      pMTX->unlock();
      Sleep(100);
   }
   pMTX->lock();
   printf("Thread_B finished ...\n");
   pMTX->unlock();
}
```

◆ 실행결과 (ver 2)

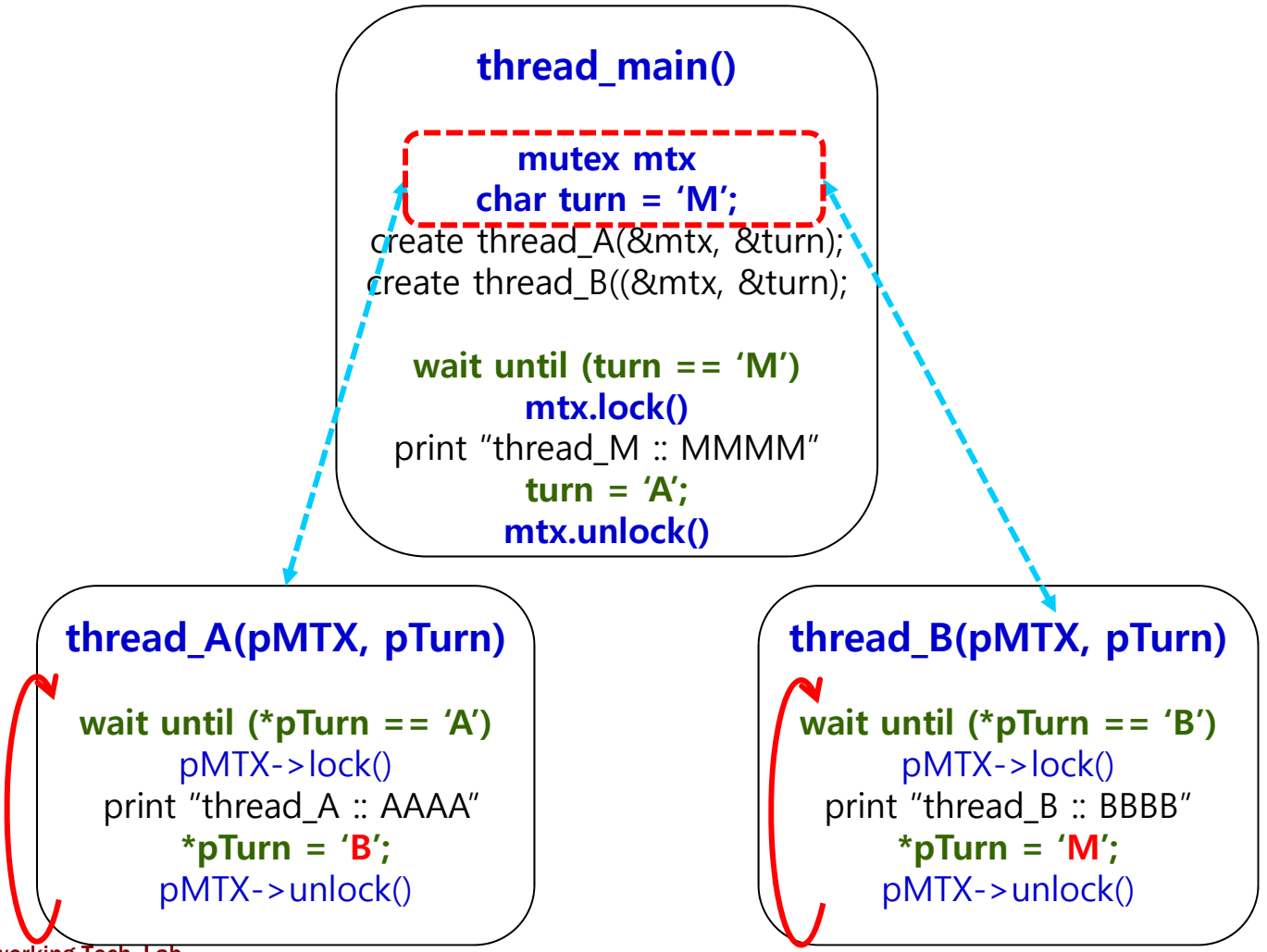
- critical section을 사용하여, 하나의 스레드가 한 줄의 출력을 완전히 완료할 때까지 다른 스레드가 실행되지 못하게 보호
- thread_A 또는 thread_B가 두 줄씩 출력하는 경우가 있음

```
main() : Thread_A() is successfully created !
main() : Thread_B() is successfully created !
round (  0)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() is activated !
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() is activated !
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  1)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  2)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
round (  3)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
round (  4)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
round (  5)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
round (  6)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
round (  7)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
round (  8)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  9)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A is terminating ...
Thread_B is terminating ...
main() : Waiting the Thread_A to terminate by itself ...
main() : Thread_A is terminated now.
main() : Waiting the Thread_B to terminate by itself ...
main() : Thread_B is terminated now.
```

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>
using namespace std;

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

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

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

    /* Thread_A에 전달 될 파라미터값 초기화 */
    thrParam_A.pMTX = &mtx_console;
    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.pMTX = &mtx_console;
    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);
        mtx_console.lock();
        printf("round(%3d) : \n", round);
        printf("main () : ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("\n");
        turn = 'A';
        mtx_console.unlock();
    }
    thrd_A.join(); /* thrd_A가 종료할 때 까지 대기 */
    thrd_B.join(); /* thrd_B가 종료할 때 까지 대기 */

}
```

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

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

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

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

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

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

◆ 실행결과 (ver 3)

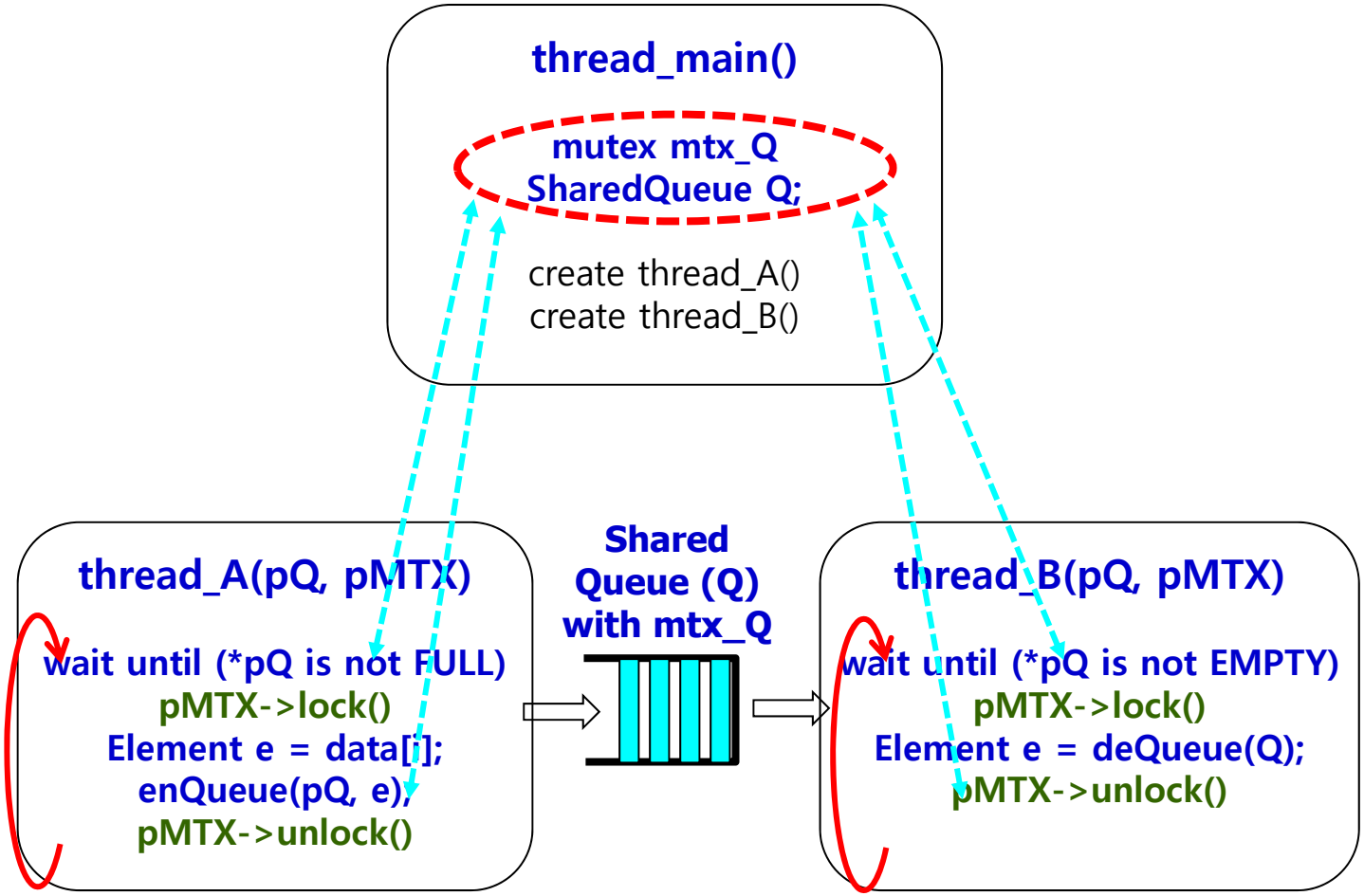
- mutex과 함께 turn semaphore를 사용
- 하나의 스레드가 한 줄의 출력을 완료한 후, 다른 스레드가 출력을 완료할 때 까지 계속 기다리게 함
- main, thread_A, thread_B가 한번에 한 줄씩만 출력하며, 번갈아 가며 출력

```
main() : Thread_A() is successfully created !
main() : Thread_B() is successfully created !
round (  0)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() is activated !
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() is activated !
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  1)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  2)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  3)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  4)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  5)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  6)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  7)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  8)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
round (  9)
main() : MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
main() : Waiting the Thread_A to terminate by itself ...
Thread_A() : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A is terminating ...
Thread_B() : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
main() : Thread_A is terminated now.
main() : Waiting the Thread_B to terminate by itself ...
Thread_B is terminating ...
main() : Thread_B is terminated now.
```

멀티스레드 간 정보 전달을 위한
큐 구성 및 활용

Multi-threads using shared Queue

◆ Two threads are sharing a Queue



Shared Queue의 구현 방법 (1)

◆ Circular Queue (CirQ)

- array 기반의 구현
- front, end가 저장된 데이터를 가르킴
  - front와 end의 초기값은 0
  - index 값은 0 ~ CAPACITY-1 범위의 값을 가지며, CAPACITY-1 이후에는 0으로 순환됨
- enQueue()
  - cirQ->data[end] = element;
  - cirQ-> end = (cirQ-> end + 1) % CAPACITY;
  - cirQ->num_data++;
- deQueue()
  - element = cirQ->data[front];
  - cirQ-> front = (cirQ-> front + 1) % CAPACITY;
  - cirQ->num_data--;

Shared Queue의 구현 방법 (2)

◆ List Queue (연결형 리스트 큐)

- Doubly Linked List 기반의 구현
- LinkNode, LinkedList 구조체 필요
- enQueue()에서는 현재의 *pEnd 뒤에 새로운 list node를 추가
- deQueue()에서는 현재의 *pFront 노드를 읽고, remove

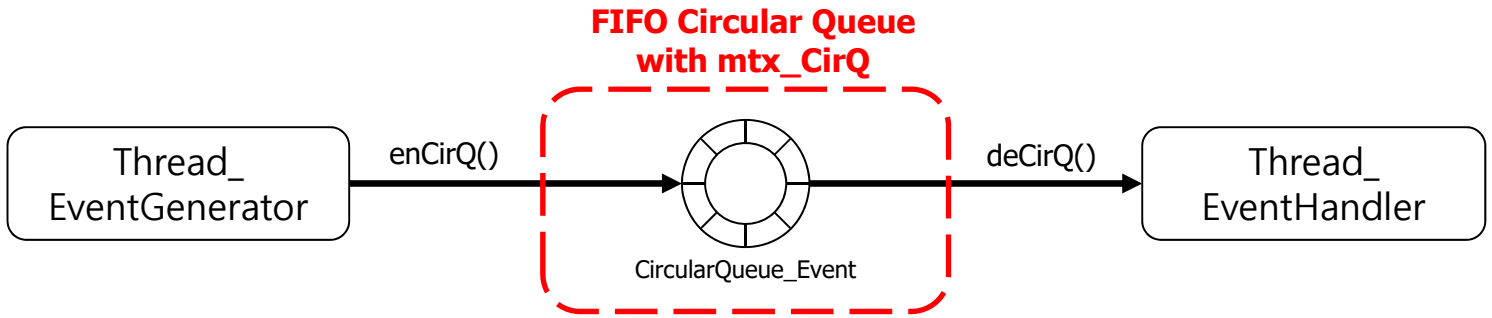
◆ Priority Queue (우선 순위 큐)

- Heap priority queue 기반의 구현
- complete binary tree로 구성
- insertHeap()에서는 upheap bubbling이 수행되며, 항상 기준이 되는 key 값이 가장 작은 (또는 가장 큰) element가 root에 위치하도록 관리됨
- removeMin()에서는 downheap bubbling이 수행되며, 남아 있는 항목들 중 기준이 되는 key 값이 가장 작은 (또는 가장 큰) element가 root에 위치하도록 관리됨

FIFO CirQ와 멀티스레드 구조의 이벤트 (event) 처리 시뮬레이션 (1)

Event Generator – Event Handler

- **Thread EventGenerator**
  - generate events, and enqueues into the shared queue
  - if queue is full, it waits
  - sleeps randomly chosen time
  - repeats the event generation for given execution duration
- **Thread EventHandler**
  - dequeues an event from the shared queue
  - if queue is empty, it waits
  - sleeps randomly chosen time
  - repeats the event processing for given execution duration
- **Functional Block diagram**



Event

```
/* Event.h */

#ifndef EVENT_H
#define EVENT_H
#include <stdio.h>
#include <Windows.h>

#define NUM_PRIORITY 100
#define EVENT_PER_LINE 5
enum EventStatus { GENERATED, ENQUEUED, PROCESSED, UNDEFINED };
extern const char *strEventStatus[];

typedef struct
{
    int ev_no;
    int ev_generator;
    int ev_handler;
    int ev_pri;  // ev_priority
    LARGE_INTEGER ev_t_gen; // for performance monitoring
    LARGE_INTEGER ev_t_handle;
    double elap_time; // for performance monitoring
    EventStatus eventStatus;
} Event;

void printEvent(Event* pEvt);
Event *genEvent(Event *pEv, int event_Gen_ID, int ev_no, int ev_pri);
void calc_elapsed_time(Event* pEv, LARGE_INTEGER freq);
void printEvent_withTime(Event* pEv);

#endif
```

```
/* Event.cpp  (1) */
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include "Event.h"

const char *strEventStatus[] = { "GENERATED", "ENQUED", "PROCESSED", "UNDEFINED" };

void printEvent(Event* pEv)
{
    char str_pri[6];

    printf("Ev(id:%3d, pri:%2d, gen:%2d, proc:%2d)  ", pEv->ev_no, pEv->ev_pri,
            pEv->ev_generator, pEv->ev_handler);
}

Event *genEvent(Event *pEv, int event_Gen_ID, int ev_no, int ev_pri)
{
    pEv = (Event *)calloc(1, sizeof(Event));
    if (pEv == NULL)
        return NULL;
    pEv->ev_generator = event_Gen_ID;
    pEv->ev_handler = -1; // event handler is not defined yet !!
    pEv->ev_no = ev_no;
    //pEv->ev_pri = eventPriority = rand() % NUM_PRIORITY;
    pEv->ev_pri = ev_pri;

    return pEv;
}
```

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

void calc_elapsed_time(Event* pEv, LARGE_INTEGER freq)
{
    LONGLONG t_diff;
    t_diff = pEv->ev_t_handle.QuadPart - pEv->ev_t_gen.QuadPart;
    pEv->elap_time = (double)t_diff / freq.QuadPart;
}

void printEvent_withTime(Event* pEv)
{
    char str_pri[6];

    //printf("Ev(no:%3d, pri:%2d, src:%2d, proc:%2d)  ", pEv->event_no, pEv->event_pri,
    //     pEv->event_gen_addr, pEv->event_handler_addr);
    printf("Ev(no:%2d, pri:%2d, elap_t:%6.0lf[ms])  ", pEv->ev_no, pEv->ev_pri, pEv->elap_time * 1000);
}
```

CircularQueue with Critical Section

```
/* CirQ_Event.h */

#ifndef CIRCULAR_QUEUE_H
#define CIRCULAR_QUEUE_H
#include <mutex>
#include "Event.h"

using namespace std;

typedef struct
{
    Event *CirBuff_Ev; // circular queue for events
    int capacity;
    int front;
    int end;
    int num_elements;
    mutex mtx_CirBuf;
} CirQ_Event;

CirQ_Event *initCirQ_Event(CirQ_Event *pCirQ, int capacity);
void printCirQ_Event(CirQ_Event *cirQ);
bool isCirQ_Ev_Full(CirQ_Event *cirQ);
bool isCirQ_Ev_Empty(CirQ_Event *cirQ);
Event *enCirQ_Event(CirQ_Event *cirQ, Event ev);
Event *deCirQ_Event(CirQ_Event *cirQ);
void delCirQ_Event(CirQ_Event *cirQ);

#endif
```

Circular Queue for Event

```
/* CirQ_Event.c (1) */
#include <stdio.h>
#include <stdlib.h>
#include "CirQ_Event.h"

CirQ_Event *initCirQ_Event(CirQ_Event *cirQ, int capacity)
{
   Event *CirBuff_Ev;

   CirBuff_Ev = (Event *)calloc(capacity, sizeof(Event));
   if (CirBuff_Ev == NULL)
   {
      printf("Error in memory allocation for Event array of size (%d)\n", capacity);
      exit;
   }
   cirQ->CirBuff_Ev = CirBuff_Ev;
   cirQ->capacity = capacity;
   cirQ->front = cirQ->end = 0;
   cirQ->num_elements = 0;
   return cirQ;
}
```

```
/* CirQ_Event.c (2) */

void printCirQ_Event(CirQ_Event *cirQ)
{
   Event ev;
   int index;

   if ((cirQ == NULL) || (cirQ->CirBuff_Ev == NULL))
   {
      printf("Error in printArrayQueue: cirQ is NULL or cirQ->array is NULL");
      exit;
   }
   cirQ->mtx_CirBuf.lock();
   printf("  Elements in CirQ_Event :\n     ");
   if (isCirQ_Ev_Empty(cirQ))
   {
      printf("CirQ_Event is Empty");
   }
   else {
      for (int i = 0; i < cirQ->num_elements; i++)
      {
         index = cirQ->front + i;
         if (index >= cirQ->capacity)
            index = index % cirQ->capacity;
         ev = cirQ->CirBuff_Ev[index];
         printEvent(&ev);
         if ((((i + 1) % EVENT_PER_LINE) == 0) && ((i + 1) != cirQ->num_elements))
            printf("\n     ");
      }
   }
   printf("\n");
   cirQ->mtx_CirBuf.unlock();
}
```

```
/* CirQ_Event.c (3) */

bool isCirQ_Ev_Full(CirQ_Event *cirQ)
{
   if (cirQ->num_elements == cirQ->capacity)
      return true;
   else
      return false;
}

bool isCirQ_Ev_Empty(CirQ_Event *cirQ)
{
   if (cirQ->num_elements == 0)
      return true;
   else
      return false;
}


void delCirQ_Event(CirQ_Event *cirQ)
{
   cirQ->mtx_CirBuf.lock();
   if (cirQ->CirBuff_Ev != NULL)
      free(cirQ->CirBuff_Ev);
   cirQ->CirBuff_Ev = NULL;
   cirQ->capacity = 0;
   cirQ->front = cirQ->end = 0;
   cirQ->num_elements = 0;
   cirQ->mtx_CirBuf.unlock();
}
```

```
/* CirQ_Event.c (4) */

Event *enCirQ_Event(CirQ_Event *cirQ, Event ev)
{

    if (isCirQ_Ev_Full(cirQ))
    {
        return NULL;
    }
    cirQ->mtx_CirBuf.lock();
    cirQ->CirBuff_Ev[cirQ->end] = ev;
    cirQ->num_elements++;
    cirQ->end++;
    if (cirQ->end >= cirQ->capacity)
        cirQ->end = cirQ->end % cirQ->capacity;
    cirQ->mtx_CirBuf.unlock();
    return &(cirQ->CirBuff_Ev[cirQ->end]);
}

Event *deCirQ_Event(CirQ_Event *cirQ)
{
    if (isCirQ_Ev_Empty(cirQ))
        return NULL;

    cirQ->mtx_CirBuf.lock();
    Event *pEv = &(cirQ->CirBuff_Ev[cirQ->front]);
    cirQ->front++;
    if (cirQ->front >= cirQ->capacity)
        cirQ->front = cirQ->front % cirQ->capacity;
    cirQ->num_elements--;
    cirQ->mtx_CirBuf.unlock();
    return pEv;
}
```

Thread.h

```
/* Thread.h  (1) */
#include "CirQ_Event.h"
#include "SimParams.h"

using namespace std;

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

typedef struct
{
    int numEventGenerated;
    int numEventProcessed;
    int totalEventGenerated;
    int totalEventProcessed;
    Event eventGenerated[TOTAL_NUM_EVENTS];
    Event eventProcessed[TOTAL_NUM_EVENTS];
    THREAD_FLAG *pFlagThreadTerminate;
} ThreadStatusMonitor;
```

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

typedef struct
{
    mutex *pMTX_main;
    mutex *pMTX_thrd_mon;
    CirQ_Event *pCirQ_Event;
    ROLE role;
    int myAddr;
    int maxRound;
    int targetEventGen;
    ThreadStatusMonitor *pThrdMon;
} ThreadParam_Event;

void Thread_EventHandler(ThreadParam_Event
    *pParam);
void Thread_EventGenerator(ThreadParam_Event
   *pParam);
#endif
```

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

#include <Windows.h>
#include <time.h>
#include "Thread.h"
#include "CirQ_Event.h"
#include "Event.h"

void Thread_EventGenerator(ThreadParam_Event* pParam)
{
    CirQ_Event *pCirQ_Event = pParam->pCirQ_Event;
    int myRole = pParam->role;
    int myAddr = pParam->myAddr;
    int maxRound = pParam->maxRound;
    int event_gen_count = 0;
    ThreadStatusMonitor *pThrdMon = pParam->pThrdMon;
    pCirQ_Event = pParam->pCirQ_Event;
    int targetEventGen = pParam->targetEventGen;
    Event* pEv;

    for (int round = 0; round < maxRound; round++)
    {
        if (event_gen_count >= targetEventGen)
        {
            if (*pThrdMon->pFlagThreadTerminate == TERMINATE)
                break;
            else {
                Sleep(500);
                continue;
            }
        }
```

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

        pEv = (Event *)calloc(1, sizeof(Event));
        pEv->ev_generator = myAddr;
        pEv->ev_handler = -1; // event handler is not defined yet !!
        pEv->ev_no = event_gen_count + NUM_EVENTS_PER_GEN*myAddr;
        //pEv->ev_pri = eventPriority = rand() % NUM_PRIORITY;
        pEv->ev_pri = targetEventGen - event_gen_count -1;
        QueryPerformanceCounter(&pEv->ev_t_gen);
        pThrdMon->eventGenerated[pThrdMon->numEventGenerated] = *pEv;
        while (enCirQ_Event(pCirQ_Event, *pEv) == NULL)
        {
            Sleep(500);
        }
        free(pEv);
        pParam->pMTX_thrd_mon->lock();
        pThrdMon->numEventGenerated++;
        pThrdMon->totalEventGenerated++;
        pParam->pMTX_thrd_mon->unlock();
        event_gen_count++;
        //Sleep(100 + rand() % 300);
        Sleep(10);
    }
}
```

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

#include <Windows.h>
#include <time.h>
#include "Thread.h"
#include "CirQ_Event.h"
#include "Event.h"

void Thread_EventHandler(ThreadParam_Event* pParam)
{
    Event *pEv, *pEvProc;
    int myRole = pParam->role;
    int myAddr = pParam->myAddr;
    CirQ_Event* pCirQ_Event = pParam->pCirQ_Event;
    ThreadStatusMonitor* pThrdMon = pParam->pThrdMon;
    int maxRound = pParam->maxRound;
    int targetEventGen = pParam->targetEventGen;

    for (int round = 0; round < maxRound; round++)
    {
        if (*pThrdMon->pFlagThreadTerminate == TERMINATE)
            break;
```

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

        if ((pEv = deCirQ_Event(pCirQ_Event)) != NULL)
        {
            //printEvent(pEv);
            //printf("\n");
            pParam->pMTX_thrd_mon->lock();
            QueryPerformanceCounter(&pEv->ev_t_handle);
            pEv->ev_handler = myAddr;
            pThrdMon->eventProcessed[pThrdMon->totalEventProcessed] = *pEv;
            pThrdMon->numEventProcessed++;
            pThrdMon->totalEventProcessed++;
            pParam->pMTX_thrd_mon->unlock();
        }
        Sleep(100 + rand() % 300);
    }

}
```

FIFO CirQ와 멀티스레드 구조의 이벤트 (event) 처리 시뮬레이션 (2)

Simulation Parameters

```
/* SimParam.h Simulation Parameters */

#ifndef SIMULATION_PARAMETERS_H
#define SIMULATION_PARAMETERS_H

#define NUM_EVENT_GENERATORS 1
#define NUM_EVENTS_PER_GEN 50
#define NUM_EVENT_HANDLERS 1
#define TOTAL_NUM_EVENTS (NUM_EVENTS_PER_GEN * NUM_EVENT_GENERATORS)

#define CIR_QUEUE_CAPACITY 10
#define PLUS_INF INT_MAX
#define MAX_ROUND 1000

#endif
```

ConsoleDisplay.h

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

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

Console Display

```
/* ConsoleDisplay.cpp */
#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);
}
```

main() – Event Handling with CirQ

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

#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <mutex>
#include "Thread.h"
#include "CirQ_Event.h"
#include "Event.h"
#include "ConsoleDisplay.h"

using namespace std;

void main()
{
    CirQ_Event cirQ_Event;
    Event *pEv;
    int myAddr = 0;
    int ev_handler, eventPriority;

    srand(time(NULL));
    initCirQ_Event(&cirQ_Event, CIR_QUEUE_CAPACITY);
```

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

    ThreadParam_Event thrdParam_EventGen, thrdParam_EventHndlr;
    mutex cs_main; // console display
    mutex cs_thrd_mon; // thread monitoring
    ThreadStatusMonitor thrdMon;
    HANDLE consHndlr;
    THREAD_FLAG eventThreadFlag = RUN;
    int count, numEventGenerated, numEventProcessed;
    LARGE_INTEGER freq;

    consHndlr = initConsoleHandler();

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

    /* Create and Activate Thread_EventHandler */
    thrdMon.numEventProcessed = 0;
    thrdParam_EventHndlr.role = EVENT_HANDLER;
    thrdParam_EventHndlr.myAddr = 1; // link address
    thrdParam_EventHndlr.pMTX_main = &cs_main;
    thrdParam_EventHndlr.pMTX_thrd_mon = &cs_thrd_mon;
    thrdParam_EventHndlr.pCirQ_Event = &cirQ_Event;
    thrdParam_EventHndlr.maxRound = MAX_ROUND;
    thrdParam_EventHndlr.pThrdMon = &thrdMon;
```

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

    thread thrd_ev_handler(Thread_EventHandler, &thrdParam_EventHndlr);
    printf("Thread_EventHandler is created and activated ...\n");

    /* Create and Activate Thread_EventGen */
    thrdMon.numEventGenerated = 0;
    thrdParam_EventGen.role = EVENT_GENERATOR;
    thrdParam_EventGen.myAddr = 0; // my Address
    thrdParam_EventGen.pMTX_main = &cs_main;
    thrdParam_EventGen.pMTX_thrd_mon = &cs_thrd_mon;
    thrdParam_EventGen.pCirQ_Event = &cirQ_Event;
    thrdParam_EventGen.targetEventGen = NUM_EVENTS_PER_GEN;
    thrdParam_EventGen.maxRound = MAX_ROUND;
    thrdParam_EventGen.pThrdMon = &thrdMon;

    thread thrd_ev_generator (Thread_EventGenerator, &thrdParam_EventGen);
    printf("Thread_EventGen is created and activated ...\n");
```

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

    for (int round = 0; round < MAX_ROUND; round++)
    {
        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");
        printf("Events generated: \n  ");
        count = 0;
        numEventGenerated = thrdMon.totalEventGenerated;
        for (int i = 0; i < numEventGenerated; i++)
        {
            pEv = &thrdMon.eventGenerated[i];
            if (pEv != NULL)
            {
                printEvent(pEv);
                if (((i + 1) % EVENT_PER_LINE) == 0)
                    printf("\n  ");
            }
        }
        printf("\n");
        printf("Event_Gen generated %2d events\n", thrdMon.numEventGenerated);
        printf("Event_Handler processed %2d events\n", thrdMon.numEventProcessed);
        printf("\n");
        printf("CirQ_Event::");  printCirQ_Event(&cirQ_Event);
        printf("\n");
```

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

        printf("Events processed: \n  ");
        count = 0;
        numEventProcessed = thrdMon.totalEventProcessed;
        for (int i = 0; i < numEventProcessed; i++)
        {
            pEv = &thrdMon.eventProcessed[i];
            if (pEv != NULL)
            {
                calc_elapsed_time(pEv, freq);
                printEvent_withTime(pEv);
                if (((i + 1) % EVENT_PER_LINE) == 0)
                    printf("\n  ");
            }
        }
        printf("\n");

        if (numEventProcessed >= TOTAL_NUM_EVENTS)
        {
            eventThreadFlag = TERMINATE; // set 1 to terminate threads
            break;
        }

        Sleep(100);
    } // end of for (round . . . . )
```

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

    /* Analyze the event processing times */
    double min, max, avg, sum;
    int min_ev, max_ev;
    min = max = sum = thrdMon.eventProcessed[0].elap_time;
    min_ev = max_ev = 0;
    for (int i = 1; i < TOTAL_NUM_EVENTS; i++)
    {
        sum += thrdMon.eventProcessed[i].elap_time;
        if (min > thrdMon.eventProcessed[i].elap_time)
        {
            min = thrdMon.eventProcessed[i].elap_time;
            min_ev = i;
        }
        if (max < thrdMon.eventProcessed[i].elap_time)
        {
            max = thrdMon.eventProcessed[i].elap_time;
            max_ev = i;
        }
    }
    avg = sum / TOTAL_NUM_EVENTS;
    printf("Minimum event processing time: %8.2lf[ms] for ", min * 1000);
    printEvent_withTime(&thrdMon.eventProcessed[min_ev]); printf("\n");
    printf("Maximum event processing time: %8.2lf[ms] for ", max * 1000);
    printEvent_withTime(&thrdMon.eventProcessed[max_ev]); printf("\n");
    printf("Average event processing time: %8.2lf[ms] for total %d events\n", avg * 1000,
        TOTAL_NUM_EVENTS);
    printf("\n");
```

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


    thrd_ev_generator.join();
    printf("Thread_EventGenerator is terminated !!\n");

    thrd_ev_handler.join();
    printf("Thread_EventHandler is terminated !!\n");

}
```

실행 결과

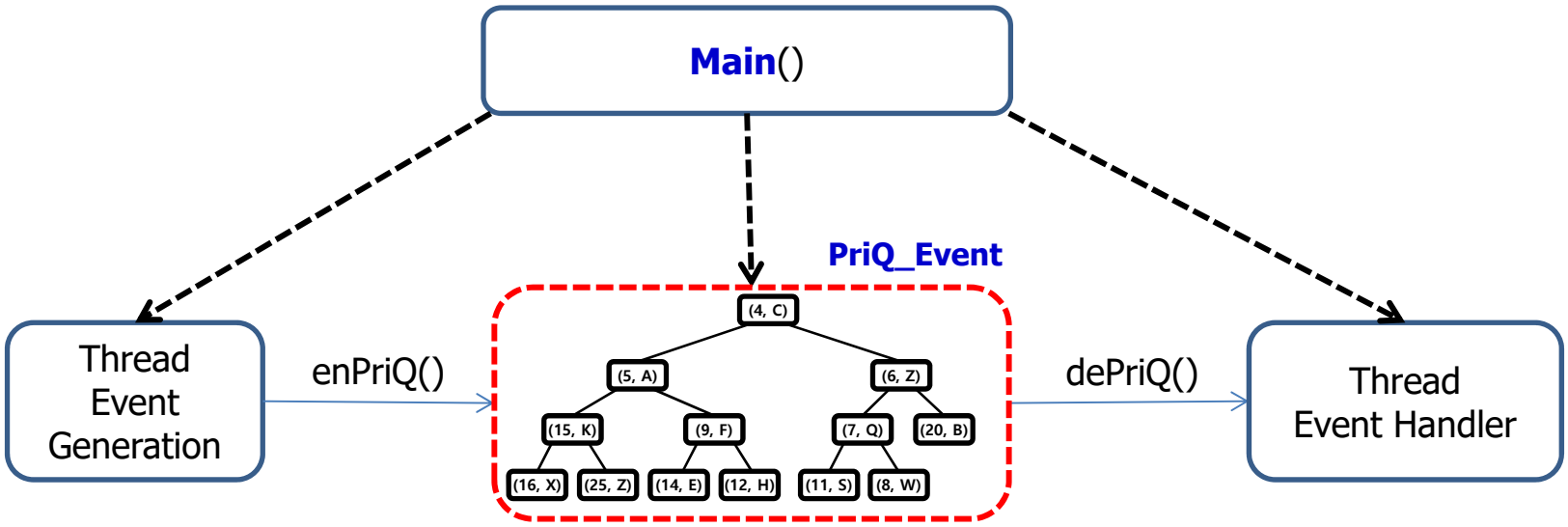
```
Thread monitoring by main() ::
  round(97): current total_event_gen (50), total_event_proc(50)

Events generated:
  Ev(id:  0, pri:49, gen: 0, proc:-1)  Ev(id:  1, pri:48, gen: 0, proc:-1)  Ev(id:  2, pri:47, gen: 0, proc:-1)  Ev(id:  3, pri:46, gen: 0, proc:-1)  Ev(id:  4, pri:45, gen: 0, proc:-1)
  Ev(id:  5, pri:44, gen: 0, proc:-1)  Ev(id:  6, pri:43, gen: 0, proc:-1)  Ev(id:  7, pri:42, gen: 0, proc:-1)  Ev(id:  8, pri:41, gen: 0, proc:-1)  Ev(id:  9, pri:40, gen: 0, proc:-1)
  Ev(id: 10, pri:39, gen: 0, proc:-1)  Ev(id: 11, pri:38, gen: 0, proc:-1)  Ev(id: 12, pri:37, gen: 0, proc:-1)  Ev(id: 13, pri:36, gen: 0, proc:-1)  Ev(id: 14, pri:35, gen: 0, proc:-1)
  Ev(id: 15, pri:34, gen: 0, proc:-1)  Ev(id: 16, pri:33, gen: 0, proc:-1)  Ev(id: 17, pri:32, gen: 0, proc:-1)  Ev(id: 18, pri:31, gen: 0, proc:-1)  Ev(id: 19, pri:30, gen: 0, proc:-1)
  Ev(id: 20, pri:29, gen: 0, proc:-1)  Ev(id: 21, pri:28, gen: 0, proc:-1)  Ev(id: 22, pri:27, gen: 0, proc:-1)  Ev(id: 23, pri:26, gen: 0, proc:-1)  Ev(id: 24, pri:25, gen: 0, proc:-1)
  Ev(id: 25, pri:24, gen: 0, proc:-1)  Ev(id: 26, pri:23, gen: 0, proc:-1)  Ev(id: 27, pri:22, gen: 0, proc:-1)  Ev(id: 28, pri:21, gen: 0, proc:-1)  Ev(id: 29, pri:20, gen: 0, proc:-1)
  Ev(id: 30, pri:19, gen: 0, proc:-1)  Ev(id: 31, pri:18, gen: 0, proc:-1)  Ev(id: 32, pri:17, gen: 0, proc:-1)  Ev(id: 33, pri:16, gen: 0, proc:-1)  Ev(id: 34, pri:15, gen: 0, proc:-1)
  Ev(id: 35, pri:14, gen: 0, proc:-1)  Ev(id: 36, pri:13, gen: 0, proc:-1)  Ev(id: 37, pri:12, gen: 0, proc:-1)  Ev(id: 38, pri:11, gen: 0, proc:-1)  Ev(id: 39, pri:10, gen: 0, proc:-1)
  Ev(id: 40, pri: 9, gen: 0, proc:-1)  Ev(id: 41, pri: 8, gen: 0, proc:-1)  Ev(id: 42, pri: 7, gen: 0, proc:-1)  Ev(id: 43, pri: 6, gen: 0, proc:-1)  Ev(id: 44, pri: 5, gen: 0, proc:-1)
  Ev(id: 45, pri: 4, gen: 0, proc:-1)  Ev(id: 46, pri: 3, gen: 0, proc:-1)  Ev(id: 47, pri: 2, gen: 0, proc:-1)  Ev(id: 48, pri: 1, gen: 0, proc:-1)  Ev(id: 49, pri: 0, gen: 0, proc:-1)

Event_Gen generated 50 events
Event_Handler processed 50 events

CirQ_Event::  Elements in CirQ_Event :
     CirQ_Event is Empty

Events processed:
  Ev(no: 0, pri:49, elap_t:   363[ms])  Ev(no: 1, pri:48, elap_t:   690[ms])  Ev(no: 2, pri:47, elap_t:   930[ms])  Ev(no: 3, pri:46, elap_t:  1079[ms])  Ev(no: 4, pri:45, elap_t:  1393[ms])
  Ev(no: 5, pri:44, elap_t:  1498[ms])  Ev(no: 6, pri:43, elap_t:  1603[ms])  Ev(no: 7, pri:42, elap_t:  1708[ms])  Ev(no: 8, pri:41, elap_t:  1977[ms])  Ev(no: 9, pri:40, elap_t:  2322[ms])
  Ev(no:10, pri:39, elap_t:  2576[ms])  Ev(no:11, pri:38, elap_t:  2320[ms])  Ev(no:12, pri:37, elap_t:  2111[ms])  Ev(no:13, pri:36, elap_t:  2425[ms])  Ev(no:14, pri:35, elap_t:  2755[ms])
  Ev(no:15, pri:34, elap_t:  2605[ms])  Ev(no:16, pri:33, elap_t:  2844[ms])  Ev(no:17, pri:32, elap_t:  3025[ms])  Ev(no:18, pri:31, elap_t:  2665[ms])  Ev(no:19, pri:30, elap_t:  2800[ms])
  Ev(no:20, pri:29, elap_t:  2545[ms])  Ev(no:21, pri:28, elap_t:  2771[ms])  Ev(no:22, pri:27, elap_t:  2486[ms])  Ev(no:23, pri:26, elap_t:  2830[ms])  Ev(no:24, pri:25, elap_t:  2425[ms])
  Ev(no:25, pri:24, elap_t:  2051[ms])  Ev(no:26, pri:23, elap_t:  2381[ms])  Ev(no:27, pri:22, elap_t:  2067[ms])  Ev(no:28, pri:21, elap_t:  2367[ms])  Ev(no:29, pri:20, elap_t:  2607[ms])
  Ev(no:30, pri:19, elap_t:  2396[ms])  Ev(no:31, pri:18, elap_t:  2741[ms])  Ev(no:32, pri:17, elap_t:  2606[ms])  Ev(no:33, pri:16, elap_t:  2711[ms])  Ev(no:34, pri:15, elap_t:  2471[ms])
  Ev(no:35, pri:14, elap_t:  2635[ms])  Ev(no:36, pri:13, elap_t:  2799[ms])  Ev(no:37, pri:12, elap_t:  2515[ms])  Ev(no:38, pri:11, elap_t:  2875[ms])  Ev(no:39, pri:10, elap_t:  2620[ms])
  Ev(no:40, pri: 9, elap_t:  3011[ms])  Ev(no:41, pri: 8, elap_t:  2681[ms])  Ev(no:42, pri: 7, elap_t:  2516[ms])  Ev(no:43, pri: 6, elap_t:  2696[ms])  Ev(no:44, pri: 5, elap_t:  2472[ms])
  Ev(no:45, pri: 4, elap_t:  2637[ms])  Ev(no:46, pri: 3, elap_t:  2936[ms])  Ev(no:47, pri: 2, elap_t:  2681[ms])  Ev(no:48, pri: 1, elap_t:  2846[ms])  Ev(no:49, pri: 0, elap_t:  2576[ms])

Minimum event processing time:   363.04[ms] for Ev(no: 0, pri:49, elap_t:   363[ms])
Maximum event processing time:  3024.59[ms] for Ev(no:17, pri:32, elap_t:  3025[ms])
Average event processing time:  2352.81[ms] for total 50 events

Thread_EventGenerator is terminated !!
Thread_EventHandler is terminated !!
```

우선 순위 큐와 멀티스레드 구조의 이벤트 (event) 처리 시뮬레이션

Event Handling with Multi-threads and Priority Queue

◆ Two Threads with Priority Queue

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



Priority Queue for Events

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

#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mutex>
#include "Event.h"

using namespace std;

#define POS_ROOT 1
#define MAX_NAME_LEN 80
#define TRUE 1
#define FALSE 0


typedef struct CBTN_Event
{
    int priority;
    Event event;
} CBTN_Event;
```

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

typedef struct PriorityQueue
{
    char PriQ_name[MAX_NAME_LEN];
    int priQ_capacity;
    int priQ_size;
    int pos_last;
    CBTN_Event *pCBT_Event;
    mutex cs_priQ;
} PriQ_Event;

PriQ_Event *initPriQ_Event(PriQ_Event *pPriQ_Event,
     const char *name, int capacity);
Event *enPriQ_Event(PriQ_Event *pPriQ_Event, Event ev);
Event *dePriQ_Event(PriQ_Event *pPriQ_Event);
void printPriQ_Event(PriQ_Event *pPriQ_Event);
void fprintPriQ_Event(FILE *fout, PriQ_Event *pPriQ_Event);
void deletePriQ_Event(PriQ_Event *pPriQ_Event);
#endif
```

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

#include "PriQ_Event.h"
#include "Event.h"

bool hasLeftChild(int pos, PriQ_Event *pPriQ_Event)
{
    if (pos * 2 <= pPriQ_Event->priQ_size)
        return TRUE;
    else
        return FALSE;
}

bool hasRightChild(int pos, PriQ_Event *pPriQ_Event)
{
    if (pos * 2 + 1 <= pPriQ_Event->priQ_size)
        return TRUE;
    else
        return FALSE;
}

PriQ_Event *initPriQ_Event(PriQ_Event *pPriQ_Event, const char *name, int capacity = 1)
{
    pPriQ_Event->cs_PriQ.lock();
    strcpy(pPriQ_Event->PriQ_name, name);
    pPriQ_Event->priQ_capacity = capacity;
    pPriQ_Event->pCBT_Event = (CBTN_Event *)calloc((capacity + 1), sizeof(CBTN_Event));
    pPriQ_Event->priQ_size = 0;
    pPriQ_Event->pos_last = 0;
    pPriQ_Event->cs_PriQ.unlock();
    return pPriQ_Event;
}
```

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

void deletePriQ_Event(PriQ_Event *pPriQ_Event)
{
    pPriQ_Event->cs_PriQ.lock();
    if (pPriQ_Event->pCBT_Event != NULL)
        free(pPriQ_Event->pCBT_Event);
    pPriQ_Event->cs_PriQ.unlock();
}

Event *enPriQ_Event(PriQ_Event *pPriQ_Event, Event ev)
{
    int pos, pos_parent;
    CBTN_Event CBTN_Ev_tmp;

    pPriQ_Event->cs_PriQ.lock();
    if (pPriQ_Event->priQ_size >= pPriQ_Event->priQ_capacity)
    {
        // Priority Queue is full
        /* Expand the capacity twice, and copy the entries */
        CBTN_Event *newCBT_Event;
        int newCapacity;

        newCapacity = 2 * pPriQ_Event->priQ_capacity;
        newCBT_Event = (CBTN_Event *)calloc((newCapacity + 1), sizeof(CBTN_Event));
        if (newCBT_Event == NULL)
        {
            printf("Error in expanding CompleteBinaryTree for Priority Queue !!₩n");
            exit(-1);
        }
        for (int pos = 1; pos <= pPriQ_Event->priQ_size; pos++)
        {
            newCBT_Event[pos] = pPriQ_Event->pCBT_Event[pos];
        }
        free(pPriQ_Event->pCBT_Event);
        pPriQ_Event->pCBT_Event = newCBT_Event;
        pPriQ_Event->priQ_capacity = newCapacity;
    } //end - if
```

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

    /* insert at the last position */
    pos = ++pPriQ_Event->priQ_size;
    pPriQ_Event->pCBT_Event[pos].priority = ev.ev_pri;
    pPriQ_Event->pCBT_Event[pos].event = ev;

    /* up-heap bubbling */
    while (pos != POS_ROOT)
    {
        pos_parent = pos / 2;
        if (pPriQ_Event->pCBT_Event[pos].priority >= pPriQ_Event->pCBT_Event[pos_parent].priority)
        {
            break;
            // if the priority of the new packet is lower than its parent's priority, just stop up-heap bubbling
        }
        else
        {
            CBTN_Ev_tmp = pPriQ_Event->pCBT_Event[pos_parent];
            pPriQ_Event->pCBT_Event[pos_parent] = pPriQ_Event->pCBT_Event[pos];
            pPriQ_Event->pCBT_Event[pos] = CBTN_Ev_tmp;
            pos = pos_parent;
        }
    } // end - while
    pPriQ_Event->cs_PriQ.unlock();
    return &(pPriQ_Event->pCBT_Event[pPriQ_Event->pos_last].event);
}
```

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

Event *dePriQ_Event(PriQ_Event *pPriQ_Event)
{
    Event *pEv, ev;
    CBTN_Event  CBTN_Ev_tmp;
    int pos, pos_root = 1, pos_last, pos_child;

    if (pPriQ_Event->priQ_size <= 0)
        return NULL;  // Priority queue is empty

    pPriQ_Event->cs_PriQ.lock();
    pEv = (Event*)calloc(1, sizeof(Event));
    *pEv = pPriQ_Event->pCBT_Event[1].event; // get the packet address of current top
    pos_last = pPriQ_Event->priQ_size;
    pPriQ_Event->priQ_size--;
    if (pPriQ_Event->priQ_size > 0)
    {
        /* put the last node into the top position */
        pPriQ_Event->pCBT_Event[pos_root] = pPriQ_Event->pCBT_Event[pos_last];

        /* down heap bubbling */
        pos = pos_root;
        while (hasLeftChild(pos, pPriQ_Event))
        {
            pos_child = pos * 2;
            if (hasRightChild(pos, pPriQ_Event))
            {
                if (pPriQ_Event->pCBT_Event[pos_child].priority >
                      pPriQ_Event->pCBT_Event[pos_child+1].priority)
                    pos_child = pos * 2 + 1; // if right child has higher priority, then select it
            }
```

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

        /* if the Event in pos_child has higher priority than Event in pos,  swap them */
        if (pPriQ_Event->pCBT_Event[pos_child].priority >= pPriQ_Event->pCBT_Event[pos].priority)
        {
            break;
        } else {
            CBTN_Ev_tmp = pPriQ_Event->pCBT_Event[pos];
            pPriQ_Event->pCBT_Event[pos] = pPriQ_Event->pCBT_Event[pos_child];
            pPriQ_Event->pCBT_Event[pos_child] = CBTN_Ev_tmp;
        }
        pos = pos_child;
      } // end while
   } // end if
   pPriQ_Event->cs_PriQ.unlock();
   return pEv;
}


void printPriQ_Event(PriQ_Event *pPriQ_Event)
{
   int pos, count;
   int eventPriority;
   int level = 0, level_count = 1;
   Event *pEv;

   if (pPriQ_Event->priQ_size == 0)
   {
      printf("  PriorityQueue_Event is empty !!\n");
      return;
   }
```

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

    pos = 1;
    count = 1;
    level = 0;
    level_count = 1; // level_count = 2^^level
    printf("\n CompBinTree :\n  ", level);
    while (count <= pPriQ_Event->priQ_size)
    {
        printf(" level%2d : ", level);
        for (int i = 0; i < level_count; i++)
        {
            pEv = &(pPriQ_Event->pCBT_Event[pos].event);
            eventPriority = pEv->ev_pri;
            //printf("Event(pri: %2d, id:%2d, src:%2d, dst: %2d) ", eventPriority, pEvent->event_no,
                  pEvent->event_gen_addr, pEvent->event_handler_addr);
            //printf("Event(pri:%2d, src:%2d, id:%3d) ", eventPriority, pEvent->event_gen_addr,
                   pEvent->event_no);
            printEvent(pEv);
            pos++;
            if ((count % EVENT_PER_LINE) == 0)
                printf("\n  ");

            count++;
            if (count > pPriQ_Event->priQ_size)
                break;
        }
        printf("\n");
        level++;
        level_count *= 2;
    } // end - while
    printf("\n");
}
```

Thread_EventGenerator() and Thread_EventHandler() with PriorityQueue

- **CircularQueue_Event를 사용하였던 Thread_EventGenerator()/Thread_EventHandler() 함수와 구조는 동일**
- **Thread_EventGenerator() with PriorityQueue**
  - ThreadParam_Event에 PriQ_Event 주소 전달
  - Event 생성 후 enPriQ_Event() 함수를 사용하여 PriQ_Event에 생성된 Event 삽입
- **Thread_EventHandler() with PriorityQueue**
  - ThreadParam_Event에 PriQ_Event 주소 전달
  - dePriQ_Event() 함수를 사용하여 PriQ_Event로 부터 Event 하나를 추출하고 이를 처리

main() - Event Handling with PriQ

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

#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <mutex>
#include "Thread.h"
#include "PriQ_Event.h"
#include "Event.h"
#include "ConsoleDisplay.h"

using namespace std;

void main()
{
    PriQ_Event priQ_Event;
    Event *pEv;
    int myAddr = 0;
    int ev_handler, eventPriority;

    srand(time(NULL));
    initPriQ_Event(&priQ_Event, "PriQ_Event", 1);

    ThreadParam_Event thrdParam_EventGen, thrdParam_EventHndlr;
    mutex cs_main; // console display
    mutex cs_thrd_mon; // thread monitoring
    ThreadStatusMonitor thrdMon;
    HANDLE consHndlr;
    THREAD_FLAG eventThreadFlag = RUN;
    int count, numEventGenerated, numEventProcessed;
    LARGE_INTEGER freq;
```

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

    consHndlr = initConsoleHandler();

    thrdMon.pFlagThreadTerminate = &eventThreadFlag;
    thrdMon.totalEventGenerated = 0;
    thrdMon.totalEventProcessed = 0;
    for (int ev = 0; ev < TOTAL_NUM_EVENTS; ev++)
    {
        thrdMon.eventProcessed[ev].ev_no = -1; // mark as not-processed
        thrdMon.eventProcessed[ev].ev_pri = -1;
    }
    QueryPerformanceFrequency(&freq);
    /* Create and Activate Thread_EventHandler */

    thrdMon.numEventProcessed = 0;
    thrdParam_EventHndlr.role = EVENT_HANDLER;
    thrdParam_EventHndlr.myAddr = 1; // link address
    thrdParam_EventHndlr.pMTX_main = &cs_main;
    thrdParam_EventHndlr.pMTX_thrd_mon = &cs_thrd_mon;
    thrdParam_EventHndlr.pPriQ_Event = &priQ_Event;
    thrdParam_EventHndlr.maxRound = MAX_ROUND;
    thrdParam_EventHndlr.pThrdMon = &thrdMon;

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

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

    /* Create and Activate Thread_EventGen */
    thrdMon.numEventGenerated = 0;
    thrdParam_EventGen.role = EVENT_GENERATOR;
    thrdParam_EventGen.myAddr = 0; // my Address
    thrdParam_EventGen.pMTX_main = &cs_main;
    thrdParam_EventGen.pMTX_thrd_mon = &cs_thrd_mon;
    thrdParam_EventGen.pPriQ_Event = &priQ_Event;
    thrdParam_EventGen.targetEventGen = NUM_EVENTS_PER_GEN;
    thrdParam_EventGen.maxRound = MAX_ROUND;
    thrdParam_EventGen.pThrdMon = &thrdMon;

    thread thrd_ev_generator (Thread_EventGenerator, &thrdParam_EventGen);
   printf("Thread_EventGen is created and activated ...\n");

    for (int round = 0; round < MAX_ROUND; round++)
    {
        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");
        printf("Events generated: \n  ");
```

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

        count = 0;
        numEventGenerated = thrdMon.totalEventGenerated;
        for (int i = 0; i < numEventGenerated; i++)
        {
            pEv = &thrdMon.eventGenerated[i];
            if (pEv != NULL)
            {
                printEvent(pEv);
                if (((i + 1) % EVENT_PER_LINE) == 0)
                    printf("\n  ");
            }
        }
        printf("\n");
        printf("Event_Gen generated %2d events\n", thrdMon.numEventGenerated);
        printf("Event_Handler processed %2d events\n", thrdMon.numEventProcessed);
        printf("\n");
        printf("PriQ_Event::");  printPriQ_Event(&priQ_Event);
        printf("\n");
        printf("Events processed: \n  ");
        count = 0;
        numEventProcessed = thrdMon.totalEventProcessed;
        for (int i = 0; i < numEventProcessed; i++)
        {
            pEv = &thrdMon.eventProcessed[i];
            if (pEv != NULL)
            {
                calc_elapsed_time(pEv, freq);
                printEvent_withTime(pEv);
                if (((i + 1) % EVENT_PER_LINE) == 0)
                    printf("\n  ");
            }
        }
        printf("\n");
```

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

        if (numEventProcessed >= TOTAL_NUM_EVENTS)
        {
            eventThreadFlag = TERMINATE; // set 1 to terminate threads
            break;
        }

       Sleep(100);
    }

    /* Analyze the event processing times */
    double min, max, avg, sum;
    int min_ev, max_ev;
    min = max = sum = thrdMon.eventProcessed[0].elap_time;
    min_ev = max_ev = 0;
    for (int i = 1; i < TOTAL_NUM_EVENTS; i++)
    {
        sum += thrdMon.eventProcessed[i].elap_time;
        if (min > thrdMon.eventProcessed[i].elap_time)
        {
            min = thrdMon.eventProcessed[i].elap_time;
            min_ev = i;
        }
        if (max < thrdMon.eventProcessed[i].elap_time)
        {
            max = thrdMon.eventProcessed[i].elap_time;
            max_ev = i;
        }
    }
```

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

    avg = sum / TOTAL_NUM_EVENTS;
    printf("Minimum event processing time: %8.2lf[ms] for ", min * 1000);
    printEvent_withTime(&thrdMon.eventProcessed[min_ev]); printf("\n");
    printf("Maximum event processing time: %8.2lf[ms] for ", max * 1000);
    printEvent_withTime(&thrdMon.eventProcessed[max_ev]); printf("\n");
    printf("Average event processing time: %8.2lf[ms] for total %d events\n", avg * 1000,
      TOTAL_NUM_EVENTS);
    printf("\n");

    thrd_ev_generator.join();
    printf("Thread_EventGenerator is terminated !!\n");

    thrd_ev_handler.join();
    printf("Thread_EventHandler is terminated !!\n");
}
```

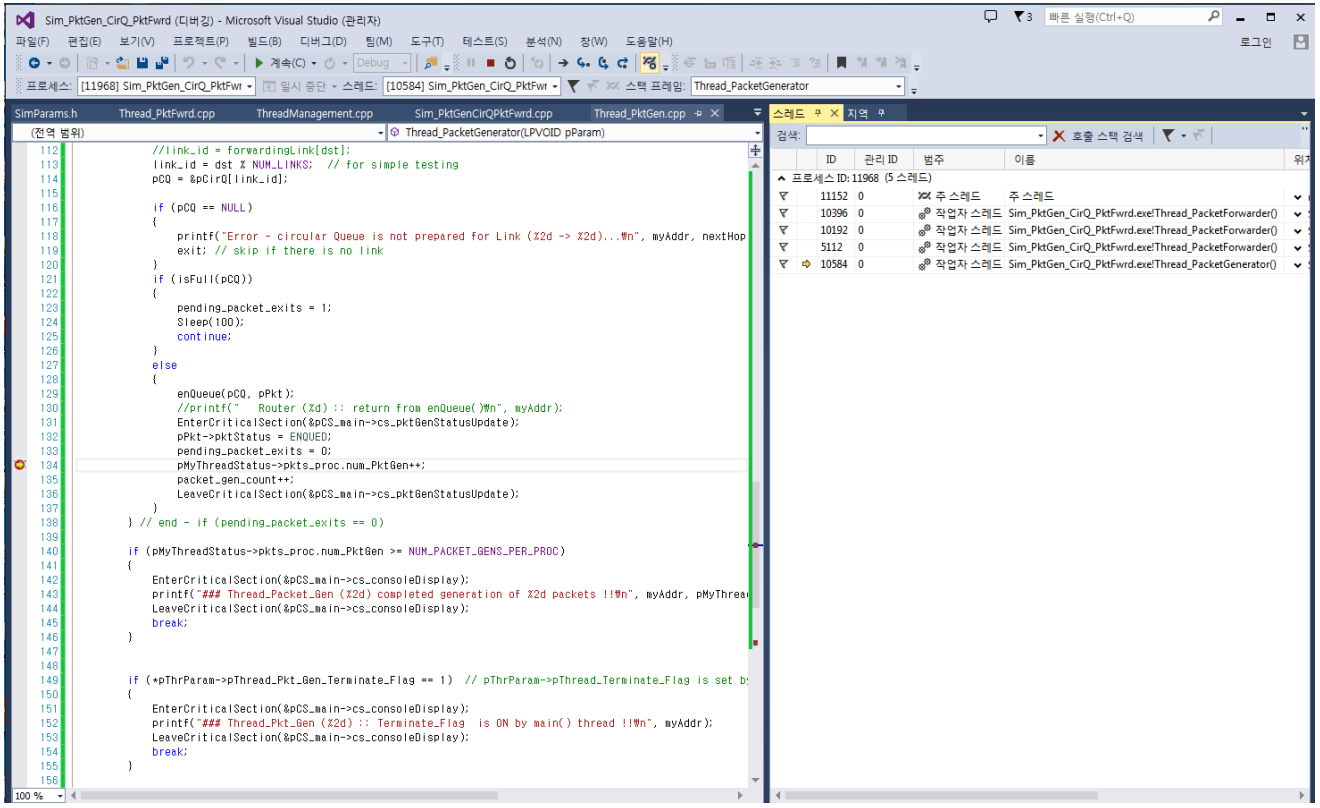
실행 결과

```
Thread monitoring by main() ::
  round(96): current total_event_gen (50), total_event_proc(50)

Events generated:
  Ev(id:  0, pri:49, gen: 0, proc:-1)  Ev(id:  1, pri:48, gen: 0, proc:-1)  Ev(id:  2, pri:47, gen: 0, proc:-1)  Ev(id:  3, pri:46, gen: 0, proc:-1)  Ev(id:  4, pri:45, gen: 0, proc:-1)
  Ev(id:  5, pri:44, gen: 0, proc:-1)  Ev(id:  6, pri:43, gen: 0, proc:-1)  Ev(id:  7, pri:42, gen: 0, proc:-1)  Ev(id:  8, pri:41, gen: 0, proc:-1)  Ev(id:  9, pri:40, gen: 0, proc:-1)
  Ev(id: 10, pri:39, gen: 0, proc:-1)  Ev(id: 11, pri:38, gen: 0, proc:-1)  Ev(id: 12, pri:37, gen: 0, proc:-1)  Ev(id: 13, pri:36, gen: 0, proc:-1)  Ev(id: 14, pri:35, gen: 0, proc:-1)
  Ev(id: 15, pri:34, gen: 0, proc:-1)  Ev(id: 16, pri:33, gen: 0, proc:-1)  Ev(id: 17, pri:32, gen: 0, proc:-1)  Ev(id: 18, pri:31, gen: 0, proc:-1)  Ev(id: 19, pri:30, gen: 0, proc:-1)
  Ev(id: 20, pri:29, gen: 0, proc:-1)  Ev(id: 21, pri:28, gen: 0, proc:-1)  Ev(id: 22, pri:27, gen: 0, proc:-1)  Ev(id: 23, pri:26, gen: 0, proc:-1)  Ev(id: 24, pri:25, gen: 0, proc:-1)
  Ev(id: 25, pri:24, gen: 0, proc:-1)  Ev(id: 26, pri:23, gen: 0, proc:-1)  Ev(id: 27, pri:22, gen: 0, proc:-1)  Ev(id: 28, pri:21, gen: 0, proc:-1)  Ev(id: 29, pri:20, gen: 0, proc:-1)
  Ev(id: 30, pri:19, gen: 0, proc:-1)  Ev(id: 31, pri:18, gen: 0, proc:-1)  Ev(id: 32, pri:17, gen: 0, proc:-1)  Ev(id: 33, pri:16, gen: 0, proc:-1)  Ev(id: 34, pri:15, gen: 0, proc:-1)
  Ev(id: 35, pri:14, gen: 0, proc:-1)  Ev(id: 36, pri:13, gen: 0, proc:-1)  Ev(id: 37, pri:12, gen: 0, proc:-1)  Ev(id: 38, pri:11, gen: 0, proc:-1)  Ev(id: 39, pri:10, gen: 0, proc:-1)
  Ev(id: 40, pri: 9, gen: 0, proc:-1)  Ev(id: 41, pri: 8, gen: 0, proc:-1)  Ev(id: 42, pri: 7, gen: 0, proc:-1)  Ev(id: 43, pri: 6, gen: 0, proc:-1)  Ev(id: 44, pri: 5, gen: 0, proc:-1)
  Ev(id: 45, pri: 4, gen: 0, proc:-1)  Ev(id: 46, pri: 3, gen: 0, proc:-1)  Ev(id: 47, pri: 2, gen: 0, proc:-1)  Ev(id: 48, pri: 1, gen: 0, proc:-1)  Ev(id: 49, pri: 0, gen: 0, proc:-1)

Event_Gen generated 50 events
Event_Handler processed 50 events

PriQ_Event::  PriorityQueue_Event is empty !!

Events processed:
  Ev(no:10, pri:39, elap_t:    15[ms])  Ev(no:18, pri:31, elap_t:    15[ms])  Ev(no:45, pri: 4, elap_t:    15[ms])  Ev(no:49, pri: 0, elap_t:   299[ms])  Ev(no:48, pri: 1, elap_t:   479[ms])
  Ev(no:47, pri: 2, elap_t:   749[ms])  Ev(no:46, pri: 3, elap_t:  1093[ms])  Ev(no:44, pri: 5, elap_t:  1439[ms])  Ev(no:43, pri: 6, elap_t:  1634[ms])  Ev(no:42, pri: 7, elap_t:  1814[ms])
  Ev(no:41, pri: 8, elap_t:  1993[ms])  Ev(no:40, pri: 9, elap_t:  2248[ms])  Ev(no:39, pri:10, elap_t:  2457[ms])  Ev(no:38, pri:11, elap_t:  2727[ms])  Ev(no:37, pri:12, elap_t:  3071[ms])
  Ev(no:36, pri:13, elap_t:  3236[ms])  Ev(no:35, pri:14, elap_t:  3477[ms])  Ev(no:34, pri:15, elap_t:  3806[ms])  Ev(no:33, pri:16, elap_t:  4150[ms])  Ev(no:32, pri:17, elap_t:  4510[ms])
  Ev(no:31, pri:18, elap_t:  4884[ms])  Ev(no:30, pri:19, elap_t:  5200[ms])  Ev(no:29, pri:20, elap_t:  5529[ms])  Ev(no:28, pri:21, elap_t:  5783[ms])  Ev(no:27, pri:22, elap_t:  6082[ms])
  Ev(no:26, pri:23, elap_t:  6307[ms])  Ev(no:25, pri:24, elap_t:  6487[ms])  Ev(no:24, pri:25, elap_t:  6906[ms])  Ev(no:23, pri:26, elap_t:  7071[ms])  Ev(no:22, pri:27, elap_t:  7415[ms])
  Ev(no:21, pri:28, elap_t:  7639[ms])  Ev(no:20, pri:29, elap_t:  7953[ms])  Ev(no:19, pri:30, elap_t:  8238[ms])  Ev(no:17, pri:32, elap_t:  8448[ms])  Ev(no:16, pri:33, elap_t:  8837[ms])
  Ev(no:15, pri:34, elap_t:  9106[ms])  Ev(no:14, pri:35, elap_t:  9241[ms])  Ev(no:13, pri:36, elap_t:  9600[ms])  Ev(no:12, pri:37, elap_t:  9780[ms])  Ev(no:11, pri:38, elap_t: 10185[ms])
  Ev(no: 9, pri:40, elap_t: 10365[ms])  Ev(no: 8, pri:41, elap_t: 10619[ms])  Ev(no: 7, pri:42, elap_t: 10919[ms])  Ev(no: 6, pri:43, elap_t: 11068[ms])  Ev(no: 5, pri:44, elap_t: 11278[ms])
  Ev(no: 4, pri:45, elap_t: 11683[ms])  Ev(no: 3, pri:46, elap_t: 12042[ms])  Ev(no: 2, pri:47, elap_t: 12461[ms])  Ev(no: 1, pri:48, elap_t: 12686[ms])  Ev(no: 0, pri:49, elap_t: 12917[ms])

Minimum event processing time:    14.93[ms] for Ev(no:10, pri:39, elap_t:    15[ms])
Maximum event processing time: 12916.99[ms] for Ev(no: 0, pri:49, elap_t: 12917[ms])
Average event processing time:  6119.16[ms] for total 50 events

Thread_EventGenerator is terminated !!
Thread_EventHandler is terminated !!
```

Debugging of Multi-Thread Operations

◆ Visual Studio Multi-thread Information

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



Homework 13

Homework 13

13.1 Multi-thread 프로그램에서 공유 자원에 Critical section (임계구역) 설정이 필요한 이유에 대하여 구체적으로 예를 들어 설명하라.

13.2 C++ 프로그래밍 환경에서 Thread 생성과 실행을 위한 파라메터/인수를 전달하는 방법에 대하여 구체적으로 예를 들어 설명하고, 일반 함수 호출에서의 인수 전달방법과의 차이점에 대하여 설명하라.

13.3 Multi-thread 의 동작 상태를 monitoring 하여, 주기적으로 상태를 출력하는 함수를 구상하고, 필요한 파라메터 전달, 출력 포맷에 따른 주기적인 출력 방법에 대하여 예를 들어 설명하라.

13.4 동적으로 할당된 배열을 기반으로 구현된 Circular FIFO Queue의 기본 기능인 enCirQ()와 deCirQ() 함수의 기본 구조 및 동작 원리에 대하여 그림으로 나타내어 설명하라. (사용되는 구조체의 모습을 그림으로 표현할 것.)

13.5 우선 순위를 고려한 Event처리를 위하여 사용되는 Priority Queue에서 우선 순위가 높은 event가 우선적으로 처리될 수 있는 구조와 동작 원리 (enPriQ()에서의 up-heap bubbling, dePriQ()에서의 down-heap bubbling 포함)에 대하여 상세히 설명하라.