프로그래밍언어 (실습)

실습 12 (보충설명) – Simulation of Event Handling with Multi-thread and Priority Queue

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)

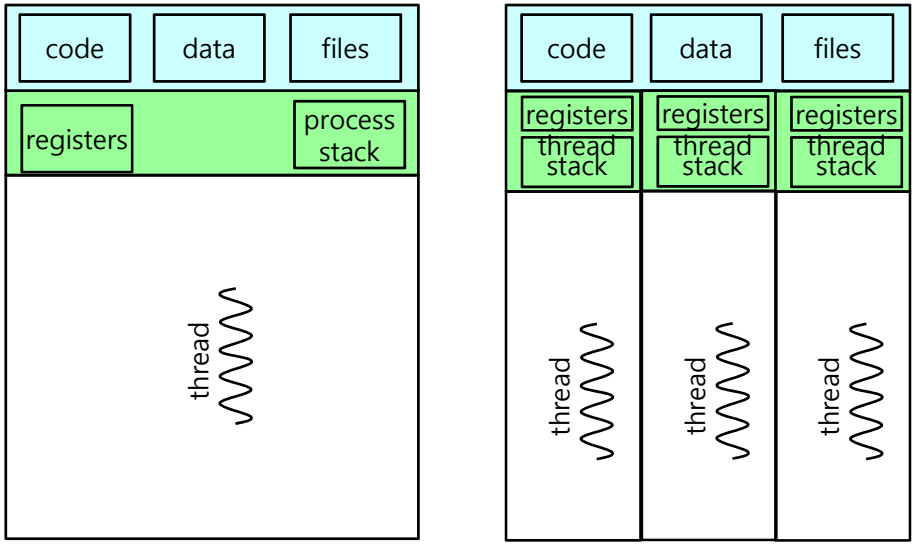
Outline

- **Simulation of Event Generation and Event Handling**
  - Example: Call Center
- **Event**
- **Priority Queue**
- **library <mutex>**
- **library <thread>**
- **Event Generator Thread**
- **Event Handler Thread**
- **Thread Monitoring**

프로세스 (Process)와 스레드 (Thread)

◆ Thread 란?

- 어떠한 프로그램 내에서, 특히 프로세스(process) 내에서 실행되는 흐름의 단위.
- 일반적으로 한 프로그램은 하나의 thread를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 thread들을 교대로 실행하게 구성할 수 있다. 이를 멀티스레드(multi-thread)라 한다.
- 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 동시에 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

C++11의 멀티스레드 관련 클래스 및 멤버함수

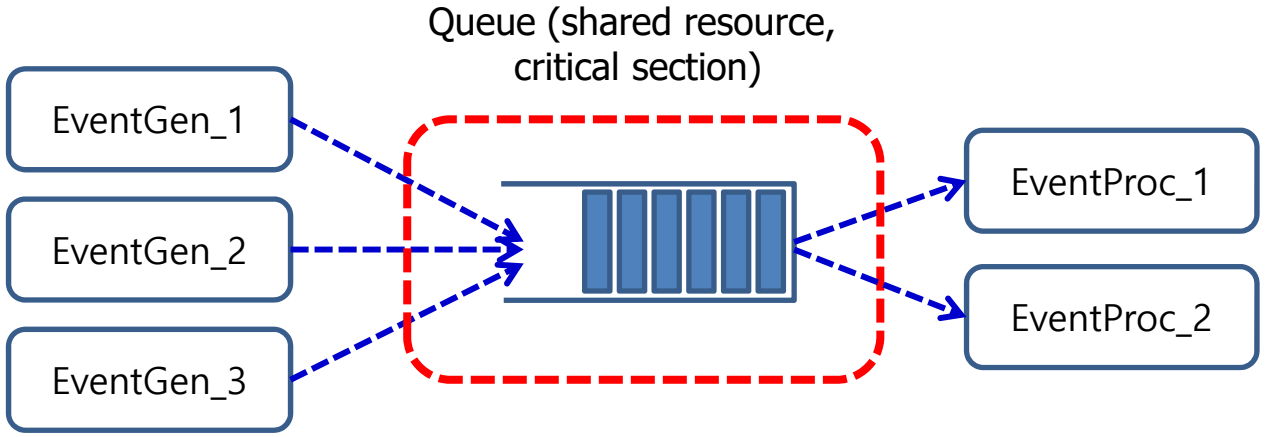
◆C++11의 스레드 관련 클래스 및 멤버 함수

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(sleep_duration_ms)	Windows 운영체제에서 제공하는 API 함수 (#include <Windows.h> 필요) sleep_duration_ms은 milli-second 단위

스레드와 스레드 간의 정보 전달

◆ 큐를 사용한 정보 전달

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



임계구역 (Critical Section)

◆ 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

은행잔고
g_account

```
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.  }
```

mutex (mutual exclusion)

- **mutex의 설정: 현재 어떤 스레드/프로세스가 실행 중에 있다는 상태를 mutex을 표시하는 변수로 표시**
  - semaphore 라고 부르기도 함
- **mutex 변수의 설정**
  - C++ 환경에서
    #include <mutex>
    using namespace std;
  - mutex mtx
    - mutex 생성
    - mutex의 lock() 및 unlock() 실행 이전에 생성되어 있어야 함
- **mutex를 사용한 critical section 영역 지정**
  - mtx.lock()
  - mtx.unlock()

임계구역 (Critical Section)

◆ 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.  }
```

mutex (mutual exclusion)

- **mutex의 설정: 현재 어떤 스레드/프로세스가 실행 중에 있다는 상태를 mutex을 표시하는 변수로 표시**
  - semaphore 라고 부르기도 함
- **mutex 변수의 설정**
  - C++ 환경에서
    #include <mutex>
    using namespace std;
  - mutex mtx
    - mutex 생성
    - mutex의 lock() 및 unlock() 실행 이전에 생성되어 있어야 함
- **mutex를 사용한 critical section 영역 지정**
  - mtx.lock()
  - mtx.unlock()

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
```

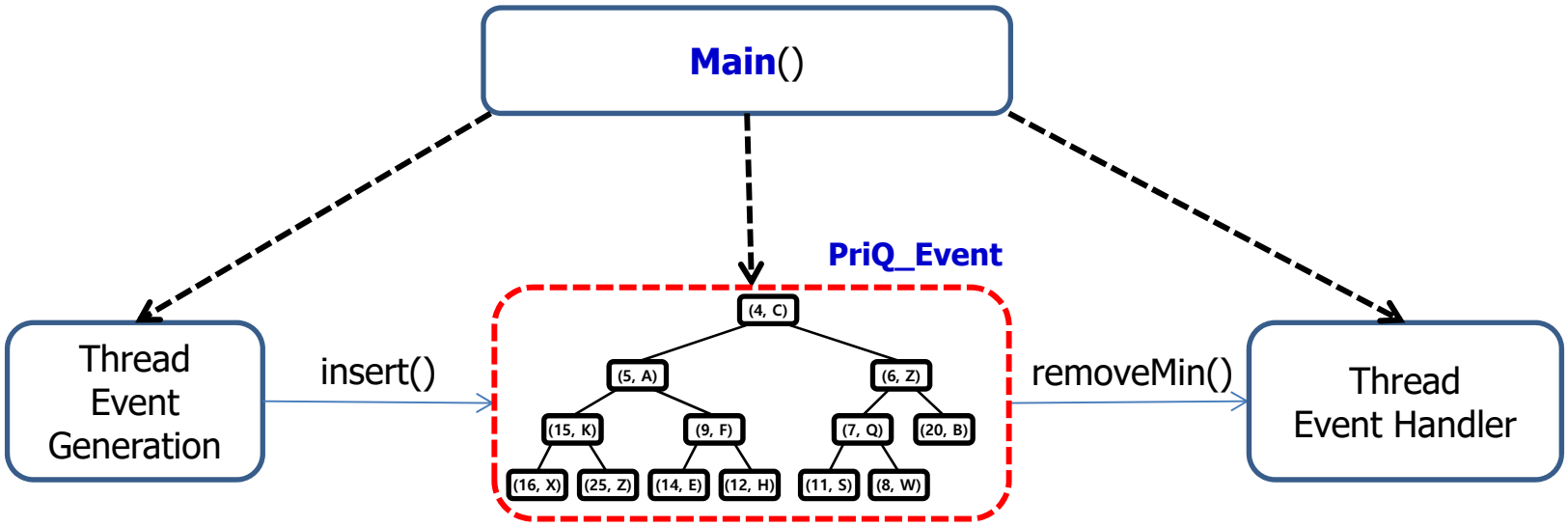
```
/* Event.cpp  (1) */
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include "Event.h"

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);
}
```

Event Handling with Multi-threads and Priority Queue

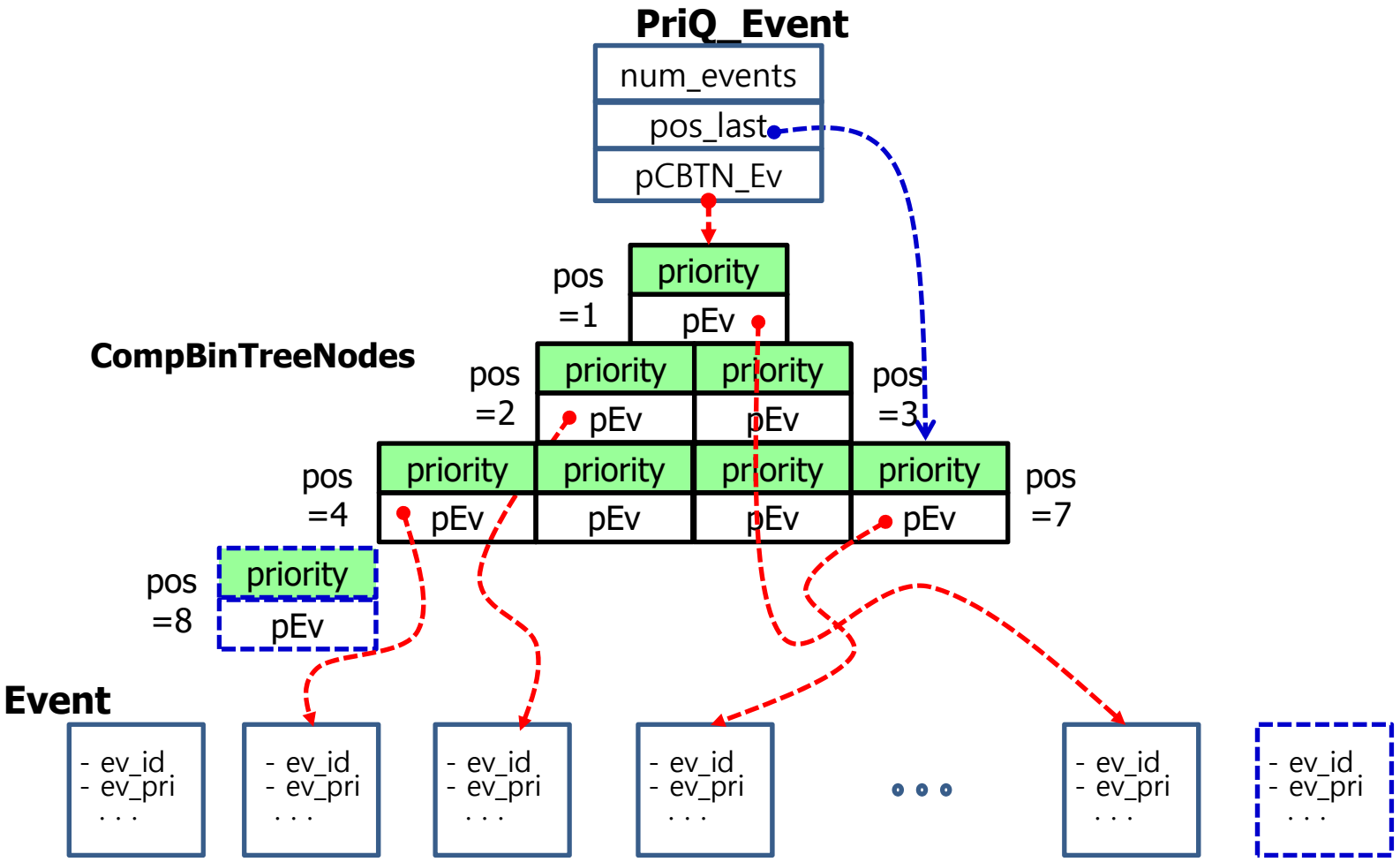
◆ Two Threads with Priority Queue

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



Heap Priority Queue

◆ Heap Priority Queue



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.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};

#define THREAD_RETURN_CODE 7
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;
    PriQ_Event *pPriQ_Event;
    ROLE role;
    int myAddr;
    int maxRound;
    int targetEventGen;
    ThreadStatusMonitor *pThrdMon;
} ThreadParam_Event;

void Thread_EventHandler(ThreadParam_Event
    *pParam);
void hread_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;

    srand(time(NULL));
    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);
        pParam->pMTX_thrd_mon->lock();
        pThrdMon->numEventGenerated++;
        pThrdMon->totalEventGenerated++;
        pThrdMon->eventGenerated[pThrdMon->numEventGenerated] = *pEv;
        pParam->pMTX_thrd_mon->unlock();

        while (enCirQ_Event(pCirQ_Event, *pEv) == NULL)
        {
           Sleep(500);
        }
        free(pEv);
        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;

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

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

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

}
```

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 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;

    initPriQ_Event(&priQ_Event, "PriQ_Event", 1);

    ThreadParam_Event thrdParam_EventGen, thrdParam_EventHndlr;
    HANDLE hThrd_EventGenerator, hThrd_EventHandler;
    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);
    cs_main.lock();
    printf("Thread_EventGen is created and activated ...\n");
    cs_main.unlock();

    for (int round = 0; round < MAX_ROUND; round++)
    {
        //cs_main.lock();
        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;
        }

        //cs_main.unlock();
        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");

}
```

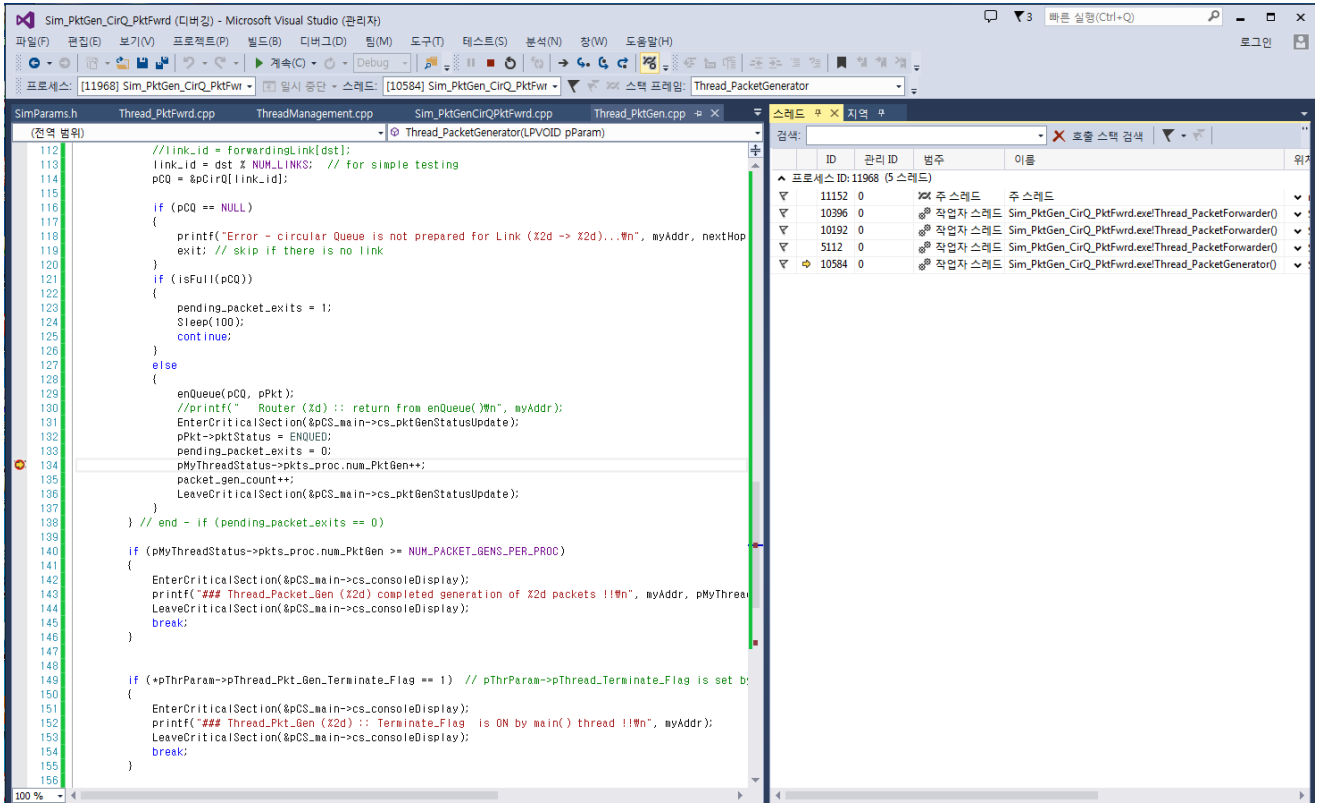
실행 결과

```
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"



Oral Test

Q 12.1 다중 스레드 구조의 프로그램에서 공유 자원의 사용에 대한 Critical section (임계구역) 설정이 필요한 이유에 대하여 예를 들어 구체적으로설명하고, 임계 구역 설정을 하기 위하여 mutex를 설정하고 사용하는 방법에 대하여 예를 들어 구체적으로 설명하라.

Q 12.2 스레드를 생성하는 방법과 생성되는 스레드에 파라메터를 전달하는 방법에 대하여 예를 들어 구체적으로 설명하라.

Q 12.3 Multi-thread의 동작 상태를 monitoring하여, 주기적으로 상태를 출력하는 방법에 대하여 예를 들어 구체적으로 설명하라. 특히 관련 구조체, 구조체의 변수를 누가 언제 변경하고, 누가 언제 출력하게 되는가에 상세하게 설명하라.

Q 12.4 우선 순위를 고려한 Event처리를 위하여 사용되는 Priority Queue에서 우선 순위가 높은 event가 우선적으로 처리될 수 있는 내부 구조와 동작 원리에 대하여 예를 들어 구체적으로 설명하라.