2015 *yuANTL*

Ch 11-2. Multi-Thread

2015년 6월 5일

교수 김 영 탁

영남대학교 공과대학 정보통신공학과

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

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

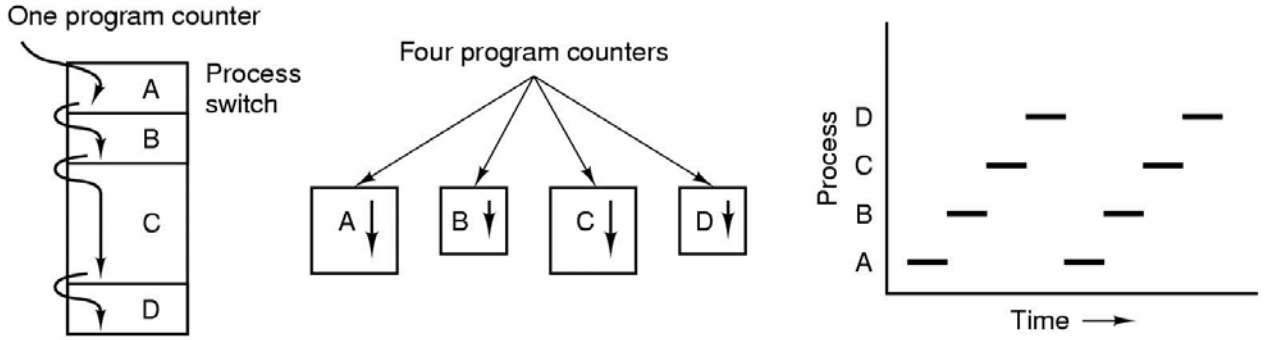
Outline

- 프로세스와 스레드
- Task 수행이 병렬로 처리되어야 하는 경우
  - 왜 다중 프로세스/다중 스레드가 필요한가?
- 다중 프로세스와 다중 스레드의 차이점
- Windows 환경에서의 다중 스레드 생성, 초기화 및 종료
- 다중 스레드로의 파라메터 전달
- 큐를 이용한 다중 스레드간의 데이터 전달
- Critical Section을 사용한 다중 스레드간의 동기화
- turn semaphore를 사용한 실행 순서 관리
- 간단한 다중 스레드의 예제

프로세스 (Process)

◆ Process

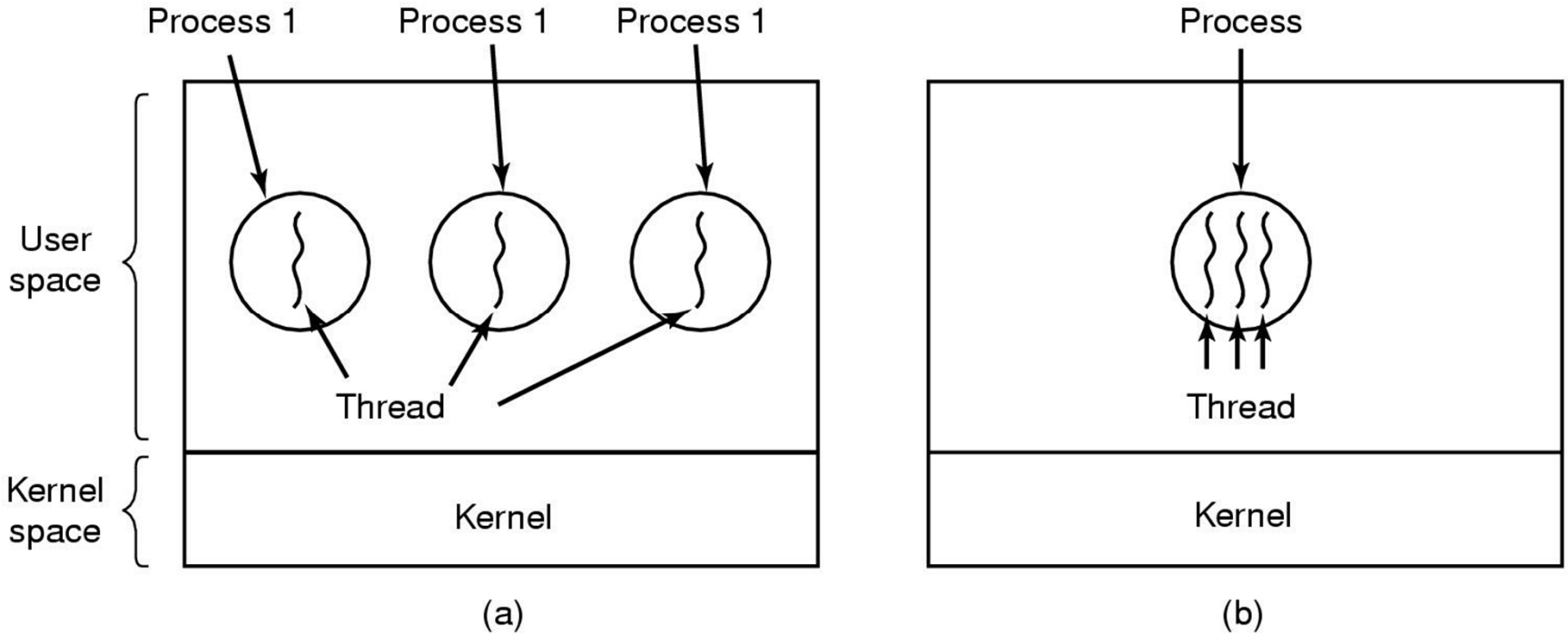
- 프로세스 (process)란 프로그램이 수행중인 상태 (***program in execution***)
  - 각 프로세스 마다 개별적으로 메모리가 할당 됨 (text core, initialized data (BSS), non-initialized data, heap (dynamically allocated memory), stack)
- 일반적인 PC나 대부분의 컴퓨터 환경에서 하나의 물리적인 CPU 상에 다수의 프로세스가 실행되는 ***Multi-tasking*** 이 지원되며, 운영체제가 다수의 프로세스를 일정 시간 마다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



스레드 (Thread)

◆ 스레드 (Thread)

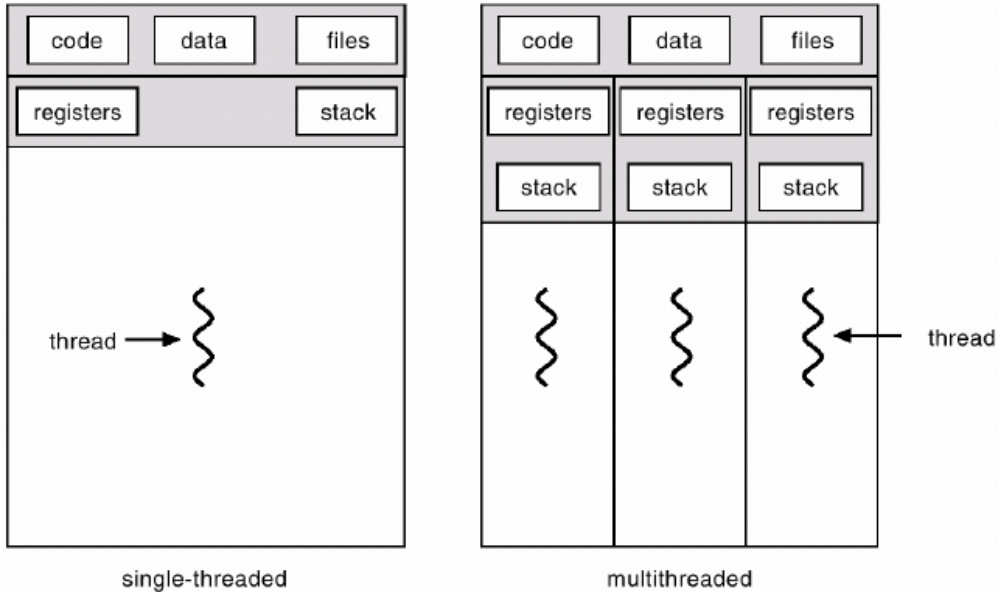
- 스레드는 하나의 프로세스 내부에 포함되는 함수들이 동시에 실행될 수 있게 한 작은 단위 프로세서 (lightweight process)
- 기본적으로 CPU를 사용하게 하는 기본 단위
- 하나의 프로세스에 포함된 다수의 스레드 들은 프로세스의 메모리 자원들 (code section, data section, Heap 등)과 운영체제의 자원들 (예: 파일 입출력 등)을 공유함



프로세스 (Process)와 스레드 (Thread)의 차이점

◆ 스레드(Thread) 란?

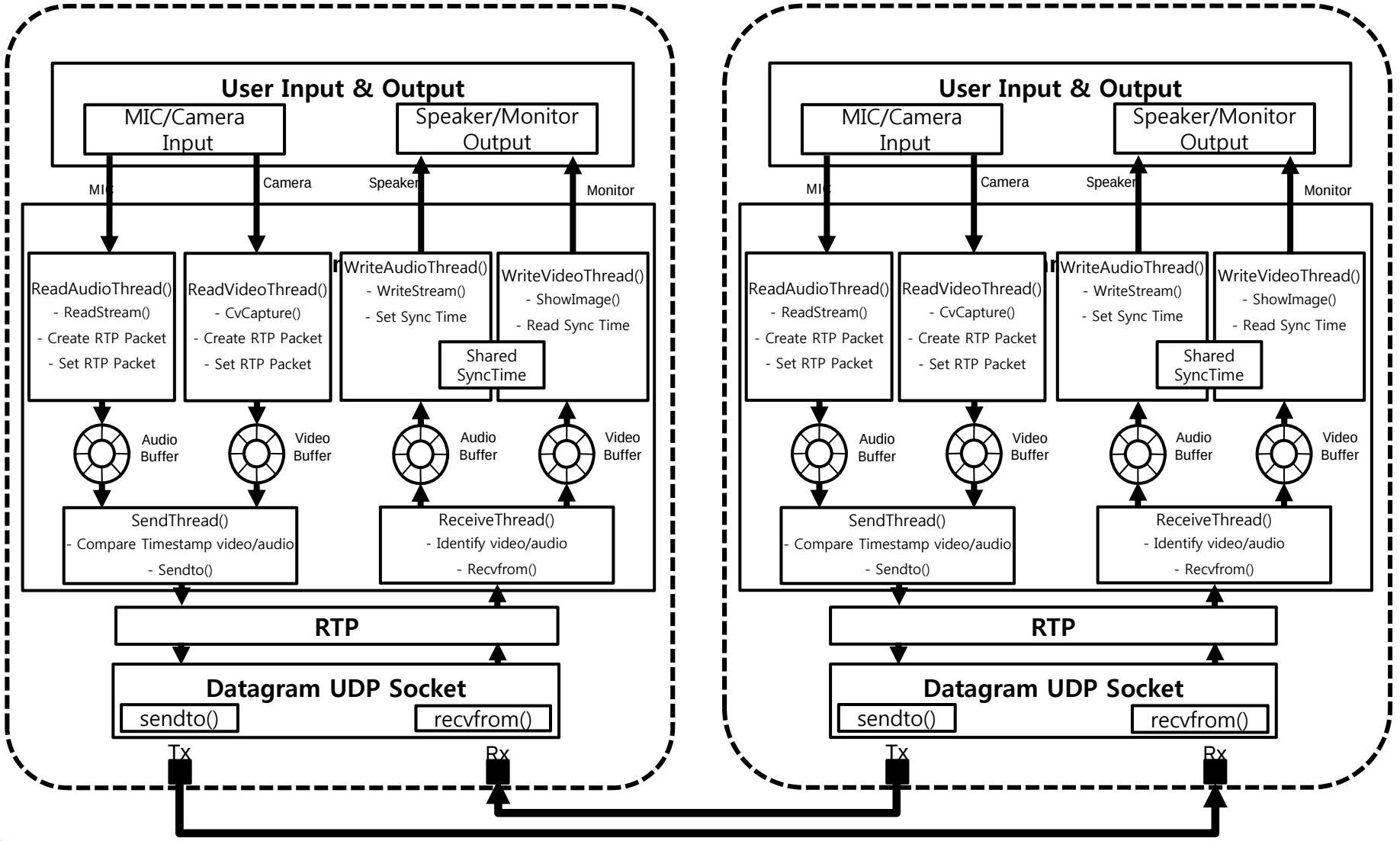
- 어떠한 프로그램 내에서, 특히 프로세스(process) 내에서 실행되는 흐름의 단위.
- 일반적으로 한 프로그램은 하나의 thread를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 thread를 동시에 실행할 수 있다. 이를 멀티스레드(multithread)라 한다.
- 프로세스는 각각 개별적인 code, data, file을 가지나, 스레드는 자신들이 포함된 프로세스의 code, data, file들을 공유함



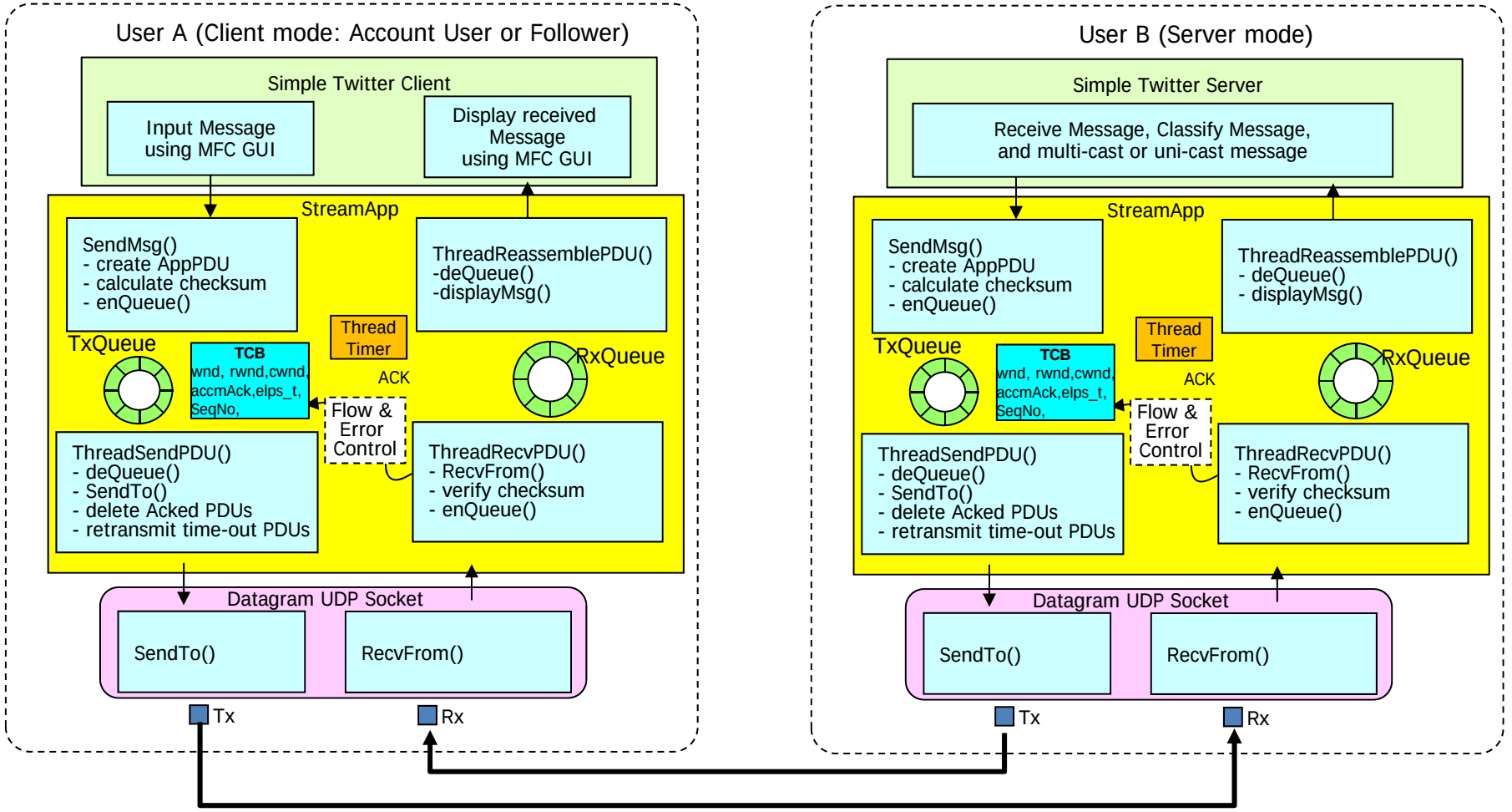
Task 수행이 병렬로 처리되어야 하는 경우

- **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

◆ 실시간 화상 전화기의 기능 블럭도



◆ 양방향 동시 전송 (full-duplex) Twitter



스레드의 생성 및 초기화

◆ 스레드 함수의 구현

- 프로그램에 포함되는 함수 중, 병렬로 실행되어야 하는 함수를 스레드로 지정
- 파라메터 구조체 포인터를 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
- 스레드 함수의 예

```
DWORD WINAPI Thread_producer(LPVOID pParam)
{
    ThreadParam *pThrParam;
    /*void* 자료형으로 스레드에 전달된 인자를 형변환을 통해 pThrParam 구조체로 변환*/
    pThrParam = (ThreadParam *)pParam;
    . . . .
    while (1)
    {

     }
}
```

스레드 함수로의 파라메터 전달

◆ 스레드 파라메터 전달을 위한 구조체 정의 (예)

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

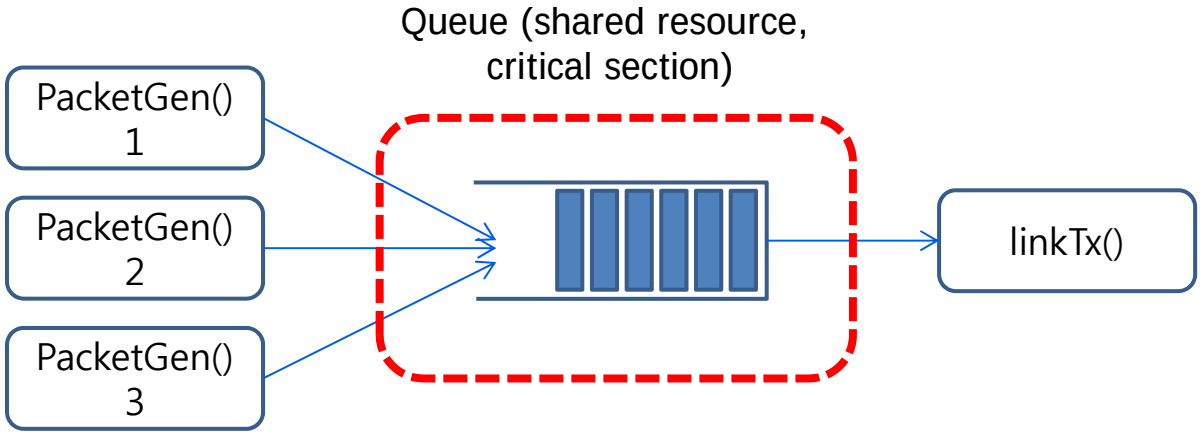
```
typedef struct ThreadParam
{
    Circular_Int_Queue *queue;
    CRITICAL_SECTION* pCS;
    int role;
} ThreadParam;
```

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

스레드 간의 정보 전달

◆ 큐를 사용한 정보 전달

- 스레드 간에 정보/메시지/신호를 전달하기 위하여 FIFO 동작을 수행하는 queue를 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



Critical Section (1)

◆ Critical section

- 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 상태가 임시 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 스레드 예에서 critical section으로 보호하여야 할 구역은 ?

```
Thread_Deposit (int deposit)
{
   // account is shared variable
   cur_account = account;

   cur_account = cur_account +
      deposit;
   account = cur_account;
   print(account);
   . . . .
}
```


```
Thread_Withdraw (int withdraw)
{
   // account is shared variable
   cur_account = account;
   cur_account = cur_account -
      withdraw;
   account = cur_account;
   print(account);
   . . . .
}
```

Critical Section (2)

- **Critical section를 현재 어떤 스레드/프로세스가 실행중에 있다는 상태를 Critical section을 표시하는 변수로 표시**
  - semaphore라고 부르기도 함
- **Critical section을 표시하는 변수의 설정**
  - InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
    - initialization of critical section
    - must be called before using EnterCriticalSection(), LeaveCriticalSection()
- **Critical section 영역 지정**
  - EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
  - LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
    - the procedures between these two functions are defined as critical section
- **Critical section을 표시하는 변수의 소멸**
  - DeleteCriticalSection(LPCRITICAL_SECTION lplCriticalSection)
    - delete the critical section flag
    - executed at the process exit

스레드의 종료 (1)

◆ MS-Windows 환경에서의 스레드 소멸 및 관리를 위한 함수들

- 생성된 스레드가 스스로 함수 실행을 종료할 때 까지 기다리거나, 지정된 시간 이후에 강제 종료를 시킬 수 있음
- WaitForSingleObject()와 TerminateThread() 함수를 사용

```
HANDLE m_hThread;

m_hThread = CreateThread(NULL, 0, Thread_Function_Name, pThrdParam, 0, NULL);
. . . . // 생성된 스레드가 별도로 실행이 됨

DWORD nExitCode = NULL;
GetExitCodeThread(m_hThread, &nExitCode);
WaitForSingleObject(m_hthread, THRD_EXE_TIME_MS); // wait for terminate thread
TerminateThread(m_hThread, nExitCode);
CloseHandle(m_hThread);
```

스레드의 종료 (2)

◆ 스레드가 스스로 종료할 때 까지 기다리는 경우

- main() 함수에서는 무한대 시간(INFINITIVE)으로 기다림

```
WaitForSingleObject(m_hthread,  INFINITIVE); // wait for terminate thread
TerminateThread(m_hThread, nExitCode);
```

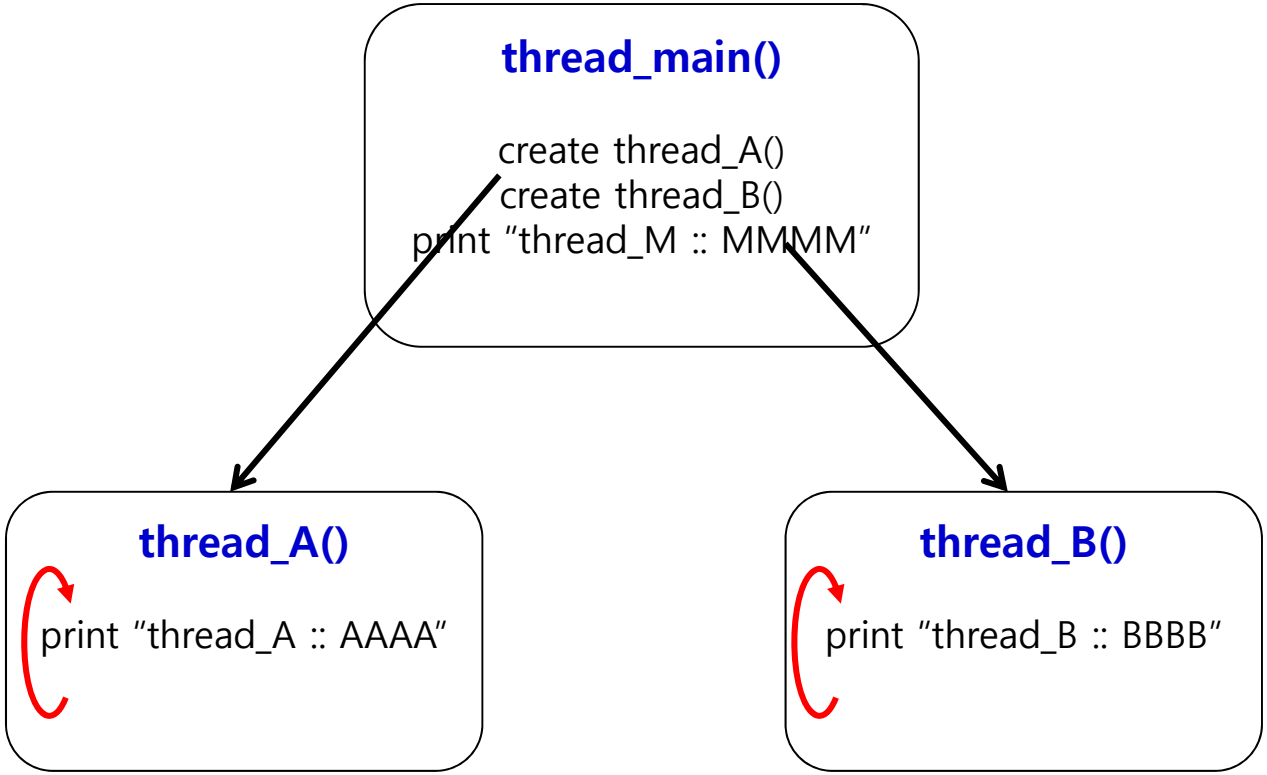
◆ 스레드를 일정 시간 동안 실행하게 한 후, 강제 종료를 시키는 경우

```
#define THRD_EXE_TIME_MS  5000 // mili-second 단위의 시간

WaitForSingleObject(m_hthread, THRD_EXE_TIME_MS); // wait for terminate thread
TerminateThread(m_hThread, nExitCode);
```

Two Simple Threads

- **version 1**



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

#include<stdio.h>
#include<windows.h>
#include<time.h>
enum ROLE { PRODUCER, CONSUMER };
typedef struct ThreadParam{
    char mark;
} ThreadParam;

DWORD WINAPI Thread_A(LPVOID pParam);
DWORD WINAPI Thread_B(LPVOID pParam);

void main()
{
    /*   변수 선언 */
    ThreadParam *pThrParam; /* 각 스레드로 전달될 파라미터 구조체*/
    HANDLE hThread_A, hThread_B; /* 스레드 정보를 관리하게 될 핸들러 변수 */
    DWORD nExitCode = NULL;

    /* thread_A에 전달 될 파라미터값 초기화 */
    pThrParam = (ThreadParam*)malloc(sizeof(ThreadParam));
    pThrParam->mark = 'A';
    /*  CreateThread API를 이용하여 Thread 생성과 전달할 인자를 전달한 후
        반환되어지는 해당 Thread에 대한 정보를 hThread_A 핸들러에 저장  */
    hThread_A = CreateThread(NULL, 0, Thread_A, pThrParam, 0, NULL);
```

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

    /* thread_B에 전달 될 파라미터값 초기화 */
    pThrParam = (ThreadParam*)malloc(sizeof(ThreadParam));
    pThrParam->mark = 'B';
    hThread_B = CreateThread(NULL, 0, Thread_B, pThrParam, 0, NULL);
    /* main() thread 실행 */
    char mark = 'M';
    for (int i = 0; i < 100; i++)
    {
        printf("Thread_main ... ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("₩n");
    }
    /* main 스레드가 생성한 thread_A가 스스로 종료할 때까지 대기*/
    WaitForSingleObject(hThread_A, INFINITE);
    GetExitCodeThread(hThread_A, &nExitCode);
    TerminateThread(hThread_A, nExitCode); /* thread_A  종료 */
    CloseHandle(hThread_A); /* 스레드 핸들러 종료 */

    /* main 스레드가 생성한 thread_B 가 스스로 종료할 때까지 대기*/
    WaitForSingleObject(hThread_B, INFINITE);
    GetExitCodeThread(hThread_B, &nExitCode);
    TerminateThread(hThread_B, nExitCode); /* thread _B 종료*/
    CloseHandle(hThread_B); /* 스레드 핸들러 종료 */

}
```

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

DWORD WINAPI Thread_A(LPVOID pParam)
{
    ThreadParam *pThrParam =
          (ThreadParam *)pParam;
    int data = 0;
    int sleep_time_ms = 0;
    char mark = pThrParam->mark;
    for (int i = 0; i < 100; i++)
    {
        printf("Thread_A ... ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("₩n");
    }
    return 0;
}
```

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

DWORD WINAPI Thread_B(LPVOID pParam)
{
    ThreadParam *pThrParam =
       (ThreadParam *)pParam;
    char mark = pThrParam->mark;

    for (int i = 0; i < 100; i++)
    {
        printf("Thread_B ... ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("₩n");
     }
    return 0;
}
```

◆ 실행결과

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

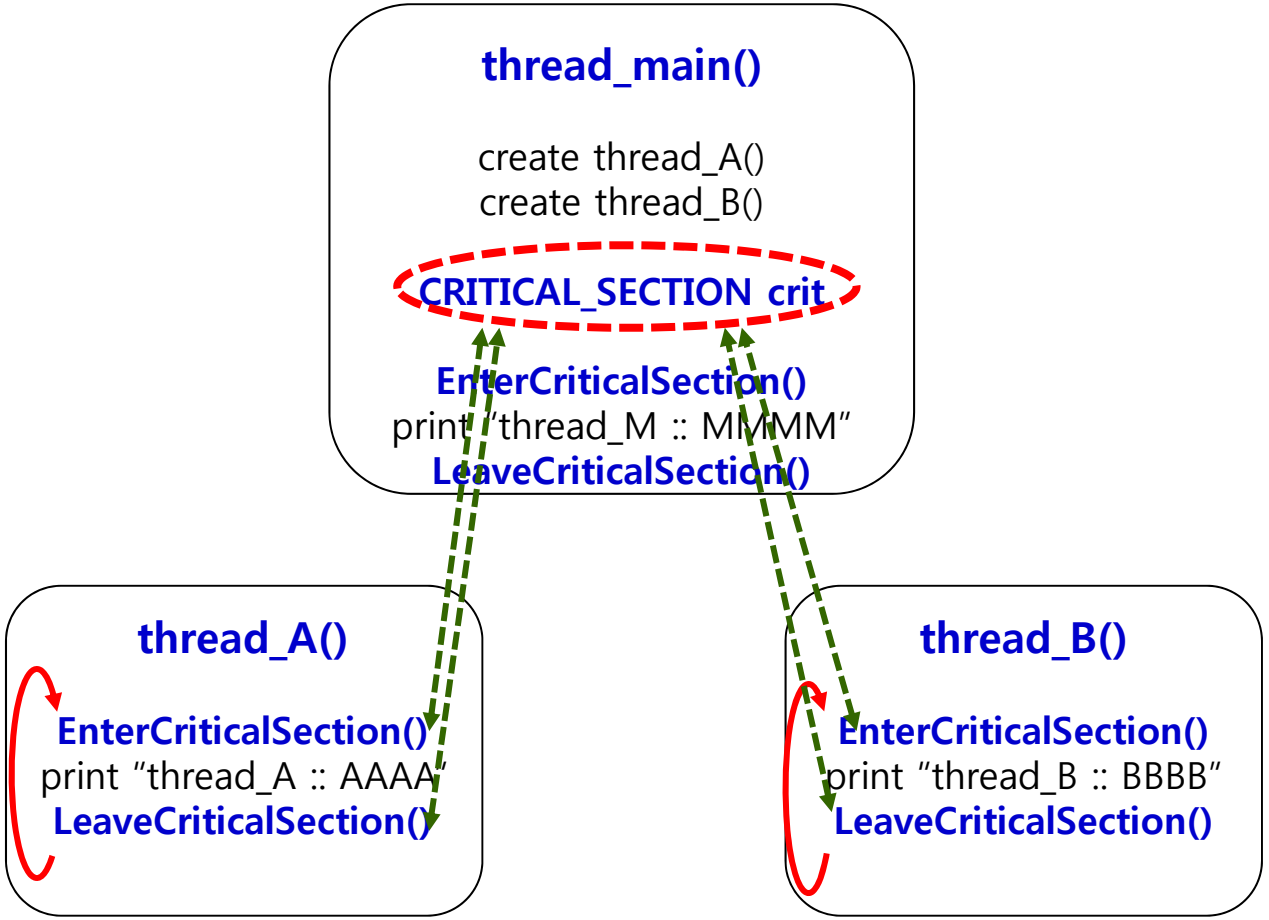
◆ 만약, 한번에 thread하나가 한 줄씩만 출력하고, 번갈아 가며 출력하기 위해서는 어떻게 수정하여야 하나?

- critical section 사용: 한 줄을 다 출력 할 때까지, 다른 스레드가 실행되지 못하도록 보호
- “turn” semaphore 사용: 한 줄을 다 출력한 후에는 반드시 다른 스레드가 실행되도록 실행 순서 (turn) 제어

```
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBThread_A ... AAAABBAABBAABBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBAABBAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBB
```

Two Simple Threads

- version 2



```
/* SimpleThreadsVer2.cpp  (1) */

#include<stdio.h>
#include<windows.h>
#include<time.h>
enum ROLE { PRODUCER, CONSUMER };
typedef struct ThreadParam{
    CRITICAL_SECTION* pCS;
    char mark;
}ThreadParam;

DWORD WINAPI Thread_A(LPVOID pParam);
DWORD WINAPI Thread_B(LPVOID pParam);

void main()
{
    /*    변수 선언 */
    CRITICAL_SECTION crit;
    ThreadParam *pThrParam;
    HANDLE hThread_A, hThread_B;
    DWORD nExitCode = NULL;

    /* 변수 초기화*/
    InitializeCriticalSection(&crit);
```

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

    /* Consumer 스레드에 전달 될 파라미터값 초기화 */
    pThrParam = (ThreadParam*)
        malloc(sizeof(ThreadParam));
    pThrParam->pCS = &crit;
    pThrParam->mark = 'A';
    hThread_A = CreateThread(NULL, 0, Thread_A,
        pThrParam, 0, NULL);

    /* Producer 스레드에 전달 될 파라미터값 초기화 */
    pThrParam = (ThreadParam*)
        malloc(sizeof(ThreadParam));
    pThrParam->pCS = &crit;
    pThrParam->mark = 'B';
    hThread_B = CreateThread(NULL, 0, Thread_B,
        pThrParam, 0, NULL);

    WaitForSingleObject(hThread_A, INFINITE);
    GetExitCodeThread(hThread_A, &nExitCode);
    TerminateThread(hThread_A, nExitCode);
    CloseHandle(hThread_A);

    WaitForSingleObject(hThread_B, INFINITE);
    GetExitCodeThread(hThread_B, &nExitCode);
    TerminateThread(hThread_B, nExitCode);
    CloseHandle(hThread_B);

    DeleteCriticalSection(&crit);
}
```

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

DWORD WINAPI Thread_A(LPVOID pParam)
{
    ThreadParam *pThrParam;
    pThrParam = (ThreadParam *)pParam;
    char turn = '₩0';
    char mark = pThrParam->mark;
    for (int i = 0; i < 20; i++)
    {
        EnterCriticalSection(pThrParam->pCS);
        printf("Thread_A ... ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("₩n");
        LeaveCriticalSection(pThrParam->pCS);
        Sleep(100);
    }
    printf("Thread_A finished ...₩n");
    return 0;
}
```

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

DWORD WINAPI Thread_B(LPVOID pParam)
{
    ThreadParam *pThrParam;
    pThrParam = (ThreadParam *)pParam;
    char mark = pThrParam->mark;
    char turn;

    for (int i = 0; i < 20; i++)
    {
        EnterCriticalSection(pThrParam->pCS);
        printf("Thread_B ... ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("₩n");
        LeaveCriticalSection(pThrParam->pCS);
        Sleep(100);
    }
    printf("Thread_B finished ...₩n");
    return 0;
}
```

◆ 실행결과 (ver 2)

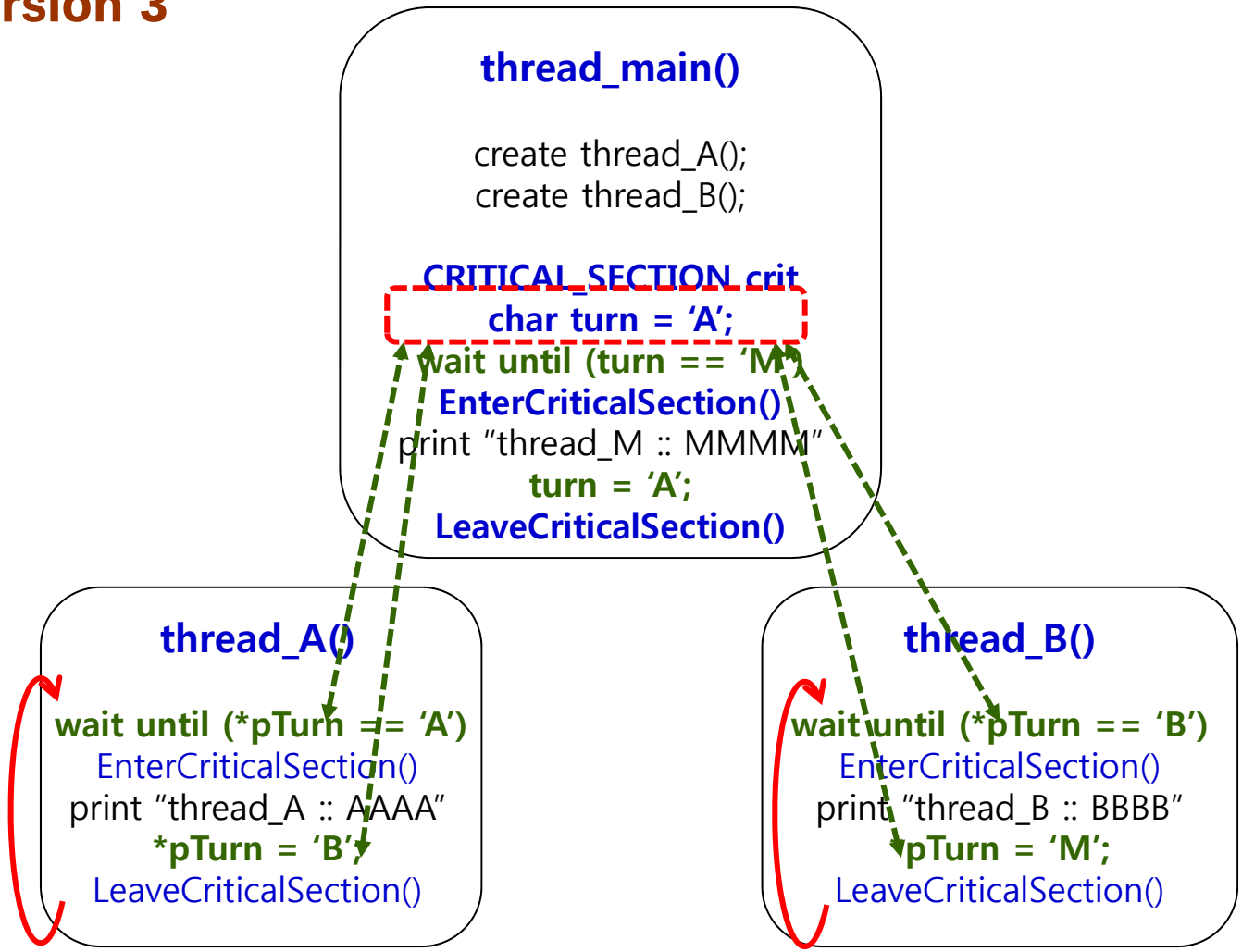
- critical section을 사용하여, 하나의 스레드가 한 줄의 출력을 완전히 완료할 때까지 다른 스레드가 실행되지 못하게 보호
- thread_A 또는 thread_B가 두 줄씩 출력하는 경우가 있음

◆ 각 스레드가 한번에 한 줄씩만 출력하게 하는 방법은 ?

```
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_A finished ...
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_B finished ...
계속하려면 아무 키나 누르십시오 . . .
```

Two Simple Threads

- **version 3**



```
/* SimpleThreadsVer3.cpp  (1) */

#include<stdio.h>
#include<windows.h>
#include<time.h>
enum ROLE { PRODUCER, CONSUMER };
typedef struct ThreadParam{
    CRITICAL_SECTION* pCS;
    char mark;
    char *pTurn;
}ThreadParam;

DWORD WINAPI Thread_A(LPVOID pParam);
DWORD WINAPI Thread_B(LPVOID pParam);

void main()
{
    /*    변수 선언 */
    CRITICAL_SECTION crit;
    ThreadParam *pThrParam;
    HANDLE hThread_A, hThread_B;
    DWORD nExitCode = NULL;
    char turn = 'A';

    /* 변수 초기화*/
    InitializeCriticalSection(&crit);
```

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

    /* Consumer 스레드에 전달 될 파라미터값 초기화 */
    pThrParam = (ThreadParam*)
        malloc(sizeof(ThreadParam));
    pThrParam->pCS = &crit;
    pThrParam->mark = 'A';
    pThrParam->pTurn = &turn;
    hThread_A = CreateThread(NULL, 0, Thread_A,
        pThrParam, 0, NULL);

    /* Producer 스레드에 전달 될 파라미터값 초기화 */
    pThrParam = (ThreadParam*)
        malloc(sizeof(ThreadParam));
    pThrParam->pCS = &crit;
    pThrParam->mark = 'B';
    pThrParam->pTurn = &turn;
    hThread_B = CreateThread(NULL, 0, Thread_B,
      pThrParam, 0, NULL);

    WaitForSingleObject(hThread_A, INFINITE);
    GetExitCodeThread(hThread_A, &nExitCode);
    TerminateThread(hThread_A, nExitCode);
    CloseHandle(hThread_A);

    WaitForSingleObject(hThread_B, INFINITE);
    GetExitCodeThread(hThread_B, &nExitCode);
    TerminateThread(hThread_B, nExitCode);
    CloseHandle(hThread_B);

    DeleteCriticalSection(&crit);
}
```

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

DWORD WINAPI Thread_A(LPVOID pParam)
{
    ThreadParam *pThrParam;
    pThrParam = (ThreadParam *)pParam;
    char mark = pThrParam->mark;
    char turn = '₩0';
    for (int i = 0; i < 20; i++)
    {
        do {
            EnterCriticalSection(pThrParam->pCS);
            turn = *pThrParam->pTurn;
            LeaveCriticalSection(pThrParam->pCS);
            if (turn == 'A')
                break;
            else
                Sleep(100);
        } while (turn != 'A');
        EnterCriticalSection(pThrParam->pCS);
        printf("Thread_A ... ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("₩n");
        *pThrParam->pTurn = 'B';
        LeaveCriticalSection(pThrParam->pCS);
        Sleep(10);
    }
    printf("Thread_A finished ...₩n");
    return 0;
}
```

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

DWORD WINAPI Thread_B(LPVOID pParam)
{
    ThreadParam *pThrParam;
    pThrParam = (ThreadParam *)pParam;
    char mark = pThrParam->mark;
    char turn = '₩0';
    for (int i = 0; i < 20; i++)
    {
        do {
            EnterCriticalSection(pThrParam->pCS);
            turn = *pThrParam->pTurn;
            LeaveCriticalSection(pThrParam->pCS);
            if (turn == 'B')
                break;
            else
                Sleep(10);
        } while (turn != 'B');
        EnterCriticalSection(pThrParam->pCS);
        printf("Thread_B ... ");
        for (int j = 0; j < 50; j++)
        {
            printf("%c", mark);
        }
        printf("₩n");
        *pThrParam->pTurn = 'A';
        LeaveCriticalSection(pThrParam->pCS);
        Sleep(100);
    }
    printf("Thread_B finished ...₩n");
    return 0;
}
```

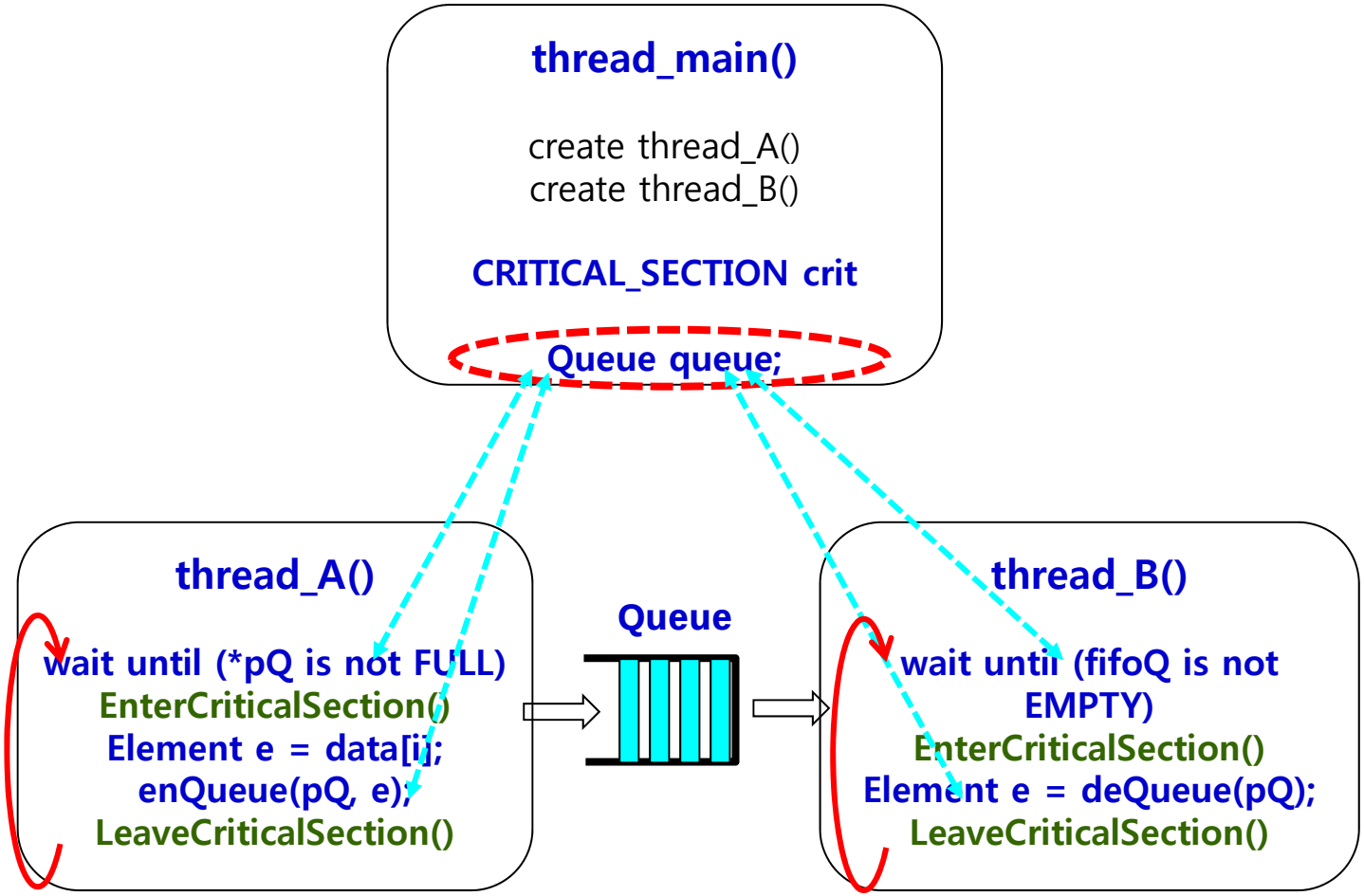
◆ 실행결과 (ver 3)

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

```
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A ... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread_B ... BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread_A finished ...
Thread_B finished ...
계속하려면 아무 키나 누르십시오 . . .
```

Two Simple Threads with shared Queue

◆ **version 4 – shared Queue 사용**



Shared Queue의 구현 방법 (1)

◆ Circular Queue

- array 기반의 구현
- first_index, last_index가 저장된 데이터를 가르킴
  - first_index와 last_index의 초기값은 0
  - index 값은 0 ~ N-1 범위의 값을 가지며, N-1 이후에는 0으로 순환됨
- enQueue()
  - pQ->data[last_index] = element;
  - pQ-> last_index = (pQ-> last_index + 1) % N;
  - pQ->num_data++;
- deQueue()
  - element = pQ->data[first_index];
  - pQ-> first_index = (pQ-> first_index + 1) % N;
  - pQ->num_data--;

Shared Queue의 구현 방법 (2)

◆ List Queue

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

◆ Priority Queue

- 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

간단한 Producer – Consumer 스레드 구현 예

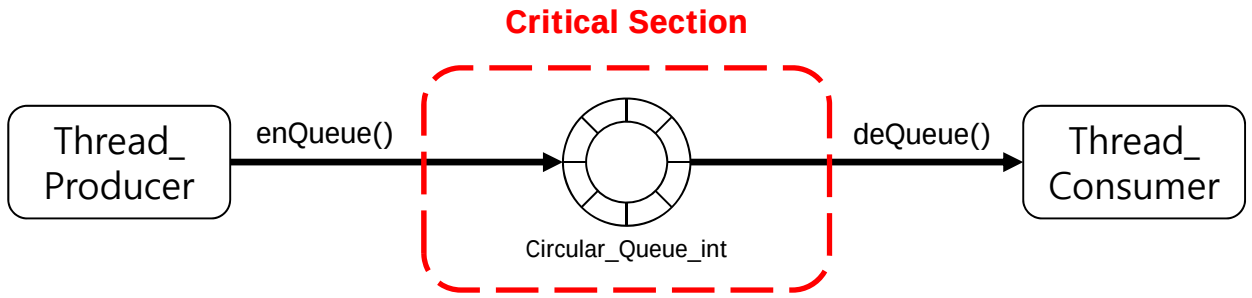
◆ Thread Producer

- generate and data randomly, and enqueue() into the shared queue
- if queue is full, it waits
- sleeps randomly chosen time
- repeats the data generation for given execution duration

◆ Thread Consumer

- dequeues a data from the shared queue
- if queue is empty, it waits
- sleeps randomly chosen time
- repeats the data processing for given execution duration

◆ Functional Block diagram



Queue - Header File

```
/* Queue.h */

#ifndef CIRCULAR_INT_Q_H
#define CIRCULAR_INT_Q_H

typedef struct Circular_Int_Queue
{
    int numData;
    int max_Q_size;
    int first_index;
    int last_index;
    int *data;
    Circular_Int_Queue(int maxQsize);  // 구조체의 초기화 생성
} Circular_Int_Queue;

int enQueue(Circular_Int_Queue* pQ, int data);
int deQueue(Circular_Int_Queue* pQ);
int isQueueEmpty(Circular_Int_Queue* pQ);
int isQueueFull(Circular_Int_Queue* pQ);
void printQueue(Circular_Int_Queue* pQ);
void freeQueueBuffer(Circular_Int_Queue* pQ);
#endif
```

Queue – 멤버 함수

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

Circular_Int_Queue::Circular_Int_Queue(int maxQsize)
{
    max_Q_size = maxQsize;
    data = (int *)malloc(sizeof(int)* max_Q_size);
    first_index = last_index = 0;
    numData = 0;
    for (int i = 0; i < max_Q_size; i++)
    {
        data[i] = -1;
    }
}

void freeQueueBuffer(Circular_Int_Queue* pQ)
{
    free(pQ->data);
}
```

Queue – 멤버 함수

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

int enQueue(Circular_Int_Queue* pQ, int data)
{
    int index = -1;
    if (isQueueFull(pQ))
    {
        printf("Queue is Full!!₩n");
        return -1;
    } else
    {
        index = pQ->last_index;
        pQ->data[index] = data;
        pQ->last_index = (pQ->last_index + 1)
              % pQ->max_Q_size;
        pQ->numData++;
        return 0;
    }
}
```

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

int deQueue(Circular_Int_Queue* pQ)
{
    int data = 0;
    int index = -1;
    if (isQueueEmpty(pQ))
    {
        printf("Queue is Empty!!₩n");
        return -1;
    } else
    {
        index = pQ->first_index;
        data = pQ->data[index];
        pQ->data[index] = -1;
        pQ->first_index = (pQ->first_index + 1)
            % pQ->max_Q_size;
        pQ->numData--;
        return data;
    }
}
```

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

int isQueueEmpty(Circular_Int_Queue* pQ)
{
    if (pQ->numData == 0)
        return 1;
    else
        return 0;
}
int isQueueFull(Circular_Int_Queue* pQ)
{
    if (pQ->numData == pQ->max_Q_size)
        return 1;
    else
        return 0;
}
void printQueue(Circular_Int_Queue* pQ)
{
    int count = 0;
    printf(" Current queue (numData: %2d) [", pQ->numData);
    for (int i = 0; i<pQ->numData; i++)
    {
        printf(" %2d", pQ->data[(pQ->first_index + i) % pQ->max_Q_size]);
        if (i < (pQ->numData - 1))
              printf(", ");
    }
    printf("]₩n");
}
```

main(), Threads

```
/* Multi-thread.c  (1) */

#include<stdio.h>
#include<windows.h>
#include<time.h>
#include"Queue.h"

#define MAX_QUEUE_SIZE 10
#define THREAD_EXECUTION_TIME 10000 // 10 sec
enum ROLE { PRODUCER, CONSUMER };
typedef struct ThreadParam
{
    Circular_Int_Queue *queue;
    CRITICAL_SECTION* pCS;
    int role;
} ThreadParam;

DWORD WINAPI Thread_producer(LPVOID pParam);
DWORD WINAPI Thread_consumer(LPVOID pParam);
```

```
/* Multi-thread.c  (2) */

void main()
{
    Circular_Int_Queue circular_IntQ(MAX_QUEUE_SIZE);  // Queue 생성 및  초기화
    /*임계구역을 나누어 스레드간의 공유자원 사용을 관리하게 될 CriticalSection 변수*/
    CRITICAL_SECTION crit;
    ThreadParam *pThrParam; /*각 스레드로 전달될 파라미터 구조체*/
    /*스레드 정보를 관리하게 될 핸들러 변수*/
    HANDLE hThreadProducer, hThreadconsumer;
    DWORD nExitCode = NULL;

    Circular_Int_Queue *pQ = &circular_IntQ; /*변수 초기화*/
    InitializeCriticalSection(&crit); /*스레드에서 사용될 CriticalSection 변수 초기화*/


    /*Consumer 스레드에 전달 될 파라미터값 초기화*/
    pThrParam = (ThreadParam*)malloc(sizeof(ThreadParam));
    pThrParam->pCS = &crit;
    pThrParam->queue = pQ;
    pThrParam->role = CONSUMER;
    /*CreateThread API를 이용하여 Thread 생성과 전달할 인자를 전달한 후
      반환되어지는 해당 Thread에 대한 정보를 hThreadconsumer 핸들러에 저장*/
    hThreadconsumer = CreateThread(NULL, 0, Thread_consumer, pThrParam, 0, NULL);
    printf("Thread consumer has been created and instantitated ..₩n");
```

```
/* Multi-thread.c  (3) */

    /*Producer 스레드에 전달 될 파라미터값 초기화*/
    pThrParam = (ThreadParam*)malloc(sizeof(ThreadParam));
    pThrParam->pCS = &crit;
    pThrParam->queue = pQ;
    pThrParam->role = PRODUCER;
    /*CreateThread API를 이용하여 Thread 생성과 전달할 인자를 전달한 후
    반환되어지는 해당 Thread에 대한 정보를 hThreadProducer 핸들러에 저장*/
    hThreadProducer = CreateThread(NULL, 0, Thread_producer, pThrParam, 0, NULL);
    printf("Thread producer has been created and instantitated ..₩n");

    /*main 스레드가 생성한 Producer 스레드가 끝날때까지 대기*/
    printf("Waiting for the termination of thread producer...₩n");
    WaitForSingleObject(hThreadProducer, THREAD_EXECUTION_TIME);
    GetExitCodeThread(hThreadProducer, &nExitCode);
    TerminateThread(hThreadProducer, nExitCode); /*Producer 스레드 종료*/
    CloseHandle(hThreadProducer); /*스레드 핸들러 종료*/

    printf("Thread producer is now terminated..₩n");

    /*main 스레드가 생성한 Comsumer 스레드가 끝날때까지 대기*/
    printf("Waiting for the termination of thread consumer...₩n");
    WaitForSingleObject(hThreadconsumer, THREAD_EXECUTION_TIME);
    GetExitCodeThread(hThreadconsumer, &nExitCode);
    TerminateThread(hThreadconsumer, nExitCode); /*Comsumer 스레드 종료*/
    CloseHandle(hThreadconsumer); /*스레드 핸들러 종료*/
    printf("Thread consumer is now terminated..₩n");
```

```
/* Multi-thread.c  (4) */

    /*스레드에서 사용된 CriticalSection 변수 Delete*/
    DeleteCriticalSection(&crit);
    freeQueueBuffer(pQ);
}

DWORD WINAPI Thread_producer(LPVOID pParam)
{
    ThreadParam *pThrParam;
    /*void* 자료형으로 스레드에 전달된 인자를 형변환을 통해 pThrParam 구조체로 변환*/
    pThrParam = (ThreadParam *)pParam;
    Circular_Int_Queue *pQ = pThrParam->queue;
    int data = 0;
    int sleep_time_ms = 0;
     int enQ_res;
    srand(time(NULL));
    while (1)
    {
        data = rand() % 100;
        /*공유자원에 접근하는 임계구역에 진입하기 전
        cs변수를 이용하여, Lock을 잡고 임계구역에 진입
        한번에 하나의 스레드만이 공유자원에 접근하도록 제한함*/
        EnterCriticalSection(pThrParam->pCS);
        printf("Thread_producer::enqueue(data = %2d) => ", data);
```

```
/* Multi-thread.c  (4) */

          enQ_res = enQueue(pQ, data);
          LeaveCriticalSection(pThrParam->pCS);
          if (enQ_res != -1)
          {
               EnterCriticalSection(pThrParam->pCS);
               printQueue(pQ);
               LeaveCriticalSection(pThrParam->pCS);
          }
          else
          {   // 만약 Queue가 FULL 상태이면, 여유 공간이 생길 때 까지 반복하여 enQueue() 시도
               do {
                    Sleep(200);
                    EnterCriticalSection(pThrParam->pCS);
                    enQ_res = enQueue(pQ, data);
                    LeaveCriticalSection(pThrParam->pCS);
               } while (enQ_res == -1);
               EnterCriticalSection(pThrParam->pCS);
               printQueue(pQ);
               LeaveCriticalSection(pThrParam->pCS);
          }

          sleep_time_ms = rand() % 500;
          Sleep(sleep_time_ms);
     }
     return 0;
}
```

```
/* Multi-thread.c  (4) */

DWORD WINAPI Thread_consumer(LPVOID pParam)
{
    ThreadParam *pThrParam;
    /*void* 자료형으로 스레드에 전달된 인자를 형변환을 통해 pThrParam 구조체로 변환*/
    pThrParam = (ThreadParam *)pParam;
    Circular_Int_Queue *pQ = pThrParam->queue;
    int sleep_time_ms = 0;
    int dequeue_data = -1;
    srand(time(NULL));
    while (1)
    {
        /*공유자원에 접근하는 임계구역에 진입하기 전 cs변수를 이용하여, Lock을 잡고 임계구역에 진입
        한번에 하나의 스레드만이 공유자원에 접근하도록 제한함*/
        EnterCriticalSection(pThrParam->pCS);
        printf("Thread_consumer::deQueue() => ");
        dequeue_data = deQueue(pQ);
        if (dequeue_data > -1)
        {
            printf("dequeue data (%2d)", dequeue_data);
            printQueue(pQ);
        }
        /*공유자원에 대한 처리를 종료한 후 critical section Lock을 놓아주어
           해당 임계구역에대한 권리를 다른 스레드가 가질 수 있도록 허락함*/
        LeaveCriticalSection(pThrParam->pCS);
        sleep_time_ms = rand() % 1000;
        Sleep(sleep_time_ms);
    }
    return 0;
}
```

실행 결과

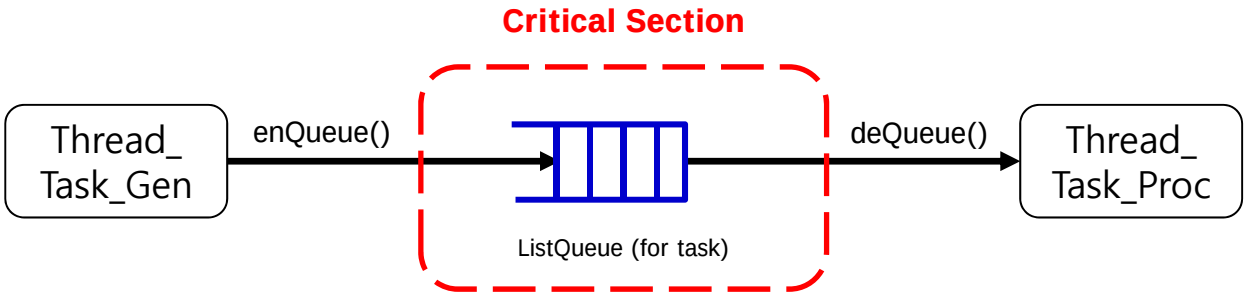
```
Thread consumer has been created and instantitated ..
Thread_consumer::deQueue() => Queue is Empty!!
Thread producer has been created and instantitated ..
Waiting for the termination of thread producer...
Thread_producer::enqueue(data = 81) =>  Current queue (num_data:  1) [ 81]
Thread_producer::enqueue(data = 31) =>  Current queue (num_data:  2) [ 81,  31]
Thread_producer::enqueue(data = 79) =>  Current queue (num_data:  3) [ 81,  31,  79]
Thread_consumer::deQueue() => dequeue data (81) Current queue (num_data:  2) [ 31,  79]
Thread_producer::enqueue(data = 54) =>  Current queue (num_data:  3) [ 31,  79,  54]
Thread_consumer::deQueue() => dequeue data (31) Current queue (num_data:  2) [ 79,  54]
Thread_producer::enqueue(data =  1) =>  Current queue (num_data:  3) [ 79,  54,   1]
Thread_producer::enqueue(data = 24) =>  Current queue (num_data:  4) [ 79,  54,   1,  24]
Thread_consumer::deQueue() => dequeue data (79) Current queue (num_data:  3) [ 54,   1,  24]
Thread_producer::enqueue(data = 41) =>  Current queue (num_data:  4) [ 54,   1,  24,  41]
Thread_producer::enqueue(data = 50) =>  Current queue (num_data:  5) [ 54,   1,  24,  41,  50]
Thread_producer::enqueue(data = 15) =>  Current queue (num_data:  6) [ 54,   1,  24,  41,  50,  15]
Thread_producer::enqueue(data = 28) =>  Current queue (num_data:  7) [ 54,   1,  24,  41,  50,  15,  28]
Thread_consumer::deQueue() => dequeue data (54) Current queue (num_data:  6) [  1,  24,  41,  50,  15,  28]
Thread_producer::enqueue(data = 49) =>  Current queue (num_data:  7) [  1,  24,  41,  50,  15,  28,  49]
Thread_producer::enqueue(data = 67) =>  Current queue (num_data:  8) [  1,  24,  41,  50,  15,  28,  49,  67]
Thread_producer::enqueue(data = 77) =>  Current queue (num_data:  9) [  1,  24,  41,  50,  15,  28,  49,  67,  77]
Thread_producer::enqueue(data = 45) =>  Current queue (num_data: 10) [  1,  24,  41,  50,  15,  28,  49,  67,  77,  45]
Thread_consumer::deQueue() => dequeue data ( 1) Current queue (num_data:  9) [ 24,  41,  50,  15,  28,  49,  67,  77,  45]
Thread_producer::enqueue(data = 36) =>  Current queue (num_data: 10) [ 24,  41,  50,  15,  28,  49,  67,  77,  45,  36]
Thread_producer::enqueue(data = 94) => Queue is Full!!!
Queue is Full!!!
Thread_consumer::deQueue() => dequeue data (24) Current queue (num_data:  9) [ 41,  50,  15,  28,  49,  67,  77,  45,  36]
 Current queue (num_data: 10) [ 41,  50,  15,  28,  49,  67,  77,  45,  36,  94]
Thread_producer::enqueue(data = 98) => Queue is Full!!!
Queue is Full!!!
Queue is Full!!!
Queue is Full!!!
Queue is Full!!!
Thread_consumer::deQueue() => dequeue data (41) Current queue (num_data:  9) [ 50,  15,  28,  49,  67,  77,  45,  36,  94]
 Current queue (num_data: 10) [ 50,  15,  28,  49,  67,  77,  45,  36,  94,  98]
Thread_consumer::deQueue() => dequeue data (50) Current queue (num_data:  9) [ 15,  28,  49,  67,  77,  45,  36,  94,  98]
Thread_producer::enqueue(data = 57) =>  Current queue (num_data: 10) [ 15,  28,  49,  67,  77,  45,  36,  94,  98,  57]
Thread_consumer::deQueue() => dequeue data (15) Current queue (num_data:  9) [ 28,  49,  67,  77,  45,  36,  94,  98,  57]
Thread_producer::enqueue(data = 96) =>  Current queue (num_data: 10) [ 28,  49,  67,  77,  45,  36,  94,  98,  57,  96]
Thread_consumer::deQueue() => dequeue data (28) Current queue (num_data:  9) [ 49,  67,  77,  45,  36,  94,  98,  57,  96]
Thread_producer::enqueue(data = 39) =>  Current queue (num_data: 10) [ 49,  67,  77,  45,  36,  94,  98,  57,  96,  39]
Thread_producer::enqueue(data = 49) => Queue is Full!!!
Queue is Full!!!
```

```
Thread_consumer::deQueue() => dequeue data (49) Current queue (num_data:  9) [ 67,  77,  45,  36,  94,  98,  57,  96,  39]
 Current queue (num_data: 10) [ 67,  77,  45,  36,  94,  98,  57,  96,  39,  49]
Thread_consumer::deQueue() => dequeue data (67) Current queue (num_data:  9) [ 77,  45,  36,  94,  98,  57,  96,  39,  49]
Thread_producer::enqueue(data = 61) =>  Current queue (num_data: 10) [ 77,  45,  36,  94,  98,  57,  96,  39,  49,  61]
Thread_producer::enqueue(data = 78) => Queue is Full!!
Queue is Full!!
Queue is Full!!
Queue is Full!!
Thread_consumer::deQueue() => dequeue data (77) Current queue (num_data:  9) [ 45,  36,  94,  98,  57,  96,  39,  49,  61]
 Current queue (num_data: 10) [ 45,  36,  94,  98,  57,  96,  39,  49,  61,  78]
Thread_producer::enqueue(data = 16) => Queue is Full!!
Thread_consumer::deQueue() => dequeue data (45) Current queue (num_data:  9) [ 36,  94,  98,  57,  96,  39,  49,  61,  78]
 Current queue (num_data: 10) [ 36,  94,  98,  57,  96,  39,  49,  61,  78,  16]
Thread_producer::enqueue(data = 44) => Queue is Full!!
Queue is Full!!
Queue is Full!!
Thread_consumer::deQueue() => dequeue data (36) Current queue (num_data:  9) [ 94,  98,  57,  96,  39,  49,  61,  78,  16]
Thread_consumer::deQueue() => dequeue data (94) Current queue (num_data:  8) [ 98,  57,  96,  39,  49,  61,  78,  16]
 Current queue (num_data:  9) [ 98,  57,  96,  39,  49,  61,  78,  16,  44]
Thread_producer::enqueue(data = 20) =>  Current queue (num_data: 10) [ 98,  57,  96,  39,  49,  61,  78,  16,  44,  20]
Thread_consumer::deQueue() => dequeue data (98) Current queue (num_data:  9) [ 57,  96,  39,  49,  61,  78,  16,  44,  20]
Thread_producer::enqueue(data = 74) =>  Current queue (num_data: 10) [ 57,  96,  39,  49,  61,  78,  16,  44,  20,  74]
Thread_producer::enqueue(data =  6) => Queue is Full!!
Queue is Full!!
Thread_consumer::deQueue() => dequeue data (57) Current queue (num_data:  9) [ 96,  39,  49,  61,  78,  16,  44,  20,  74]
Thread_consumer::deQueue() => dequeue data (96) Current queue (num_data:  8) [ 39,  49,  61,  78,  16,  44,  20,  74]
 Current queue (num_data:  9) [ 39,  49,  61,  78,  16,  44,  20,  74,   6]
Thread producer is now terminated..
Waiting for the termination of thread consumer...
Thread_consumer::deQueue() => dequeue data (39) Current queue (num_data:  8) [ 49,  61,  78,  16,  44,  20,  74,   6]
Thread_consumer::deQueue() => dequeue data (49) Current queue (num_data:  7) [ 61,  78,  16,  44,  20,  74,   6]
Thread_consumer::deQueue() => dequeue data (61) Current queue (num_data:  6) [ 78,  16,  44,  20,  74,   6]
Thread_consumer::deQueue() => dequeue data (78) Current queue (num_data:  5) [ 16,  44,  20,  74,   6]
Thread_consumer::deQueue() => dequeue data (16) Current queue (num_data:  4) [ 44,  20,  74,   6]
Thread_consumer::deQueue() => dequeue data (44) Current queue (num_data:  3) [ 20,  74,   6]
Thread_consumer::deQueue() => dequeue data (20) Current queue (num_data:  2) [ 74,   6]
Thread_consumer::deQueue() => dequeue data (74) Current queue (num_data:  1) [  6]
Thread_consumer::deQueue() => dequeue data ( 6) Current queue (num_data:  0) []
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread_consumer::deQueue() => Queue is Empty!!
Thread consumer is now terminated..
계속하려면 아무 키나 누르십시오 . . .
```

Task Generator – Task Processor

◆ Functional Block diagram

- 2개의 threads: Task_Gen, Task_Proc
- 1개의 Linked List 기반 Queue



◆ Thread Task_Gen

- Thread_Task_Gen()는 차례대로 0 ~ 19값의 task_id와 임의로 생성되는 task_name를 가지는 Task 구조체 변수를 생성하며, 이 생성된 task를 공유되는 queue에 저장(enqueue) 시킨다.
- 항상 isFull() 함수를 사용하여, queue가 FULL 상태인 가를 먼저 확인하고, 만약 queue가 FULL 상태이면, 0.1초를 sleep한 후 다시 queue 상태를 점검한 후, 여유 공간이 생기면 저장한다.
- Queue에 데이터를 저장한 후에는, printQueue() 함수를 사용하여 현재의 queue 상태를 출력시키며, 0.1 ~ 1초 사이의 값을 임의로 선정하여 그 기간 동안 sleep한 후, 데이터 생성 및 enqueue 동작을 반복한다.

◆ Thread Task_Proc

- Thread_Task_Proc()는 isEmpty() 함수를 사용하여 queue의 상태를 확인하여, 만약 EMPTY 상태가 아니면 queue에 저장되어 있는 데이터를 추출 (deQueue)하여 Task 정보 (task_id, task_name)을 출력하고,
- printQueue() 함수를 사용하여 queue 상태를 출력한 후, 0.1 ~ 1초 사이의 값을 임의로 선정한 후, 그 기간 동안 sleep하고, queue로 부터의 데이터 추출을 반복한다.
- 만약 queue가 EMPTY 상태이면, 0.1 초 동안 sleep한 후, queue를 상태를 다시 점검하고, EMPTY 상태가 아니면 데이터를 추출한다.

◆ main() 함수

- main() 함수는 MAX_CAPACITY 10인 queue를 구조체 변수의 초기화 기능을 사용하여 설정하며, 두 개의 스레드 (Thread_Task_Proc(), Thread_Task_Gen())를 생성하고,
- 이들 스레드들이 queue와 CRITICAL_SECTION criticalSection를 공유하도록 하며, 각각의 role을 TASK_PROC와 TASK_GEN으로 지정한다.
- 생성된 Thread_Task_Gen() 는 지정된 개수 (예: 20)의Task를 생성하여, enqueue 시킨 후, 종료를 한다.
- Thread_Task_Proc()는 지정된 개수의 task를 차례로 dequeue한 후, 이를 처리 (화면에 출력) 한 후, 스스로 종료한다.
- main()함수에서는 WaitForSingleObject()함수를 사용하여, Thread_Task_Gen() 이 스스로 종료할 때 까지 기다린다.
- Thread_Task_Gen() 이 종료되면, main()함수에서는 WaitForSingleObject()함수를 사용하여, Thread_Task_Proc() 이 스스로 종료할 때 까지 기다린다.

Task_Gen, Task_Proc, ListQueue의 구현

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

#ifndef LIST_QUEUE_H
#define LIST_QUEUE_H

/* Queue based on Linked List */

#include <stdio.h>
#include <stdlib.h>
typedef struct Task
{
    int task_id;
    char task_name[16];
} Task;

typedef struct ListNode
{
    Task *pTask;
    ListNode *prev;
    ListNode *next;

} ListNode;
```

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

typedef struct ListQueue
{
    ListNode *pFront;
    ListNode *pEnd;
    int size;
    int max_capacity;
    ListQueue(int max_cap);
} ListQueue;

bool isEmpty(ListQueue *pQ);
bool isFull(ListQueue *pQ);
void initQueue(ListQueue *pQ);
int enQueue(ListQueue *pQ, Task *pTsk);
Task *deQueue(ListQueue *pQ);
void printQueue(ListQueue *pQ);

#endif
```

```
/* ListQueue.c (1) */

#include "ListQueue.h"

ListQueue::ListQueue(int max_cap)
{
    max_capacity = max_cap;
    size = 0;
    pFront = pEnd = NULL;
}

bool isEmpty(ListQueue *pQ)
{
    if (pQ->size == 0)
        return true;
    else
        return false;
}

bool isFull(ListQueue *pQ)
{
    if (pQ->size >= pQ->max_capacity)
        return true;
    else
        return false;
}
```

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

void printQueue(ListQueue *pQ)
{
    Task *pTsk;
    if (isEmpty(pQ))
    {
        printf(" Exception::Queue is Empty !!₩n");
        return;
    }
    printf(" Queue status (size:%2d) : ", pQ->size);
    ListNode *pLN = pQ->pFront;
    for (int i = 0; i < pQ->size; i++)
    {
        pTsk = pLN->pTask;
        printf(" task (%2d, %8s)",
            pTsk->task_id, pTsk->task_name);
        pLN = pLN->next;
    }
    printf("₩n");
}
```

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

int enQueue(ListQueue *pQ, Task *pTsk)
{
    ListNode *pNewLN;

    pNewLN = (ListNode *)
      malloc(sizeof(ListNode));
    pNewLN->pTask = pTsk;
    pNewLN->next = NULL;

    if (isFull(pQ))
        return -1; // queue is full
    else if (isEmpty(pQ))
    {   // queue is empty
        pNewLN->prev = NULL;
        pQ->pFront = pQ->pEnd = pNewLN;
    }
    else {
        pNewLN->prev = pQ->pEnd;
        pQ->pEnd->next = pNewLN;
        pQ->pEnd = pNewLN;
    }
    pQ->size++;
    return pQ->size;
}
```

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

Task *deQueue(ListQueue *pQ)
{
    Task *pTsk;
    if (isEmpty(pQ))
    {
        printf(" Exception::Queue is Empty !!₩n");
        return NULL;
    }
    pTsk = pQ->pFront->pTask;
    pQ->size--;

    ListNode *pLN = pQ->pFront;
    pQ->pFront = pQ->pFront->next;
    free(pLN);
    return pTsk;
}
```

```
/* main.c (1) */
#include<stdio.h>
#include<windows.h>
#include<time.h>
#include"ListQueue.h"
#define MAX_CAPACITY 4
#define TASK_NAME_LEN 8
#define TASK_NAME_LEN_MIN 4
#define NUM_TASK_GEN 20
#define THRD_TASK_GEN_INTERVAL 500 // 500ms
#define THRD_TASK_PROC_INTERVAL 1000 // 1000ms
#define THREAD_EXECUTION_TIME 5000 // 5 sec
enum ROLE { TASK_GEN, TASK_PROC };
typedef struct ThreadParam
{
    ListQueue *pQ;
    CRITICAL_SECTION* pCS;
    int role;
} ThreadParam;

DWORD WINAPI Thread_TaskGen(LPVOID pParam);
DWORD WINAPI Thread_TaskProc(LPVOID pParam);
```

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

void main()
{
    /*    변수 선언     */
    ListQueue queue(MAX_CAPACITY);
    Elm_t data;
    ListQueue *pQ = &queue;

    /*임계구역을 나누어 스레드간의 공유자원 사용을 관리하게 될 CriticalSection 변수*/
    CRITICAL_SECTION crit;
    /*각 스레드로 전달될 파라미터 구조체*/
    ThreadParam *pThrParam;
    /*스레드 정보를 관리하게 될 핸들러 변수*/
    HANDLE hThreadTaskGen, hThreadTaskProc;
    DWORD nExitCode = NULL;

    /*스레드에서 사용될 CriticalSection 변수 초기화*/
    InitializeCriticalSection(&crit);

    /*Task_Processor 스레드에 전달 될 파라미터값 초기화*/
    pThrParam = (ThreadParam*)malloc(sizeof(ThreadParam));
    pThrParam->pCS = &crit;
    pThrParam->pQ = pQ;
    pThrParam->role = TASK_PROC;
    /*CreateThread API를 이용하여 Thread 생성과 전달할 인자를 전달한 후
    반환되어지는 해당 Thread에 대한 정보를 hThreadTask_Processor 핸들러에 저장*/
    hThreadTaskProc = CreateThread(NULL, 0, Thread_TaskProc, pThrParam, 0, NULL);
    printf("Thread Task_Proc has been created and instantitated ..₩n");
```

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

    /*Task_Generator 스레드에 전달 될 파라미터값 초기화*/
    pThrParam = (ThreadParam*)malloc(sizeof(ThreadParam));
    pThrParam->pCS = &crit;
    pThrParam->pQ = pQ;
    pThrParam->role = TASK_GEN;
    /*CreateThread API를 이용하여 Thread 생성과 전달할 인자를 전달한 후
    반환되어지는 해당 Thread에 대한 정보를 hThreadTask_Generator 핸들러에 저장*/
    hThreadTaskGen = CreateThread(NULL, 0, Thread_TaskGen, pThrParam, 0, NULL);
    printf("Thread Task_Gen has been created and instantitated ..₩n");

    /*main 스레드가 생성한 Task_Generator 스레드가 끝날때까지 대기*/
    printf("Waiting for the termination of thread Task_Gen...₩n");
    WaitForSingleObject(hThreadTaskGen, INFINITE);
    GetExitCodeThread(hThreadTaskGen, &nExitCode);
    TerminateThread(hThreadTaskGen, nExitCode); /*Task_Generator 스레드 종료*/
    CloseHandle(hThreadTaskGen); /*스레드 핸들러 종료*/
    printf("Thread Task_Gen is now terminated..₩n");

    /*main 스레드가 생성한 Task_Gen 스레드가 끝날때까지 대기*/
    printf("Waiting for the termination of thread Task_Proc ...₩n");
    WaitForSingleObject(hThreadTaskProc, INFINITE);
    GetExitCodeThread(hThreadTaskProc, &nExitCode);
    TerminateThread(hThreadTaskProc, nExitCode); /* Task_Proc 스레드 종료*/
    CloseHandle(hThreadTaskProc); /*스레드 핸들러 종료*/
    printf("Thread Task_Proc is now terminated..₩n");

    DeleteCriticalSection(&crit); /*스레드에서 사용된 CriticalSection 변수 Delete*/
}
```

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

DWORD WINAPI Thread_TaskGen(LPVOID pParam)
{
    ThreadParam *pThrParam;
    Task *pTask;
    char t_name[TASK_NAME_LEN];
    int t_name_len;
    int i, j;
    /* void * 자료형으로 스레드에 전달된 인자를 형변환을 통해 pThrParam 구조체로 변환*/
    pThrParam = (ThreadParam *)pParam;
    ListQueue *pQ = pThrParam->pQ;
    int data = 0;
    int sleep_time_ms = 0;
    int enQ_res;
    srand(time(NULL));
    for (i = 0; i < NUM_TASK_GEN; i++)
    {
        pTask = (Task *)malloc(sizeof(Task));
        pTask->task_id = i;
        for (int j = 0; j < TASK_NAME_LEN; j++)
            t_name[j] = '₩0';
        t_name_len = rand() % (TASK_NAME_LEN - TASK_NAME_LEN_MIN) + TASK_NAME_LEN_MIN;
        t_name[0] = rand() % 26 + 'A';
        for (j = 1; j < t_name_len; j++)
            t_name[j] = rand() % 26 + 'a';
        t_name[j] = '₩0';
        strcpy(pTask->task_name, t_name);
```

```
/* main.c (5) */

        EnterCriticalSection(pThrParam->pCS);
        printf("TaskGen::enqueue(%2d, %8s) => ", pTask->task_id, pTask->task_name);
        enQ_res = enQueue(pQ, pTask);
        LeaveCriticalSection(pThrParam->pCS);
        if (enQ_res != -1)
        {
            EnterCriticalSection(pThrParam->pCS);
            printQueue(pQ);
            LeaveCriticalSection(pThrParam->pCS);
        } else  // queue is full
        {
            EnterCriticalSection(pThrParam->pCS);
            printf("Queue is Full !!₩n");
            LeaveCriticalSection(pThrParam->pCS);
            do {
                Sleep(200);
                EnterCriticalSection(pThrParam->pCS);
                enQ_res = enQueue(pQ, pTask);
                LeaveCriticalSection(pThrParam->pCS);
            } while (enQ_res == -1);
            EnterCriticalSection(pThrParam->pCS);
            printf("   enQueue() after queue has been Full =>");
            printQueue(pQ);
            LeaveCriticalSection(pThrParam->pCS);
        }
         sleep_time_ms = rand() % THRD_TASK_GEN_INTERVAL;
        Sleep(sleep_time_ms);
    } // for
    printf("TaskGen:: Total %d tasks have been generated !!₩n", NUM_TASK_GEN);
    return 0;
}
```

```
/* main.c (6) */

DWORD WINAPI Thread_TaskProc(LPVOID pParam)
{
    ThreadParam *pThrParam;
    /*void* 자료형으로 스레드에 전달된 인자를 형변환을 통해 pThrParam 구조체로 변환*/
    pThrParam = (ThreadParam *)pParam;
    ListQueue *pQ = pThrParam->pQ;
    int sleep_time_ms = 0;
    int dequeue_data = -1;
    int count = 0;
    Task *pTsk;
    srand(time(NULL));
    while (1)
    {
        /*공유자원에 접근하는 임계구역에 진입하기 전, cs변수를 이용하여, Lock을 잡고 임계구역에 진입
        한번에 하나의 스레드만이 공유자원에 접근하도록 제한함*/
        EnterCriticalSection(pThrParam->pCS);
        printf("TaskProc::deQueue() => ");
        pTsk = deQueue(pQ);
        if (pTsk != NULL)
        {
            printf("task(%2d, %8s)", pTsk->task_id, pTsk->task_name);
            printQueue(pQ);
            count++;
        }
```

```
/* main.c (7) */

        /*공유자원에 대한 처리를 종료한 후 Lock을 놓아주어
         해당 임계구역에대한 권리를 다른 스레드가 가질 수 있도록 허락함*/
         LeaveCriticalSection(pThrParam->pCS);
         if (count >= NUM_TASK_GEN)
         {
             printf("TaskProc:: Total %d tasks have been processed !!₩n", count);
             break;
         }

         sleep_time_ms = rand() % THRD_TASK_PROC_INTERVAL;
         Sleep(sleep_time_ms);
    }
    return 0;
}
```

```
Thread Task_Proc has been created and instantitated ..
TaskProc::deQueue() =>  Exception::Queue is Empty !!
Thread Task_Gen has been created and instantitated ..
Waiting for the termination of thread Task_Gen...
TaskGen::enqueue( 0,  Bwowdbl) =>  Queue status (size: 1) :  task ( 0,  Bwowdbl)
TaskProc::deQueue() => task( 0,  Bwowdbl) Exception::Queue is Empty !!
TaskGen::enqueue( 1,   Arbchc) =>  Queue status (size: 1) :  task ( 1,   Arbchc)
TaskProc::deQueue() => task( 1,   Arbchc) Exception::Queue is Empty !!
TaskGen::enqueue( 2,   Hugfxb) =>  Queue status (size: 1) :  task ( 2,   Hugfxb)
TaskGen::enqueue( 3,  Xbeutvq) =>  Queue status (size: 2) :  task ( 2,   Hugfxb) task ( 3,  Xbeutvq)
TaskGen::enqueue( 4,   Hhixt) =>  Queue status (size: 3) :  task ( 2,   Hugfxb) task ( 3,  Xbeutvq) task ( 4,   Hhixt)
TaskGen::enqueue( 5,   Uqink) =>  Queue status (size: 4) :  task ( 2,   Hugfxb) task ( 3,  Xbeutvq) task ( 4,   Hhixt) task ( 5,   Uqink)
TaskProc::deQueue() => task( 2,   Hugfxb) Queue status (size: 3) :  task ( 3,  Xbeutvq) task ( 4,   Hhixt) task ( 5,   Uqink)
TaskGen::enqueue( 6,   Auetb) =>  Queue status (size: 4) :  task ( 3,  Xbeutvq) task ( 4,   Hhixt) task ( 5,   Uqink) task ( 6,   Auetb)
TaskGen::enqueue( 7,   Blesw) => Queue is Full !!
TaskProc::deQueue() => task( 3,  Xbeutvq) Queue status (size: 3) :  task ( 4,   Hhixt) task ( 5,   Uqink) task ( 6,   Auetb)
   enQueue() after queue has been Full => Queue status (size: 4) :  task ( 4,   Hhixt) task ( 5,   Uqink) task ( 6,   Auetb) task ( 7,   Blesw)
TaskGen::enqueue( 8,    Gkec) => Queue is Full !!
TaskProc::deQueue() => task( 4,   Hhixt) Queue status (size: 3) :  task ( 5,   Uqink) task ( 6,   Auetb) task ( 7,   Blesw)
   enQueue() after queue has been Full => Queue status (size: 4) :  task ( 5,   Uqink) task ( 6,   Auetb) task ( 7,   Blesw) task ( 8,    Gkec)
TaskGen::enqueue( 9,  Ecyati) => Queue is Full !!
TaskProc::deQueue() => task( 5,   Uqink) Queue status (size: 3) :  task ( 6,   Auetb) task ( 7,   Blesw) task ( 8,    Gkec)
   enQueue() after queue has been Full => Queue status (size: 4) :  task ( 6,   Auetb) task ( 7,   Blesw) task ( 8,    Gkec) task ( 9,  Ecyati)
TaskGen::enqueue(10,  Veoygph) => Queue is Full !!
TaskProc::deQueue() => task( 6,   Auetb) Queue status (size: 3) :  task ( 7,   Blesw) task ( 8,    Gkec) task ( 9,  Ecyati)
   enQueue() after queue has been Full => Queue status (size: 4) :  task ( 7,   Blesw) task ( 8,    Gkec) task ( 9,  Ecyati) task (10,  Veoygph)
TaskGen::enqueue(11,   Scdiio) => Queue is Full !!
TaskProc::deQueue() => task( 7,   Blesw) Queue status (size: 3) :  task ( 8,    Gkec) task ( 9,  Ecyati) task (10,  Veoygph)
   enQueue() after queue has been Full => Queue status (size: 4) :  task ( 8,    Gkec) task ( 9,  Ecyati) task (10,  Veoygph) task (11,   Scdiio)
TaskGen::enqueue(12,   Zmnee) => Queue is Full !!
TaskProc::deQueue() => task( 8,    Gkec) Queue status (size: 3) :  task ( 9,  Ecyati) task (10,  Veoygph) task (11,   Scdiio)
   enQueue() after queue has been Full => Queue status (size: 4) :  task ( 9,  Ecyati) task (10,  Veoygph) task (11,   Scdiio) task (12,   Zmnee)
TaskGen::enqueue(13,   Arbyw) => Queue is Full !!
TaskProc::deQueue() => task( 9,  Ecyati) Queue status (size: 3) :  task (10,  Veoygph) task (11,   Scdiio) task (12,   Zmnee)
   enQueue() after queue has been Full => Queue status (size: 4) :  task (10,  Veoygph) task (11,   Scdiio) task (12,   Zmnee) task (13,   Arbyw)
TaskProc::deQueue() => task(10,  Veoygph) Queue status (size: 3) :  task (11,   Scdiio) task (12,   Zmnee) task (13,   Arbyw)
TaskGen::enqueue(14,  Fqnxyeo) =>  Queue status (size: 4) :  task (11,   Scdiio) task (12,   Zmnee) task (13,   Arbyw) task (14,  Fqnxyeo)
TaskProc::deQueue() => task(11,   Scdiio) Queue status (size: 3) :  task (12,   Zmnee) task (13,   Arbyw) task (14,  Fqnxyeo)
TaskGen::enqueue(15,  Szcdzex) =>  Queue status (size: 4) :  task (12,   Zmnee) task (13,   Arbyw) task (14,  Fqnxyeo) task (15,  Szcdzex)
TaskProc::deQueue() => task(12,   Zmnee) Queue status (size: 3) :  task (13,   Arbyw) task (14,  Fqnxyeo) task (15,  Szcdzex)
TaskGen::enqueue(16,   Pnzfio) =>  Queue status (size: 4) :  task (13,   Arbyw) task (14,  Fqnxyeo) task (15,  Szcdzex) task (16,   Pnzfio)
TaskGen::enqueue(17,    Qwmh) => Queue is Full !!
TaskProc::deQueue() => task(13,   Arbyw) Queue status (size: 3) :  task (14,  Fqnxyeo) task (15,  Szcdzex) task (16,   Pnzfio)
   enQueue() after queue has been Full => Queue status (size: 4) :  task (14,  Fqnxyeo) task (15,  Szcdzex) task (16,   Pnzfio) task (17,    Qwmh)
TaskGen::enqueue(18,   Wyful) => Queue is Full !!
TaskProc::deQueue() => task(14,  Fqnxyeo) Queue status (size: 3) :  task (15,  Szcdzex) task (16,   Pnzfio) task (17,    Qwmh)
   enQueue() after queue has been Full => Queue status (size: 4) :  task (15,  Szcdzex) task (16,   Pnzfio) task (17,    Qwmh) task (18,   Wyful)
TaskProc::deQueue() => task(15,  Szcdzex) Queue status (size: 3) :  task (16,   Pnzfio) task (17,    Qwmh) task (18,   Wyful)
TaskGen::enqueue(19,   Jvtnt) =>  Queue status (size: 4) :  task (16,   Pnzfio) task (17,    Qwmh) task (18,   Wyful) task (19,   Jvtnt)
TaskGen:: Total 20 tasks have been generated !!
Thread Task_Gen is now terminated..
Waiting for the termination of thread Task_Proc ...
TaskProc::deQueue() => task(16,   Pnzfio) Queue status (size: 3) :  task (17,    Qwmh) task (18,   Wyful) task (19,   Jvtnt)
TaskProc::deQueue() => task(17,    Qwmh) Queue status (size: 2) :  task (18,   Wyful) task (19,   Jvtnt)
TaskProc::deQueue() => task(18,   Wyful) Queue status (size: 1) :  task (19,   Jvtnt)
TaskProc::deQueue() => task(19,   Jvtnt) Exception::Queue is Empty !!
TaskProc:: Total 20 tasks have been processed !!
Thread Task_Proc is now terminated..
계속하려면 아무 키나 누르십시오 . . .
```

Homework 11

11.1 Critical section이 필요한 이유에 대하여 설명하라.

11.2 Queuing System and Multi-Threads

(1) 다음과 같은 typedef 을 정의하라:

```
typedef unsigned int UINT_32;
typedef unsigned short UINT_16;
typedef unsigned char UINT_8;
```

(2) 구조체 struct Packet은 다음과 같은 멤버를 가진다:

```
UINT_32 srcAddr; // source address
UINT_32 dstAddr; // destination address
UINT_8 priority; // priority of protocol data unit (사용자/프로토콜 정보의 우선순위)
UINT_32 seqNo; // sequence number
UINT_32 payloadLength; // length of payload
UINT_8 *pPayload; // payload
```

(3) 다음 함수들이 구조체 struct Packet를 위하여 사용된다:

```
void initPacket(Packet *pPkt, unsigned int sAddr, unsigned int sN);
FILE * fprintPacket(FILE *fout, Packet* pPkt);
```

11.2 Queuing System and Multi-Threads (2)

(4) 구조체 struct ListNode 는 다음과 같은 데이터 멤버를 가진다:

```
Packet *pPkt;
ListNode *pNext;
ListNode *pPrev;
```

(5) 구조체 struct Queue 는 다음과 같은 데이터 멤버를 가진다:

```
int numPkts;
ListNode* pFirst;
ListNode* pLast;
```

(6) 다음 함수들은 구조체 struct Queue 와 관련되어 사용된다:

```
int enQueue(Queue *pQ, Packet* pPkt); // enQueue a packet into the queue.
   // The list node is pointing a packet.
Packet* deQueue(Queue *pQ); // remove a list node from the queue, and return a Packet.
void printQueue(FILE *fout, Queue *pQ); // print all list node in the queue
```

11.2 Queuing System and Multi-Threads (3)

(7) 다중 스레드의 초기화를 위하여 다음과 같은 구조체 (ThreadParam)가 스레드로 파라메터를 전달하기 위하여 사용된다:

```
typedef struct ThreadParam
{
        CRITICAL_SECTION *pCS; // pointer to the shared critical section
        Queue *pQ; // pointer to the shared queue
        int role; // packetGen or linkTx
        UINT_32 addr;
        int max_queue;
        int duration; // duration of the thread execution (1 minute).
        FILE *fout; // pointer to the output file stream
}ThreadParam;
```

(8) Thread packetGen() 는 주기적으로 패킷들을 생성하며, 생성된 패킷들을 queue에 삽입 (enqueue) 한다. 각 스레드 packetGen()은 지정된 송신주소 (srcAddr) 를 사용한다. 패킷 생성 시간 간격은 1 ~ 5 초 사이의 값이 임의로 설정된다.

(9) Thread linkTx()는 queue를 지속적으로 점검하며, 만약 queue가 empty 상태이면 1 초를 sleep 한다. Queue가 empty가 아니면, queue로 부터 패킷 1개를 추출 (dequeue) 하여 그 패킷의 정보를 지정된 파일로 출력 한다.

11.2 Queuing System and Multi-Threads (4)

(10) main() 함수는 다음과 같은 내용을 실행한다:

3의 critical section이 생성되어 다수의 packetGen() 스레드와 linkTx() 스레드간의 공유 자원 (3개의 queue)에 대한 사용을 제어한다.

5개의 패킷 생성 스레드를 생성하고, 초기화 한다.

1개의 link transmission 스레드를 생성하고, 초기화 한다.

main() 함수는 3의 구조체 struct Queue 변수를 생성하고, 3의 critical section을 생성하여, 5개의 packet generation 스레드와 1개의 link transmission 스레드가 공유하게 한다.

각 패킷 생성 스레드는 각각 30개의 패킷 (목적지 주소는 random 하게 설정하며, 우선 순위 (priority)는 0 ~ 2의 값을 random하게 설정)을 생성하여, 우선순위에 따라 지정된 queue에 넣는다.

linkTx() 스레드는 항상 우선 순위가 높은 queue를 먼저 점검하며, 만약 패킷이 존재하는 경우, 이를 우선 처리한다.

패킷 생성 스레드에서 생성되어 enqueue된 패킷들은 linkTx() 스레드에 의하여 dequeue된 후, 송신측 주소에 따라 분류되어 개수가 점검된다. 각 송신 주소로 부터의 우선 순위별 총 패킷 개수는 프로그램의 종료단계에서 출력되어, 정확하게 전달되었는지를 확인한다.

프로그램의 수행 내용은 출력파일 “output.txt”에 출력된다.