공학 연구논문집 제28권 제2호 pp. 67~80, 울산대학교.
Journal of Engineering Research Vol. 28, No. 2, pp. 67~80, 1997
University of Ulsan.

Win32 환경 하에서 병행성을 지원하는 재사용 가능한 C++ 클래스 설계 및 구현

추경환 · 김규년
컴퓨터 · 정보통신공학부

<요 약>

실시간 시스템(Real-Time System)은 요구되는 기능들을 주어진 시간 제약아래 모두 정확하게 수행하는 시스템을 말한다. 병행성(concurrency)은 실시간 시스템을 구현하는데 있어서 가장 중요한 개념중의 하나이다. 이러한 병행성을 구현하기 위한 많은 방법론들이 개발되었고, Modula-2, Ada, CHILL등의 프로그래밍 언어는 병행성의 구현을 언어차원에서 지원하도록 개발되었다. 반면에 많은 수의 사용자를 가진 C++는 순차적 프로그래밍 언어로서 실시간 시스템의 개발에 많이 사용되어지는 프로그래밍 언어이다. C++는 병행성을 지원하지 않는 객체지향 프로그래밍 언어이다. 순차적 언어를 이용하여 실시간 시스템을 개발할 때에는 실시간 운영체제의 시스템 서비스를 이용하여 구현한다. 따라서 C++를 이용하여 실시간 시스템을 개발하기 위해서는 병행성을 지원하기 위한 실시간 운영체제의 시스템 서비스를 C++의 객체지향적인 특징들과 어떻게 잘 조화시키느냐가 중요한 문제이다.

본 논문에서는 운영체제의 병행성 지원 시스템 서비스를 C++의 클래스로 캡슐화하여 C++를 이용하여 실시간 시스템을 개발하는데 있어서 쉽게 병행성을 구현하기 위한 기반을 제공하고자 한다.

Design and implementation of reuseable C++ class that support concurrency in Win32 environment

Gyung-Hwan Chu · Kyoo-Nyun Kim
School of Computer Engineering And Information Technology

<Abstract>

A Real-Time System is a system that is required to react to stimuli from the nvironment including the passage of physical time within time intervals dictated by he environment. Concurrency is an inherent feature of real-time system. To implement oncurrency in real-time system many software develop method is introduced, and nany programming language that provide concurrency in language level such as Modula-2, Ada, CHILL are developed. C++ that have many user but don't support oncurrency in language level is most popular programming language in developing eal-time system. C++ that have many user is most popular programming language in leveloping real-time system. C++ is object-oriented language but don't support oncurrency in language level. To develop a concurrent tasking application using a sequential language, it is necessary to use a multi-tasking kernel. In developing eal-time system with C++ how to combine the concepts of concurrency and object nust be very important subject.

In this article I will show easy way of implementing concurrency in developing Real-Time system with encapsulated concurrency by C++ class.

1. 서 론

실시간 시스템(Real-Time System)은 요구되는 기능들을 주어진 시간 제약아래 모두 정확하게 수행하는 시스템을 말한다. 이러한 시스템을 개발함에 있어 주요 고려 사항은 외부 환경과의 실시간 통신문제, 외부로부터의 요구가 짧은 시간에 집중적으로 발생될 때의 우선 처리문제, 시스템 오류 발생 시 대응책, 병행처리 프로세스들의 처리문제 등 고려해야 될 사항들은 다양하다[7]. 실시간 시스템은 외부 환경과 밀접하게 관련되어 동작하는 경우가 대부분이며 내장 실시간 시스템(Embedded Real-Time System)인 경우는 더더욱 그러하다. 외부 환경으로부터의 요구사항 발생은 순차적으로 발생된다는 보장이 없으며, 동시 다발적으로 발생되는 것이 일반적이다. 실시간 시스템은 외부 환경의 요구에 대하여 주어진 시간 제한 안에 응답하여야 하고 이러한 응답은 다른 작업을 처리하는 도중이라도 이루어져야한다. 따라서 동시에 발생되는 요구사항에 대하여 올바른 응답성을 보장하기 위하여 실시간 시스템은 병행성이 필수적으로 지원되어야 한다. 또한 효율적인 하드웨어의 사용과 모델링의 편리성도 실시간 시스템에 있어서 병행성의 중요성을 높여준다[2].

실시간 시스템은 가전제품에서부터 산업 생산 설비에 이르기까지 현대의 인간 생활 전반에 걸쳐서 많은 부분에 적용되고 있다. 실시간 소프트웨어는 실시간 시스템에 주어진 여러 가지 제약 조건하에 실시간 시스템을 효과적으로 제어하기 위한 실시간 시스템의 핵심적인 부분이다. 효과적인 실시간 소프트웨어를 개발하기 위한 많은 연구가 있어왔다. CODARTS, JSD, ROOM, OCTOPUS 등과 같은 실시간 소프트웨어를 개발하기 위한 분석 및 디자인 방법론이 제안되었고, Ada, CHILL, Modular-2 등의 프로그래밍 언어는 실시간 소프트웨어 개발을 용이하게 하는 여러 가지 특징들을 가지고 있다[4]. C++는 C 언어로부

터 파생된 범용 프로그래밍 언어로서 객체지향 개념을 지원하며, 용이한 확장성과, C언어와의 호환성, 그리고 가장 널리 사용되고있다는 특징들로 인하여 현재 실시간 소프트웨어를 구현하기 위하여 많이 사용되고 있는 언어이다[2].

병행성을 지원하기 위한 기능은 크게 운영체제에서 또는 프로그래밍 언어에서 지원 받을 수가 있다. C++와 같은 순차적으로 수행하는 언어는 병행성을 지원하는 기능이 없다. 따라서 C++를 이용하여 병행성을 사용하는 소프트웨어를 작성하려면 병행성을 지원해주는 기능이 있는 운영체제의 도움을 받아야 한다[1]. 운영체제가 지원해 주는 병행성 지원 기능에는 다음과 같은 것이 있다.

첫째, 태스크들간의 우선 순위에 따라 실행 권을 박탈할 수 있는 기능
둘째, 메시지를 통한 태스크들간의 통신기능(SendMessage, WaitMessage)
셋째, 세마포어를 이용하여 상호배제를 달성할 수 있는 기능(WaitSemaphore, SignalSemaphore)
넷째, Event를 이용하여 태스크들간의 동기화를 이룰 수 있는 기능 또는 메시지를 이용하여 동기화를 이룰 수 있는 기능(SignalEvent, WaitEvent)
다섯째, 인터럽트 처리기능과 기본적인 I/O 서비스
여섯째, 메모리 관리기능. 강 실시간 시스템(Hard Real-Time System)에서는 디스크 I/O에 의한 응답시간을 줄이기 위해서 보통 모든 태스크들이 메모리에 상주하게 된다.

2장에서는 순차적 언어인 C++를 이용하여 실시간 시스템을 개발하는데 있어서의 문제점을 해결하기 위한 방법에 대하여 살펴보고, 3장에서는 C++를 이용하여 병행성을 실제 구현한다. 4장에서는 구현된 병행성 지원 C++ 클래스를 실제 실시간 시스템의 구현에 적용하는 예를 보이고, 5장에서는 본 연구가 가지는 의미에 대하여 기술하며 앞으로 보완해야 할 내용 및 향후과제에 대하여 논한다.

2. C++를 이용하여 실시간 소프트웨어를 개발하는데 있어서의 문제점과 해결 방안

본 논문에서는 순차적 실행 언어인 C++를 이용하여 실시간 시스템을 개발하기 위하여 Win32 시스템 서비스를 이용하여 병행성을 구현한다. Win32는 현재 Windows 95와 Windows NT 운영체제의 API(Application Programming Interface)이다. Win32는 앞에서 언급한 병행성을 지원하는 운영체제의 기능들을 모두 갖추고 있다[8].

병행성을 구현하는 모델로는 Win32에서 지원하는 쓰레드 개념을 사용한다. 쓰레드는 프로세스 내에서 제어의 다중 경로를 생성하여 수행되는 것을 지원하는 응용 프로그램 인터페이스이다[8].

Win32는 여러 가지 병행성 구현을 위한 시스템 서비스를 지원하지만 기본 인터페이스는 C 언어와 같은 함수 호출 인터페이스이므로 객체지향 언어인 C++와의 통합이 자연스럽지 못하다. 그리고, 독립적으로 수행되는 쓰레드들 사이의 정보를 주고받는 메시지 전송

체계는 윈도우의 생성과 깊은 관련이 있어서 윈도우를 가지지 않는 쓰레드 또는 프로세스 사이에는 동기적 메시지 전송이 제한된다. 이러한 문제점을 해결하기 위하여서는 재사용 가능한 C++ 클래스를 만들어서 Win32 API의 병행성 구현에 대한 사항은 클래스에 감추도록 하는 것이 바람직하다.

병행성 구현을 위하여 Win32 API에서 지원하는 쓰레드를 C++ 클래스로 캡슐화할 때의 고려 사항을 살펴보면, 우선 쓰레드에 의한 C++ 클래스 객체의 잘못된 메모리 접근을 방지하기위하여 쓰레드의 생존기간과 C++ 클래스의 생존 기간을 동기화 시킬 필요가 있다. 그리고 단순히 동기적 메시지 전송만을 위하여 윈도우를 만드는 비효율성을 줄이기 위하여 윈도우를 사용하지 않는 동기적 메시지 전송을 지원할 필요가 있다.

3. Win32 환경하의 병행성을 지원하는 재사용 가능한 C++ 클래스의 구현

다음은 병행성을 지원하는 재사용 가능한 C++ 클래스에 대한 정의 부분이다. 클래스의 이름은 CTask이다.

```
class CTask
{
public:
    CTask();
    virtual ~CTask();
    BOOL CreateTask(DWORD dwCreateFlags=0,
                    UINT nStackSize=0,
                    LPSECURITY_ATTRIBUTES lpSecurityAttrs=NULL);
    HANDLE GetHandle() {return m_hThread;}
    BOOL IsBusy() {return m_bThread;}
    void Stop() {m_bEndThread = TRUE;}
    // 동기적 메시지 전송 메쏘드
    BOOL SendSyncMessage(UINT message,
                         UINT wParam, LONG lParam, LONG &rlReply,
                         BOOL bReply=TRUE,
                         DWORD dwTimeout=0xFFFFFFFF);
    // 비동기 메시지 전송 메쏘드
    BOOL SendAsyncMessage(UINT message, UINT wParam, LONG lParam);
protected:
    CEvent* m_pExitEvent;          // 쓰레드 종료 시 동기화에 사용
    BOOL m_bEndThread;             // 쓰레드 종료 여부
    BOOL m_bThread;                // 내부 쓰레드가 활성인가?
    HANDLE m_hThread;              // 내부 쓰레드 핸들
    unsigned m_usThreadAddr;       // 내부 쓰레드 주소
    // 동기적 메시지 저장소
    CMessageInfoPool m_SyncMessagePool;
    // 작업 수행 부분 순수 가상함수
    // 파생 클래스에서 재정의하여 작업을 수행한다.
    virtual void DoWork() {}

    // 동기적 메시지 전송을 기다리는 함수
    BOOL WaitSyncMessage(UINT message,
                         UINT &rwParam, LONG &rlParam, BOOL &rbReply);
    // 동기적 메시지 전송에 대한 회답을 보내는 함수
```

```
    BOOL ReplySyncMessage(UINT message, LONG lReply);
    // 비동기 메시지를 메시지 큐로부터 받아들인다.
    BOOL GetAsyncMessage(UINT &rmessage, UINT &rwParam, LONG &rlParam);
    BOOL PeekAsyncMessage(UINT &rmessage, UINT &rwParam, LONG &rlParam, BOOL bRemove=TRUE);
    void KillTask();
    int Run();
    static unsigned int __stdcall Start(void* pv);
};
```

CTask 클래스를 이용하여 태스크를 수행하기 위해서는 CTask 클래스로부터 상속되는 하위 클래스를 선언하고, 객체 파괴자(destructor)와 DoWork() 메쏘드를 재정의(overriding)하면 된다.

3.1 쓰레드의 시작

CTask 로부터 상속받는 클래스의 인스턴스를 선언하면 메모리 상에 C++ 객체가 생성된다. 그리고 CreateTask() 메쏘드를 호출하여 쓰레드를 생성한다.

다음은 CreateTask() 메쏘드의 정의 부분이다.

```
BOOL CTask::CreateTask(DWORD dwCreateFlags,
                       UINT nStackSize, LPSECURITY_ATTRIBUTES lpSecurityAttrs)
{
    // 데이터 멤버 초기화
    m_bThread = TRUE;
    m_bEndThread = FALSE;
    m_pExitEvent->ResetEvent();

    // 내부 종속 스레드를 시작한다.
    m_hTask = reinterpret_cast<HANDLE>(_beginthreadex(lpSecurityAttrs,nStackSize,Start,this,1,
                                                      &m_usThreadAddr));
    return reinterpret_cast<unsigned long> (m_hTask);
}
```

CreateTask() 메쏘드에서 하는 일은 먼저 데이터 멤버들의 값을 초기화한 다음, _beginthreadex 함수를 사용하여 Win32 쓰레드를 시작한다. _beginthreadex 함수는 _beginthread 함수보다 다양한 옵션을 줄 수 있고, C++ 런타임 라이브러리를 멀티쓰레딩 환경에 맞게 초기화하는 기능을 가지고 있다. _beginthreadex 함수가 호출되면 쓰레드가 생성되어 인자에 주어진 Start 메쏘드를 실행한다.

Start 메쏘드는 콜백 함수로 사용되기 위하여 static으로 선언되고, 수행하는 일은 Run 메쏘드를 호출하는 것이다. 따라서 실질적인 작업은 Run메쏘드에서 행해진다.

3.2 메시지 전송

CTask 클래스는 실시간 시스템상의 각 태스크간의 동기적 메시지 전송과 비동기적 메시지 전송 모두를 지원한다.

3.2.1 동기적 메시지 전송

CTask 클래스에서 동기적 메시지 전송을 수행하는 SendSyncMessage 메쏘드의 정의는 다음과 같다.

```
BOOL CTask::SendSyncMessage(UINT message,
                            UINT wParam, LONG lParam,
                            LONG &rlReply, BOOL bReply, DWORD dwTimeout)
{
    // 메시지 정보 저장
    m_SyncMessagePool.AddMessageInfo(message,wParam,lParam,bReply,0);

    char szevtMsg[32], szevtReceive[32], szevtReply[32];
    HANDLE hevtMsg, hevtReceive, hevtReply;
    // 메시지 전송 Event 객체 생성
    itoa(message, szevtMsg, 10);
    if(!(hevtMsg=CreateEvent(NULL, TRUE, FALSE, szevtMsg)))
        return FALSE;
    if(bReply){    // 회답이 요구되는 경우
        // 메시지 회답 통보 Event 객체 생성
        strcpy(szevtReply, szevtMsg);
        strcat(szevtReply, "_Reply");
        if(!(hevtReply=CreateEvent(NULL, TRUE, FALSE, szevtReply)))
            return FALSE;
    }
    else{     // 회답이 요구되지 않는 경우
        // 메시지 접수 통보 Event 객체 생성
        strcpy(szevtReceive, szevtMsg);
        strcat(szevtReceive, "_Receive");
        if(!(hevtReceive=CreateEvent(NULL, TRUE, FALSE, szevtReceive)))
            return FALSE;
    }

    // 메시지 전송을 알린다.
    SetEvent(hevtMsg);

    if(bReply){    // 회답이 요구되는 경우
        // 메시지에 대한 회답을 기다린다.
        WaitForSingleObject(hevtReply, dwTimeout);
        // 메시지 회답을 얻어낸다.
        if(!m_SyncMessagePool.GetMessageInfo(message,wParam,lParam,bReply,rlReply))
                return FALSE;
    }
    else{     // 회답이 요구되지 않는 경우
        // 메시지를 접수 했다는 통보를 기다린다.
        WaitForSingleObject(hevtReceive, dwTimeout);
    }

    // Event에 대한 핸들을 닫는다.
    CloseHandle(hevtMsg);
    if(bReply)
        CloseHandle(hevtReply);
    else
        CloseHandle(hevtReceive);

    // 메시지 정보를 제거
    m_SyncMessagePool.RemoveMessageInfo(message);
```

```
    return TRUE;
}
```

SendSyncMessage 메쏘드는 메시지 정보 저장소에 메시지 정보를 저장한다. 그리고 메시지의 전송을 메시지를 받는 태스크에 알려주는 메시지 전송 통보 이벤트를 생성하여 시그널 상태로 만든다. 그리고 메시지 전송 형태에 따라 메시지가 회답을 요구하는 형태일 때는 메시지 회답 통보 이벤트를 생성하여 기다리고 그렇지 않은 경우는 메시지 접수 통보 이벤트를 기다린다. 메시지를 받는 태스크로부터 메시지에 대한 적절한 응답이 생성된 각 이벤트를 통하여 전달되게 되면, 메시지 정보를 메시지 정보 저장소로부터 읽어낸다.

동기적 메시지를 받아들이는 함수 WaiteSyncMessage 메쏘드에 대한 정의는 다음과 같다.

```
BOOL CTask::WaitSyncMessage(UINT message,
                            UINT &rwParam, LONG &rlParam, BOOL &rbReply)
{
    char szevtMsg[32], szevtReceive[32];
    HANDLE hevtMsg, hevtReceive;
    // 메시지 전송 Event 객체 생성
    itoa(message, szevtMsg, 10);
    if(!(hevtMsg=CreateEvent(NULL, TRUE, FALSE, szevtMsg)))
        return FALSE;

    // 메시지를 기다린다.
    WaitForSingleObject(hevtMsg, 0xFFFFFFFF);

    // 메시지 정보를 얻어낸다.
    LONG lReply;
    if(!m_SyncMessagePool.GetMessageInfo(message,rwParam,rlParam,rbReply,lReply))
        return FALSE;

    if(!rbReply){  // 회답이 요구되지 않는 경우
        // 메시지 접수 통보 Event 객체를 연다.
        strcpy(szevtReceive, szevtMsg);
        strcat(szevtReceive, "_Receive");
        if(!(hevtReceive=OpenEvent(EVENT_ALL_ACCESS, FALSE, szevtReceive)))
            return FALSE;

        // 메시지 접수를 통보한다.
        SetEvent(hevtReceive);
    }

    // Event에 대한 핸들을 닫는다.
    CloseHandle(hevtMsg);
    if(!rbReply)
        CloseHandle(hevtReceive);

    return TRUE;
}
```

WaiteSyncMessage 메쏘드는 메시지를 기다리는 태스크에서 호출한다. 첫 번째 인자인 메시지 상수를 이용하여 메시지 전송 이벤트를 생성하고 이벤트가 시그널 상태가 되기를 기다린다. 이벤트가 시그널 상태가 되었을 때 메시지 저장소로부터 메시지를 읽어낸다. 전송된 메시지가 회답을 원하지 않는 메시지인 경우에는 곧바로 메시지 접수 이벤트를 생성

하여 이벤트를 시그널 상태로 만들어 메시지를 전송한 태스크에 메시지가 접수되었음을 통보를 한다. 메시지 정보를 메시지 정보 저장소로부터 읽어내어 UINT &rwParam, LONG &rlParam, BOOL &rbReply 인자들에 정보를 복사한다.

동기적 메시지에 대한 회답을 보내는 함수 ReplySyncMessage 메쏘드의 정의는 다음과 같다.

```
BOOL CTask::ReplySyncMessage(UINT message, LONG lReply)
{
    // 회답 정보를 저장한다.
    m_SyncMessagePool.AddMessageInfo(message,0,0,1,lReply);

    char szevtMsg[32], szevtReply[32];
    HANDLE hevtReply;
    // 메시지 회답 통보 Event 객체를 연다.
    itoa(message, szevtMsg, 10);
    strcpy(szevtReply, szevtMsg);
    strcat(szevtReply, "_Reply");
    if(!(hevtReply=OpenEvent(EVENT_ALL_ACCESS, FALSE, szevtReply)))
        return FALSE;

    // 메시지 회답을 통보한다.
    SetEvent(hevtReply);

    CloseHandle(hevtReply);
    return TRUE;
}
```

ReplySyncMessage 메쏘드는 회답 정보를 메시지 정보 저장소에 쓴 후 회답 통보 이벤트를 생성하여 시그널 상태로 만들어 메시지를 전송한 태스크에 회답을 전송되었다는 것을 통보한다.

메시지 저장소를 구현하는 클래스인 CMessageInfoPool 클래스에 대한 정의와 CMessageInfoPool 클래스를 구성하는 CMapMessageInfo 및 StMessageInfo 구조체에 대한 정의가 다음과 같다.

```
struct StMessageInfo
{
    UINT m_message;
    UINT m_wParam;
    LONG m_lParam;
    BOOL m_bReply;
    LONG m_lReply;
};

typedef CMap<UINT,UINT,StMessageInfo*,StMessageInfo*&> CMapMessageInfo;

class CMessageInfoPool
{
public:
    CMessageInfoPool();
    virtual ~CMessageInfoPool();
```

```
    void AddMessageInfo(UINT message,
                        UINT wParam,
                        LONG lParam,
                        BOOL bReply,
                        LONG lReply);
    BOOL GetMessageInfo(UINT message,
                        UINT &rwParam,
                        LONG &rlParam,
                        BOOL &rbReply,
                        LONG &rlReply);
    void RemoveMessageInfo(UINT message);

protected:
    CMapMessageInfo m_mapMessageInfo;
    CMutex m_mutx;
};
```

CMessageInfoPool 클래스는 메시지를 저장하는 객체인 CMapMessageInfo 클래스의 인스턴스인 m_mapMessageInfo 데이터 멤버와 m_mapMessageInfo 데이터 멤버에 메시지를 저장하는 메쏘드인 AddMessageInfo, 메시지를 얻어내는 메쏘드인 GetMessageInfo들로 구성된다.

CMapMessageInfo 클래스는 CMap 클래스의 템플릿 인스턴스 클래스인데 해싱함수를 이용하여 저장된 데이터에 대한 빠른 접근을 가능하게 하는 클래스이다.

StMessageInfo 클래스는 메시지 데이터를 나타내는 구조체이다.

3.2.2 비동기적 메시지 전송

비동기 메시지 전송을 지원하는 멤버 함수 SendAsyncMessage 메쏘드의 정의는 다음과 같다.

```
BOOL CTask::SendAsyncMessage(UINT message, UINT wParam, LONG lParam)
{
    if(m_usThreadAddr)
        return PostThreadMessage(m_usThreadAddr, message, wParam, lParam);
    return FALSE;
}
```

이 함수는 PostThreadMessage Win32 API 함수를 호출하여 메시지가 전달되는 쓰레드의 메시지 큐에 메시지를 넣는다.

비동기 메시지를 메시지 큐로부터 읽어들이는 함수 GetAsyncMessage 메쏘드는 다음과 같다.

```
BOOL CTask::GetAsyncMessage(UINT &rmessage, UINT &rwParam, LONG &rlParam)
{
    MSG msg;
    if(!GetMessage(&msg,NULL,0,0))
        return FALSE;
    rmessage = msg.message;
```

```
    rwParam = msg.wParam;
    rlParam = msg.lParam;
    return TRUE;
}
```

GetAsyncMessage 메쏘드는 GetMessage Win32 API 함수를 호출하여 메시지 큐로부터 메시지를 얻어낸다. GetAsyncMessage 메쏘드는 메시지 큐가 비어있을 경우 메시지 큐에 메시지가 들어올 때까지 기다린다.

비동기 메시지를 메시지 큐로부터 읽어들이는 멤버 함수 PeekAsyncMessage에 대한 정의는 다음과 같다.

```
BOOL CTask::PeekAsyncMessage(UINT &rmessage, UINT &rwParam, LONG &rlParam, BOOL bRemove)
{
    MSG msg;
    if(!PeekMessage(&msg,NULL,0,0,(bRemove?PM_REMOVE:PM_NOREMOVE)))
        return FALSE;
    rmessage = msg.message;
    rwParam = msg.wParam;
    rlParam = msg.lParam;
    return TRUE;
}
```

PeekAsyncMessage 메쏘드는 PeekMessage Win32 API를 사용하여 메시지 큐로부터 메시지를 얻어낸다. PeekAsyncMessage 메쏘드는 메시지 큐가 비어있을 경우 GetAsyncMessage 메쏘드와 달리 바로 반환한다.

3.3 쓰레드의 종료

CTask로부터 상속된 클래스의 인스턴스가 소멸되어 객체 파괴자가 호출되면 KillTask() 메쏘드가 호출되어 내부 쓰레드를 종료시키고, 내부 쓰레드가 종료될 때까지 기다린 다음 계속 상위 클래스의 객체 파괴자를 수행한다. 이렇게 함으로서 C++ 인스턴스가 내부 쓰레드보다 먼저 파괴되는 것을 방지한다. 만약 그렇지 않고 내부 쓰레드가 C++ 인스턴스보다 오래 유지될 경우에는 내부 쓰레드가 이미 사라져버리고 없는 C++ 인스턴스를 참조하여 원하지 않는 메모리 접근 시스템 오류가 발생될 수 있다.

4. 적용 예

개발된 병행성을 지원하는 C++ 클래스 CTask를 Robot Controller System[1]상의 Robot Command Processor 태스크에 대하여 적용한 예를 보인다.

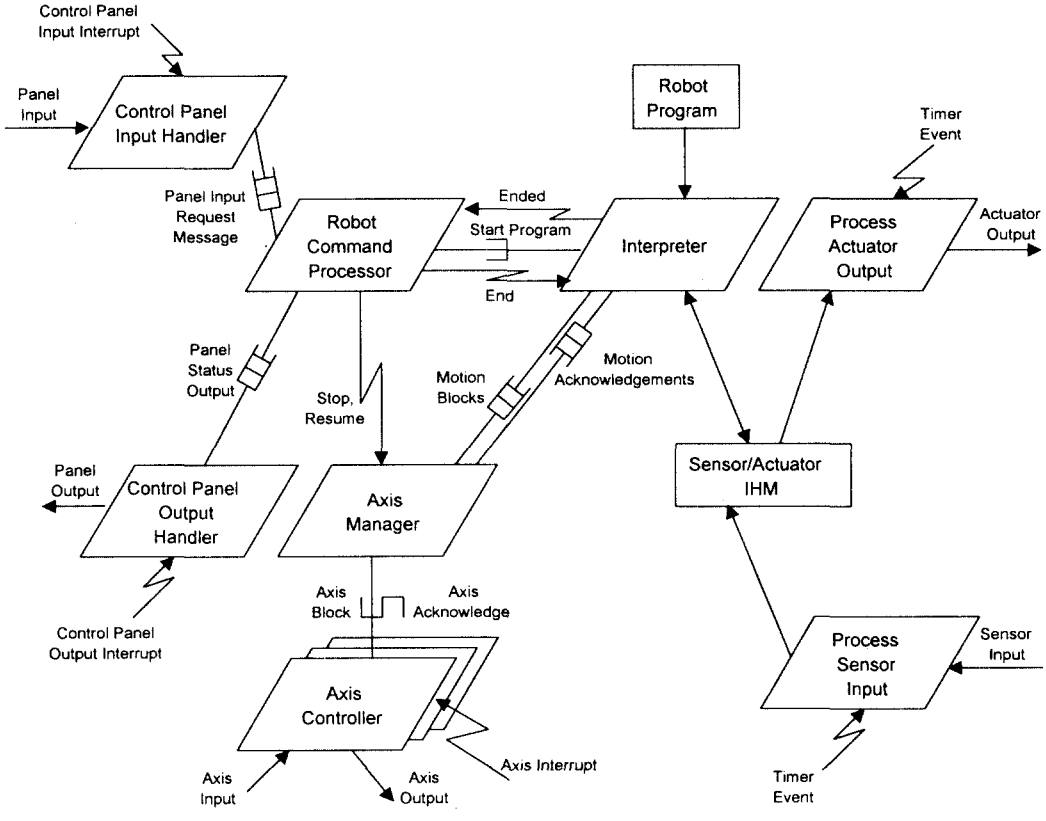

그림 1 Robot Controller System Task Architecture Diagram

그림 1에는 Robot Controller System에 대한 Task Architecture Diagram을 보여주고 있다.

Robot Command Processor 태스크는 Controller Panel Input Handler 태스크로부터 선입선출 메시지 큐(FIFO-Message Queue) 타입의 약결합 메시지(Loosely-Coupled Message)인 Panel Input Request Message를 받아들여 요구의 형태에 따라 Interpreter 태스크에게는 Start Program 메시지와 End 이벤트를 출력하고, Control Panel Output Handler 태스크에게는 Panel Status Output 메시지를 보낸다. 그리고 Axis Manager에는 Stop,Resume 이벤트를 출력한다. Interpreter 태스크로부터는 Ended 이벤트를 입력받는다.

Robot Command Processor 태스크를 본 논문에서 개발된 병행성을 지원하는 C++ 클래

스인 CTask를 이용하여 구현한 클래스인 CRobotCommandProcessor의 C++ 코드 리스트는 다음과 같다.

```
class CRobotCommandProcessor : public CTask
{
public:
     CRobotCommandProcessor();
     ~CRobotCommandProcessor();
protected:
     enum STD
     {
          MANUAL, RUNNING, SUSPENDED, TERMINATING
     } m_std;
     void ChangeProgram();

     // 작업 수행 가상함수 재정의
     virtual void DoWork();
};
```

CTask 클래스로부터 상속된 DoWork 멤버 함수를 재정의하여 Robot Command Processor 태스크가 수행해야할 일을 구현한다. DoWork 멤버 함수 재정의 C++ 코드 리스트는 다음과 같다. 이 예제 코드에서는 시스템 상에 Interpreter 태스크를 구현한 클래스 CInterpreter의 인스턴스 interpreter, Control Panel Output Handler 태스크를 구현한 CControlPanelOutputHandller 클래스의 인스턴스 control_panel_output_handler, Axis Manager 태스크를 구현한 CAxisManager 클래스의 인스턴스 axis_manager가 있다고 가정한다.

```
void CRobotCommandProcessor::DoWork()
{
     // 입력 Loosely-Coupled Message 처리 문
     while(1){
          GetAsyncMessage(message, wParam, lParam);

          switch(message){
               case RUN:
                    if(m_std == MANUAL){
                         interpreter.SendSyncMessage(START_PROGRAM,0,0);
                         control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
                                                                       STATUS_MANUAL_OFF,0);
                         control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
                                                                       STATUS_RUN_ON,0);
                    }
                    else
                         return 0L;
               break;
               case STOP:
                    if(m_std == RUNNING){
                         axis_manager.SendSyncMessage(STOP,0,0);
                         control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
                                                                       STATUS_RUN_OFF,0);
                         control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
```

```
                                                    STATUS_STOP_ON,0);
            }
            else
                return 0L;
        break;
        case END:
            if(m_std == RUNNING){
                interpreter.SendSyncMessage(END,0,0);
                control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
                                                    STATUS_END_ON,0);
            else if(m_std == SUSPENDED){
                axis_manager.SendSyncMessage(RESUME,0,0);
                control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
                                                STATUS_STOP_OFF,0);
                control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
                                                STATUS_RUN_ON,0);
            }
            else
                return 0L;
        break;
        case PROGRAM_SELECT:
            if(m_std == MANUAL){
                ChangeProgram();
                interpreter.SendSyncMessage(START_PROGRAM,0,0);
                control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
                                                    STATUS_MANUAL_OFF,0);
                control_panel_output_handler.SendAsyncMessage(PANEL_STATUS_OUTPUT,
                                                    STATUS_RUN_ON,0);
            }
            else
                return 0L;
        break;
        default:
            return;
    }

    if(m_bEndThread)
        break;
  }
}
```

5. 결론 및 향후 연구방향

병행성을 지원하지 않는 범용 객체지향 언어인 C++ 와 객체지향 프로그래밍 인터페이스를 지원하지 않는 멀티 프로세스/쓰레드 개발환경인 Win32를 재사용 가능한 클래스를 이용하여 통합하였다.

본 논문에서는 실시간 시스템의 구현을 위한 병행성에 대한 논의만 있었는데, 이후의 연구 방향은 시스템의 성능을 최대화하면서 프로그래머가 사용하기 쉬운 실시간 시스템을 위한 클래스 라이브러리를 구축하여야 하겠다. 또한 객체 지향적으로 잘 구축된 라이브러리는 실시간 모델링 도구와 연계하여 실시간 시스템 코드 자동 생성도구를 구현하는데 있

어서도 밑바탕이 될 것이다.

5. 참 고 문 헌

[1] Hassan Gomaa, Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley, 1993
[2] Maher Award, Object-Oriented Tecjnology for Real-Time System, Prentice Hall PTR, 1996
[3] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley, 1991
[4] Alan Burns, Real-Time systems and Programming Languages, Addison-Wesley, 1996
[5] Michael J. Pont, Software Engineering with C++ and CASE Tools, Addison-Wesley, 1996
[6] Tom Cargill, C++ Programming Style, Addison-Wesley, 1992
[7] 이광용, "실시간 Ada 소프트웨어 개발을 위한 객체행위 설계 방법," 정보과학회논문지 (B), 제 24 권, 제 1 호, pp.43-61, 1997
[8] Joel Powell, Multitask Windows NT, Waite Group PressTM, 1993
[9] Jeffrey Richter, Advanced Windows The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95, Microsoft Press, 1995
[10] Shem-Tov Levi, Real-Time System Design, McGraw-Hill, 1990
[11] Hassan Gomaa, "The Calibration and Validation of a Hybrid Simulation/Regression Model of a Batch Computer System," Software, Practice and Experience, vol.8, no.1, pp.11-28, 1978
[12] Hassan Gomaa, "Software Development of Real-Time Systems," CACM, vol.29, no.7, pp.657-668, 1986
[13] Hassan Gomaa, "A Software Design Method for Real-Time Systems," CACM, vol.27, no.9, pp.938-949, 1984
[14] Liu, C. L.,"Scheduling Algorithms for Multiprogramming in Hard Real-Time Environments", Journal of the ACM, vol.20, no.1, pp.46-61, 1973