내장형 실시간 시스템에서의 실시간 쓰레드와 자바 쓰레드간의 동기화

임종구[0] 박성호 강순주
경북대학교 전자전기컴퓨터공학부
unlimited@palgong.knu.ac.kr[0], slblue@palgong.knu.ac.kr, sjkang@ee.knu.ac.kr

Synchronization between Real-time threads and Java threads in embedded real-time systems.

Jong-Koo Lim[0] Sung-Ho Park Soon-Ju Kang
School of Electrical Engineering and Computer Science, Kyungpook National University

요 약

내장형 실시간 시스템을 위한 프로그래밍의 방법으로, 플랫폼에 의존적이며 실시간성이 고려되어야 하는 태스크는 실시간 쓰레드를 이용하고, GUI나 인터넷통신과 같은 실시간성이 고려되지 않는 태스크는 자바 쓰레드를 이용해서 프로그래밍하는 것이 편리하다. 이런 경우, 서로 다른 이들 쓰레드간에 동기화를 위한 방법이 필요하게 된다. 따라서, 본 논문에서는 실시간 쓰레드와 자바 쓰레드간의 동기화를 위한 방법을 제시하고 이를 위한 API(Application Programming Interface)를 설계 및 구현한다. 즉, 실시간 운영체제인 RT-Linux 상에서 수행되는 실시간 쓰레드들이 RT-FIFO와 네이티브 IPC(Inter-process communication) 메커니즘을 사용해 자바 쓰레드들과 동기화 되어질 수 있게 구현한 것이다. 구현된 이 네이티브 IPC API들은 재사용 가능한 공유 라이브러리와 클래스파일로 제작되어 활용될 수도 있다.

1. 서 론

내장형 시스템이란 휴대폰, 세탁기, 셋톱박스, 휴대용 장치 등과 같은 제어용 장치에 사용되는 특수화된 컴퓨터로 제한된 메모리를 갖고 있으며, 한정된 기능을 수행하게 된다. 또한 실시간 반응을 제공하여 고신뢰성과 안정성을 보장해야 한다[1]. 이러한 내장형 시스템의 응용 프로그램을 작성할 때, 예전에는 주로 C, 어셈블러등의 언어가 사용되어져 왔으나, 개발의 어려움과 더불어 기존의 작성된 모듈을 재사용 할 수 없다는 문제점을 가지고 있었다. 이에 대한 해결책으로 높은 이식성과 네트웍 이동성, 코드의 재사용성을 제공하는 자바라는 언어가 등장하게 되었다. 이처럼 자바가 여러가지 장점을 가지고 있지만, 자바만을 이용해 내장형 시스템의 응용 프로그램을 작성하기에는 아직까지 문제점들이 있다. 즉, 가비지 콜렉터에 의한 메모리 관리로 인해 발생하는 예측할 수 없는 지연시간, 물리적인 메모리 액세스의 불가, 표준화된 실시간 API의 부재등과 같은 자바언어가 가지는 비실시간적인 여러가지 특징들은 자바가 내장형 실시간 응용 프로그램의 작성에 적합하지 않은 점을 보여준다. 그래서 최근에는 자바의 이런 점을 해결하기 위해 RTSJ(Real-Time Specification for Java)가 제안되었고, 이 분야의 연구도 활발히 진행되고 있다[2]. 그리고, 내장형 응용 프로그램을 작성하는 또 다른 방법으로는, 하드웨어 의존적이며 실시간성을 요구하는 태스크를 처리할 때는 C 코드를 사용해 실시간 쓰레드로 처리하고, GUI나 인터넷통신과 같은 실시간성이 고려되지 않는 태스크는 자바 쓰레드를 이용해서 프로그래밍하는 것이 편리하다. 이런 경우 C 코드로 작성된 실시간 쓰레드와 자바 코드로 작성된 자바 쓰레드들 사이에서, 데이터나 메시지를 주고 받거나 이를 동기화 시키는 과정이 필요하게 된다. 하지만 이를 위한 표준화된 방법이나 라이브러리들이 없다. 그래서, 본 논문에서는 이를 위한 방법을 제시하고 실제 구현과정을 상세히 소개한다.

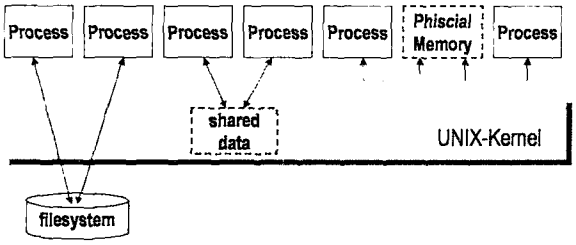
본 논문의 구성은 다음과 같다. 2.1절에서는 설계 및 구현을 위한 기본 개념으로서 UNIX 쓰레드들 사이의 동기화를 설명하고, 2.2절에서는 이를 확장하여 실시간 쓰레드와 자바 쓰레드간의 동기화가 가지는 의미를 살펴본다. 2.3절에서는 구현한 네이티브 IPC API의 대상이 된 시스템 V 계열의 IPC 시스템 콜을 살펴보고 2.4절에서는 시스템 V IPC 시스템 콜을 네이티브 IPC API로 실제 구현하는 과정을 소개한다. 2.5절에서는 구현된 이 네이티브 IPC API들을 실제로 실시간 쓰레드와 자바 쓰레드의 동기화에 이용해 봄으로써 테스트 및 검증한다. 마지막으로 3절에서는 결론 및 향후 연구계획에 대해 논할 것이다.

2. 설계 및 구현

2.1 UNIX 쓰레드들 사이의 동기화

먼저, 실시간 쓰레드와 자바 쓰레드간의 동기화를 이해

하기 위해서는, 기본 개념이 되는 UNIX 쓰레드들 사이에서 정보를 공유하는 방법을 알아야 한다. 전통적인 UNIX 프로그래밍 모델에서 보면, 하나의 시스템에는 자신의 주소공간을 가지는 여러 개의 쓰레드들이 수행 될 수 있다. 이들 쓰레드들은 제한된 리소스들을 서로 공유하면서 메시지 큐, 세마포, 공유메모리등과 같은 메커니즘을 통해 동기화되어 수행되어진다. 이를 흔히 프로세스들간의 통신, 즉 IPC(inter-process Communication) 라고 한다[3].

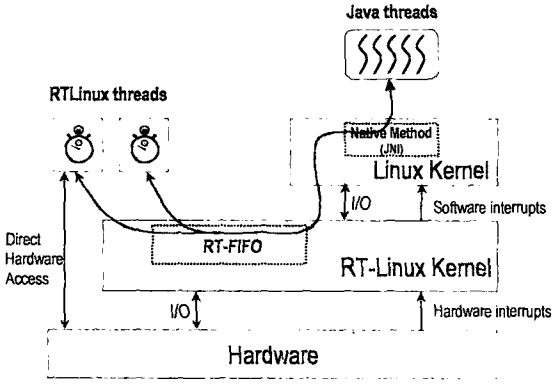


[그림 1] UNIX 프로세스들 사이의 데이터 공유

[그림 1]은 UNIX 프로세스들 사이에서 공유대상이 되는 IPC객체가 존재하는 장소가 파일 시스템이나 커널 내부 또는 물리적인 메모리의 일부가 될 수 있음을 보여준다. 파일 시스템 내의 IPC객체들을 이용하기 위한 방법으로는 read(), write(), lseek()등과 같은 시스템 콜이 있고, 커널내부의 IPC 객체들을 이용하기 위한 방법으로는 메시지큐, 세마포와 같은 메커니즘이 있으며, 물리적인 메모리내의 IPC객체들을 이용하기 위한 방법으로는 커널을 거치지 않는 공유메모리와 같은 방법이 있다.

2.2 실시간 쓰레드와 자바 쓰레드간의 동기화

본 논문에서 채택한 방법은 커널내부에 존재하는 리소스인 메시지큐와 세마포를 통해서 실시간 쓰레드와 자바 쓰레드를 동기화 할 수 있는 방법을 제시하였다. 먼저 아래 그림을 통해 실시간 쓰레드와 자바 쓰레드간의 동기화라는 개념을 살펴본다.



[그림 2] 실시간 쓰레드와 자바 쓰레드간의 동기화

[그림 2]를 보면 실시간 쓰레드와 자바 쓰레드사이의 IPC를 하기 위해 RT-Linux 커널의 RT-FIFO와 네이티브 IPC 메소드가 사용되어짐을 알 수 있다. 일반적으로, RT-FIFO 하나는 단방향성의 큐이므로, 2개의 RT-FIFO를 사용해서 양방향의 큐로 사용한다

2.3 System V IPC 시스템 콜

이 절에서는 네이티브 IPC 메소드의 구현 대상이 되는 메시지 큐, 세마포, 공유 메모리에서 지원되는 시스템 콜을 살펴본다. 모든 IPC 객체들은 공통적으로 생성과 제어 및 실제 동작을 위한 시스템 콜들을 가진다. 메시지 큐를 예를 들어보면, msgget()을 통해 메시지 큐를 생성하고, 리턴된 메시지큐의 ID를 통해서 msgsnd()나 msgrcv()를 통해 메시지를 주고 받을 수 있다. 그리고 더 이상 필요치 않은 메시지 큐는 msgctl()을 통해서 삭제하게 된다. [표 1]은 각 IPC 객체의 시스템 콜에 대해서 보여준다.

	Message Queue	Semaphore	Shared Memory
Header	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
Function to create or open	msgget	semget	shmget
Function for control operations	msgctl	semctl	shmctl
Function for IPC operations	msgsnd msgrcv	semop	shmat shmdt

[표 1] SYSTEM-V IPC 시스템 콜

2.4 네이티브 IPC 메소드의 구현

이제 메시지큐, 세마포, 공유메모리등의 IPC객체를, 실제 네티이브 IPC 메소드로 구현하는 방법에 대해 알아본다. 전체과정을 간단히 설명하면, 먼저 구현하고자 하는 native 메소드를 포함하는 클래스를 정의한다. 그리고 나서 이 클래스 내부에 선언한 native 메소드를 실제 네이티브 코드로 작성한다. 마지막으로 이 코드를 컴파일해서 생성한 공유라이브러리를 자바 쓰레드에 포함시키면 네이티브 메소드를 호출할 수 있게 된다[5]. 메시지큐를 예로 들어, 실제 수행된 과정을 보이도록 하겠다.

```
int msgget (key_t key, int msgflg);
int msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
int msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

[그림 3] 시스템 V의 메시지 큐 함수들

[그림 3]은 시스템 V 메시지큐 함수들의 프로토타입을 나타낸 것이다. 이를 자바 쓰레드에서 호출할 수 있는 형태인 네이티브 메시지큐 함수로 선언한 JavaMsgQ라는 새로운 클래스가 [그림 4]이다.

```
public class JavaMsgQ{
    public native int Create(int key, int msgflg);
    public native int Send(int msqid, MsgBuf mbuf, int msgsz, int msgflg);
    public native int Recv(int msqid, MsgBuf mbuf, int msgsz, int msgtyp, int msgflg);
    public native int Ctrl(int msqid, int cmd, int buf);
}
```

[그림 4] 자바 메시지큐 네이티브 메소드

[그림 5]는 네이티브 메소드의 구현을 위한 JNI함수들의 프로토타입을 나타낸다.

```
JNIEXPORT jint JNICALL Java_JavaMsgQ_Create(JNIEnv *, jobject, jint, jint);
JNIEXPORT jint JNICALL Java_JavaMsgQ_Send(JNIEnv *, jobject, jint, jobject, jint, jint);
JNIEXPORT jint JNICALL Java_JavaMsgQ_Recv(JNIEnv *, jobject, jint, jobject, jint, jint, jint);
JNIEXPORT jint JNICALL Java_JavaMsgQ_Ctrl(JNIEnv *, jobject, jint, jint, jint);
```

[그림 5] JNIFunction Prototype (JavaMsgQ.h)

이 각각의 JNI함수들은 내부에서 다음과 같은 일을 수행한다. 자바 쓰레드에서 넘어온 파라미터를 실시간 쓰레드가 해석할 수 있는 형태로 변환해서 실제 시스템 V 메시지큐 함수들을 내부적으로 호출하게 된다. 이 네이티브 메소드의 첫번째 인자로 공통적으로 들어있는 JNIEnv * 는 JNI 함수 테이블내에 있는 함수를 가리킬 수 있는 함수 포인터이다. 이 함수 테이블에는 대략 200가지 정도의 함수들이 존재한다. 메시지큐를 생성하고, 제어하는 JavaMsgQ_Create()와 JavaMsgQ_Ctrl()는 파라미터의 변환없이 간단히 구현되어지므로, JavaMsgQ_Send()와 JavaMsgQ_Recv()메소드에 대해서만 살펴보기로 한다.

```
JNIEXPORT jint JNICALL
Java_JavaMsgQ_Send(JNIEnv* env, jobject thisObj, jint msqid, jobject jObz,
                   jint msgsz, jint msgflg)
{
        jclass aClazz; jfieldID aFid, bFid; jstring aStr; jint aVal;
        const char* utf_string; struct MBuf msg_buf;
        // 자바코드에서 넘어온 객체(jObz)의 int필드인 mtype값을 받아온다.
        aClazz = (*env)->GetObjectClass(env, jObz);
        aFid = (*env)->GetFieldID(env, aClazz, "mtype", "I");
        aVal = (jint)(*env)->GetObjectField(env, jObz, aFid);
        // 자바코드에서 넘어온 객체(jObz)의 String필드(mtext)에 들어있는
        // 자바스트링을 C에서 사용할 수 있는 문자열로 변환.
        bFid = (*env)->GetFieldID(env, aClazz, "mtext", "Ljava/lang/String;");
        aStr = (jstring)(*env)->GetObjectField(env, jObz, bFid);
        utf_string = (*env)->GetStringUTFChars(env, aStr, 0);
        strcpy(msg_buf.mtext, utf_string);
        (*env)->ReleaseStringUTFChars(env, aStr, utf_string);
        msg_buf.mtype = aVal;
        return msgsnd(msqid, &msg_buf, msgsz, msgflg);
}
```

[그림 6] JavaMsgQ_Send()의 구현코드

[그림 6]은 자바 쓰레드에서 msgsnd()함수를 호출할 수 있게 구현된 자바 네이티브 메소드의 일부이다.

```
JNIEXPORT jint JNICALL
Java_JavaMsgQ_Recv(JNIEnv* env, jobject thisObj, jint msqid, jobject jObz,
                   jint msgsz, jint msgtyp, jint msgflg)
{
        int rcvRetval; jclass aClazz; jfieldID aFid, bFid; jstring aStr;
        jint aVal; struct MBuf msg_buf;
        // 먼저, 메시지큐에 들어있는 메시지를 읽어온다.
        rcvRetval = msgrcv(msqid, &msg_buf, msgsz, msgtyp, msgflg);
        aVal = (jint)msg_buf.mtype;
        // 메시지큐 버퍼(msg_buf)의 int 필드인 mtype값(int)을
        // 받아서 jint로 변환.
        aClazz = (*env)->GetObjectClass(env, jObz);
        aFid = (*env)->GetFieldID(env, aClazz, "mtype", "I");
        (*env)->SetObjectField(env, jObz, aFid, aVal);
        // 메시지큐 버퍼(msg_buf)의 String필드인 mtext(const char *)를
        // jstring으로 변환.
        bFid = (*env)->GetFieldID(env, aClazz, "mtext", "Ljava/lang/String;");
        aStr = (*env)->NewStringUTF(env, msg_buf.mtext);
        (*env)->SetObjectField(env, jObz, bFid, aStr);
        return rcvRetval;
}
```

[그림 7] JavaMsgQ_Recv()의 구현코드

[그림 7]은 자바 쓰레드에서 msgrcv()함수를 호출할 수 있게 구현된 자바 네이티브 메소드의 일부이다.

2.5 네이티브 IPC 메소드의 사용 및 검증

본 논문에서 설계 및 구현한 네이티브 IPC 메소드를 통해서 실시간 쓰레드와 자바 쓰레드간의 동기화 검증을 위한 환경으로 RT-Linux 커널 2.2.18-rtl이 사용되었다. 자바 쓰레드 테스트를 위해 사용된 컴파일러 및 가상머신(JVM : Java Virtual Machine)은 SUN의 j2sdk-1.3.1이 사용되었다.

```
[root@Linuxia rtipc]# uname -a
Linux Linuxia 2.2.18-rtl #2 SMP Wed Feb 20 16:31:53 GMT 2002 i586 unknown
```

[그림 8] 테스트 및 검증에 사용된 RT-Linux 커널 버전

```
[root@Linuxia rtipc]# insmod rtipc_module.o
[root@Linuxia rtipc]# lsmod
Module                  Size  Used by
rtipc_module            1556   0  (unused)
psc                    18264   0  (unused)
rtl_fifo                9080   0  [rtipc_module psc]
rtl_posixio             6928   0  [rtl_fifo]
rtl_sched              40776   0  [rtipc_module psc]
rtl_time               10100   0  [rtipc_module psc rtl_posixio rtl_sched]
rtl                    25856   0  [rtipc_module psc rtl_fifo rtl_posixio rtl_sch
ed rtl_time]
mbuff                   5128   0  (unused)
rt18139                11300   1  (autoclean)
[root@Linuxia rtipc]# ./rtipc_app
-------------------------------------------------
IPC testing between RT-task and Java-task!
-------------------------------------------------
msgQid = 768
select()'s retval = 2
Data from FIFO_1 : RTLinux_IPC_Testing
Data from FIFO_2 : From RTLinux_task To Java_task

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status

------ Message Queues --------
key        msqid      owner      perms      used-bytes  messages
0x000004d2 768        root       660        2048        2

[root@Linuxia rtipc]# java MsgRecv
-------------------------------------------------
IPC testing between Java-tasks and RT-tasks!
-------------------------------------------------
msgQid = 768
Message from RT-FIFO-1 : RTLinux_IPC_Testing
Message from RT-FIFO-2_: From RTLinux_task To Java_task
```

[그림 9] 테스트를 수행한 결과 화면

[그림 9]는 실제 테스트를 수행한 결과 화면으로써, RT-Linux환경에서 실시간 쓰레드와 자바 쓰레드가 RT-FIFO와 네이티브 IPC 메소드를 사용해 동기화가 가능함을 보여준다. 이를 위해 사용자가 정의한 모듈인 rtipc_module은 로딩된 후에 RT-FIFO와 핸들러를 생성하여 수행준비를 마친 후, 핸들러를 호출하여 실시간 쓰레드와 자바 쓰레드, RT-FIFO를 관리하면서 동기화를 수행하게 된다.

3. 결 론

본 논문에서는 내장형 시스템을 위한 응용 프로그램을 작성하는 방법으로서, 플랫폼에 의존적이며 실시간성이 고려되어야 하는 태스크는 실시간 쓰레드를 이용하고, GUI나 인터넷통신과 같은 실시간성이 고려되지 않는 태스크는 자바 쓰레드를 이용해서 프로그래밍하는 방법을 제시하였고, 서로 다른 이 두 쓰레드간의 동기화를 지원하기 위해 네이티브 IPC 메소드를 구현하는 과정을 상세히 소개하였다. 또한 이를 검증한 결과를 보여주었다. 현재는 실시간 쓰레드와 자바 쓰레드간의 IPC가 로컬에서 수행되지만 , 여기에 소켓이나 RMI(Remote Method Invocation)를 이용한 원격지상에서의 실시간 쓰레드와 자바 쓰레드간의 IPC에 대해서도 계속 연구가 진행중이다.

참고문헌

[1] ETRI, " Development of Java Embedded S/W"
[2] Mark Stolz, " Implementing the Real-Time Specification for Java"
[3] W.Richard Stevens, " UNIX Network Programming" vol2.
[4] Richard Stones and Neil Matthew, "Beginning Linux Programming"
[5] Rob Gordon, " essential Java Native Interface"
[6] http://www.jblend.com/en/jtron/2-e.html " JTRON"