실시간 운영체제 QNX 인터페이스용 미들웨어 설계 및 구현

이승열°, 이철훈
충남대학교 컴퓨터공학과
{sy-lee°, clee}@cnu.ac.kr

Design and Implementation of the Portable Middleware on Realtime Operating Systems QNX

Soong-Yeol Lee°, Cheol-Hoon Lee
Dept. of Computer Engineering, Chungnam National University*

요 약

응용프로그램은 특정 운영체제에 의존적으로 개발되기 때문에 다른 운영체제를 사용하는 시스템에 그대로 이식하는 것이 불가능하며, 운영체제가 변경될 경우 응용프로그램을 다시 개발하여야 하는 한계를 가지고 있다. 또한 응용프로그램의 시스템 종속성으로 인해 동일한 기능을 제공하는 응용프로그램을 중복 개발함으로써 개발 단계뿐 아니라 유지 및 관리에 많은 노력과 비용을 필요로 하게 된다. 응용프로그램의 운영체제에 대한 의존성을 줄이고 플랫폼이 변경되더라도 응용프로그램의 정상 동작을 지원하기 위해서는 미들웨어가 필요하다. 본 논문에서는 실시간 운영체제들의 대표적인 API 함수를 선택하여 기본 API 를 선정하고 대표적 실시간 운영체제인 QNX 위에서 POSIX 기반의 미들웨어를 설계 및 구현하였다.

1. 서 론

최근 임베디드 시스템들은 이전 시스템들과는 달리 지능화 및 정밀화되고 있으며 그 기능이 날로 복잡해지고 있다. 또한 임베디드 시스템은 신뢰성과 안전성이 매우 중요한 시스템이기 때문에 시스템의 기능 구현 후에도 오랜 기간의 안전성 시험과 시험 운용 단계를 통해 결국 많은 비용과 기간이 소요되는 것이 일반적이다.

응용프로그램은 특정 운영체제에 의존적으로 개발되기 때문에 다른 운영체제를 사용하는 시스템으로의 이식성이 문제가 되며 운영체제가 변경될 경우 응용프로그램을 다시 개발하여야 하는 단점을 가지고 있다. 또한 응용프로그램의 시스템 종속성으로 인해 응용프로그램을 중복 개발함으로써 유지 및 관리에 많은 노력과 시간을 필요로 한다. 응용프로그램의 운영체제에 대한 의존성을 줄이고 응용프로그램의 정상 동작을 지원하려면 미들웨어(middleware)의 채택이 필요하다. 즉 실시간 운영체제 인터페이스용 미들웨어는 응용프로그램과 실시간 운영체제 사이에 독립적인 인터페이스 계층을 추가함으로써 실시간 운영체제가 바뀌어도 응용프로그램의 수정 없이 미들웨어를 통하여 사용 할 수 있게 한다[1][3].

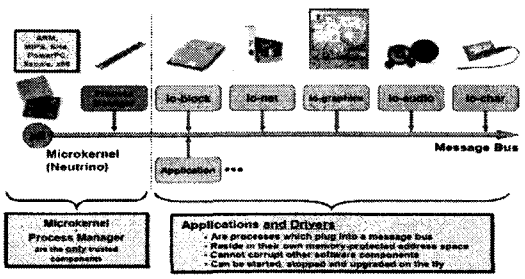
본 논문에서는 이와 같은 필요성에 따라 POSIX 기반의 미들웨어 기본 API 를 정의하고 이 API 를 바탕으로 미들웨어를 설계하여 실시간 운영체제인 QNX 에서 구현하였다.

본 논문의 구성은 2 장에서 QNX 운영체제의 특성과 POSIX 의 기능적 특징을 기술하고 3 장에서는 QNX 인터페이스를 바탕으로 미들웨어 기본 API 를 정의하고 미들웨어 구조를 설계 및 구현한다. 4 장에서는 정의된 미들웨어 기본 API 를 실시간 운영체제 QNX 위에 구현한 환경 및 결과를, 5 장에서는 결론 및 향후 연구 과제를 기술한다.

2. 관련 연구

2.1 QNX 커널

QNX 의 마이크로 커널인 Neutrino 는 운영체제의 핵심 역할만을 수행하는 순수 커널이라고 볼 수 있다. 따라서 다른 운영체제의 커널보다 매우 작으며, [그림 1]은 QNX Neutrino Microkernel 이 Message Bus 를 통해 다른 모듈과 통신하는 모습을 보여주고 있다.



[그림 1] Neutrino Microkernel

* 본 논문은 국방과학연구소의 실시간 운영체제 인터페이스용 미들웨어 연구과제의 수행결과임.

커널 이외의 파일시스템, 드라이버, 프로토콜, 네트워크 서버 등의 프로세스는 응용프로그램의 하나로서 메모리 보호가 이루어지는 고유 메모리영역에 각각 존재한다. 이러한 프로세스들은 메시지 패싱(Message Passing)이라는 통신 방법을 통하여 프로세스들간의 통신을 수행하는데, 패싱된 메시지들은 프로세스의 단위를 넘어 네트워크 상에 존재하는 다른 QNX 시스템의 어떠한 자원에도 같은 방법으로 쉽고 투명하게 전달될 수 있다.

2.1.1 프로세스 모델

QNX 는 Neutrino 마이크로 커널과 시스템 서비스를 제공하는 여러 프로세스간에 IPC(Inter Process Communication)를 통해 구성된다. 쓰레드는 하나의 실행 또는 제어 흐름을 의미하며, 한 프로세스 내의 여러 쓰레드들은 상호간에 아래와 같은 자원을 공유할 수 있다[2][5].

- 스택에 위치하지 않는 변수
- Signal handler
- Signal ignore mask
- Channels
- Connections

2.1.2 우선순위와 스케줄링 기법

QNX 는 각 쓰레드가 각자의 우선순위를 가지고 실행되며, 우선순위에 기반하여 CPU 를 점유하는 우선순위 기반 선점형(Preemptive) 커널이다. 각 쓰레드는 스케줄링 정책과 독립된 우선순위를 갖는데, 사용자의 권한에 따라 다른 범위를 가질 수 있다. 예를 들면 일반 사용자의 경우에는 "1 ~ 63", root 인 경우에는 "1 ~ 255" 의 우선순위를 지정할 수 있으며 숫자가 클수록 높은 우선순위를 나타낸다.

QNX 에서는 아래와 같은 스케줄링 알고리즘을 제공한다.

- FIFO 스케줄링(SCHED_FIFO)
- Round-Robin 스케줄링(SCHED_RR)
- Sporadic 스케줄링(SCHED_SPORADIC)

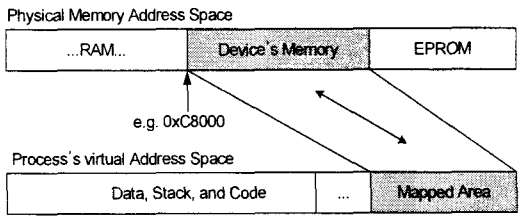
2.1.3 태스크간 통신 및 동기화

QNX 는 마이크로 커널 구조로서, 운영체제의 성능은 프로세스 간의 통신 성능과 밀접한 관계가 있다. IPC 는 일명 소프트웨어 버스라고 불리며, 이는 마치 하드웨어 버스와 같이 커널을 포함한 S/W 컴포넌트들 사이의 데이터 전송에 대한 중요한 통로 역할을 한다. 따라서 QNX 는 다양하고 강력한 IPC 들을 지원하고 있으며, 현재 제공하는 IPC 는 다음과 같다[2][5].

- QNX Messaging
- QNX Pulses
- Shared Memory
- POSIX Semaphore
- Pipes
- POSIX Message Queues
- Traditional Signals
- POSIX Realtime Signals

2.1.4 메모리 관리

메모리에 접근하기 위하여 실제 사용될 물리주소를 프로세스의 고유 가상 주소공간에 mmap*()함수를 사용하여 할당한다. [그림 2]는 메모리 매핑의 파라미터와 할당된 공간의 상태를 보여주며, 보통 연속되지 않는 물리공간에 메모리를 할당하기 때문에 MAP_PHYS 와 MAP_ANON 과 같은 플래그를 사용하여 DMA(Direct Memory Access)와 같은 연속적인 공간을 할당 받을 수 있다[2][5].



[그림 2] 메모리 할당 방법

2.2 POSIX

POSIX 는 컴퓨터 처리 환경을 위한 이식 가능한 운영체제 인터페이스로, IEEE P1003 기술 위원회에서 작성한 컴퓨터 운영체제 서비스의 기본 규격이다. POSIX 기본은 유닉스를 기반으로 하였으나 다른 운영체제에서도 쉽게 이식될 수 있다[7].

3. 미들웨어 구조 설계 및 구현

3.1 미들웨어 구조 설계

미들웨어는 분산 환경에서 네트워크 운영체제와 응용프로그램 상에 위치하여 시스템의 이질적인 환경을 추상화 시켜주고 서로 다른 기능을 정합시키는 계층이다. 즉 다양한 시스템과 응용프로그램을 통합하여 사용자에게 시스템 투명성을 제공하는 역할을 한다. [그림 3.1]은 미들웨어의 구조를 보여주고 있다.

3.1.1 시스템 추상화 계층(SAL)

인터페이스용 미들웨어의 이식성과 재사용성을 높이기 위해 정의된 계층으로서 미들웨어가 각 실시간 운영체제와 상호 작용을 할 수 있게 각 실시간 운영체제에 의존적인 부분을 집약한 계층이다.

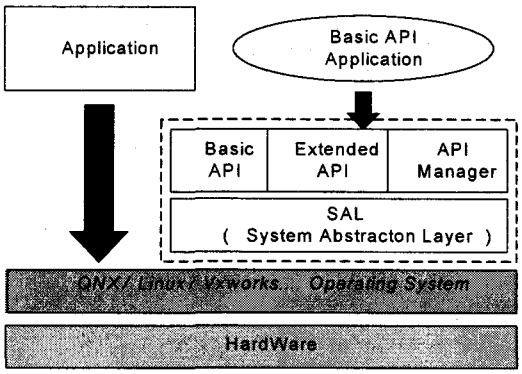
3.1.2 API 관리자(API Manager)

시스템 자원의 성능 향상을 위해 운영체제의 모든 API 를 적재하지 않고 연동되는 운영체제에 필요한 API 를 동적으로 추가/갱신함으로써 시스템의 확장성 및 변경에 대처할 수 있다. API 관리자는 응용프로그

램이 필요로 하는 운영체제의 서비스리스트(service list)를 유지해야 하며, 해당 서비스를 지원하기 위한 API 의 존재 유무를 파악하여 네트워크를 통해 동적으로 재구성하는 기능 구조를 가진다.

3.1.3 기본 API 및 확장 API

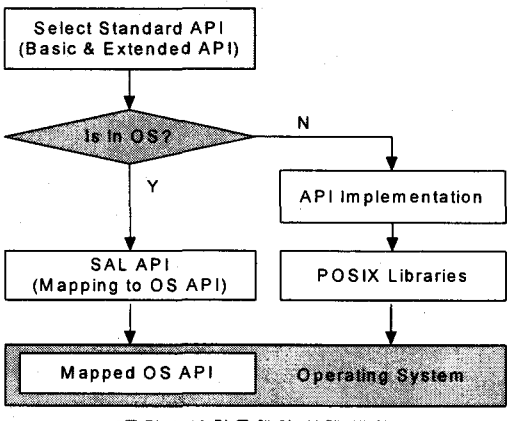
API 규모가 커지면 API 를 구현한 미들웨어의 크기가 커짐에 따라 메모리의 부담과 실행 성능의 저하를 가져온다. 따라서 자주 사용되는 API 만을 기본 API 로 정의하여 항상 메모리에 적재하도록 하고, 기타 API 는 확장 API(Extended API)로 정의하여 시스템의 메모리 상황과 응용프로그램에 따라 필요한 경우에만 선택적으로 적재하도록 한다.



[그림 3.1] MiddleWare 구조

3.2 미들웨어 세부 설계

실시간 운영체제 인터페이스용 미들웨어의 기본 API 는 대상 운영체제가 제공하는 API 에 매핑 가능한 경우와 그렇지 못한 경우가 있다. 기본 API 에 해당하는 공통적인 기능을 제공하는 API 가 대상 운영체제에 존재할 경우, 기본 API 는 SAL 계층을 거쳐 대상 운영체제에 의존적인 부분들이 추상화된다.

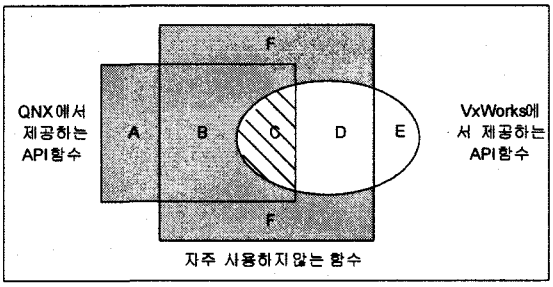


[그림 3.2] 미들웨어 설계 방안

이에 반해 공통 기능을 제공하는 API 가 대상 운영체제에 존재하지 않을 경우, 동일한 기본 API 를 대상 운영체제에서도 이용할 수 있도록 하기 위하여 미들웨어 계층에서 POSIX 라이브러리를 이용하여 구현한다. 이와 같은 실시간 운영체제 인터페이스용 미들웨어에서 기본 API 를 제공하기 위한 미들웨어 설계는 [그림 3.2]와 같다.

3.3 기본 API(Basic API) 정의

본 논문에서는 QNX 와 Vxwork 의 API 를 비교하여 실시간 운영체제 인터페이스용 미들웨어를 위한 공통 API 를 정의하는 방법을 다음 두 개의 운영체제를 예로 들어 설명한다.



[그림 3.3] 공통 API 정의

[그림 3.3]에서 QNX 환경에서 지원되는 API 함수를 A,B,C 부분이라 하고, VxWorks 환경에서 지원되는 API 함수를 C,D,E 부분이라고 하자. C 는 QNX 와 VxWorks 에서 그 기능이 서로 같은 함수를 의미한다. 서로 다른 운영체제에서 공통적으로 제공되는 기능은 그만큼 기본적이며 중요한 API 라는 의미이다.

B 는 VxWorks 에는 해당되는 기능이 없는 API 로서 QNX 에서 일반적으로 사용되는 API 집합을 의미한다. A 는 QNX API 로 정의는 되어 있으나 잘 사용하지 않은 API 들을 의미한다. VxWorks 환경에서 지원되는 API 함수는 공통부분 C 그리고 B 에 해당하는 D, A 에 해당하는 E 로 표현할 수 있다. F 는 공통 API 함수에 새로이 추가될 필요가 있는 함수이다.

QNX 환경에서 지원되는 API 함수(A,B,C)를 모두 조사하고, VxWorks 환경에서 지원되는 API 함수(C,D,E)를 모두 조사한 후, 공통으로 기능을 갖는 함수(C)와 추가할 필요가 있는 함수(F)를 파악하여 기본 API 로 정의한다. 기타 API 들은(A,B)를 QNX 를 위한 확장 API 로 정의하고, (D,E)를 VxWorks 를 위한 확장 API 로 정의하는 방법을 이용한다.

위에서 설명한 방법으로 두 가지 운영체제들을 위한 기본 API 를 정의하였다. 태스크, 인터럽트, 메모리, 세마포어, 타이머, 메시지 큐, 메시지 포트, 시그널, WatchDog 타이머, 총 9 개의 그룹으로 분류되어 있으며 [표 3-1]과 같이 총 69 개로 구성되어 있다.

[표 3-1] 기본 API 수

API 모듈	개수	API 모듈	개수
Task Management	23	Signal	5
Interrupt	5	Message Port	4
Memory Management	12	Timer	6
Semaphore	6	WatchDog Timer	4
Message Queue	4		

3.4 미들웨어 구현

[표 3-2]는 VxWorks 와 QNX 의 Semaphore 에 대한 API 의 함수를 이용하여 기본 API 를 정의한 것이다.

[표 3-2] Semaphore API 비교

기본 API	VxWorks	QNX
semaphoreCreate()	semCCreate()	Sem_open()
semaphoreReceive()	semTake()	Sem_wait()
semaphoreSend()	semGive()	Sem_post()
semaphoreReset	semFlush()	Sem_init()
semaphoreDelete()	semDelete()	Sem_close()

실질적으로 VxWorks 와 QNX 의 커널 소스를 접근할 수 없기 때문에 각 운영체제에 대한 Reference API 를 참조하여 인자값과 리턴값을 참고하여 기본 API 를 구현하였다.

```
int sem_post(sem_t * sem)                    //QNX
{ … }
STATUS semGive(SEM_I semId)                  //VxWorks
{ … }
```

[그림 3.4] 다른 매개변수 타입을 가진 세마포어

두 운영체제에서 함수의 역할은 같지만 매개 변수의 형태가 다르다. 따라서 이를 해결해야 두 함수를 통합할 수 있을 것이다. 통합을 원활히 하기 위해 POSIX 기반으로 구현하였으며 POSIX 가 지원 안되는 부분은 직접 매핑하는 방법으로 기본 API 를 만들었다.

아래 [그림 3.5]는 [그림 3.4]의 세마포어 기능을 통합하는 기본 API 를 구현한 것이다. 따라서 이 SemaphoreSend 함수는 같은 매개변수를 가지고 QNX, Linux, VxWorks 에서 모두 동작하도록 설계되어 있다.

```
/* 기본 API */
MW_Status_t SemaphoreSend( SEM_ID semaphore )
{ … }
```

[그림 3.5] 세마포어 기본 API

QNX 에서 SemaphoreSend 기본 API 의 구현은 [그림 3-6]과 같으며, POSIX 기반으로 Linux 와 같은 다른 운영체제에서도 동작 가능하다.

```
MW_Status_t SemaphoreSend(MW_SEMA *semaphore) {
if ( sema_exist(semaphore) ){
our_tcb = my_tcb();
if(((semaphore ->flags&SEM_TYPE_MASK) == MUTEX_SEMA4)
        && (semaphore->current_owner != our_tcb) )
{  error = IFM_ERROR_INTERNAL; }
else{
if ( semaphore->recursion_level > 0 )
{
    if ( (--(semaphore->recursion_level)) == 0 )
    {
        semaphore->count++;
        semaphore->current_owner = (MW_TCB *)NULL;
      if(semaphore->flags & SEM_DELETE_SAFE)
…
        if ( semaphore->flags & SEM_INVERSION_SAFE )
           …
 else
semaphore->count++;
 if(semaphore->first_susp != (MW_TCB *)NULL ) {
pthread_cond_broadcast(&(semaphore->sema_send));
 }
 …
 Return ( error );
}
```

[그림 3.6] 세마포어 기본 API 구현(for QNX)

VxWorks 에서 SemaphoreSend 기본 API 구현은 [그림 3.7]과 같다.

```
MW_Status_t SemaphoreSend( SEM_ID *semaphore )
{
        MW_INT_t sendSem;
        sendSem = semGive( semaphore );
        return sendSem;
}
```

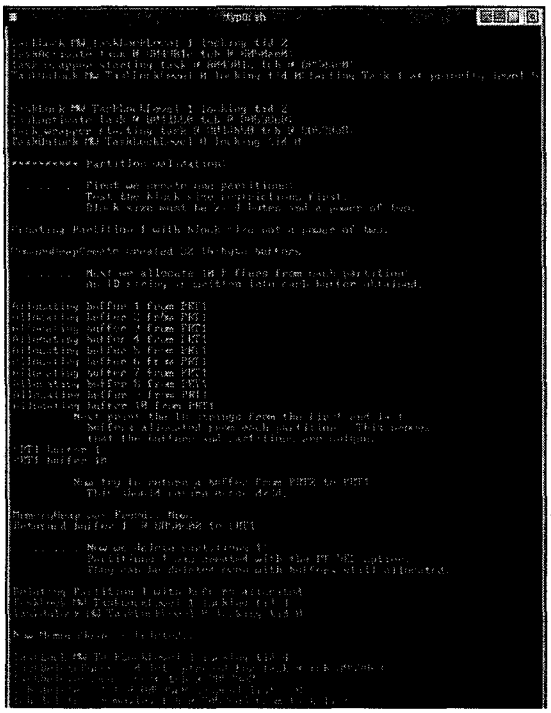
[그림 3.7] 세마포어 기본 API 구현(for VxWorks)

운영체제마다 세마포어를 보내는 함수의 기능이 같더라도 실제로는 함수 이름도 다르고 반환 타입이나 매개변수가 어느 하나로 통일할 수 없을 정도로 전혀 상관관계가 없을 수도 있다. 또한 매개변수의 개수가 다를 수도 있다. 그래서 이 과정이 공통 API 제작에 있어 가장 어려운 부분이면서도 중요한 부분이다.

4. 테스트 환경 및 결과

본 논문에서는 펜티엄 3-933, 256MB 메모리 기반의 컴퓨터에 실시간 운영체제 QNX 6.2.1 버전의 운영체제를 설치하여 테스트하였다. 그리고 Task 를 비롯하여 9 개의 API 모듈을 POSIX 기반으로 설계 및 구현하였

다.



[그림 4.1] 테스트 결과(For QNX)

프로그램 테스트는 QNX 운영체제 위에 본 논문에서 구현한 미들웨어를 올려 우선 순위가 5 인 태스크를 생성하고, 생성과 동시에 Partition Validation 함수를 실행시키는 순서로 실험하였다.

먼저 메모리로부터 파티션(Partition 1)을 하나 할당 받고 할당 받은 파티션으로부터 16 바이트 크기의 버퍼 32 개를 생성하였다. 버퍼를 1 번부터 10 번까지만 할당해 주었고 다음으로 파티션에 할당된 버퍼들을 Free 시켜주고 해당 파티션을 삭제하는 것을 마지막으로 메모리 기본 API 를 이용한 미들웨어 테스트를 마쳤다. [그림 4.1]은 테스트 프로그램의 결과이며, 이 예제 프로그램은 리눅스와 VxWorks 에서도 같은 결과를 보여준다.

5. 결론 및 향후 과제

본 논문은 시스템 의존적인 응용프로그램들의 이식성을 해결하기 위해 현재 가장 보편적으로 이용되는 대표적 실시간 운영체제들이 가지고 있는 API 를 이용하여 기본 API 를 정의하고 미들웨어를 설계 및 구현하였다. 기본 API 를 이용하여 구현한 응용프로그램은 운영체제에 의존적이지 않고 실행 가능하다.

향후, 본 연구의 성과를 바탕으로 미들웨어의 사용으로 인한 성능상의 오버헤드를 테스트할 것이다. 더 나아가 다른 실시간 운영체제에도 확장 가능한 미들웨어 프로그램의 개발이 진행되어야 한다.

참고 문헌

[1] Qusay H.Mahmoud, "Middleware for communications", WILEY, 2004. 6

[2] Herman Bruyninckx, "Real-Time and Embedded Guide"

[3] 김선자 외, "임베디드 운영체제 기본화 동향", 정보처리학회지, 제 9 권 제 1 호, 2002, 한국정보처리학회

[4] Embedded System & RTOS, http://www.windriver.com

[5] QNX Software Systems, http://www.qnx.com

[6] MapuSoft Technologies LLC, http://www.oschanger.com

[7] POSIX Library Functions, "http://docs.sun.com/app/docs/doc/806-7021