

설계와 구현을 통한 임베디드 OS의 이해와 응용 _#1

Overview

본 기사는 임베디드 시스템을 구성하는데 있어 가장 중요한 운영체제(OS)의 흐름을 이해하는데 있다. 실제로 임베디드 OS인 RTOS를 구현해 봄으로써 독자들의 이해를 돕고자 한다. 앞으로 많은 관심을 바라며, 이번 호에서는 앞으로 어떤 내용들이 연재되는지 살펴보자.

글: 서민우/과학기술 정보연구소(www.stii.co.kr)
minucy@hanmail.net

임베디드 시스템은 알게 모르게 우리 주변에 널리 퍼져 있다. 유비쿼터스, 홈네트워킹과 같은 용어들은 모두 임베디드 시스템을 기본 전제로 한다. 우리가 가지고 다니는 휴대전화나 MP3 플레이어, 휴대용 게임기 등도 모두 임베디드 시스템이다. 이러한 임베디드 시스템은 CPU, 메인 메모리, I/O와 같은 하드웨어, RTOS와 같은 소프트웨어로 구성된다.

임베디드 시스템 개발자들은 특정한 목적에 맞게 임베디드 시스템을 구성하며, 그러한 임베디드 시스템의 특징은 표면적으로 I/O 디바이스에 의해 나타난다.

임베디드 시스템 개발자들이 해야 할 일들 중에는 이러한 I/O 디바이스를 제어하는 디바이스 드라이버를 작성하는 일도 있다. 디바이스 드라이버는 OS의 큰 흐름 내에서 동작한다. 따라서 OS의 동작을 정확히 이해하고 있어야만 디바이스 드라이버의 정확한 작성이 가능하다. 디바이스 드라이버를 작성하여 구동 시킬 경우 동기화의 문제가 발생할 수 있는데, 이러한 문제는 OS의 흐름을 정확히 모르고 디바이스를 작성할 때 발생한다.

본 기사는 궁극적으로 정확한 디바이스 드라이버의 작성을 위해 OS의 흐름을 이해하는데 그 목적을 둔다. OS의 흐름을 이해하는데 OS를 직접 구현해보는 것만큼 좋은 방법은 없다. 따라서 본 기사에서는 OS를 구현해 나가는 과정에 대해 구체적으로 소개하고자 한다.

Overview

OS는 보통 프로세스 관리, 메모리 관리, 파일 시스템, 네트워크, 디바이스 드라이버로 구성된다. 이 중 OS의 흐름은 프로세스 관리와 밀접하게 관련되어 있다. 따라서 프로세스 관리를 중심으로 OS를 구현해 보고자 한다.

OS는 ARM core를 기본으로 구현한다. 따라서 ARM 명령어와 ARM 아키텍처의 programmer's model에 대해서는 어느 정도 익숙해야 본 기사를 이해할 수 있을 것이다.

Host 개발 환경은 i386 PC, 리눅스 파란 7.3 OS, ARM용 GNU toolchain을 사용하기로 한다. 보다 구체



적인 내용은 다음 기사에서 OS를 본격적으로 구현해 나가면서 언급하기로 하겠다.

본 기사를 쓰는 동안에는 OS, RTOS, kernel 이라는 용어가 차이점이 없다라는 가정 하에 혼용해서 사용하기로 한다. 또한 프로세스와 태스크라는 용어도 혼용해서 사용하기로 한다.

OS의 구현은 다음과 같은 순서로 진행한다.

그럼 어떤 내용을 구현해 나갈 지 좀 더 구체적으로 알아보자.

1. 개발 환경 구성 방법 및 부트로더
2. 기본 커널의 설계와 구현
3. 하드웨어 인터럽트 루틴의 설계와 구현
4. multitasking의 설계와 구현
5. 동기화 문제의 발생원인: 코드간 간섭
6. 동기화 문제의 해결 방법 설계 및 구현
7. IPC의 설계 및 구현

1. 개발 환경 구성 방법 및 부트로더

a) 리눅스의 설치 구성 요소를 알아보고 toolchain의 설치 방법을 알아본다.

여기서는 OS를 개발하기 위해 리눅스를 설치하고, GNU ARM toolchain의 설치 방법을 알아본다. toolchain이란 하나의 프로그램을 개발하기 위해 프로그램을 작성하여 컴파일하고 디버깅하고 최종 이미지를 만드는 과정에서 순서대로 사용하는 프로그램들을 말한다. 간단히 말해서 컴파일러, 디버거 등을 생각하면 되겠다. 이러한 toolchain에 대해서도 구체적으로 언급을 하고자 한다.

b) 부트로더를 작성해 보고 이해해 본다.

부트로더의 역할은 커널이 정상적으로 부팅을 수행하고 최종적으로 부팅을 완료하여 시스템을 관리할 수 있는 상태가 될 수 있도록 최소한의 환경을 구성해 주는 역할을 한다.

임베디드 시스템에서의 부트로더는 PC에서의 BIOS와 부트로더(예를 들어 lilo와 같은)의 합으로 볼 수 있다. 이러한 임베디드 시스템에서의 부트로더가 하는 역할을 알아

보고 구현하여 테스트 해 보기로 하자.

c) 부트로더를 작성하는 과정에서 cache, mmu 등의 특성을 이해해 보고, 구현은 어떻게 이루어지는지, 어떻게 나타나는지를 살펴보기로 한다.

소프트웨어가 동작하는 환경에서 cache와 mmu가 하는 기능은 중요하다. cache의 경우 시스템의 성능을 높이기 위해 사용하는 하드웨어 장치이다. 가상주소를 관리하는 mmu의 역할은 여러 프로세스간에 주소영역을 분리해 냄으로써 프로세스간에 상호 간섭을 막는 역할을 한다. 그런데 mmu의 경우, 역할은 중요하긴 하지만 그 동작 원리를 제대로 이해하지 못하고 쓰는 경우가 많다. 그러다 보니 프로그램을 이해하는 데에도 뭔가 명확하지 않은 요소를 제공한다. 따라서 여기서는 mmu의 특성을 명확하게 이해하고 가상주소가 갖는 의미를 생각해 보기로 한다.

2. 기본 커널의 설계와 구현

a) 어셈블리 루틴에서 C로 진입하는 루틴을 작성해 본다.

일반적으로 OS의 앞부분은 어셈블리어로 작성한다.

이유는 CPU architecture 의존적인 부분의 경우 C로 표현할 수 없으며, 따라서 필수적으로 어셈블리어로 시작할 수 밖에 없다. 예를 들어 CPU의 인터럽트를 막는 동작은 C로 표현할 수 없다. 이러한 이유로 부팅 초기에 약간의 어셈블리 루틴을 작성해야 한다. 이후에는 대부분 C로 구현해 나간다. 이러한 과정에서 어셈블리 루틴에서 어떻게 C 루틴으로 진입하는지 구현을 통해 알아보자.

b) C 루틴상에서 커널 메시지를 출력해 본다.

OS를 개발해 나가는 과정에서 디버깅 메시지를 찍어보는 건 필수적이다. 그렇지 않을 경우 우리는 눈을 감고 개발해 나가야 하는 상황을 맞게 된다. OS의 개발은 무척 정밀한 작업이며, 따라서 개발해 나가는 순간순간 디버깅 메시지를 출력해 볼 수 있어야 한다.

c) interrupt vector table을 작성해 본다.

Interrupt vector table이란 용어는 많이 들어봤을 것이다. 그래서 그 개념이 어떤지는 알지만 뭔가 명확치 않은 부분도 있었으리라 생각한다. 여기서는 interrupt

vector table의 역할이 무엇인지 구체적으로 구현을 통해 이해해 본다.

d) system call handler를 설계 및 구현해 본다.

사용자 영역에서 커널 영역으로 진입하는 방법을 시스템 콜이라 한다. 시스템 콜은 프로세스가 커널에 작업을 요청할 경우에 사용하는 인터페이스로 커널의 반 정도를 차지한다. 따라서 시스템 콜을 이해한다는 건 적어도 커널의 반 정도를 이해한다는 것과 같다. 여기서는 그러한 시스템 콜을 처리하기 위한 커널 루틴이 구체적으로 어떻게 구현되는지 자세히 살펴보기로 하자.

3. 하드웨어 인터럽트 루틴의 설계와 구현

하드웨어의 요청에 의해 임의의 사용자 영역에서 커널 영역으로 진입하는 방법을 하드웨어 인터럽트라고 한다. 하드웨어 인터럽트는 I/O 등의 디바이스가 커널에 작업을 요청할 경우에 사용하는 인터페이스로 커널의 나머지 반 정도를 차지한다. 따라서 하드웨어 인터럽트를 이해한다는 건 커널의 나머지 반을 이해하는 것과 같다. 시스템 콜의 콜이라는 용어와 하드웨어 인터럽트의 인터럽트라는 용어의 차이점은 무엇지도 생각해 보기로 하자.

a) Interrupt controller를 초기화하는 루틴을 작성해 본다.

Interrupt controller는 I/O 디바이스에서 요청하는 하드웨어 인터럽트를 CPU로 연결시켜 주는 역할을 하며, 하드웨어 인터럽트를 이용하기 위해서 interrupt controller를 초기화 시켜 주어야 한다. 여기서는 interrupt controller의 구체적인 역할이 무엇인지, 그에 따라 초기화 루틴의 구현은 어떻게 이루어지는지 알아보기로 하자.

b) timer controller를 초기화하는 루틴을 작성해 본다.

OS의 심장과 같은 역할을 하는 timer controller는 주기적으로 하드웨어 인터럽트를 발생시키는 역할을 한다. 보통 RTOS의 경우는 초당 1000번 정도의 하드웨어 인터럽트를 발생시킨다. OS에서 중요한 역할을 하는 timer 디바이스를 사용할 수 있도록 초기화 하는 루

틴을 작성해 본다.

c) timer interrupt handler를 설계 및 구현해 본다.

일반적인 하드웨어 인터럽트 처리루틴을 이해하고 구현하기 위해 구체적으로 timer 인터럽트를 처리하는 루틴을 작성해 본다. timer 인터럽트는 OS 내에서 여러 가지 중요한 역할을 한다. 예를 들어 round robin 방식의 스케줄링은 time slice라는 개념이 필요하며, 이러한 time slice는 timer 인터럽트 처리 루틴에서 관리하는 요소 중에 하나이다. 또, 시간과 관련한 여러 가지 OS의 기능 역시 timer interrupt handler에서 제공한다. 예를 들어 리눅스 시스템 프로그래밍에서 alarm()과 같은 시스템 콜 함수 등은 OS 내부에서 timer interrupt handler가 제공하는 기능을 이용한다.

4. multitasking의 설계와 구현

OS가 일반적으로 펌웨어와 다른 점은 multitasking을 수행한다는 점이다. 즉, 일반적인 펌웨어는 하나의 태스크를 수행하는 반면에, OS는 여러 태스크를 관리한다. 따라서 multitasking은 OS가 수행해야 할 필수적인 기능이다.

a) 커널 태스크를 생성해 본다.

OS 역시 부팅 초기에는 하나의 논리적인 루틴으로 수행을 시작한다. 부팅이 완료될 즈음에는 multitasking을 수행하기 위한 여러 가지 데이터를 초기화하여 multitasking의 수행을 시작한다. 이 과정에서 부팅 초기에 수행하던 루틴 자체도 하나의 태스크로 관리해야 하며, 이를 여기서는 커널 태스크라 했다. 이 커널 태스크를 위한 데이터를 초기화한다.

b) 문맥 전환 루틴을 설계 및 구현해 본다.

태스크간에 전환을 문맥 전환이라 한다. 문맥이란 어느 시점에서의 태스크의 수행 상태를 말한다. 문맥 전환은 현재 수행하고 있는 태스크의 문맥을 저장하는 문맥 저장 부분과 바로 다음에 수행할 태스크의 문맥을 복구하는 문맥 복구 부분으로 구성된다. 문맥 전환 루틴을 기준으로 태스크간의 논리적인 흐름이 바뀌게 된다.

multitasking을 수행하기 위해 OS의 가장 안쪽에는 이러한 문맥전환 루틴이 자리하고 있다. 따라서 OS를 이해하기 위해서는 문맥전환의 원리를 반드시 이해해야 한다.

c) 스케줄링 루틴을 설계 및 구현해 본다.

태스크간 전환하는 시점에서 새로운 태스크를 선택하는 동작을 스케줄링이라 한다. 이 부분을 어떻게 설계하고 구현하느냐에 따라 우선순위 기반의 스케줄링, 라운드 로빈 방식의 스케줄링 등이 결정된다. 구현의 관점에서 보면 스케줄링 루틴 자체는 어렵지 않다. 여기서는 RTOS에서 필수적인 우선순위 기반의 스케줄링 루틴을 구현해 보기로 한다. 일반적으로 스케줄링 루틴은 문맥전환 루틴을 포함하고 있다.

d) 시스템 콜 처리 루틴에 태스크 스케줄링 루틴을 추가해 본다.

e) 하드웨어 인터럽트 처리 루틴에 태스크 스케줄링 루틴을 추가해 본다.

보통 우리가 응용프로그램을 작성할 때, 다른 프로세스로의 전환이 어떻게 이루어지는지 잘 모르는데, 그 이유는 프로세스간 전환을 수행하는 프로세스 스케줄링 루틴이 커널 루틴의 한 가운데 있기 때문이다. 이러한 프로세스 스케줄링 루틴을 시스템 콜 처리 루틴과 하드웨어 인터럽트 처리 루틴에 넣어 보고 시험해 본다.

5. 동기화문제의 발생원인: 코드 간 경쟁

a) 공유 영역과 critical section의 개념을 명확히 이해한다.

하나 이상의 태스크 등에서 공통으로 접근하는 데이터 영역을 공유 영역이라 하며, 이 공유 영역을 접근하는 루틴을 critical section이라고 한다. 일반적으로 OS 내에서 동기화의 문제가 이 공유 영역을 중심으로 발생한다. OS 내의 디바이스 드라이버에서 관리하는 I/O 디바이스 내부의 레지스터도 공유 영역이며, 따라서 디바이스 드라이버는 critical section을 포함하게 된다. 보통은 공유 영역을 하나 이상의 태스크가 접근할 때는 순서대로 접근해야 하는데 OS 내에서는 그 순서

가 겹치는 문제가 발생할 수 있다.

b) 하드웨어 인터럽트 처리 루틴과 태스크 루틴간의 간섭 원인을 정확히 이해한다.

하드웨어 인터럽트는 비동기적으로 발생한다. 즉, 하드웨어 인터럽트란 언제 발생할지 알 수 없다. 태스크를 수행하던 도중에 하드웨어 인터럽트가 발생할 경우 마치 태스크가 함수를 호출하는 모양으로 하드웨어 인터럽트 처리 루틴으로 뛰어 들어가게 되는데, 이 때 태스크 루틴과 하드웨어 인터럽트 루틴이 겹칠 수 있다. 보통은 문제가 없는데, 겹치는 루틴이 같은 공유 영역을 접근할 경우 논리적인 문제가 발생할 수 있다.

c) 태스크 루틴간의 간섭 원인을 정확히 이해한다.

하드웨어 인터럽트 루틴에서 태스크 스케줄링을 수행할 수 있다. 즉, 하드웨어 인터럽트가 발생할 경우 비동기적으로 스케줄링이 발생할 수 있다. 하드웨어 인터럽트 루틴에서 수행하는 스케줄링 루틴으로 인해 태스크 루틴과 태스크 루틴간에 간섭이 있을 수 있다. 이 경우도 두 태스크가 공유 영역을 접근하는 부분에서 태스크 루틴간 간섭이 있을 경우 논리적인 문제가 발생할 수 있다. 태스크 루틴간에 어떻게 간섭이 있게 되는지는 OS를 구현해 나가며 구체적으로 알아보기로 하자.

6. 동기화문제의 해결 방법 설계 및 구현

a) cli, sti 명령어를 이용하여 인터럽트 처리 루틴과 태스크 루틴간에 발생하는 코드간 간섭을 해결해 본다.

하드웨어 인터럽트 처리 루틴과 태스크 루틴간의 간섭을 해결하는 방법이다. 즉, 하드웨어 인터럽트가 발생하면 문제가 발생할 수 있는 지점에서 하드웨어 인터럽트를 막는 방법이다. 그렇게 함으로써 문제가 되는 지점을 태스크 루틴과 하드웨어 인터럽트 루틴이 순서대로 접근하게 하는 방법이다. 이 방법이 구체적으로 어떻게 사용되는지 구현을 통해 알아보기로 하자.

b) key와 flag의 개념을 이용하여 태스크 루틴간 발생하는 코드간 간섭을 해결해 본다.

태스크 루틴과 태스크 루틴간의 간섭을 해결하는 방법이다. 즉, key나 flag 역할을 하는 변수를 두어 문제가 발생할 수 있는 지점을 태스크간에 순차적으로 접근

하게 하는 방법이다. 세마포어나 뮤텍스(Mutex) 변수 역시 이러한 key나 flag의 개념을 가지고 있다.

c) 세마포어와 뮤텍스의 구현 원리를 알아보고 구현해 본다.

디바이스 드라이버를 작성하는 과정에서는 세마포어와 뮤텍스를 사용해야 할 경우가 많다. 세마포어나 뮤텍스의 개념을 정확히 알지 못하고는 디바이스 드라이버를 구현하는 과정에서 이러한 객체들을 적절히 쓸 수 없으며, 결론적으로 버그를 내포하고 있는 코드를 구성하게 된다. 여기서는 세마포어와 뮤텍스를 직접 구현해 봄으로써 그것들의 동작 특성을 정확히 이해해 보고자 한다.

b) sleep_on 함수와 wake_up 함수를 설계 및 구현해 본다.

디바이스 드라이버를 구현하는 과정에서 흔히 사용하는 개념의 함수들이다. sleep_on 함수의 경우 현재 수행하던 태스크를 논리적으로 더 이상 진행할 수 없을 경우, 예를 들어 I/O 디바이스에서 도착하는 데이터를 기다려야 할 경우, 현재 태스크를 wait queue에 넣고 스케줄링을 수행하는 루틴으로 구성된다.

wake_up 함수의 경우는 sleep_on 함수를 이용해 wait queue에서 대기하고 있는 태스크를 꺼내는 역할을 한다. 즉, sleep_on 함수는 wait queue에서 조건을 기다리던 태스크를 꺼내서 ready queue로 넣고, 필요할 경우 스케줄링을 수행하는 루틴으로 구성된다. sleep_on 함수의 경우는 그 특성상 시스템 콜 루틴에서만 사용할 수 있다. wake_up 함수의 경우는 일반적으로 하드웨어 인터럽트 루틴과 시스템 콜 루틴 모두에서 사용할 수 있다.

7. IPC의 설계 및 구현

a) 생산자 소비자 문제를 이해한다.

일반적으로 하드웨어 인터럽트 루틴과 태스크 루틴 간, 태스크 루틴과 태스크 루틴간에 데이터를 주고받게 된다. 예를 들어 네트워크 패킷을 기다리던 태스크에게 하드웨어 인터럽트 루틴에서 그 패킷을 전달해 줄 수가 있다. 또 파이프를 이용하여 태스크간 통

신을 하는 경우도 데이터를 주고받게 된다.

이 때, 데이터를 주는 루틴이 데이터에 대한 생산자, 받는 루틴이 데이터에 대한 소비자 역할을 한다. 즉, OS 내부의 많은 루틴이 이 생산자 소비자 관계에 있다. 이렇게 데이터를 주고 받는 과정에서 발생할 수 있는 동기화 문제를 생각해 보고, 앞에서 구현한 동기화 함수를 이용하여 해결해 보기로 한다. 실제로 OS 수업 중에 배운 잠자는 이발사 등의 문제도 OS 내부에서 디바이스 드라이버 등을 작성하는 과정에서 발생할 수 있는 문제에 대해 인간 세상에서 그 모델을 찾은 것이다.

b) 하드웨어 인터럽트 처리 루틴과 태스크간 통신을 설계 및 구현해 본다.

네트워크 패킷을 기다리는 태스크와 그 패킷을 전달하는 네트워크 인터럽트 루틴, 하드디스크에 있는 파일을 기다리는 태스크와 그 패킷을 전달하는 하드디스크 인터럽트 루틴, 키보드의 문자를 기다리는 태스크와 그 문자를 전달하는 키보드 인터럽트 루틴은 모두 데이터 통신을 한다. 이러한 루틴간에 데이터 통신을 어떻게 구현할 지 생각해 보자.

c) 태스크간 통신을 설계 및 구현해 본다.

파이프나 시그널 등은 태스크간 통신을 수행하기 위해 OS 내부에서 제공하는 기능이다. 따라서 OS 내에는 태스크간 통신을 위한 기능도 구현되어야 한다.

이상에서 앞으로 구현하고자 하는 OS의 내용을 간략하게 알아보았다.

다음 호에는 개발 환경을 구성하고 부트로더를 작성해 보기로 한다. 구체적으로는 GNU toolchain을 설치하고, 부트로더를 작성해 보기로 하자. 그 과정에서 mmu의 정확한 특성도 이해해 보기로 하겠다. 참고로 이후에 구현할 OS의 이름은 'cuteOS'라 하겠다. ^{Real Time}