목 차

그림 목차

표 목차

1. 서론

오늘날 네트워킹 디바이스들은 정보화 사회의 뼈대를 이루는 기반 시설로서의 역할을 하고 있다. LAN, WAN을 구성하는 스위치나 라우터에서부터 상위 레이어의 컨트롤을 담당하는 캐쉬 엔진, L4 스위치, 나아가서는 개개인에게 정보를 제공하는 터미널로서의 역할을 하는 휴대폰이나 PDA까지 네트워킹 디바이스들은 현재의 정보화 사회에서 빼놓을 수 없는 중요한 구실을 하고 있다.

또한 각종 프로세서와 메모리, 주변 장치의 발달로 인해 대부분의 네트워킹 디바이스를 구성하는 요소들이 갈수록 소형화, 대용량화 되어가면서 정보화 사회의 필요에 의해 새롭게 등장하는 장비에서 뿐만 아니라 현존하는 대부분의 전자 장비에 네트웍 기능이 추가되어져서 진화 혹은 재탄생하는 추세이다. 이에 따라 앞으로는 무선 네트웍 기술이나 전력선 네트웍과 같은 새로운 피지컬 레이어의 출현에 따라 대부분의 전자 장비에 네트웍 기능이 추가될 것으로 기대되어 지고 있다.

1.1 배경

그런데 이런 네트웍 기능을 가진 디바이스들의 컴퓨팅 환경은 과거의 한가지 일만 할 수 있으면 되는 단순한 전자장비의 그것과는 상당한 차이를 보이고 있다. 우선 기본적으로 프로세서가 처리해야 하는 외부정보의 양이 기존과는 상당한 차이가 있다. 네트웍 기능이 들어있다는 의미는 일단 그 장비가 네트웍 데이터의 패스로서 의존하는 피지컬 레이어에 대해 그 표준에 의거하여 피지컬 프로토콜을 처리할 수 있는 능력을 가진 컨트롤러가 내재해야 한다는 것을 의미하며 결론적으로 그 컨트롤러를 다룰수 있고 컨트롤러로의 데이터 전송과 인터럽트를 처리할 수 있어야 한다는 것을 의미한다. 또한 현재 대부분의 네트웍 디바이스들과 애플리케이션들이 TCP/IP를 트랜스포트 레이어로

사용하고 있다는 점을 볼 때 TCP/IP 스택을 소프트웨어적으로 혹은 하드웨어적으로 내장하고 있어서 이 스택을 통한 데이터 플로우에 대해 다른 네트워킹 디바이스들과 충분히 통신할 수 있을 만큼의 성능을 낼 수 있는 계산 능력이 필요하다. 그리고 이런 네트웍 하부 구조에 대한 처리 기능에 더불어 본래 자신의 기능에 충실할 수 있도록 애플리케이션을 처리하고 관리하는 능력이 있어야 한다. 즉, 네트워킹 디바이스 혹은 네트웍 기능을 가지는 디바이스 들은 과거의 전자장비들과는 달리 기본적으로 하드웨어 리소스를 더욱 요구하며 프로세서의 프로세싱 파워를 더욱 필요로 하고있다. 거기에 더불어 이런 리소스들을 관리하고 TCP/IP 스택, 애플리케이션 등과 같은 여러 프로세스 엘리먼트들을 관리하고 조정하여 보다 높은 성능을 낼 수 있도록 하는 운영체계가 필요하다.

물론 운영체계 없이도 이런 네트워킹 디바이스들이 제대로 움직이도록 할 수 있지만 디바이스 스케일러빌러티의 빈곤, 애플리케이션의 안정성 부재, 개발과정의 비효율성 등으로 인해 네트워킹 디바이스에 있어서는 전혀 호감을 끌 수 없는 방법이라고 할 수 있다. 현재 대부분의 네트워킹 디바이스들은 그 자체에 운영체계를 가지고 있으며 이런 운영체계로서는 하나의 운영체계가 거의 독점하고 있다시피한 PC용 운영체계 시장과는 달리 여러 종류의 독특한 특성을 가진 운영체계가 공존하고 있는 상황이다. Xinu[8], uC/OS[10]와 같이 개인이나 학계등에서 공개한 교육용 혹은 평가용 운영체계를 채용하고 있는 몇몇 장비들을 제외하곤 대부분 pSOS[14], VxWorks[17]와 같은 상용 운영체계를 구입하여 실장하고 있으며 시스코와 같은 일부 대형 장비업체에서는 IOS[20]와 같이 자신들이 직접 개발한 운영체계를 사용하고 있다. 현재 네트워킹 디바이스의 운영체계로 많이 사용되고 있는 운영체계들은 다음과 같이 구분할 수 있다[1].

✍✍ **공개 또는 공개에 가까운 운영체계**: Xinu, uC/OS, etc.

✍✍ **상용 운영체계** : pSOS, VxWorks, Nucleus[18], LynxOS[47], VRTX[16], QNX[15], etc.

✍✍ **자체 개발 운영체계**: IOS, Enterprise OS[21], etc.

1.2 문제점

이렇게 다양한 운영체계들이 존재하지만 이를 자신들이 개발하고 있는 네트워킹 디바이스에 채택하려 할 때 산, 학계의 많은 개발자들은 다음과 같은 여러가지 문제점을 지적하고 있다[2]. 우선 교육용 운영체계를 채택하는 것은 무료로 공개가 되어있는 경우가 대부분이므로 가격적인 면에서 부담이 없고 구조가 간단하므로 자체적으로 애플리케이션을 개발하는 것이 쉽다. 하지만 사용자가 적거나 혹은 그 운영체계를 이용해 애플리케이션을 개발하는 데 있어 어떠한 대가도 지불하지 않았으므로 문제가 생겼을 때 구체적으로 도움을 받거나 책임 요소를 분명히 할 수가 없다. 즉, 운영체계 자체를 사용하는 데 있어 위험성이나 불안 요소를 떠안고 갈 수 밖에 없으며 이런 특성은 특히 제품의 성능과 안정성이 금전적인 손익과 직결되는 산업계에 있어 크게 환영을 받지 못하는 요인이 되고 있다.

따라서 제품화 되고 있는 대부분의 네트워킹 디바이스들은 상용 운영체계를 사용하고 있는데 이런 운영체계들은 구입과 동시에 개발하려 하는 해당 아키텍쳐에 대한 초기화 코드, 운영체계 소스코드와 개발 툴등을 함께 제공하므로 쉽게 애플리케이션 개발에 들어갈 수 있는 이점이 있다. 그리고 운영체계에 대한 지원 체계가 확실하므로 문제가 생겼을 때 체계적인 지원을 신속하게 받을 수 있으며 또한 운영체계 자체의 성능이 보장되어 있으므로 안심하고 사용할 수 있는 장점이 있다. 하지만 이런 여러가지 장점에도 불구하고 상용 운영체계는 가격적인

면과 확장성 면에서 상당한 제약을 가진다[3]. 대부분의 상용 운영체계는 운영체계 자체만으로도 수천불 혹은 수만불에 달하는 가격이며 거기다 개발 툴을 합하면 개인이나 학계, 중소 규모의 회사에서는 감당하기 힘든 금액을 필요로 하게 된다. 거기다 네트웍 기능을 위해서 TCP/IP 스택, 특정 디바이스 드라이버, 특정 네트웍 애플리케이션등이 구입 옵션으로 제공되기 때문에 어떤 애플리케이션을 위한 네트워킹 디바이스를 만들기 위해 상용 운영체계를 구입하게 될 경우 그 가격대는 수만불대에 다다르게 된다.

그리고 그 운영체계를 사용한 제품이 시장에 나와 판매를 통해 이윤을 얻게 될 경우 일반적으로 제품 한대당 특정 규모의 운영체계에 대한 라이센스 비가 책정이 되므로 개발자의 부담은 더욱 가중되게 된다. 또한 상용 운영체계의 개발 툴 자체는 그 가격만큼 잘 정비가 되어 있고 개발자에게 뛰어난 기능들을 제공하지만 그 자체를 돌릴 수 있는 곳, 즉 개발을 위한 머신이 제한적이다. 대부분 특정 유닉스나 윈도우 기반의 운영체계를 개발 툴을 위한 필요조건으로 요구하고 있으며 이는 개발자에게 있어 또 다른 장벽이 될 수 있다. 그리고 이런 상용 운영체계위에서 돌아가는 애플리케이션 코드를 만드는 경우 한 운영체계를 위해 개발된 코드는 쉽게 다른 운영체계를 위해 포팅이 될 수 없다. 즉, 현재의 네트워킹 디바이스에 많이 채택되어 있는 상용 운영체계는 운영체계와 그 상에서 돌아가는 애플리케이션이 밀접한 관계를 이루고 있어 다른 특성을 가진 운영체계로 쉽게 포팅이 되지않는다. 결국 상용 운영체계는 상용인 만큼 성능과 개발을 위한 환경이 보장되지만 상당한 규모의 대가를 요구하고 그 확장성, 호환성에 있어서 미약한 면을 보이고 있으며 또한 개발에 익숙해지기 위해서는 많은 노력이 필요한 것이 사실이다.

1.3 임베디드 리눅스

이런 상황에서 임베디드 리눅스의 등장은 많은 문제점의 해결과 더불어 수많은 이득을 개발자에게 가져다 주고 있다. 여기서 임베디드 리눅스란 일반적으로 PC의 운영체계로서 쓰이는 GNU/Linux 시스템의 일반적인 명칭인 리눅스와는 다소 차이가 있다. PC에서의 리눅스는 운영체계의 기본 하부구조로서의 리눅스 커널과 더불어 그 상위에서 유닉스 운영체계와 유사한 구조를 이루어주는 Glibc C 라이브러리 시스템과 이를 기반으로 한 수많은 조그만 유틸리티들로 이루어지며 이를 총괄적으로 통합하여 리눅스 운영체계라 부르고 있다[4]. 이런 커널과 라이브러리, 시스템 유틸리티들의 집합을 GNU/Linux 시스템이라는 정식 명칭으로 부르고 있으며 이는 현재 언급하고 있는 임베디드 리눅스에서의 리눅스와는 차이가 있다.

임베디드 리눅스는 기본적으로 특정 임베디드 애플리케이션에 맞도록 리눅스 커널의 크기와 성능을 최소, 최적화 시켜 만들어낸 커널을 의미하며 여기에 C 라이브러리나 유틸리티등은 필요에 따라 추가되거나 교체되어 하나의 시스템을 만드는 형태가 된다. 이런 커널의 최소, 최적화에는 커널의 내부 구조의 변화도 가능하며 이를 극단화 시키면 일반적인 상용 운영체계의 성능을 뛰어 넘는 것도 가능하다.

임베디드 리눅스의 장점은 다음과 같이 수많은 점을 열거할 수 있다[2]. 우선 리눅스 커널 자체가 태생적으로 모듈형식의 코드로 이루어져 있으므로 쉽게 최소, 최적화가 가능하다. 또한 오픈 소스이기 때문에 소스 코드 자체가 공개가 되어있고 이의 적용에 따른 로열티 자체가 없으므로 부담없이 쉽게 자체 개발 아키텍쳐에 적용이 가능하다. 그리고 산, 학계에 걸친 방대한 수의 사용자들로부터 지난 수년간 직접 사용되며 검증된 코드들이므로 효율적이고 안정적이며 성숙된 내용을 가지고 있다.

또한 네트워킹 애플리케이션에 필수적인 TCP/IP, PPP, X.25, HDLC,

FDDI등의 네트웍 코드를 가지고 있으며[48] 그 종류에 있어서도 다양하고 무료로 사용할 수 있으므로 개발에 있어 엄청난 이득을 가져올 수 있다. 또한 네트워킹 장비에서 쓰이는 현존하는 거의 대부분의 프로세서에 포팅이 되어있으며 대부분의 컨트롤러를 위한 디바이스 드라이버가 제공이 되므로 그 적용 영역또한 방대할 수밖에 없다. 그리고 사용자 뿐만 아니라 개발자 그룹의 크기도 오픈 소스이면서도 개발을 위한 자체 툴과 라이브러리를 가지고 있는 현대적인 운영체계라는 특성에 맞물려 전세계적으로 거대한 수의 개발자들을 가지고 있다. 따라서 애플리케이션의 수도 방대하며 자체 문서화도 거의 모든 영역에 대해 상용 운영체계 못지 않게 잘 이루어져 있으므로 개발자에게 커다란 장점을 가져다 준다[5, 6].

또한 이미 PC상에서 리눅스 애플리케이션과 커널의 개발에 익숙한 사람들에게 있어 임베디드 리눅스는 자신에게 익숙한 환경을 제공하므로 쉽게 임베디드 애플리케이션을 개발할 수 있게 해주고 자체 개발툴은 어떤 운영체계나 아키텍쳐상에도 대부분 포팅이 되어있으므로 한번 그 개발툴에 익숙해지면 개발환경의 제약이 없어진다는 장점이 있다. 결국 임베디드 리눅스는 개발자에게 네트워킹 디바이스의 개발에 있어 금전적인 제약과 개발 환경에 익숙해지기 위한 시간, 그리고 제품의 신뢰성에 대한 장벽을 넘을 수 있게 해주는 운영체계라 할 수 있다.

1 년 전에는 임베디드 리눅스가 하나의 실험 단계로서의 운영체계로서 사람들에게 깊이 인식되지는 못했지만 지난 1 년 동안 수많은 회사들이 임베디드 리눅스를 이용해 네트워킹 디바이스들을 많들어내고 임베디드 리눅스와 그에 따르는 개발 환경을 갖추어 판매하는 회사들이 전세계적으로 그 수를 늘리고 있으며 현재도 계속 생겨나고 있다[7].

1999 년 9 월에 있었던 임베디드 시스템 컨퍼런스의 자료들[49]을 살펴보면 임베디드 리눅스 마켓이란 단어를 찾아볼 수 없을 정도로 임

베디드 리눅스는 흔히 볼 수 없는 현상의 하나였고 개발자의 창조에 의한 결과물일 뿐이었다. 즉, 철저한 마켓팅 계획과 프로모션 전략에 의한 것이 아니라 개인적 취향의 하나였다는 것이다. 하지만 지금은 수많은 회사들이 상업적인 성공을 바라보면서 임베디드 리눅스 시장에 뛰어들고 있고 새롭게 설립된 임베디드 리눅스 컨소시엄에는 114 개 이상의 영리, 비영리 단체가 멤버로 가입해 있다. 그중에는 메이저 임베디드 운영체계 회사인 WindRiver와 QNX도 포함되어 있으며 EE Times에 따르면 개발자간의 리눅스 사용이 작년에 비해 1400%이상 늘어났다고 한다[50].

1.4 논문의 목적

이 논문은 이런 수많은 장점을 가진 임베디드 리눅스를 네트워킹 장비에 특화되고 최적화된 형태로 디자인하고 그에 따르는 라이브러리와 애플리케이션을 갖춰 하나의 임베디드 리눅스 시스템으로서 디자인하고 특정 네트워킹 애플리케이션을 실현해내는 네트워킹 디바이스를 이 디자인을 이용해 구현하고 실제 제품을 만들어보며 그 결과물을 다른 상용 운영체계나 상용 임베디드 리눅스 패키지들의 디자인과 비교해보고 평가해서 네트워킹 디바이스를 위해 보다 나은 임베디드 리눅스 시스템을 만들어내는데 그 목적이 있다.

1.5 구성

논문의 전체적인 구성은 다음과 같다. 2장에서는 현재의 네트워킹 디바이스들을 위한 임베디드 운영체계들을 구체적으로 분석 및 비교하고 3장에서는 2장에서의 기능 비교, 분석을 토대로 네트워킹 디바이스들을 위한 운영체계 디자인을 위한 필요 기능요건들을 추출한다. 4장에서는 3 장의 결과들을 요구 조건으로 삼아 네트워킹 디바이스를 위한

임베디드 리눅스와 라이브러리, 애플리케이션등의 전체적인 시스템 디자인을 한다. 5 장에서는 구현결과와 이에 대해 평가하고 6 장에서는 결론과 앞으로의 연구목표에 대해 다룬다.

2. 네트워킹 디바이스용 내장형 운영체계의 현황

일반적으로 네트워킹 디바이스에 많이 쓰이고 있는 운영체계들은 교육용 혹은 그와 비슷한 이유에서 무료로 소스 코드가 공개되어 있는 것들, 그리고 운영체계 회사에서 전문적으로 만든 상용 운영체계, 마지막으로 네트워킹 디바이스를 만드는 회사에서 직접 만들어낸 자체 개발 운영체계의 세 종류로 나눠볼 수 있다. 이 장은 이들을 분류하고 특징들을 정리하여 네트워킹 디바이스를 위한 운영체계에 필요한 요소와 기능들을 추출해내기 위한 분석의 목적을 가진다.

2.1 무료 혹은 무료에 가까운 내장형 운영체계

교육용의 목적으로 개발되어 소스가 무료이거나 특정 이유로 비영리 목적에는 무료이고 영리목적일 경우도 거의 대가없이 소스 코드를 사용할 수 있는 운영체계들은 다음과 같다.

2.1.1 Xinu

Xinu는 Purdue University의 Douglas E. Comer가 교육용의 목적으로 개발한 운영체계였다[1, 8]. 실제로 지금도 수많은 교육기관에서 이를 운영체계 학습을 위해 사용하고 있다. 그리고 최근 버전에서는 TCP/IP도 구현이 되어있어 많은 교육기관에서 데이터 커뮤니케이션이나 컴퓨터 네트웍 코스에서 활용하고 있다. Xinu는 병렬 프로세싱, 메시지 패싱, 세마포어, 메모리 관리, 통일된 디바이스 I/O 와 같은 세련된 운영체계적 특징을 가지고 있으면서 TCP/IP스택도 가지고 있으며 교육용으로 제작된 것이라 잘 정리된 레퍼런스 문서가 존재한다는 장점을 가지고 있다. 거기다 인텔, 모토롤라, 스팍등의 여러 CPU에 포팅이 되어있고 아주 간단하고 크기가 작으며 무료이기 때문에 이를 네트워킹 디바이

스용 운영체계로 사용하여 장비를 제작하는 회사들이 있다.

2.1.2 uC/OS and uC/OS-II

uC/OS는 Jean J. Labrosse가 만든 POSIX표준 호환의 실시간 운영체계이며 uC/OS-II는 이의 다음 버전이다[1, 9]. 비영리 목적이라면 소스코드를 홈페이지에서 무료로 다운로드 받을 수 있고 라이센스 비도 거의 무료에 가깝다. 이렇게 저렴한 가격의 실시간 운영체계이기 때문에 많은 계측장비, 산업용 장비의 운영체계로 쓰이고 있다. uC/OS와 uC/OS-II는 4000 라인 정도의 C코드와 약간의 아키텍쳐 의존적인 어셈블리 코드로 이루어져 있어 여러 CPU로의 포팅이 쉬우며 실제로도 Intel, Motorola, ARM, SPARC, Hitachi SH등의 32 비트 CPU와 Zilog의 Z80 과 같은 8 비트 CPU, 그리고 TI의 여러 DSP등에 포팅이 되어있다. 그리고 C 프리프로세서 매크로로 필요없는 기능을 제거할 수 있는 스케일러빌러티를 제공하며 완전한 선점형 실시간 멀티 태스킹을 할 수 있는 구조를 지니고 있다. 또 여러 메일 박스, 큐, 세마포어와 같은 시스템 서비스를 제공하고 있다. 하지만 TCP/IP스택을 가지고 있지 않기 때문에 네트워킹 디바이스의 운영체계로 사용하기 위해서는 uC/OS, uC/OS-II용 스택을 판매하는 회사의 제품을 구매해야 하는 단점이 있다.

2.1.3 eCOS

eCOS는 지금은 RedHat에 합병된 Cygnus에서 만든 오픈 소스 실시간 운영체계로서 로열티가 없고 제조사에서 잘 정리된 지원체계를 갖고 있다는 장점이 있다[1, 11]. 리눅스가 실장되지 못할 만큼 극히 제한된 임베디드 디바이스를 위해 제작되었으며 모토롤라의 파워 피씨, 스팍, 암, 밉스 계열의 CPU로 포팅이 되어있으며 커널 자체를 쉽게 특정 목적에 맞게 구성할 수 있는 GUI툴이 제공된다. 그리고 애플리케이션 개발을 위한 자체 C 라이브러리를 가지고 있고 리눅스 호환을 위한

EL/IX라는 리눅스 호환 API를 가지고 있어 EL/IX를 이용해 리눅스에서 프로그래밍한 애플리케이션은 수정없이 eCOS에서 돌릴 수 있다. 이외에 ITRON[12] 호환 실시간 라이브러리를 제공하며 TCP/IP스택을 제공한다. 하지만 아직까지 TCP/IP스택 자체가 베타버전이어서 네트워킹 디바이스용 운영체계로서의 안정성에 대한 검증이 이루어지지 않았다. 그리고 운영체계 자체는 오픈 소스이지만 개발 툴과 지원체계를 이용하기 위해서는 상용 운영체계 만큼의 부담이 든다. 그림 1 은 eCOS의 아키텍처를 보여준다.

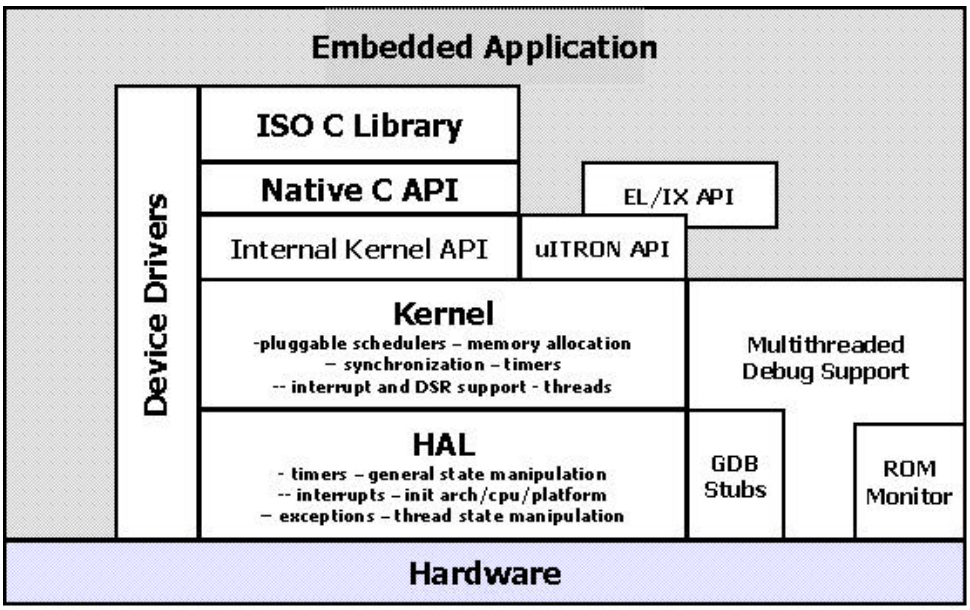

그림 1 eCOS 아키텍처

2.2 상용 내장형 운영체계

상용 임베디드 운영체계의 수는 거의 100 가지에 이르며 그 중에서 TCP/IP스택을 자체적으로 가지고 있거나 외부 회사를 통해 지원하

여 기본적인 네트워킹 기능을 보유하고 있는 운영체계만도 50 가지가 넘는다[3]. 하지만 자체적으로 지원되는 아키텍처의 수가 적어 네트워킹 디바이스에 많이 쓰이는 Motorola, ARM, MIPS계열의 프로세서들에는 개발자가 직접 포팅을 해야하는 운영체계들과 상업적으로 그다지 성공하지 않은 운영체계들을 제외하여 현재의 네트워킹 디바이스에 많이 쓰이고 있는 운영체계들을 찾아보면 다음과 같은 약 6 가지의 운영체계가 나온다.

2.2.1 pSOS+

pSOS는 지금은 WindRiver Systems, Inc.로 합병된 Integrated Systems, Inc.에서 만든 임베디드 디바이스용 실시간 커널이다[13, 14]. 이를 이용한 운영체계를 pSOSystem이라고 부른다. 이 커널은 기본적으로 클라이언트-서버의 구조를 가지고 있으며 소프트웨어 버스를 이용해 운영체계 각 모듈간의 통신을 한다. 모든 쓰레드들이 하나의 어드레스 스페이스를 공유하고 있으며 따라서 컨텍스트 스위칭 타임은 매우 적으나 서로 다른 쓰레드간의 메모리 보호가 이루어지지 않으므로 프로그래밍 테크닉이 시스템의 안정성에 영향을 준다. 이를 피하기 위해 메모리 보호와 관리를 위한 라이브러리를 따로 판매한다. 멀티 프로세서를 지원하기 위한 버전인 pSOS+m도 존재하며 이 경우 프로세서간은 하나의 마스터와 다수의 슬레이브 관계를 요구하며 공유 메모리, 메시지 기반의 소프트웨어 버스등의 방법을 통해 프로세서간의 통신을 하도록 되어있다. 하지만 pSOS+m커널은 인텔의 x86 계열만을 지원하며 마스터-슬레이브라는 구조때문에 fault tolerant하지 않다. priority레벨은 256 단계이며 쓰레드의 수는 메모리 크기에 의해 제한된다. 그리고 prioritized FIFO와 prioritized round-robin의 두가지 스케줄링 메커니즘을 가진다. MMU지원은 메모리 보호와 함께 옵션 사항이며 피지컬 페이지 크기는 4KB이지만 가상 메모리를 지원하지 않는다. 외부 인터럽트 처리는 인터럽트된

쓰레드 내부 컨텍스트에서 처리되므로 시스템 안정성을 위해서는 숙련된 프로그래밍 태크닉이 요구된다. 태스크 관리와 메모리 관리 기능이 부족하므로 이를 위한 옵션 라이브러리들이 많이 제공된다. 네트웍 쪽으로는 TCP/IP를 비롯한 다양한 프로토콜 스택들이 유료로 선택가능하며 FTP, Telnet, TFTP, NFS, HTTP 서버등이 애플리케이션으로 존재한다. 그리고 웹 브라우저도 여러 써드파티 회사에서 이용 가능하다. pSOSystem용 애플리케이션은 pRISM+라는 개발 툴을 이용해 개발가능하며 이 툴은 여러 유닉스 시스템과 윈도우 95/98, NT등에서 운용할 수 있다. pRISM+는 CORBA를 하부구조로 가지고 있어 개발툴 자체가 폭넓은 확장성을 가지고 있다. 에디터로는 SNiFF++라는 툴이 있으며 pROBE+라는 GUI디버거를 제공한다. 그림 2 는 pSOSystem의 아키텍처를 보여준다.

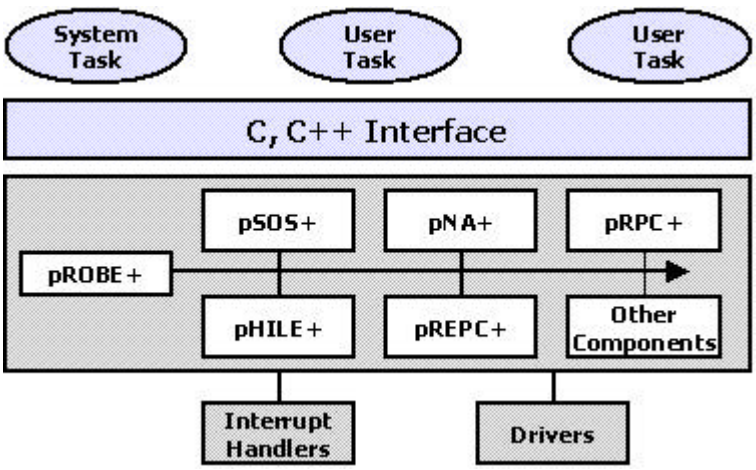

그림 2 pSOSystem 아키텍처

2.2.2 QNX

QNX는 QNX Software Systems Ltd.에서 만든 운영체계로 마이크로 커널과 그 상위에서 시스템 서비스를 하는 여러 프로세스들로 이루어진 클라이언트-서버 구조를 가진다[13, 15]. QNX의 마이크로 커널은 스케줄링, 프로세스 디스패칭, 기본 인터럽트 핸들링, IPC 라우팅과 같은 핵심 서비스들만을 가지고 있으며 그외의 서비스는 유저모드 프로세스들이 담당하고 있다. 이 프로세스들은 파일 시스템 매니저, 프로세스 매니저, 디바이스 매니저, 네트웍 매니저등의 서버로 구성되어 있으며 클라이언트 개념의 애플리케이션 프로세스로부터의 서비스 요청을 받아들여 동작하게 되어있다. 그리고 마이크로 커널과 시스템 서비스 서버들, 애플리케이션 프로그램들은 서로 다른 CPU모드에서 동작하게 되어있어 구조적으로 안정성이 보장되고 있다. QNX는 메시지 기반의 운영체계이며 IPC도 메시지를 통해 이루어지고 애플리케이션과 시스템 서비스 프로세스간의 통신 역시 메시지를 통해 이루어진다. 이런 메시지 패싱 메커니즘은 네트웍 메시징으로 확장되며 이를 이용해 QNX는 완전한 fault-tolerant, 분산 운영체계로 검증되고 있다. QNX는 여러 상용 실시간 운영체계로서는 드물게 실제 프로세스 개념을 가지고 있다. 모든 프로세스는 자신의 메모리 영역을 가지고 있으며 이 때문에 애플리케이션에 대해 운영체계의 안정성을 확보하고 있다. 하지만 쓰레드의 개념은 가지고 있지 않다. QNX는 32 개의 priority 레벨을 가지고 있으며 프로세스의 수는 2000 개로 제한되어 있다. 그리고 prioritized FIFO, prioritized round-robin, adaptive 세 종류의 스케줄링 정책을 가진다. QNX는 메모리 보호를 위해 반드시 MMU를 필요로 하며 4KB의 물리적 페이지 사이즈를 가지고 가상메모리를 지원한다. 인터럽트 처리는 인터럽트 핸들러를 인스톨하는 형태로 처리되지만 이를 인스톨한 프로세스의 컨텍스트에서 다루어진다. 인터럽트 핸들링과 타이머 관련 옵션 라이브러리가 많이 제공된다. 네트워킹은 기본적으로 TCP/IP스택이 제공되며 제공되는 프로토콜이 그리 다양하지 못하다. 그림 3 은 QNX의 아키텍

처를 보여준다.

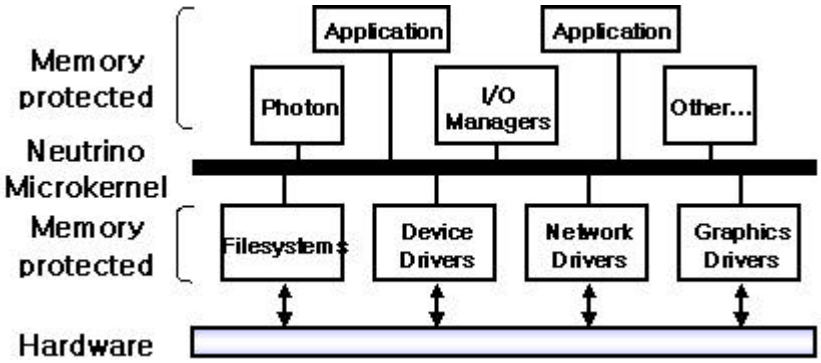

그림 3 QNX 아키텍처

2.2.3 VRTX

VRTX는 Mentor Graphics에서 만든 실시간 운영체계로 극히 제약된 리소스를 가진 간단한 시스템을 위한 VRTXoc (on-chip)와 메모리 보호를 필요로 하는 복잡한 시스템을 위한 VRTXsa (scalable architecture) 두종류의 커널을 가지고 있다[16]. 커널상의 시스템 서비스가 모듈화 되어 있기 때문에 커널 크기를 필요에 맞게 줄일 수 있으며 옵션으로 POSIX/ANSI 호환 라이브러리를 제공해 프로그램의 호환성을 높일 수 있고 MS-DOS호환 파일 시스템을 제공한다. 기본적으로 TCP/IP 스택을 제공하며 유닉스와 유사한 소켓 인터페이스를 가지고 있다. SNMP와 TMN/CMIP 툴킷이 옵션으로 제공된다. 네트워킹, I/O, 파일 시스템은 스트림 기반으로 통합되어 있다. MMU를 제공하는 프로세서에 대해 MMU를 지원하므로 시스템 안정성의 향상을 기대할 수 있다. 그리고 멀티 태스킹 디버깅을 지원하는 XRAY 디버거가 제공된다. XRAY는 다수의 쓰레드를 동시에 추적하여 실행상태

를 디스플레이 할 수 있고 서로 다른 종류의 프로세서들에서 돌아가는 쓰레드들도 동시에 디버깅이 가능하다. 또한 여러 쓰레드간의 디버깅 명령을 동기화 할 수 있어 통신이나 메시지 패싱 같은 복잡한 문제에 대한 대응을 할 수 있고 결과적으로 멀티 프로세서 시스템도 지원가능하다. 그림 4는 VRTX의 아키텍처를 보여준다.

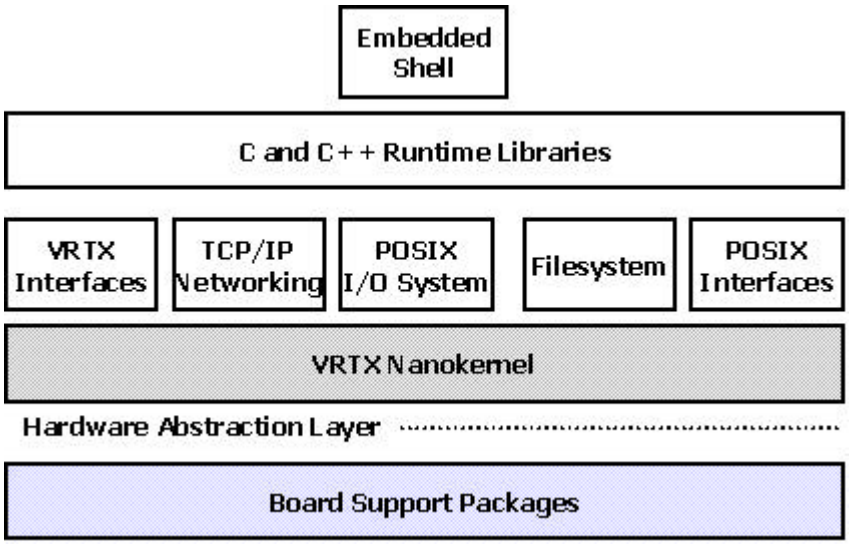

그림 4 VRTX 아키텍처

2.2.4 VxWorks

VxWorks는 처음에 WindRiver Systems, Inc.에서 VRTX와 pSOSystem을 위한 개발환경과 네트웍 환경을 제공하기 위해 만들어졌었다. 이런 시스템 밑에 후에 WindRiver Systems, Inc.가 자신들의 마이크로 커널을 구현하여 만든것이다[13, 17]. VxWorks도 마이크로 커널 기반의 클라이언트-서버 구조의 운영체계로서 VxWorks의 마이크로 커널은 wind 마이크

로 커널이라 불리는 것으로 멀티 태스킹, 스케줄링, 태스크간 동기화/통신, 메모리 관리 기능이 포함되어 있는 실시간 커널이다. 이 이외의 다른 기능들은 서비스 프로세스들로 구현되어 있다. 모든 프로세스들은 커널과 같은 레벨의 privilege를 가지고 모든 프로세스와 커널이 하나의 어드레스 영역을 공유하므로 시스템의 안정성이 약화될 수 있으나 옵션으로 VxVMI라는 보호 컴포넌트를 통해 커널과 사용자 프로세스간의 메모리 보호를 할 수 있다. Priority 레벨은 256 단계를 지원하며 최대 태스크 수는 메모리 크기에 의해 제한된다. 스케줄링 정책은 POSIX와 wind라는 proprietary 스케줄링 기법을 쓴다. MMU는 VxVMI 컴포넌트에 의해 지원되며 물리적 페이지 사이즈는 기본 8KB이며 VxVMI 컴포넌트를 통해 가상 메모리를 지원한다. 인터럽트 핸들링은 해당 핸들러 컨텍스트에서 이루어진다. 옵션 라이브러리는 태스크 관리, 메모리 관리, 타이머 관련이 많이 존재한다. VxWorks는 네트웍 관련 각종 프로토콜과 자바 VM, 각종 서버들, 네트웍 하드웨어 디바이스 드라이버들, 웹 브라우저들과 시스템 개발환경을 함께 제공하는 Tornado라는 제품을 따로 가지고 있다. 이 패키지는 Tcl기반으로 구현되어 있다. 이 이외에 시스템을 시뮬레이트해 볼 수 있는 유닉스 기반 툴인 VxSim이 있다. VxWorks는 다중 프로세서를 지원하지만 멀티 프로세서간 통신에 있어 하드웨어 아키텍처의 영향을 많이 받는다. 예를 들어 프로세서간 메시지 전달은 공유된 메모리 영역이 있을때만 가능하며 이런 기능또한 VxMP라는 추가 구입 옵션 컴포넌트로서 존재한다. 소켓 기반의 통신을 지원하므로 소켓을 통해 멀티 프로세서간 통신을 해도 되지만 소켓은 자료구조를 지원하지 못하는 등의 문제로 인해 메시지 패싱에 비해 단점이 많다. 또한 TCP/IP스택을 거쳐야 하므로 성능에도 문제가 될 수 있다. 그림 5는 VxWorks의 아키텍처를 보여준다.

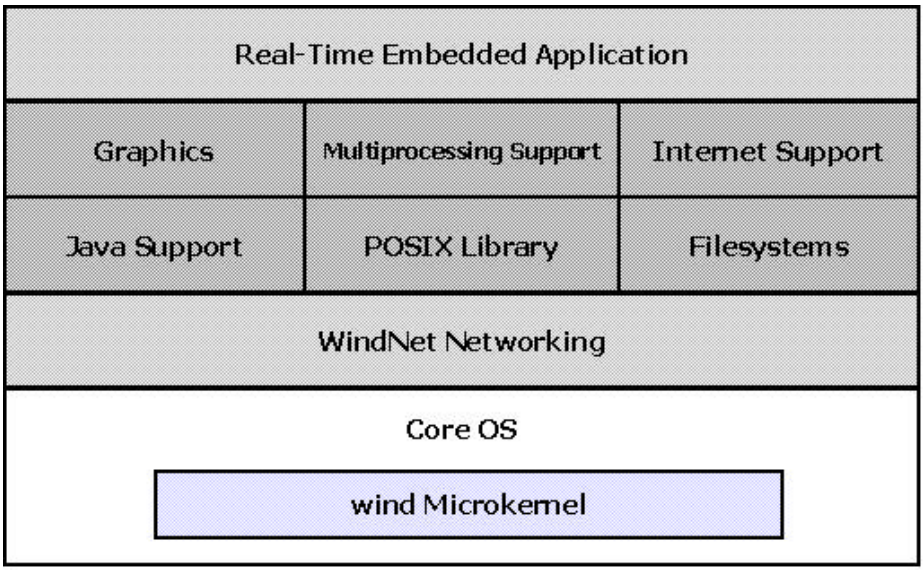

그림 5 VxWorks 아키텍처

2.2.5 Nucleus PLUS

Nucleus는 Accelerated Technology Inc. 에서 제작한 실시간 운영체계 제품군으로서 그중에서 네트워킹 디바이스 같이 복잡한 애플리케이션에 걸맞은 커널이 Nucleus PLUS이다[18]. 이 커널내에는 네트웍 기능이 들어있지 않으므로 Nucleus NET이라는 TCP/IP스택을 따로 구입해야 네트워킹 디바이스에 쓸 수 있다. Nucleus PLUS는 대부분의 코드가 ANSI C로 짜여져있기 때문에 포팅이 용이하며 태스크간 통신, 태스크 동기화, 메모리 관리, 타이머 관리등의 기능과 다수의 디바이스 드라이버를 제공하여 개발의 편의성을 도모한다. Nucleus NET은 내부에 TCP/IP와 그위의 수많은 프로토콜을 소스레벨에서 제공하며 telnet, ftp, web등의 애플리케이션을 제공한다. Nucleus NET의 TCP/IP스택은 그 크기가 60KB 정도이며 유닉스 소켓 인터페이스와 비슷한 인터페이스를 제공한다. Nucleus PLUS은 한 서비스 내에서 데이터끼리 각자의 메모리 영역을 침

범하지 않도록 태스크마다 자신들이 가지고 있는 메모리 포인터들의 리스트를 관리하여 메모리 보호를 하게 된다. 하지만 아키텍처 자체가 하나의 어드레스 스페이스에서 모든 프로세스가 CPU의 수퍼바이저 모드에서 돌아가므로 안정성에 문제가 있다. 그러나 이 방법은 태스크 스위칭이나 함수 콜에 있어 오버헤드가 적어진다는 장점은 있다. Nucleus PLUS는 메일 박스, 큐, 파이프의 태스크간 통신 메커니즘을 가지며 인터럽트 처리는 애플리케이션 컨텍스트에서 이루어진다. MMU는 아직 지원하지 못하며 따라서 가상 메모리를 지원하지 않는다.

2.2.6 OS-9

OS-9 은 Microware의 실시간 임베디드 운영체계이다[19]. OS-9 은 운영체계가 모듈단위로 나뉘어져 있으며 각 모듈들은 헤더, 코드와 데이터, CRC 체크섬으로 이루어져 있어 모듈의 안정성과 높은 보안성을 제공한다. 그리고 각 모듈은 런타임에 로딩할 수 있기 때문에 특정 기능을 커널에 추가한다거나 모듈 개발시에 많은 시간을 절약할 수 있다. OS-9 의 프로세스 모델은 유저레벨과 시스템 레벨의 프로세스를 분리하는 선점형 멀티 태스킹 모델로서 모듈의 CRC를 이용해 에러를 체크하기 때문에 높은 안정성을 기대할 수 있고 시스템 오류의 빠른 발견과 오류로부터의 회복을 위한 환경을 제공한다. OS-9 은 각각의 프로세스가 각자의 보호된 메모리 영역에서 돌아가므로 애플리케이션의 오류가 시스템 전체에 영향을 끼치지 않는다. OS-9 은 유닉스와 흡사한 C/C++ 인터페이스와 소켓을 제공하며 HTML을 임베디드 애플리케이션에 사용하기 위한 API도 제공한다. OS-9 은 I/O의 확장성을 위해 하드웨어 디바이스 드라이버 역할을 하는 I/O 매니저 모듈과 하드웨어 독립적인 파일 매니저 모듈을 통해 모든 디바이스를 파일로 맵핑하는 방법을 쓰고 있다. SoftStax라는 I/O 프레임 워크를 통해 네트워킹 기능을 OS-9 에 부여할 수 있으며 여러 표준 프로토콜이 구현되어 있고 모듈형태의 구조로

인해 새로운 스택을 추가하기 쉬우며 네트워킹 애플리케이션의 필요에 따라 프로토콜 스택을 런타임에 교체할 수 있다. 그림 6 은 OS-9 의 아키텍처를 보여준다.

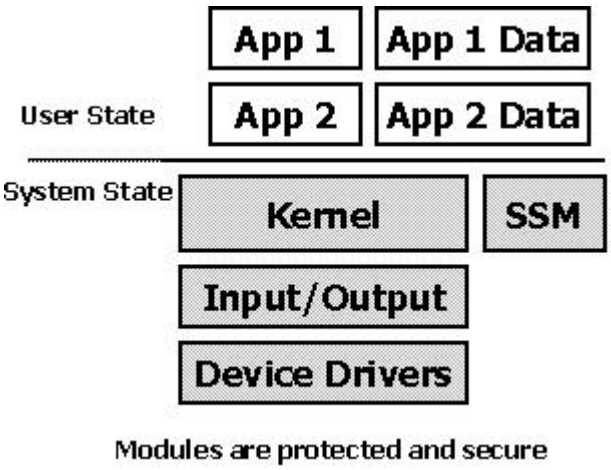

그림 6 OS-9 아키텍처

2.3 자체 개발 운영체계

Cisco, 3Com등의 거대 네트워킹 디바이스 하드웨어 회사에서는 자신들의 운영체계를 직접 만들어 쓰고 있다. 이런 운영체계중에서 가장 많이 보급되어 있는 운영체계는 Cisco의 IOS[20] 로서 하드웨어 자체의 높은 점유율과 네트워킹 디바이스이외의 종류에는 포팅되어있지 않으므로 네트워킹 디바이스에 가장 특화된 운영체계라고 볼 수 있다. 이외에 3Com의 Enterprise OS[21] 가 많이 알려져 있다. 하지만 모든 회사들이 독점적으로 운영체계를 소유하고 있기 때문에 운영체계 내부의 특성이나 구조, 성능등에 대한 자료를 구하기가 매우 힘들다.

2.4 내장형 운영체계 들의 종합 비교

지금까지 살펴봤던 교육용, 상용 네트워킹 디바이스용 운영체계의 특징들을 분류하여 네개의 표로 나타내어 보았다.

표 1 은 내장형 운영체계의 지원 하드웨어와 언어 지원에 대해 나타내고 있다.

Name	Supported CPUs	Supported Languages
Xinu	x86, SPARC, i960, 68k, PowerPC	Assembly, C, C++
uC/OS – I, II	Most	Assembly, C, C++
eCOS	ARM, MIPS, SPARC, PowerPC	Assembly, C, C++
pSOS+	68k, ARM, M Core, ColdFire, i960, MIPS, PowerPC, SH, SPARC, x86	Assembly, C, C++, Java, Fortran, ADA
QNX	MIPS, PowerPC, x86	Assembly, C, C++
VRTX	68k, ARM, PowerPC, x86, M Core	Assembly, C, C++, Java
VxWorks	PowerPC, 68k, ColdFire, M Core, x86, i960, ARM, MIPS, SH, SPARC	Assembly, C, C++, Java, Fortran, ADA
Nucleus PLUS	68k, ARM, M Core, ColdFire, i960, MIPS, PowerPC, SH, x86	Assembly, C, C++, Java
OS-9	68k, ARM, PowerPC, SH, x86	Assembly, C, C++, Java

표 1 내장형 운영체계의 지원 하드웨어와 지원 언어 비교

표 2는 내장형 운영체계들의 자원 요구량과 스케줄링 방법, MMU 지원에 대해 나타내고 있다.

Name	Req. ROM (min)	Req. RAM (min)	Multitasking Strategy	MMU support
Xinu	Unknown	Unknown	Fixed Priority	No
uC/OS – I, II	4 ~ 8KB	3KB	Fixed Priority	No
eCOS	Unknown	Unknown	Unknown	No
pSOS+	15KB	5KB	Round Robin, Time Slice, Cooperative,Rate Monotonic,Proprietary	Yes
QNX	32 ~ 64KB	8KB	Round Robin, Time Slice, Fixed Priority, Proprietary,	Yes
VRTX	Unknown	Unknown	Round Robin, Time Slice, Fixed Priority	Yes
VxWorks	Unknown	Unknown	Round Robin, Fixed Priority	Yes
Nucleus PLUS	3 ~ 45KB	2KB	Time Slice, Cooperative, Prorietary	Yes
OS-9	128 ~ 6000KB	128KB	Round Robin, Time Slice, Rate Monotonic, Cooperative, Proprietary	Yes

표 2 내장형 운영체계의 자원 요구량과 스케줄링 방법, MMU 지원 비교

표 3 은 내장형 운영체계들의 모듈성, 다중 프로세서 지원, POSIX 호환, 컴파일러 지원에 대해 나타내고 있다.

Name	Modular link	Multiprocessor support	POSIX compliant	Compilers supported
Xinu	No	No	No	All ANSI compatible compilers
uC/OS – I, II	Yes	No	No	All ANSI compatible compilers
eCOS	Unknown	No	Yes	GNU gcc
pSOS+	Yes	Yes	Yes	Diab Data, pRISM+
QNX	No	Yes	Yes	GNU gcc, Watcom C/C++
VRTX	Yes	No	Yes	Microtec C/C++,
VxWorks	Yes	Yes	Yes	GNU gcc, Diab Data, Tornado
Nucleus PLUS	Yes	Yes	No	GNU, ARM, Diab Data, SDS, MetaWare, Watcom, Borland, Microsoft, SSI
OS-9	Yes	No	No	ucc, GNU gcc

표 3 내장형 운영체계의 모듈성, 다중 프로세서 지원, POSIX 호환성, 컴파일러 지원 비교

표 4 는 내장형 운영체계들의 디버거 지원, 보급량, 가격에 대해 나타낸다.

Name	Debuggers supported	Approx. installed base	Price
Xinu	Post-Mortem Debugger	Unknown	Free
uC/OS – I, II	Unknown	2,000	$50
eCOS	GNU GDB	Unknown	Free
pSOS+	Cross compiler and source level debugger included	7,000	$16,500
QNX	CodeWarrior IDE for QNX Neutrino, GNU GDB, Watcom C/C++	One million	$795
VRTX	XRAY debugger	50,000	$1,000
VxWorks	GNU GDB, Tornado,	Unknown - Very Large	$90,000 ~$200,000
Nucleus PLUS	Nucleus UDB, SDS, EST VisionCLICK, MetaWare, SSI, CodeWarrior	2,300	$7,495 ~ $12,495
OS-9	Hawk, CodeWarrior	Three million	Unknown

표 4 내장형 운영체계의 디버거 지원, 보급량, 가격 비교

3. 네트워킹 디바이스를 위한 내장형 운영체계의 요구조건

지금까지 현장에서 사용되고 있는 수많은 운영체계들의 특징과 장,단점들을 비교해 보았고 이제 이 장에서는 현재의 네트워킹 디바이스 필요 자원량과 그 구현 추세에 대해 알아본후 2 장에서의 결과를 이용해 네트워킹 디바이스를 위한 운영체계의 필요조건으로는 어떤 것이 있을 수 있는가를 알아보고 이를 정리해 네트워킹 디바이스를 위한 임베디드 리눅스의 디자인에 이용할 수 있도록 한다.

3.1 네트워킹 디바이스의 자원 요구량

현재의 네트워킹 디바이스들은 과거의 임베디드 시스템과는 그 질적 측면에서 많은 차이를 보이고 있다. 우선 프로세서의 성능이 비약적으로 향상되고 그 가격도 저렴해지면서 네트워킹 디바이스에 들어가는 대부분의 프로세서가 32bit 아키텍처에 MMU를 가지고 있고 수십 MIPS 이상의 속도를 가지게 되었고 플래쉬나 DRAM과 같은 메모리 디바이스들도 마찬가지로 그 용량은 크게 확대되면서도 가격은 점점 하락하게 되어 현재는 대부분의 네트워킹 디바이스가 수MB ~ 수십MB 단위의 메모리를 가지고 있다[22]. 현재의 임베디드 컴퓨팅 디바이스의 보유 메모리양을 애플리케이션 레벨별로 나눠보면 표 5 와 같이 나누어진다[23].

	Tiny	Midrange	High-end	Embedded PC	Embedded Server
Flash	100K ~ 4MB	2 ~ 8MB	8 ~ 32MB	16 ~ 64MB	64+ MB or disk
RAM	500K ~ 2MB	2 ~ 4MB	4 ~ 16MB	16+ MB	64+ MB

표 5 네트워킹 디바이스의 자원 요구량

3.2 소프트웨어의 크기 혹은 품질

대부분의 상용 운영체계는 100KB 이하의 램으로도 시스템을 운용할 수 있다고 되어있는데 이는 표 5 와는 많은 차이를 보인다. 하지만 이런 리소스 요구량은 네트워킹 디바이스에 필요한 TCP/IP스택, 애플리케이션들과 이들을 위한 시스템 라이브러리들을 생각하지 않은 것이고 실제로 애플리케이션을 탑재하여 쓸만큼의 운영체계 구성을 하려면 커널크기가 보통 128 ~ 256KB정도로 커지는 것이 대부분이며 네트워킹 디바이스로서 역할을 하기 위해서 TCP/IP스택과 소켓라이브러리를 추가하면 여기다 100 ~ 200KB정도 더 늘어나게 된다. 결국 상용 운영체계를 사용하더라도 네트워킹 디바이스는 적어도 800KB정도의 저장공간이 필요하고 해당 애플리케이션에 따라 그 이상의 런타임 메모리가 필요하게 된다[24].

따라서 이런 상황과 맞물려 실제로 네트워킹 디바이스용 운영체계가 얼마나 작은 리소스를 필요로 하는가는 중요하지 않은 일이 되어가고 있다. 현재의 추세는 거의 PC급에 맞먹는 하드웨어 스펙을 가진 네트워킹 디바이스들이 줄이어 나오고 있는 상황이고 실제로 PC아키텍처위에서 구현된 라우터, 캐쉬 엔진 들도 시장에 다량 출시되어 있기 때문이다. 이런 현상들의 원인은 앞에서 언급한대로 전체적인 하드웨어 가격에서 램, 플래쉬 메모리등의 용량 변화에 따른 비용 변화가 그리 큰 부분을 차지하지 못하고 있는데 기인하며 따라서 과거처럼 제한된 리소스에 얽매여 소프트웨어의 크기를 줄이는데 많은 노력을 기울이는 것보다는 소프트웨어의 질적 향상에 그 시간을 투자하는 것이 훨씬 효율적인 방법이 되어가고 있다고 볼 수 있다.

특히 네트워킹 애플리케이션들은 데이터 수집, 기계장비 컨트롤 등의 과거 임베디드 애플리케이션들과는 달리 TCP/IP스택과 같은 다중 프로세스 지향적인 복잡한 서브시스템이 고도로 안정화 되어있어야 하고 네트웍 자체가 고속화되어 가면서 처리해야하는 데이터양도 매우

크며 여러 서비스 리퀘스트를 동시에 안정적으로 처리할 수 있어야 한다. 따라서 신뢰성있는 프로그램 구조를 필요로 하며 이는 최소한으로 제한된 크기라는 임베디드 프로그래밍의 중요한 필요조건보다 네트워킹 애플리케이션에서 있어서 훨씬 그 의미가 크다고 할 수 있다.

3.3 실시간성

2 장에서 살펴보았던 운영체계들중에 Xinu를 제외하고는 모든 운영체계가 실시간 운영체계를 표방하고 있다. IEEE의 정의에 따르면 "실시간 시스템이란 기능적 정확함뿐만 아니라 반응시간의 정확함을 함께 가지고 있는 시스템"이다[25]. 즉, 실시간 시스템은 어떤 시간안에 주어진 일을 반드시 해내야 하는 시스템이고 이런 실시간 시스템은 hard real-time과 soft real-time으로 다시 나뉜다. Hard real-time은 최대한의 주어진 worst-case time내에서 주어진 일을 해내는 것을 의미하며 soft real-time은 시간 제약을 요구하기는 하나 그 제약이 시스템에 치명적이지 않은 것을 의미한다. 즉 soft real-time은 어떤 시스템이 주어진 일을 주어진 시간내에서 해내는 것이 바람직하긴 하나 만약 실패한다고 해도 시스템 전체에는 문제될 것이 없음을 뜻하며 결과적으로 평균적인 시간제약을 가진다.

2 장에서 보았던 대부분의 운영체계들은 hard real-time 특성을 가지고 있으며 Xinu는 실시간 운영체계는 아니다. 여기서 운영체계가 실시간이라 함은 대부분 그 운영체계 자체의 모든 태스크들이 시간 제약내에 주어진 일을 해낸다는 것보다는 그 운영체계 위에서 돌아가는 애플리케이션들의 실시간성을 보장한다는 의미이다. 그렇다면 네트워킹 장비에 들어가는 애플리케이션들은 실시간성을 가져야 하는 것들이 많은지 알아볼 필요가 있다.

네트워킹 디바이스는 크게 네트웍의 연결점으로서의 역할을 하는

라우터, 스위치등의 장비와 네트웍의 터미널 역할을 하는 장비들로 구분할 수 있는데 이런 장비들은 real-time을 필요로 하지 않는 경우가 대부분이다. 예를 들어 VoIP장비의 경우 주어진 시간안에 들어온 패킷을 처리하지 못한다고 해서 시스템의 운영에 문제가 되지는 않고 다만 성능에 문제가 생길 뿐이며 이런 경우의 해결책은 시스템의 애플리케이션에 실시간성을 부여하는 것보다는 시스템 하드웨어, 소프트웨어의 성능을 향상시키는 것이 훨씬 효율적인 대응책이 된다.

대부분의 내장형 운영체계들이 실시간성을 가지고 있는 이유는 과거에 이런 운영체계를 필요로 했던 애플리케이션들이 인공위성이나 로켓, 군사용 장비등과 같이 시간제약에 맞게 일을 끝내지 못하면 시스템 자체가 실패하게 되는 극단적으로 시간제약이 심한 것들이기 때문이었다[25]. 네트워킹 디바이스용 애플리케이션은 이런 범주에 속하지 않으며 실제로 네트웍 애플리케이션을 실시간 운영체계로 만드는 엔지니어들도 운영체계의 실시간성을 이용하지는 않는다는 사실과 Xinu와 같은 운영체계가 네트워킹 디바이스를 잘 운용하고 있는것에서 알 수 있듯이 실시간성이 반드시 필요한 것은 아니라 할 수 있다.

3.4 요구조건

이제 2 장에서 살펴보았던 운영체계들의 특징들과 위에서 알아본 몇가지 사실들을 전제로 실제 네트워킹 디바이스를 위한 운영체계는 어떤 요구조건들을 가져야 하는지 나열한다.

3.4.1 새로운 프로세서로의 이식성

내장형 운영체계를 설계하는 데 있어 가장 중요한 것은 portability 이다. 날이 갈수록 더욱더 빠르고 성능이 강화된 프로세서들이 등장하면서 네트워킹 디바이스에 사용할 수 있는 프로세서의 수도 점점 증가

하고 있고 따라서 얼마나 쉽고 빠르게 새로운 프로세서를 지원할 수 있느냐 하는 것이 중요한 요구조건중의 하나이다. 현재 대부분의 내장형 운영체계는 많은 부분이 C로 만들어져 있다. 초기에 내장형 운영체계는 애플리케이션에 빠른 응답성과 최소한의 오버헤드만을 주도록 대부분이 어셈블리어로 만들어졌고 이런 구조의 운영체계를 다른 아키텍처에 포팅하는 것은 결코 쉬운일이 아니었다. 또한 성능좋고 저렴한 RISC프로세서들이 나오면서 RISC프로세서들의 내장형 장치에 대한 채택이 늘어나게 되었는데 RISC의 특성상 그 명령어 수가 많지않아 어셈블리로 직접 운영체계를 만드는 것은 무리한 일이 되어갔다. 그래서 최적화가 잘 되어있는 코드를 생산하는 컴파일러가 중요해졌고 그에 따른 컴파일러 기술의 발달이 뒤따르면서 결과적으로 RISC의 운영체계는 대부분 C로 만들어지게 되었다. Wind River. Inc,의 VxWorks의 RISC 프로세서로의 포트인 경우 98%의 코드가 CISC 포트인 경우 80%정도의 코드가 C로 이루어져 있다[26]. 따라서 portability를 결정하는 중요한 요소중 하나는 C와 같은 아키텍처 중립적인 언어가 얼마나 사용되었나 하는 것이다.

3.4.2 다양한 응용 영역에 쓰일 수 있는 스케일러빌러티

네트워킹 디바이스의 종류는 갈수록 다양해지고 있다. 기능적, 물리적으로 크게는 Gbps급 백본용 라우터에서부터 작게는 성냥갑 크기의 웹서버에 이르기까지 하드웨어 리소스의 크기가 가변적이기 때문에 운영체계 자체의 스케일러빌러티가 아주 중요하다. 스케일러빌러티를 결정하는 요소로는 크게 사용되는 언어와 하드웨어 아키텍처, 그리고 운영체계의 modularity를 들 수 있다. 예를 들어 많은 부분이 C로 짜여진 커널은 그 크기가 어셈블리어로 이루어진 커널보다 크기도 크고 메모리도 많이 요구한다. 이 부분은 포터빌러티와 상반되는 현상이라 할 수 있다. 프로세서가 RISC인 경우 RISC의 명령어 구조로 인해 그 차이는

더욱 커지며 따라서 CISC 아키텍처에서 같은 커널이라도 더욱 스케일러빌러티를 높일 수 있다[26]. pSOSystem의 경우 CISC아키텍처인 모토롤라 68k계열 포팅이 가장 스케일러빌러티가 높다고 알려져 있다. 또한 pSOSystem의 경우 구조적으로 세마포어의 유무, 개수 같은 상세한 운영체계 특성까지도 결정할 수 있으며 런타임에 필요한 ROM과 RAM의 크기를 표시해주는 구성 툴까지 개발 툴에 포함되어 있다. 하지만 pSOSystem과 같이 극도로 세밀하게 스케일러빌러티를 제공하는 운영체계는 몇종류 많지는 않으며 대부분의 운영체계는 기본적인 커널위에 네트웍, 파일 시스템, 디바이스 드라이버 등의 기능 단위로 모듈화하여 스케일러빌러티를 제공한다. 예를 들어 VxWorks의 경우 커널위에 네트웍, I/O서브시스템 등을 라이브러리 형태로 제공하여 선택한 기능을 커널, 애플리케이션과 함께 링크하여 운영체계를 만들게 되어 있다.

3.4.3 다중 프로세서 지원

Gbps급 라우터와 같이 높은 퍼포먼스를 요구하는 하이엔드급 애플리케이션에서는 운영체계 자체의 스케일러빌러티와는 다른 특성을 요구한다. 이런 애플리케이션의 경우는 하드웨어 처리성능이 일차적으로 요구되며 따라서 하드웨어 리소스의 크기에 있어서는 전혀 제약조건을 가질 수 없게 된다. 이 종류의 하드웨어는 대부분 하나의 프로세서로는 처리를 할 수 없기 때문에 결국 멀티 프로세서 시스템으로 나갈 수 밖에 없으며 따라서 이를 지원하는 소프트웨어 환경이 필요하게 된다. 이상적으로는 운영체계가 애플리케이션 레벨에서 다수의 프로세서를 하나의 프로세서로 인식할 수 있도록 투명함을 제공해 줘야 하며 애플리케이션간 통신도 태스크가 어느 프로세서에서 돌아가든 상관없이 공통된 통신 수단을 제공해줘야 한다. VxWorks와 pSOSystem의 경우 이런 이상적인 멀티 프로세싱 환경을 제공한다. VxWorks 환경인 VxMP라는 제품은 공유 메모리를 이용하는 tightly-coupled 멀티 프로세서 시

스템이며 총 20 개 까지의 프로세서를 지원한다. pSOSystem 환경인 pSOS+m은 각자의 프로세서가 로컬 메모리에서 애플리케이션을 실행하는 loosely-coupled 멀티 프로세서 시스템이며 이들 애플리케이션간의 통신은 LAN, bus, 공유 메모리등 pSOSystem에서 지원하는 어떤 디바이스로도 가능하다.

3.4.4 확장 서비스 지원

스케일러빌러티, 멀티 프로세서 지원과 더불어 그 운영체계가 기본적인 시스템 서비스외에 얼마나 많은 종류와 수의 확장가능한 서비스를 가지는 가도 아주 중요한 요구조건이라 할 수 있다. 일반적으로 내장형 운영체계를 구성하는 가장 간단한 요소로는 멀티 태스킹 스케줄러, 메모리 관리 서비스, 인터럽트 핸들러, 그리고 세마포어, 메일박스, 큐와 같은 프로세스간 통신, 동기화 메커니즘을 들 수 있다. 반면에 실제 네트워킹 디바이스를 구성하는 운영체계의 경우 여러 애플리케이션의 필요에 맞게 다양한 종류의 I/O 라이브러리와 파일 매니저, 디바이스 드라이버등을 운영체계 회사나 써드파티에서 제공하고 있다. 예를 들어 OS-9, pSOSystem, VxWorks의 경우 모두 TCP/IP와 NFS에 대한 지원을 하고 있으며 특히 OS-9 의 경우 모든 I/O 서브시스템을 파일로 접근하는 특징과 맞물려 FIFO, character device, random and sequential block device, file system, internet device, NFS, X-Windows, ISDN등의 다양한 기능을 확장할 수 있도록 되어있다.

3.4.5 다양한 애플리케이션 프로그램의 존재

현재 많은 상용 운영체계 회사들은 운영체계 자체의 성능 뿐만 아니라 그 운영체계에서 돌릴 수 있는 특정 목적에 특화된 애플리케이션을 서비스하는 것을 하나의 세일즈 포인트로 삼고 있다. 이런 애플리

케이션으로는 웹 서버나 SNMP 에이전트 등을 들 수 있는데 이런 운영체계 회사로부터 제공되는 vertical application들은 개발자들로 하여금 많은 시간을 절약할 수 있게 해준다. 이는 애플리케이션의 수가 운영체계의 운명을 지배하고 있는 PC계의 현상을 생각해보면 쉽게 알 수 있다. 예를 들어 pSOSystem의 경우 SNMP 에이전트를 구현해놓고 판매하고 있으며 OS-9 은 kiosk 시스템들을 위한 mpeg decoder를 판매하고 있다. 이렇게 어떤 특정한 목적의 네트워킹 디바이스를 제작하는데 있어 쉽게 구해서 목적에 맞게 고쳐서 쓸 수 있는 애플리케이션이 존재하느냐 아니냐 라는 것은 개발자에게 있어 운영체계를 선택하는데 큰 영향을 준다.

3.4.6 표준/POSIX 호환성

네트워킹 디바이스를 개발하는데 있어 만약 그 애플리케이션 개발 환경이 기존의 표준을 따르지 않는다면 개발자들이 이에 숙달되는데 많은 시간이 필요해지며 따라서 제품 개발의 지연을 가져올 수 있다. 하지만 만약 애플리케이션 개발 환경이 기존의 표준을 따르고 있다면 다른 운영체계에서 개발을 해오던 사람들도 쉽게 새로운 환경에 익숙해질 수 있다.

이런 표준으로 많이 쓰이고 있는 것으로 POSIX(Portable Operating System Interface)라는 것이 있다[27]. POSIX는 다양한 유닉스 운영체계 사이의 호환성을 위해 표준 인터페이스를 제정한 것으로 그 중에서도 C언어 레벨에서의 애플리케이션 프로그램 인터페이스의 표준을 정한 것이 POSIX.1 이며 이는 유닉스 운영체계 사이에서 C 소스레벨에서의 호환성을 제공하는 것을 목표로 하고 있다. 쉽게 얘기해 운영체계가 POSIX호환이라면 유닉스에서 개발한 C프로그램은 아무 문제없이 그 운영체계에서 컴파일되어 돌아갈 수 있다는 의미이다. 하지만 실제 상용 운영체계중에서 POSIX 호환을 표방하고 있는 것은 많지 않은데 그

이유는 POSIX가 실시간 시스템을 위한 이상적인 표준은 아니라는 데 있다. POSIX는 최소 latency를 목표로 정해진 부분이 많아서 hard real-time성을 가지기 힘들어 실시간 운영체계를 표방하는 대부분의 상용 운영체계에서 잘 채택되지 못하고 있다[28]. POSIX 1003.1 과 POSIX 1003.1B 두 섹션이 실시간에 대한 인터페이스를 제공하고 있지만 아직 완벽하지는 않으며 일본에서 제정한 TRON(The Real-time Operating system Nucleus)[12] 이라는 표준을 따르고 있는 운영체계가 많다. 네트워킹 디바이스를 개발하는데 있어 실시간성이 거의 필요가 없다는 것은 앞에서 알아본 바와 같고 따라서 내장형 운영체계를 설계하는데 있어 POSIX호환을 따른다면 애플리케이션 개발의 용이성이 더욱 증대될 수 있다.

3.4.7 다양한 프로그래밍 언어 지원

앞에서 본 바와 같이 내장형 애플리케이션을 개발하는데 있어 대부분의 운영체계 사이에 정해진 API표준이 없는 상태이기 때문에 개발중인 애플리케이션이 높은 포터빌러티를 가지기 위해서는 개발자가 전적으로 표준 언어에 의존하는 수밖에 없다. 이런 상황에서 C는 내장형 시스템에서 가장 보편적인 언어 표준이며 이 외에 C++도 오브젝트의 재사용성을 장점으로 조금씩 쓰이고 있다. 하지만 C/C++의 높은 인기와 언어적 장점에도 불구하고 특정 영역에서는 다른 언어의 사용이 필요한 경우가 있다. 예를 들어 과거에 존재했던 수많은 과학 계산 알고리즘들은 Fortran으로 짜여진 경우가 대부분이며 군사영역에서 만들어져 사용되었던 많은 알고리즘들은 ADA 소스로 되어있는 것이 많다. 만약 운영체계가 제공하는 개발환경에서 특정 언어를 지원하지 않는다면 그 언어를 사용하는 소스는 지원되는 언어로 변환해야 하므로 이는 개발자에게 큰 부담으로 작용한다. 순수하게 네트웍에 관련된 네트워킹 디바이스에서는 이런 분야의 코드를 사용할 일이 없겠지만 만약 이런 분

야의 장비에 네트웍 기능을 실장하여 네트웍 터미널로서의 역할이 필요해진다면 그 필요성은 분명해진다. 언어지원에 대해서 대부분의 상용 운영체계 회사들은 두가지의 정책을 사용하는데 첫번째는 C와 같이 필수적이고 많이 쓰이는 컴파일러는 자신들의 컴파일러를 만들어 제공하고 두번째로 그다지 많이 쓰이지 않는 언어의 개발도구는 써드파티에 맡기는 정책을 쓴다. 예를 들어 OS-9 을 만드는 Microware의 경우 운영체계 자체의 특이성과 맞물려 Ultra C라는 OS-9 에 극도로 최적화된 C 컴파일러만을 제공하며 그 외에는 써드파티도 인정하지 않고 있다. 이런 경우 언급한 바와 같이 특정 분야에는 적용되기 힘든 운영체계가 될 수밖에 없다. pSOSystem이나 VxWorks의 경우는 자체에서 컴파일러를 만들지 않고 언어 지원은 써드파티에 맡기며 이로 인해 그들의 운영체계에서는 C, C++, Fortran, ADA등의 언어가 지원된다. 결국 지원되는 언어의 다양성은 그 운영체계를 선택할 수 있는 네트워킹 디바이스의 종류를 늘릴 수 있게 한다.

3.4.8 개발 환경 지원

개발환경을 구성하는 것은 컴파일러만이 아니다. 컴파일러, 디버거, 에디터 그리고 소스코드 컨트롤 시스템과 같이 개발자를 도와주는 여러 프로그램들이 모여 개발환경이 구성되며 이 환경이 얼마나 잘 갖춰져 있느냐에 따라 커널의 포팅에서부터 애플리케이션까지 그 개발기간에 큰 영향을 주기 때문에 네트워킹 디바이스에서 뿐만 아니라 모든 내장형 장비의 개발에 있어 개발환경이 차지하는 비중은 아주 크다. 예를 들어 커널의 포팅과 안정화, 그리고 애플리케이션 초기 개발 단계에서는 디버거가 아주 중요하며 소스 레벨 디버거의 존재는 필수이다. 그리고 커널과 애플리케이션이 정상동작에 들어가는 단계에서부터는 성능 최적화를 위해서 프로파일러, 메모리 누수 탐지 프로그램등이 중요해진다. 거기다 소스 코드 컨트롤 시스템은 개발단계 전체에 영향을 끼

치며 공동작업시 아주 큰 효과를 발휘한다. 이와 같은 개발환경도 컴파일러의 비중이 상당하기 때문에 상용 운영체계의 언어 지원 정책에 크게 영향을 받는다. 따라서 써드파티에 개발환경 전체를 맡기는 경우도 있고 스스로 개발환경을 만드는 경우도 있으며 혼합된 경우도 존재한다. 어떤 경우든 네트워킹 디바이스용 운영체계를 설계하는데 있어 잘 구성된 개발환경의 지원은 필수이다.

3.4.9 유연한 라이센스 정책과 저렴한 가격

개발자들이 운영체계를 선택하는데 있어 그 운영체계의 라이센스 비는 중요한 가늠자가 된다. 교육용이나 공개 운영체계를 쓰는 겨우 이는 크게 문제가 되지 않으나 상용 운영체계의 경우 운영체계의 라이센스 비는 개발 코스트에 그대로 반영이 되며 특히 판매되는 장비마다 부과되는 런타임 라이센스 비는 대량 생산을 목적으로 개발되는 장비를 위한 운영체계 선택에 많은 영향을 준다. 런타임 라이센스비는 회사마다 경쟁적으로 그 가격을 낮추고는 있지만 주로 판매회사와의 협상을 통해 결정되며 대량으로 생산되는 장비의 경우 대당 1 달러 이하로 떨어지는 경우도 있고 소량 생산 장비인 경우 대당 1000 달러 이상으로 치솟기도 한다. 거기다 일부 유명한 상용 운영체계의 경우 라이센스 계약을 한번만 하는 것으로 계속 그 운영체계의 사용권한을 갖는 것이 아니라 프로젝트 기반으로 계약을 하여 개발할 제품이 달라지면 다시 한번 더 라이센스를 얻어야 한다. 또한 사용자 수 혹은 프로그램이 인스톨되는 머신의 수에 따라 라이센스 비가 달라지기도 한다. 이렇듯 개발시에 운영체계의 라이센스 비는 개발을 방해하는 요소가 되기도 하며 일부 프로젝트에서 교육용이나 공개 운영체계가 선택되는 주 요인이기도 하다. 따라서 네트워킹 디바이스용 운영체계의 디자인에 있어 개발자에게 라이센스에 대한 부담을 줄여주는 것이 하나의 필요조건이 된다.

4. 네트워킹 디바이스를 위한 임베디드 리눅스의 디자인

이 장에서는 임베디드 리눅스가 네트워킹 디바이스를 위한 운영체계로서 어떻게 설계하면 3 장의 요구조건들을 충족시킬 수 있는지 리눅스의 구조적 분석과 병행해 알아보고 커널내부에서 서브시스템들간의 의존성을 파악하여 내장형 운영체계로서 최적화할 수 있는 곳이 있는지 분석해본다. 커널과 함께 임베디드 리눅스 시스템은 어떤 요소들로 이루어져야 네트워킹 디바이스에 적합한지 설명하고 설계요소들에 대해 파악하고 이를 이용해 시스템을 구성한다.

4.1 요구조건의 만족

리눅스가 3 장에서 제시했던 네트워킹 디바이스를 위한 운영체계의 요구조건에 대해 만족할 수 있는지에 대해 알아본다.

4.1.1 새로운 프로세서로의 이식성

리눅스는 이미 수많은 프로세서로 포팅이 되어있다. 현재 커널이 소스 형태로 배포되면서 기본적으로 지원하는 프로세서 아키텍처로는 Alpha, ARM, i386, Motorola 68k, PowerPC, MIPS, S390, SPARC, SPARC64 가 있으며 이 이외에 스탠다드 커널 소스에 포함되어 있지는 않지만 개발버전 소스에 들어있거나 독립 소그룹에서 포팅중인 프로젝트로는 Motorola 의 DragonBall, ColdFire, MC68360, MC68EN302 등의 마이크로 컨트롤러, 커뮤니케이션 컨트롤러와 ETRAX, Intel i960, Hitachi SH 등의 프로세서들이 있다[29]. 거기다 현재 막 개발이 끝난 Crusoe같은 프로세서나 새로 등장할 Intel IA-64 같은 경우는 개발단계에서부터 리눅스 지원을 염두에 두고 운영체계 포팅과 프로세서 개발이 동시에 진행될 정도이다[30, 31]. 따라서 리눅스는 네트워킹 디바이스에 사용할 수 있는 대부

분의 프로세서에 이미 포팅이 되어있으며 결국 포터빌러티에 대한 요구조건은 거의 충족된다고 할 수 있다. 하지만 위에서 언급한 프로세서 이외의 새로운 프로세서에 포팅을 원할 경우 포터빌러티는 아키텍처에 무관한 코드가 얼마나 사용되었나를 통해 알아볼 수 있다. 리눅스 커널 2.2.17 버전의 경우 전체 코드는 86,597,632 bytes 이며 이중 아키텍처 의존적인 코드는 18,829,312 bytes 이다. 결국 전체 코드에서 아키텍처 의존적인 코드는 약 21.74%인데 이렇게 아키텍처 의존적인 코드가 많은 이유는 리눅스 커널 소스가 위에서 언급한 모든 아키텍처를 다 지원하기 위해서 아키텍처마다 필요한 모든 코드를 다 포함하고 있기 때문이다. 따라서 하나의 아키텍처만을 선택하여 그 크기를 비교하면 PowerPC 프로세서의 경우 아키텍처 의존적인 코드는 2,531,328 bytes 이며 전체에서 차지하는 비중은 2.92%에 지나지 않는다. 따라서 나머지 아키텍처도 크게 이 범위를 벗어나지 않는다고 하면 한 아키텍처로의 포팅시에 아키텍처 중립적인 코드는 전체 코드의 97%이상이며 이는 VxWorks등 상용 운영체계에 결코 뒤지지않는 포터빌러티를 가졌다는 의미이다.

4.1.2 다양한 응용 영역에 쓰일 수 있는 스케일러빌러티

리눅스는 원천적으로 modular 한 성격을 가지므로 필요없는 기능에 대해서는 간단하게 제공되는 configuration tool을 이용해 커널에서 그 기능을 삭제해낼 수 있다. 이런 모듈중에 아키텍처 무관한 모듈로는 크게 다음과 같은 것들이 있다.

- Loadable Module Support
- Networking Support
- Floppy, IDE, SCSI, RAM disk Support
- File System Support

✍Kernel Debugger Support

✍Etc.

각각의 모듈내에는 수많은 종류의 디바이스 드라이버, 네트웍 프로토콜, 파일 시스템 등을 선택할 수 있게 되어 있으며 그 외에도 아키텍처 의존적인 선택사항들도 많이 있다. i386 아키텍처의 경우 모든 선택사항을 제거하고 코어 리눅스 커널의 사이즈만을 구해보면 2.2.17 버전의 경우 402,183 bytes 로서 약 393KB의 크기를 가진다. 이는 다른 상용 운영체계의 최소 크기에 비해 상당히 크나 최소 크기 상태에서의 상용 운영체계들과 리눅스는 특성과 기능이 완전히 다르다. 게다가 1장에서 살펴본 바와 같이 네트워킹 디바이스의 리소스는 점점 그 제약이 줄어들고 있기 때문에 이만한 크기는 결코 문제가 될 수 없는 경우가 대부분이다. 실제 이 정도의 크기이면 자동화 업계에서 내장형 운영체계로 많이 쓰이고 있는 DOS보다 약간 큰 정도이다.

4.1.3 다중 프로세서 지원

리눅스는 Alpha, i386, PowerPC, S390, SPARC, SPARC64 아키텍처에서 하나의 메모리와 운영체계 서비스를 공유하는 tightly-coupled된 symmetric multiprocessing을 지원한다. i386 아키텍처에서 16개 까지의 프로세서를 지원하며 이는 tightly-coupled 시스템에서 일반적인 최대치이며 VxMP에 비해서는 약간 모자라나 다른 운영체계에 비해 모자라지 않는다. 위에 언급한 이외의 아키텍처에서는 프로세서 자체가 멀티 프로세싱을 지원하지 않는 경우가 대부분이며 따라서 리눅스는 네트워킹 디바이스에 사용될 수 있는 프로세서 중에 멀티 프로세싱이 가능한 종류를 거의 다 지원한다고 할 수 있다. 멀티 프로세서 시스템에서 프로세스 간 통신은 IPC를 통해 이루어지며 이는 System V IPC를 지원한다.

4.1.4 확장 서비스 지원

리눅스는 오픈 소스 운영체계이고 그 특성 때문에 사용자가 곧 개발자로 이어질 수 있는 특징을 가지고 있다. 따라서 리눅스가 가지는 시스템 서비스의 대부분은 사용자가 필요에 의해 만들어 낸 것이 대부분이며 리눅스 사용자와 사용환경이 그 수를 헤아리기 힘들 정도로 큰 것을 보면 리눅스에서 확장 가능한 서비스가 얼마나 많은지 짐작해 볼 수 있다. 멀티 태스킹 스케줄러, 메모리 관리, 인터럽트 핸들러, IPC 등은 기본적으로 내장되어 있으며 이 위에 Floppy, IDE, SCSI, IrDA, 사운드 컨트롤러, 그래픽 컨트롤러, 비디오 컨트롤러, serial, parallel port 컨트롤러 등의 디바이스 드라이버들이 포함되어 있고 파일 시스템에 있어서도 DOS, HFS, ISO9660, Minix, NTFS, HPFS, ROMFS, ext2, UFS, NFS등의 대부분을 제공하며 네트웍의 경우에도 Ethernet, PPP, SLIP, PLIP, X.21, X.25, V.24, V.35, V.36, Wireless LAN, Token ring, Fibre Channel, ARCnet, Frame relay등의 현존하는 대부분의 L2 프로토콜 디바이스 드라이버를 제공하고 IP, IPX, Appletalk등의 L3 프로토콜을 커널 자체적으로 지원한다. 거기다 이런 요소들이 모두 모듈화 되어있어 그 확장성이 방대하다 할 수 있다.

4.1.5 다양한 애플리케이션 프로그램의 존재

현재 리눅스에서 개발이 진행중인 오픈소스 프로젝트는 웹페이지를 가진 것만 해도 12,000 개가 넘는다[32, 33]. 그 중에서 네트워킹 디바이스에 직접적으로 쓰일 수 있는 애플리케이션으로는 1,000 개 이상의 FTP, telnet, HTTP, SMTP, POP3, DNS, NNTP, SNMP등의 TCP/IP프로토콜을 이용하는 클라이언트와 서버 프로젝트들, 100 개 이상의 각종 멀티미디어 플레이어, 100 개 이상의 DB, 1,000 개 이상의 웹 애플리케이션들, 500 개 이상의 각종 라이브러리, 200 개 이상의 자바 패키지, 10 여 종류의 ORB등 네트워킹 디바이스를 구성할 수 있는 거의 모든 영역의 애플리

케이션들이 소스와 함께 공개되어 있으므로 개발에 크나큰 도움을 줄 수 있다. 실제로 교육용, 상용 내장형 운영체계에서 개발되는 많은 애플리케이션들이 이런 오픈 소스 애플리케이션의 소스를 이용해 만들어지는 경우가 많다. 따라서 리눅스는 vertical application의 지원이 가장 잘 되어있는 운영체계라 할 수 있다.

4.1.6 표준/POSIX 호환성

리눅스는 유닉스 운영체계이며 POSIX 호환성을 목표로 만들어졌다[34]. 따라서 POSIX에 대한 호환성은 다른 어떤 내장형 운영체계에 비해 좋다고 할 수 있다.

4.1.7 다양한 프로그래밍 언어 지원

리눅스에서 개발이 가능한 언어로는 C, C++, Fortran, ADA, 각 아키텍처의 assembly, Java, Forth, Prolog, Dylan, Eiffel, Modula-2 등 대부분의 언어를 지원하며 이 이외에도 특수용도로 사용되는 100 종류 이상의 언어들의 컴파일러와 인터프리터가 오픈소스로서 존재한다. 또한 새로운 언어가 등장할 때 그 언어를 위한 개발도구는 대부분 리눅스용으로 나오고 있는 실정이다. 따라서 언어지원에 있어 리눅스에 비교될 수 있는 운영체계는 없다고 할 수 있다.

4.1.8 개발 환경 지원

리눅스는 개발자를 위한 운영체계라고도 불린다. 운영체계 자체가 오픈소스이기 때문만 아니라 개발환경이 가장 잘 되어있는 운영체계이기 때문이다. 앞서 보았던 수많은 종류의 사용가능 언어들의 컴파일러, 인터프리터와 함께 gdb등의 디버깅 툴, RCS등의 소스 코드 컨트롤 툴,

gprof과 같은 프로파일링 툴, 각종 언어를 위한 메모리 누수, 성능 테스트 툴등이 존재한다. 이런 여러가지 툴들이 리눅스에서만 돌아가는 것이 아니라 윈도우, OS/2, IRIX, HP-UX, Solaris, AIX등의 여러 다른 운영체계에서도 사용가능하도록 포팅이 되어있다. 예를 들어 리눅스 애플리케이션뿐만 아니라 커널 자체를 윈도우에서 컴파일하여 그 커널을 이용해 부팅할 수 있으며 서로 다른 아키텍처를 위한 바이너리를 생성하고 디버깅, 프로파일링 할 수 있도록 되어있다. 즉, 아키텍처에 상관없이 애플리케이션을 개발할 수 있도록 해주며 이는 네트워킹 디바이스를 위한 애플리케이션 개발에 큰 도움이 된다. 이런 장점에도 불구하고 이런 대부분의 툴이 텍스트 기반이라는 점이 단점이 될 수 있으나 이런 텍스트 기반 툴들을 좀더 사용하기 쉽도록 GUI환경으로 만들어주는 수많은 wrapper들이 존재하며 여러 툴들을 하나로 모아 IDE를 구성해주는 애플리케이션들도 있다. 이중에는 상용 IDE에 버금가는 성능을 가진 오픈소스 IDE들이 많다.

4.1.9 유연한 라이센스 정책과 저렴한 가격

일반적으로 소프트웨어와 같은 지적 재산권을 가진 저작물에 대한 라이센스는 목적상 다음 몇가지 종류로 권한을 구분한다[35].

1. 소프트웨어를 사용할 수 있는 권한
2. 소프트웨어를 수정할 수 있는 권한
3. 소프트웨어를 복제할 수 있는 권한
4. 소프트웨어를 배포할 수 있는 권한
5. 위의 권한을 타인에게 승인할 수 있는 권한
6. 소프트웨어로 이득을 얻을 수 있는 권한
7. 위의 권한을 타인에게 제한할 수 있는 권한

8. 위의 권한을 양도하거나 라이센스 할 수 있는 권한

이런 권한에 대해 Open Source Initiative는 오픈 소스 소프트웨어가 어떠해야 하는지를 규정해 놓았는데 이에 따르면 오픈 소스의 자격을 얻기 위해서는 다음과 같은 사항을 라이센스에 넣어야 한다[36].

1. 무제한의 복사권이 허용되어야 한다.
2. 무제한의 사용권이 허용되어야 한다.
3. 무제한의 개인적인 용도를 위한 수정이 허용되어야 한다.

여기에 덧붙여 오픈 소스 가이드라인은 수정된 바이너리의 재배포를 금지한다. 이러한 특성을 가지고 오픈 소스 소프트웨어의 자격을 가지게 하도록 인정된 라이센스로는 GPL, LGPL, BSD, MIT, Artistic 등이 있으며 각각의 세부적인 사항은 약간씩 다르다[35, 37]. 리눅스는 이러한 룰을 따르는 GPL에 의해 라이센스된 오픈소스 운영체계이며 따라서 사용에 있어 어떠한 제한도 받지 않는다. 이는 위에서 언급한 수많은 오픈소스 애플리케이션들에도 마찬가지로 해당된다. 따라서 개발자에게 있어 가장 부담이 없는 운영체계라 할 수 있다.

4.2 리눅스 서브 시스템간의 자원 의존성

지금까지 리눅스가 네트워킹 디바이스를 위한 운영체계로서 3 장에서 언급한 요구조건을 모두 만족하는 것을 보았다. 이제 리눅스가 어떻게 하면 3 장의 요구조건을 충족시킬 수 있는 특성을 그대로 유지하면서 내장형 운영체계로서 적합하도록 그 사이즈를 줄일 수 있는지 리눅스 서브시스템의 구조와 그 의존성을 통해 파악하여 본다.

리눅스의 개념적 구조는 그림 7과 같이 나타낼 수 있다[38].

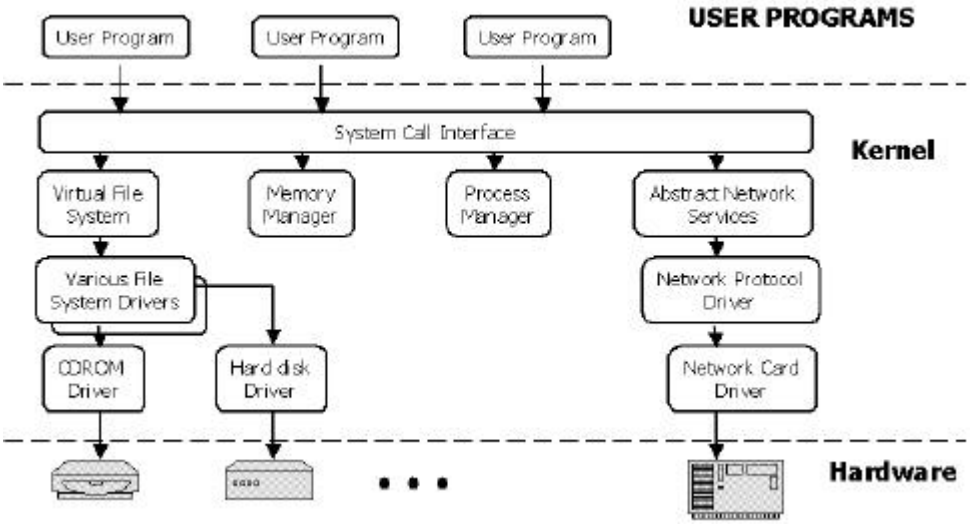

그림 7 리눅스 커널의 개념적 구조

운영체계는 크게 monolithic, layered, virtual machine, client-server 의 네가지 아키텍처로 구분해 볼 수 있는데[45] 리눅스는 그림 7 로 볼 때 layered 구조로 생각할 수 있으나 실제로는 각 모듈간에 데이터를 공유할 수 있는 시스템으로 되어 있기 때문에 monolithic 구조이다. 따라서 운영체계의 속도에 있어서 크게 이득을 가질 수 있지만 컴포넌트간의 의존성으로 인해 코어 커널의 기능을 제거하는 일은 상대적으로 쉬운 일이 아니다. 따라서 리눅스 코어 커널의 내부 구조를 살펴보고 각 subsystem간의 의존성을 알아보면 큰 도움이 될 수 있다.

리눅스는 크게 다음 5 가지의 개념적 subsystem들로 나누어 진다 [39].

✍✍ Process Scheduler

✍✍ Memory Manager

✍✍ Virtual File System

✍✍ Inter-Process Communication

✍✍ Network Interface

이제 각 subsystem의 구조적 의존성에 대해서 알아보고 사이즈를 줄이기 위해 어떤 부분을 제거할 수 있는지 알아보자.

4.2.1 프로세스 스케줄러

프로세스 스케줄러는 그림 8과 같은 구조를 가진다.

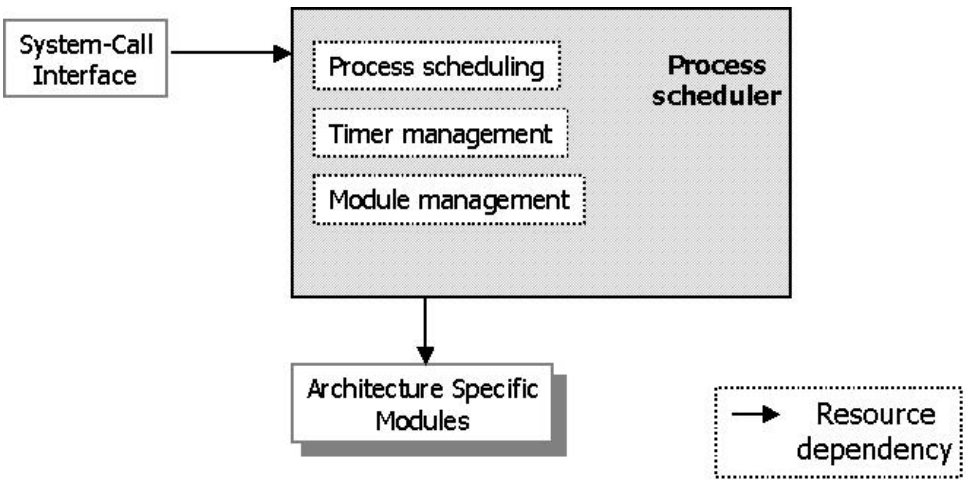

그림 8 프로세스 스케줄러 구조

이 구조에서 알 수 있듯이 프로세서 스케줄러는 아키텍처 의존적인 코드와 단단히 묶여있으며 내부적으로 타이머 관리, 모듈 관리와 함께 프로세스 스케줄러의 역할을 하고 있다. 여기서 타이머는 시스템 전체에 영향을 미치는 요소이므로 제거할 수 없고 모듈 관리 기능은 삭제할 수 있는 기능이다.

4.2.2 메모리 매니저

메모리 관리 모듈의 구조는 그림 9와 같다.

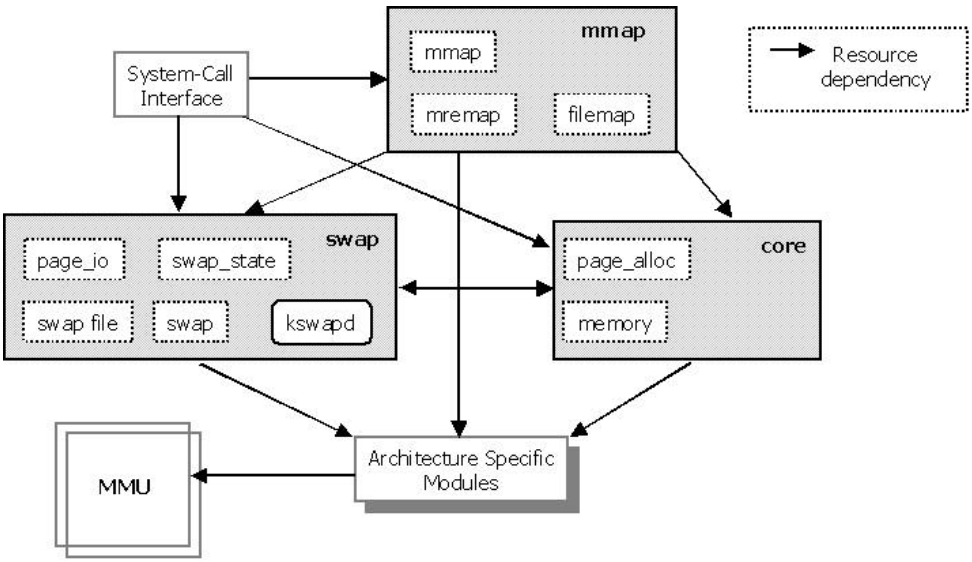

그림 9 메모리 매니저 구조

실제로 이 구조에서 시스템 콜과 관련된 부분은 vertical application 들과의 호환성, POSIX호환성 유지를 위해 제거할 수 없는 부분이다. 따라서 결국 MMU와 아키텍처 의존적인 코드를 목표로 할 수 있다. 실제로 MMU가 없는 마이크로 컨트롤러에 포팅된 리눅스 커널이 많이 있고 이런 경우 MMU에 의존적인 코드를 아예 삭제하고 메모리 관리를 소프트웨어로 하게 된다[29]. 이 경우 안정성에 문제가 생길 수 있고 성능에도 영향을 끼치게 된다. 하지만 커널 코드의 사이즈는 크게 줄일 수 있다.

4.2.3 가상 파일 시스템

가상 파일 시스템의 구조는 그림 10과 같다.

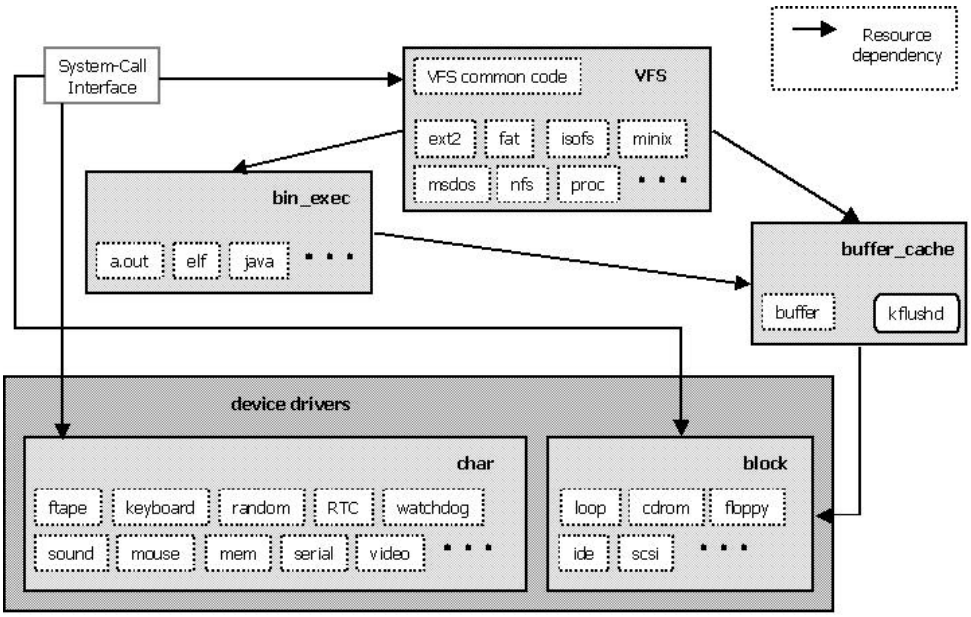

그림 10 가상 파일 시스템 구조

가상 파일 시스템에서 내부 모듈중 exended service 지원을 조금 줄이면 삭제할 수 있는 모듈이 상당히 많다. 예를 들어 가상 파일 시스템 서브시스템의 여러 파일 시스템 지원 코드는 필요없는 부분에 대해 모두 삭제할 수 있으며 bin_exec 서브시스템에서도 커널내에서 지원하지 않을 바이너리 포맷에 대한 코드를 삭제할 수 있다. 디바이스 드라이버 부분에서도 필요한 드라이버 모듈만을 사용하면 된다. 만약 파일 시스템 자체를 쓰지 않고 여러 애플리케이션들을 커널 내부에 포함시킨다면 가상 파일 시스템 이하의 관련 서브 시스템 자체를 삭제할 수도 있다. 물론 POSIX호환성은 그만큼 감소하게 된다.

4.2.4 Inter-Process Communication

IPC 서브시스템의 구조는 그림 11과 같다.

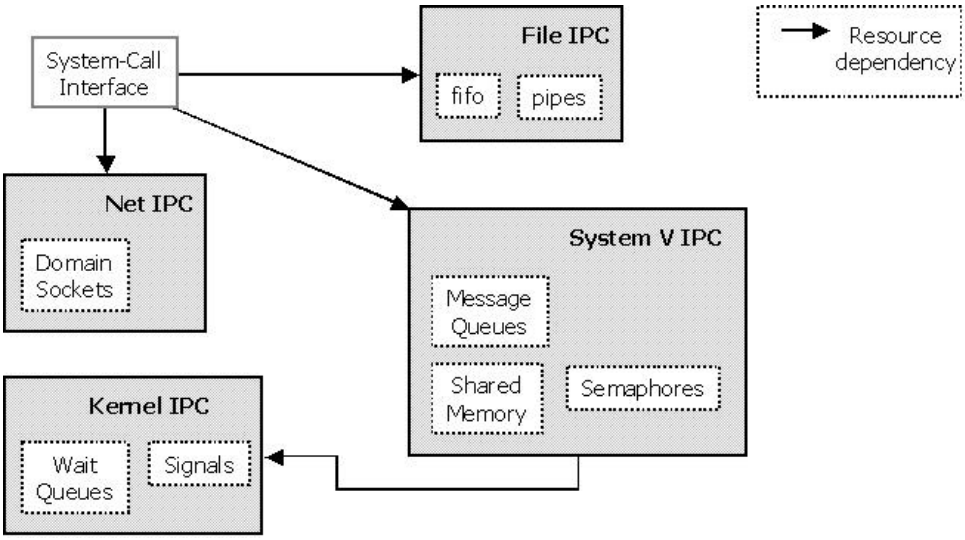

그림 11 IPC 서브시스템 구조

실제로 File IPC를 사용하는 application들은 그다지 많지 않다. 그리고 직접 애플리케이션을 만들때도 File IPC는 사용하지 않으면 되므로 삭제할 수 있다. 그 이외에 System V IPC나 Net IPC는 많은 vertical application들이 이를 사용하므로 삭제하기가 힘드며 Kernel IPC도 커널 내부의 모듈들이 통신시 사용하므로 삭제 하기 힘들다. 하지만 System V IPC중에서 사용하지 않는 IPC기법이 있다면 삭제할 수 있다.

4.2.5 네트웍 인터페이스

네트웍 인터페이스 서브시스템의 구조는 그림 12와 같다.

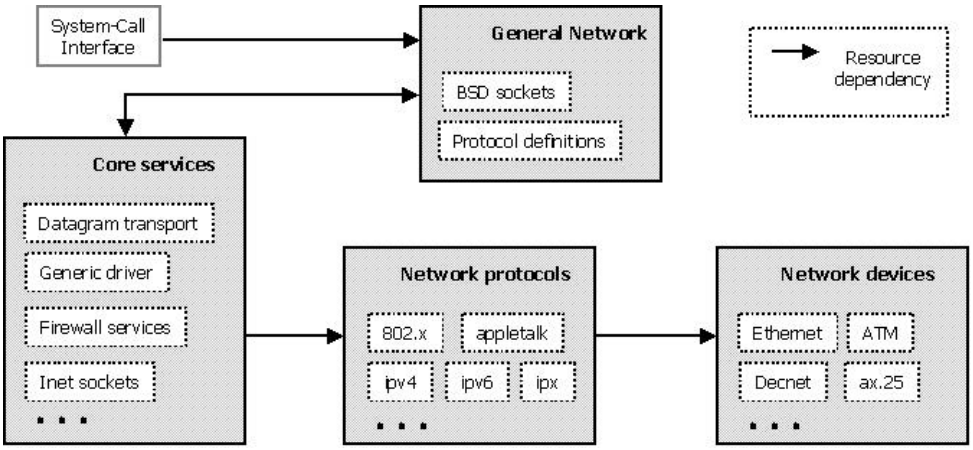

그림 12 네트웍 서브시스템 구조

이 부분은 실제로 가장 커스터마이즈하기 쉬운 부분이면서 리눅스의 확장성을 보여주는 부분이기도 하다. 각 서브시스템의 내부 모듈은 필요에 따라 모두 제거할 수 있는 부분이고 따라서 리눅스 커널의 크기를 줄일 수 있는 중요한 요소가 된다.

4.3 내장형 운영체계 구성요소로서의 리눅스 시스템 컴포넌트

여기서는 커널이외에 리눅스 시스템을 구성하는데 필요한 중요한 요소들인 라이브러리, 루트 파일시스템등이 어떻게 구성이 되어야 내장형 운영체계 시스템으로서 적절한 지 생각해보고 여러가지 디자인 선택요소들에 대해 알아본후 최적의 설계를 한다.

4.3.1 런타임 라이브러리

리눅스 시스템에 있어 커널 다음으로 중요한 구성요소가 있다면

그 중의 하나로 시스템 런타임 라이브러리를 들 수 있다. 애플리케이션을 컴파일할 때 반드시 필요하며 또한 vertical application들을 포팅하는데 있어 어떤 라이브러리를 선택하느냐에 따라 포팅의 난이도가 결정된다. 만약 애플리케이션들을 스스로 만들고 모두 라이브러리와 정적링킹을 한다면 런타임 라이브러리는 필요하지 않으나 애플리케이션이 많아질수록 중복되는 코드의 크기가 늘어나 리소스의 낭비가 생기게 된다. 결국 이를 피하기 위해서는 런타임 라이브러리가 필수이며 따라서 리눅스 시스템이 내장형 운영체계로 쓰이기 위해서는 이의 선택이 아주 중요하다. 리눅스에서 시스템 런타임 라이브러리로 쓰일 수 있는 것들은 다음을 들 수 있다[40].

✍✍ glibc – GNU C Library

✍✍ sglibc – Small glibc

✍✍ libc5 – Older GNU C Library

✍✍ dietlibc – To create small statically linked apps

✍✍ Newlib – C library intended for use on embedded systems

glibc는 GNU에서 만든 C라이브러리이며 거의 모든 리눅스 시스템의 기본 런타임 라이브러리이다. 따라서 그 크기가 내장형 장비에는 적합하지 않지만 내부적인 디버깅 심볼들을 제거하면 내장형 장비에도 사용가능할 정도로 크기를 줄일 수 있다. 대부분의 오픈소스 애플리케이션들이 이 라이브러리를 이용해 만들어지므로 애플리케이션의 포팅시 아주 많은 장점을 가지고 있다. sglibc는 플로피 디스크로 이루어진 리눅스 배포판이나 내장형 장비에 사용하기 위해 glibc에서 많은 부분을 잘라낸 라이브러리로서 glibc에 대한 패치형태로 되어있다. libc5 는 glibc가 버전 2로 넘어가기 전의 버전명칭이다. 즉 glibc 2.0대가 나오기 전에 리눅스 시스템에서 표준으로 쓰였던 라이브러리며 대부분의 애플리케이션들이 이 라이브러리로 컴파일 되면서도 glibc보다 크기가 작다.

하지만 현재 나오는 애플리케이션들중 libc5 에서 컴파일 되지 않는것도 많다. dietlibc는 정적 링킹된 애플리케이션들의 크기를 줄이기 위한 목적에서 만들어진 라이브러리이다. Newlib은 레드햇에서 내장형 장비에서 사용될 수 있도록 여러 라이브러리 소스에서 크기가 작은 부분을 가져와 만든 라이브러리이다.

4.3.2 루트 파일 시스템

리눅스 시스템이 사용하는 리소스를 극도로 제한시키려 한다면 파일시스템을 사용하지 않을수도 있다. 즉, 커널 내부에 각종 애플리케이션들을 포함시켜 커널과 하나의 바이너리 이미지로 만든후 커널 쓰레드의 하나로 이들 애플리케이션들을 돌리는 방법이다. 하지만 이 방법은 애플리케이션을 커널 쓰레드로 변환해야 하는 어려움이 있고 디버깅도 커널 디버깅 레벨로 내려가므로 상당히 힘들게 된다. 반면에 루트 파일시스템을 가지게 되면 시스템 자체가 일반적인 PC용 리눅스 시스템과 그 구성이 비슷해지고 애플리케이션의 개발에 있어서 다른 아키텍처에서 시뮬레이션이 가능해지며 수많은 오픈 소스 애플리케이션들이 예상하는 시스템과 같아지므로 시스템의 구성이 훨씬 쉬워진다. 따라서 리눅스 시스템을 네트워킹 디바이스에 내장하더라도 루트 파일시스템을 가지는 것이 좋으며 결국 약간의 리소스 희생으로 더욱 큰 이득을 얻을 수 있다는 장점이 있다.

내장형 장비에서 파일시스템을 구성할 수 있으려면 PC의 하드디스크와 같은 저장장치가 필요하다. 이런 목적으로 쓰일 수 있는 것들로는 RAM, ROM, Flash등을 들 수 있으며 리눅스에서는 각각의 장비를 디스크처럼 사용할 수 있도록 하는 다양한 종류의 디바이스 드라이버가 나와있다.

리눅스에서 지원하는 파일시스템은 그 수가 20 종류 이상에 이르나 실제로 내장형 장비에 적합한 파일시스템과 그 구성방법으로는 다

음과 같은 것들이 있다[40].

✍✍ Network File System (NFS)

✍✍ Initial Ramdisk

✍✍ ROM File System (ROMFS)

✍✍ RAM File System (RAMFS)

✍✍ Compressed RAM File System (CRAMFS)

✍✍ Journaling Flash File System (JFFS)

NFS는 실제로 네트워킹 디바이스에 내장되기 위해서라기 보다는 시스템 구성 디버깅을 위해서 많이 쓰인다. 리눅스는 자체적으로 NFS root mount 기능이 있으므로 처음에 리눅스 커널만을 개발중인 시스템에 올리고 루트 파일시스템은 NFS로 마운트를 하여 시스템의 재부팅이 필요없이 원격으로 시스템을 구성해 나아갈 수 있다. 이 방법은 개발 시간을 단축시킬 수 있는 장점이 있다. Initial Ramdisk는 리눅스에서 지원하는 램디스크를 이용해 루트 파일시스템을 구성하는 방법이다. 램디스크는 물리적으로 램의 일부분을 디스크로 사용할 수 있는 방법이며 이때 생성된 디스크는 리눅스의 VFS인터페이스를 통해 마운트 되므로 일반적인 하드디스크와 다를 바 없이 보이게 된다. 이때 파일시스템은 어떤 종류이든 상관없다. 일반적으로 리눅스 시스템에서 가장 많이 쓰이는 ext2 파일시스템을 많이 쓴다. 램디스크는 전원 공급이 끊기면 내용을 보존할 수 없는 비영구적인 저장공간이므로 처음 부팅시에 영구적 저장공간에서 그 내용을 읽어오고 시스템을 멈출 때 저장이 필요한 내용은 따로 저장해야 된다는 오버헤드가 따른다. 하지만 램을 디스크로 사용하므로 속도면에서 가장 빠르다는 장점을 가지고 있다. ROM File System은 EEPROM과 같은 ROM을 위해서 만들어진 파일시스템이며 ROM을 디스크로서 사용하게 된다. 이 파일시스템은 ROM의 특성상 Read-Only로 쓰일 수 밖에 없지만 여러 파일시스템중에 가장 크기가 작

다. 실제로 일반적인 파일시스템중 가장 크기가 작은것중 하나인 Minix 파일시스템의 경우 그 드라이버 크기가 20KB이상이지만 ROMFS의 드라이버 크기는 4KB정도이다. RAM File System은 리눅스 커널 2.4 에서 램디스크에 가장 적합한 파일시스템으로 설계되었으며 성능도 최적화 되어있다. Compressed RAM File System은 RAMFS와 마찬가지로 램디스크에 최적화된 파일시스템으로서 파일시스템 자체가 압축된 상태라는 것이 특징이다. 즉, 파일 접근시 런타임에 파일시스템 드라이버 레벨에서 압축이 풀려 VFS인터페이스로 전해지는 구조로 이루어져 있으며 따라서 저장공간이 넓어지는 효과를 볼 수 있다. 하지만 Read-Only File System이라는 점과 RAMFS보다 속도가 떨어지는 단점이 있다. RAMFS와 CRAMFS 둘다 리눅스 커널 2.4 에 포함되어 있는 커널이지만 커널 2.2 로도 포팅되어 질 수 있다. Journaling Flash File System은 실제로 네트워킹 디바이스에 많이 쓰이는 플래쉬에 적합한 파일시스템을 목표로 만들어진 파일시스템이다. 그리고 Read-Write로 마운트 될 수 있으면서 Journaling 파일시스템이기 때문에 시스템이 예기치않게 다운되더라도 파일시스템의 내용이 파괴되지 않는다는 장점이 있다.

5. 구현

이 장에서는 앞에서 디자인했던 네트워킹 디바이스를 위한 내장형 리눅스를 실제로 구현하고 이를 상용 장비에 포팅하여 하나의 네트워킹 디바이스 시스템을 구현하는 것에 대해 다룬다. 구체적으로 구현할 장비가 목표로 하는 기능과 하드웨어, 소프트웨어적 특징을 먼저 알아보고 이에 따라 어떤 형태로 리눅스 커널을 구성하고 런타임 라이브러리, 파일시스템을 갖추었는지, 그리고 어떤 애플리케이션들을 이용하여 장비를 구현하였는지에 대해 알아볼 것이다.

5.1 구현 대상

리눅스를 내장형 운영체계로 하여 구현해볼 네트워킹 디바이스는 인터넷 공유를 할 수 있도록 해주는 장비이다. 인터넷 공유란 하나의 IP 어드레스를 이용해 여러대의 컴퓨터가 인터넷을 사용할 수 있도록 하는 기술로서 NAT(Network Address Translation)를 이용한다[41].

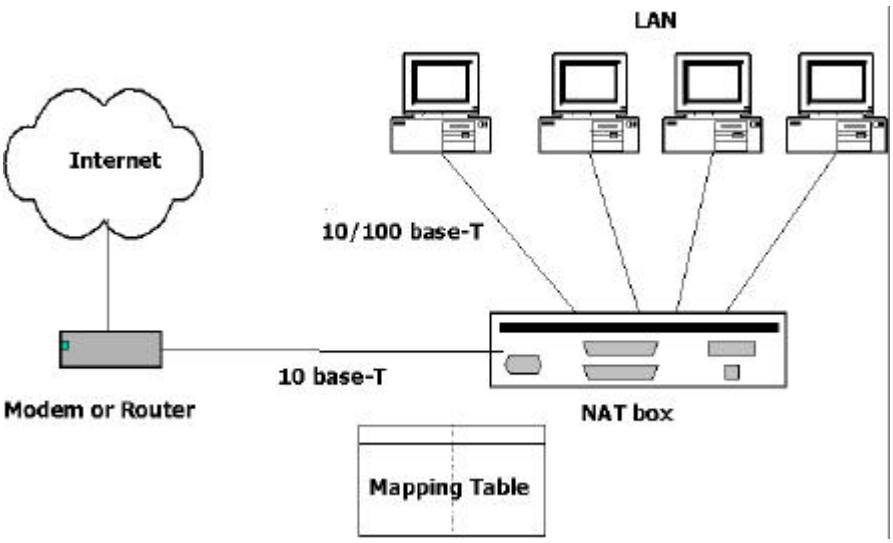

그림 13 Network Address Translation 구조

NAT는 RFC1631 에 정의되어 있으며 그 동작의 원리는 위 그림에서 나타낸 바와 같이 내부적으로 사설 IP를 할당받은 각 머신이 외부 인터넷에 접근하려 할 때 그 패킷의 소스 IP와 포트번호를 외부 인터넷에 연결된 인터페이스의 IP와 포트번호로 변경한 후 <소스 IP> : <소스 포트번호> : <데스티네이션 IP> : <데스티네이션 포트번호> 매칭 테이블을 저장하고 있다가 인터넷에서 들어오는 패킷에 대해 테이블과 들어맞는 경우 이의 데스티네이션 IP와 포트번호를 테이블에 저장된 정보에 따라 변환시켜 해당 컴퓨터로 되돌리게 된다. 기본적으로 이런 간단한 IP헤더 변환 조작으로 인터넷을 공유할 수 있게 되며 다음과 같은 장점을 얻을 수 있다[42].

✍✍ Solve A World Shortage of IP Addresses

✍✍ Satisfy Security Needs

✍✍ Ease and Flexibility of Network Administration

이런 장점 때문에 대부분의 방화벽, 라우터 장비에는 NAT기능이 들어있으며 NAT기능만을 내장한 가정용 라우터도 출시되어 있다.

지금부터 인터넷 공유 장비를 구현하는데 필요한 소프트웨어의 기능과 하드웨어에 대해 알아본다.

5.1.1 하드웨어의 특징

인터넷 공유 기능을 수행하기 위해서는 WAN쪽으로 하나의 이더넷 인터페이스, LAN쪽으로도 적어도 하나의 이더넷 인터페이스가 있어야 한다. 따라서 두개의 이더넷 컨트롤러가 필요하며 LAN쪽에 허브 기능을 제공하면 여러대의 컴퓨터를 이 하나의 장비로 연결할 수 있으므로 허브를 내장하도록 설계한다. 그리고 NAT기능은 모두 소프트웨어로 구현되므로 LAN쪽의 컴퓨터에 충분한 인터넷 속도를 보장하기 위해서는 빠른 속도의 CPU가 필요하게 된다. 이런 여러 요구조건에 적합하도

록 실제 하드웨어를 구성하여 다음과 같은 최종 하드웨어 스펙이 나오게 되었다.

- ✍✍ WAN interface : 10Base-T Ethernet 1 Port
- ✍✍ LAN interface : 10/100Base-T Ethernet 4 Ports, Auto-Sensing
- ✍✍ CPU : MPC850DE (have 2 Ethernet Controllers)
- ✍✍ Memory : 2MB Flash, 16MB RAM
- ✍✍ 2 MII Transceivers
- ✍✍ 1 Switched Repeater (for HUB function)

MPC850DE는 PowerPC계열의 프로세서이며 모토롤라에서 만든 프로세서이다. 모토롤라에서는 PowerPC코어와 커뮤니케이션 프로세서인 MC68360 QUICC의 통신 코어를 합하여 PowerQUICC이라는 이름으로 시장에서 내놓았으며 이 계열 프로세서로는 MPC860/850 시리즈가 있으며 MPC850 은 MPC860 의 저가형 모델이다. MPC850 시리즈 중 하나인 MPC850DE는 다음과 같은 특징을 가진다.

- ✍✍ 105MIPS at 80MHZ (maximum speed)
- ✍✍ 2KB instruction cache and 1KB data cache
- ✍✍ Instruction and data MMUs
- ✍✍ Two SCCs (Serial Communication Controller with Ethernet Support)
- ✍✍ Two SMCs (Serial Management Channels)
- ✍✍ Parallel interface port
- ✍✍ PCMCIA interface
- ✍✍ Background Debug Mode(BDM)[43] debug interface

✍✍ 3.3V operation

MPC860/850 시리즈와 이 외의 다른 PowerPC시리즈들은 수많은 상용 라우터에 쓰였으며 대표적인 네트워킹 장비로는 Cisco 4500 시리즈가 있다.

5.1.2 소프트웨어의 기능

기본적으로 NAT는 어드레스, 포트번호 맵핑을 통해 LAN쪽에서 보낸 패킷에 대한 WAN쪽의 응답만을 변환시켜 LAN으로 뿌려준다. 기존의 Telnet, HTTP, SMTP등의 well-known 포트들을 통하면서 정상적으로 소스 포트를 통해 연결하는 서비스들은 이런 방법만으로 충분히 통신이 가능하나 이런 방법을 쓰지않고 다른 방법으로 연결을 시도하는 인터넷 애플리케이션들이 현재 많기 때문에 이에 대한 대응이 필요하다. 또한 LAN쪽에서 서버 운영을 원할 경우, LAN쪽의 접근권한에 대한 관리가 필요할 경우들도 생각할 수 있다. 그리고 LAN쪽의 머신들에 대한 IP어드레스 할당 방법, WAN쪽의 연결 방법에 대해서도 생각해 두어야 한다. 따라서 여러 다른 상용 인터넷 공유 장비들의 기능들에 대해 뽑아서 소프트웨어적으로 필요한 기능들에 대해 나열하면 다음과 같다.

✍✍ NAT (Network Address Translation)

✍✍ Virtual Server

✍✍ Access Control

✍✍ Special Internet Application

✍✍ Exposed Host

✍✍ DNS Cache

✍✍ DHCP Client and Server

✍✍ Static Routing

✍✍ PPPoE (PPP over Ethernet)

Virtual Server는 LAN에 특정 기능을 하는 서버를 지정하여 WAN에서 접근할 수 있는 기능을 제공하는 것이고 Access Control은 LAN에서 WAN으로 접근할 수 있는 권한을 제한하는 것이며 Special Internet Application은 NAT기능으로 처리하지 못하는 인터넷 애플리케이션에 대한 처리를 위한 것이다. Exposed Host는 LAN의 한 머신을 WAN에 완전히 노출시키는 기능이며 DNS캐쉬는 DNS 질의에 대한 응답을 캐싱하는 기능이고 DHCP 클라이언트와 서버는 WAN쪽에서 DHCP를 이용해 IP어드레스를 부여받는 기능과 LAN쪽의 머신에 IP어드레스를 부여하는 역할을 한다. Static Routing은 LAN쪽에 라우팅해야 할 랜 세그먼트가 생겼을 때 라우팅할 수 있도록 하는 기능이고 PPPoE는 WAN에 연결되는 방법이 이더넷 뿐만 아니라 다른 방법도 제공하도록 하기 위한 생각에서 조사를 한 결과 대부분의 가정에 연결되는 ADSL장비가 PPPoE를 통해 인증과 연결을 하기 때문에 추가한 기능이다.

5.2 개발 환경

일반적으로 리눅스를 내장형 장비를 위한 운영체계로 포팅하기 위해서 개발 환경을 구축할 때는 크게 두가지 단계를 거치게 된다. 그 첫번째는 개발 툴 구축이다. 운영체계를 포팅할 때는 개발을 위한 개발 플랫폼과 타겟 호스트가 구별되어 있는 경우가 대부분인데 그 이유는 우선 포팅의 목표 아키텍처를 가진 타겟 호스트는 내장형 장비를 위한 프로세서를 사용하므로 개발용 장비로 쓰기에는 그 프로세싱 파워가 너무 부족하며 게다가 그 프로세서에 개발환경을 구축할만한 운영체계가 올라가 있지 않은 경우가 많기 때문이다. 따라서 개발 플랫폼은 강력한 프로세싱 파워와 안정적인 운영체계를 가진 PC나 유닉스 워크스

테이션에 구축하고 타겟을 위한 바이너리를 만들어내는 크로스 컴파일러 툴셋을 설치하게 된다. 이런 구조는 그림 14와 같이 표현된다.

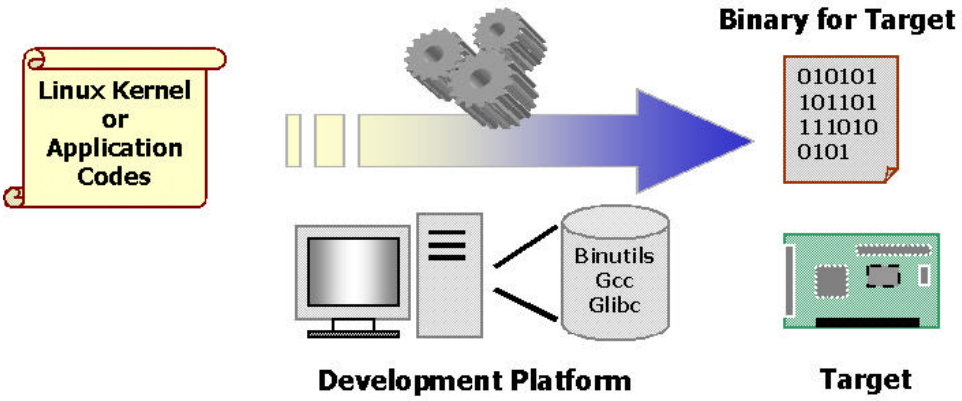

그림 14 Cross 개발 환경

개발 툴 구축의 두번째 단계는 디버깅 환경 구축이다. 내장형 운영체계를 위한 디버깅은 소프트웨어 디버깅과 하드웨어를 이용한 디버깅 두가지로 나뉠수 있는데 소프트웨어 디버깅은 소스레벨에서 브레이크 포인트나 특정 변수의 값들을 출력해 보면서 디버깅을 하는 일반적인 방법이고 하드웨어 디버깅은 프로세서에 정의되어 있는 물리적 디버깅 인터페이스를 통해 프로세서를 외부에서 완전히 콘트롤하여 내부 상태를 관찰해 가면서 디버깅을 하는 방법이다. 소프트웨어 디버깅은 디버깅을 위한 플랫폼이 따로 필요없지만 일단 타겟 호스트에서 프로그램의 진행상태를 외부로 표현할 수 있는 장치를 우선 살려야 한다는 단점이 있다. 이런 장치로는 일반적으로 LED, 7-segment LED와 같은 저수준 장치에서 LCD 디스플레이, RS-232 인터페이스까지 다양하다. 하드웨어 디버깅은 고가의 특별한 하드웨어가 필요하고 따로 디버깅 플랫폼도 필요하다는 단점이 있지만 실제로 하드웨어의 어느 부분도 초기화 시키는 노력을 할 필요가 없으므로 운영체계의 포팅에 아주 많은

도움을 줄 수 있다. 따라서 운영체계의 포팅엔 특수한 하드웨어를 이용한 디버깅을 많이 하고 운영체계가 올라간 후부터는 소프트웨어 디버깅을 많이 한다. 리눅스 커널의 경우는 일단 리눅스가 올라간 후부터는 커널과 애플리케이션 디버깅을 RS-232 나 이더넷 인터페이스로 할 수 있다는 특징이 있다. 즉, 하드웨어 없이도 디버깅 플랫폼을 따로 구성할 수 있다는 의미이며 이는 내장형 장비 개발에 큰 도움이 된다. 리눅스 상에서 디버깅 환경의 구축은 그림 15와 같은 구조를 가진다.

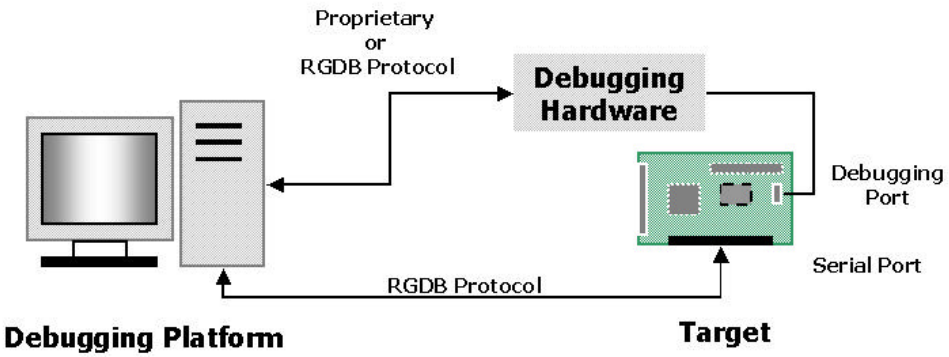

그림 15 디버깅 환경

인터넷 공유 장비 구현에서는 표 6 에 나와있는 PowerPC용 크로스 컴파일러 툴셋과 디버깅 툴들을 사용했으며 하드웨어 디버깅 툴로는 PowerPC의 디버깅 인터페이스로 정의되어 있는 BDM(Background Mode Debugger)포트를 사용하는 여러 BDM 디버거중에 리눅스의 리모트 gdb프로토콜을 지원하는 Abatron의 BDI2000 for GDB를 사용하였다 [44].

Name	Functions
powerpc-linux-gcc **powerpc-linux-g++**	GNU C and C++ compiler
powerpc-linux-as	GNU assembler
powerpc-linux-gdb	GNU debugger
powerpc-linux-objdump **powerpc-linux-objcopy** **powerpc-linux-strings** **powerpc-linux-strip** **powerpc-linux-size**	GNU object file tools
powerpc-linux-ar **powerpc-linux-nm** **powerpc-linux-ranlib**	GNU library tools
gdbserver	PowerPC Binary that enables application debugging with remote gdb protocol

표 6 Cross 개발 툴 셋

5.3 리눅스 커널

4 장에서 리눅스 서브시스템 내에서의 의존성을 검사해 보았고 이에 따라 인터넷 공유 장비에 쓰이도록 리눅스 커널 코어의 서브시스템에서 다음과 같은 부분들을 제거할 수 있었다.

- 프로세스 스케줄러
  - MPC850 과 무관한 아키텍처 의존적인 코드
  - 동적 로딩 모듈 관리 코드

✍✍ 메모리 매니저

✍✍ MPC850 과 무관한 아키텍처 의존적인 코드

✍✍ 가상 파일 시스템

✍✍ ext2 를 제외한 파일 시스템 드라이버

✍✍ elf와 script를 제외한 바이너리 실행 포맷 드라이버

✍✍ 시리얼과 가상 터미널을 제외한 캐릭터 디바이스 드라이버

✍✍ 램디스크를 제외한 블락 디바이스 드라이버

✍✍ Inter-Process Communication

✍✍ 없슴

✍✍ 네트웍 인터페이스

✍✍ TCP/IP와 파이어 월을 제외한 프로토콜과 네트워킹 코드

✍✍ PPP와 MPC850 의 이더넷 컨트롤러를 제외한 네트웍 디바이스 드라이버

각 코어 서브시스템에서 제거한 부분이외에도 코어에 속하지 않는 많은 부분을 제거하였으며 이를 통해 필요한 기능만을 모은 최소한의 커널을 얻을 수 있었다.

이렇게 필요없는 부분을 제거한 후에 실제로 커스텀 하드웨어에 포팅하는 작업을 했으며 이때 리눅스 커널에 다음과 같은 몇가지 큰 수정 및 첨가를 하였다.

✍✍ MPC850DE의 serial communication channel을 이용하여 2 개의 이더넷 인터페이스를 사용할 수 있게 해주는 디바이스 드라이버

- ✍✍ MPC850DE의 serial management channel을 이용한 시리얼 디바이스 드라이버
- ✍✍ AMD와 Intel의 플래쉬에 read/write/erase를 가능하게 해주는 디바이스 드라이버
- ✍✍ PPPoE 드라이버

이들에 대해 설명을 하면 인터넷 공유 기능을 위해 2 개의 이더넷 컨트롤러, 시리얼 콘솔을 통한 관리를 위해 시리얼 컨트롤러, 플래쉬를 통한 파일시스템 저장과 런타임 정보 저장을 위한 플래쉬 디바이스 드라이버, PPPoE지원을 위한 드라이버 등이 주요 수정 내용이다.

이들을 모두 첨가하여 컴파일하면 하나의 커널 이미지가 나오며 그 크기는 대략 1MB정도가 되었다. 이를 다시 압축하여 압축된 커널을 플래쉬에 담아 부팅초기에 압축을 풀어 커널 이미지를 램에 올려 실제 부팅을 하게 되는데 이렇게 압축된 커널의 크기는 약 400KB 미만이 된다.

5.4 애플리케이션

애플리케이션은 크게 시스템 운영을 돕는 시스템 유틸리티들과 인터넷 공유를 위한 기능을 제공하는 네트워킹 애플리케이션들이 있다. 시스템 유틸리티들로는 표 7 과 같은 유틸리티들이 쓰였고 이들 유틸리티들은 대부분 일반적인 리눅스 혹은 유닉스 시스템에서의 같은 이름의 유틸리티들과 같은 역할을 한다. 다른 것이 있다면 내장형 장비에 어울리도록 많은 기능을 제거하고 그 크기를 줄인 데 있다.

Name	Functions
ash	Shell
busybox	Multi-function binary Works as [, basename, cat, chgrp, chmod, cp, cut, date, df, dmesg, echo, false, free, grep, hostname, id, kill, killall, ln, ls, lsmod, mkdir, mount, mv, nslookup, ping, ps, rm, rmmod, sleep, sync, syslogd, test, touch, true, uname, update and uptime
tinylogin	Multi-function binary Works as getty, login, passwd, sulogin
route	Show/manipulate the IP routing table
insmod	Install loadable kernel module
expr	Evaluate expressions
stty	Set the options for a terminal
make_scripts	Make configuration scripts from configuration DB file
flash_utils	Multi-function binary Works as file2flash, flash2file, flash2img, flash_info, img2flash, tftp2flash
killall5	Send a signal to all processes
halt, reboot, poweroff	Stop the system

표 7 시스템 유틸리티

이런 유틸리티들의 목적은 개발단계와 시스템 안정화 단계에서 운영체계의 디버깅을 위한 것이 많으며 따라서 실제 시스템이 안정되면 대부분을 없애도 된다. 하지만 make_scripts와 같은 몇몇 유틸리티들은 시스템의 정보를 플래쉬에서 읽어오거나 저장하는 기능을 가지고 있거나 웹 인터페이스의 CGI에서 사용하기 때문에 없앨 수 없다. 이들

시스템 유틸리티들은 전체 500KB정도의 크기를 차지한다.

네트워킹 애플리케이션들로는 크게 NAT기능을 해주는 각종 프로그램들과 웹 기반의 장비관리를 위한 웹 서버, 그리고 그에 따르는 CGI와 웹 페이지들이 있다. 이런 애플리케이션들은 표 8에 나와있다.

Name	Functions
dhcpd	DHCP daemon
pump	DHCP client for embedded purpose
ens	Embedded domain name server
inetd	Internet super server
ipchains	IP firewall administration
ipmasqadm	IP Masquerading additional modules administration
pppd, pppoe	Poin to Point Protocol daemon and PPPoE wrapper
telnetd	Telnet server
webs	GoAhead web server

표 8 네트워킹 애플리케이션

이런 애플리케이션들은 인터넷 공유의 핵심적인 부분이며 그 기능또한 쉽게 구현하기는 힘이 든다. 하지만 리눅스를 위한 NAT관련 프로그램들과 웹 서버 등이 모두 존재하며 모두 수많은 사용자들을 통해 검증이 된 프로그램들이므로 실제로 이들 기능을 구현하는데 있어 들인 노력의 대부분은 크로스 컴파일시의 오류 수정과 웹 인터페이스 작업, 그리고 시스템에 통합하기 위한 coordination을 위한 것이었다. 이들 코드들은 인터넷 공유 장비의 코어 소프트웨어이므로 제거할 수 없고 그 크기는 전체 200KB정도를 차지한다.

5.5 그 밖의 시스템 컴포넌트

여기서는 커널과 애플리케이션외에 리눅스 시스템을 구성하는 중요 요소인 런타임 라이브러리와 파일시스템의 구현에 대해서 다룬다.

5.5.1 런타임 라이브러리

표 8 의 애플리케이션들은 모두 스탠다드 glibc를 통해 링킹된 것이다. 따라서 런타임 라이브러리는 glibc이다. glibc는 내장형 장비를 위해 나온 라이브러리가 아니므로 그 크기는 상당히 크나 리눅스에서 사용되는 거의 모든 오픈소스 소프트웨어가 이를 통해 개발된 것이므로 수많은 애플리케이션들을 별다른 어려움없이 포팅해서 사용할 수 있다는 장점을 가진다. 따라서 개발기간을 상당히 단축시킬 수 있으며 그 동작에 있어서도 PC에서만큼의 신뢰성을 확보할 수 있다. 스탠다드 C 라이브러리 이외에 다른 프로그램에 필요한 몇가지 라이브러리를 사용하였으며 이들 라이브러리들은 표 9와 같다.

Name	Functions
ld-2.1.2.so **libc-2.1.2.so** **libcrypt-2.1.2.so** **libdl-2.1.2.so** **libnss_db-2.1.2.so** **libnss_dns-2.1.2.so** **libnss_files-2.1.2.so** **libutil-2.1.2.so** **libresolv-2.1.2.so**	Portion of GNU C library
libgdbm.so.2.0.0	GNU database indexing library
libproc.so.2.0.0	procps system information library

표 9 런타임 라이브러리

이들 라이브러리들의 크기는 전체 1.1MB정도를 차지한다.

5.5.2 파일 시스템

파일시스템의 구현은 램디스크상에 ext2 파일시스템을 만드는 방법을 택했다. 이 방법을 택한 이유는 이 장비가 한 인터페이스로 들어오는 IP패킷을 처리하여 다른 인터페이스로 내보내는 장비이며 이를 소프트웨어적으로 처리해야 한다는 점에 있어서 속도를 최우선 목표로 삼았기 때문이다. 내장형 장비에 쓰이는 저장장치중에 램이 가장 빠르다는 것은 주지의 사실이며 파일시스템상에 모든 애플리케이션을 구현한 현재의 시스템 구성에서는 램디스크가 가장 적절한 선택이라 할 수 있다. 램디스크 상에 올릴 수 있는 파일시스템은 리눅스가 지원하는 모든 파일시스템중에서 리눅스를 위한 파일시스템인 ext2 파일 시스템을 사용한다. 크기를 줄이기 위해서는 minix파일시스템이나 msdos파일시스템을 쓸 수도 있으나 속도와 호환성에 있어 뒤떨어지며 RAMFS를 사용할 수도 있지만 포팅에 시간이 걸리므로 사용하지 않았다. 시스템 구성에 대한 런타임 데이터들은 램디스크의 특성상 전원공급이 끊기면 사라져 버리므로 파일시스템에 가지고 있다가 시스템 부팅/셧다운시에 플래쉬로 저장하도록 했다. 결과적으로 파일시스템은 그림 16 과 같이 PC용 리눅스 시스템과 비슷한 구조를 가진다.

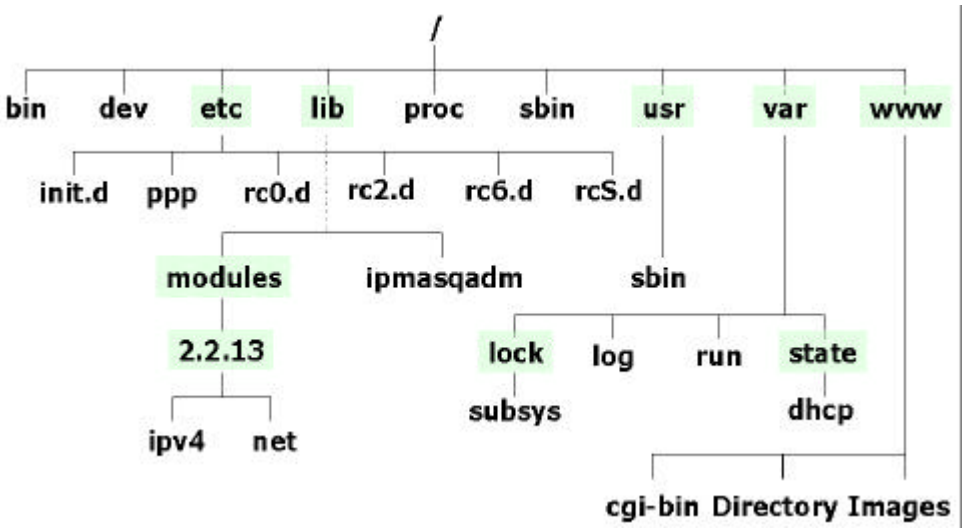

그림 16 루트 파일 시스템 디렉토리 구조

시스템을 구성하는 모든 파일들을 다 합하여 파일시스템 크기를 구해보면 약 3MB정도가 나오게 된다. 실제로 이 파일시스템은 압축되어 저장되므로 압축된 파일시스템의 크기는 1.2MB정도가 된다.

5.5.3 저장 장치

지금까지 구현한 커널과 파일시스템을 저장하는 저장공간으로는 플래쉬 메모리를 선택하였다. 플래쉬 메모리는 전원공급이 끊어져도 그 내용이 보장되며 속도는 다소 느리지만 섹터 단위로 쓸 수 있다. 대부분의 네트워킹 디바이스는 저장을 위한 장치로 eeprom이나 플래쉬 메모리를 사용하는데 플래쉬 메모리의 사용이 늘고 있는 추세이다. 이 장비에서 사용한 플래쉬 메모리는 AMD와 인텔의 것이며 둘다 기본 섹터 사이즈는 64KB이고 크기는 2MB이다. 인텔의 경우 bottom boot타입의 플래쉬로 처음 64KB섹터는 8 개의 8KB섹터로 나뉘어져 있다. 커널과 파일시스템을 플래쉬에 그림 17 과 같이 구성하였으며 실제로 시스템

가동시 쓰기 기능을 활용하는곳은 마지막 두 섹터이다.

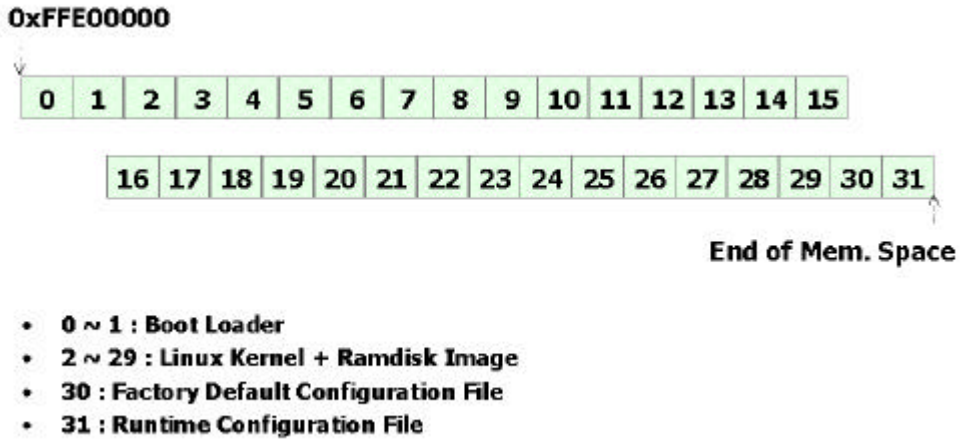

그림 17 플래쉬 메모리 내용

5.6 구현 결과

구현된 인터넷 공유 장비에 내장된 커널은 전체 크기가 압축된 커널 이미지일 때 약 400KB정도의 플래쉬 영역을 차지하며 부팅이 되고난 후에는 램에서 약 1MB정도의 커널 코드, 400K정도의 스택 영역, 1.5MB정도의 데이터 영역을 차지한다. 결국 커널의 런타임 리소스 요구량은 약 3MB정도의 램이며 플래쉬 영역은 요구하지 않는다. 그리고 웹서버, DNS 캐쉬등이 유저 애플리케이션으로 운용되며 이들은 약 2MB정도의 시스템 메모리를 차지하고 있다. 여기에 램디스크 영역이 고정적으로 4MB가 할당되므로 구현된 임베디드 리눅스 시스템은 전체적으로 약 3MB + 2MB + 4MB = 9MB의 런타임 리소스를 사용하고 있다. 하드웨어가 16MB의 램을 가지고 있으므로 충분히 시스템이 실패없이

돌아갈 수 있는 환경이며 실제로 기능 테스트 결과 15 대의 PC를 이 장비밑에서 운영했을 때 아무런 문제없이 일주일 이상을 운영할 수 있었다.

그리고 성능 측정을 위해 파일 전송 속도를 여러 상용 인터넷 공유 장비들과 비교하여 보았는데 어떠한 제품들보다도 빠른 속도를 보여주었다. 물론 각 제품들마다 하드웨어 스펙이 다르므로 운영체계의 성능에 대한 절대적인 비교는 될 수 없겠지만 테스트한 제품들의 프로세서가 대부분 ARM계열이고 이 프로세서들의 성능이 구현된 장비의 프로세서인 MPC850 보다는 떨어지지만 그 차이가 그리 크지 않다는 점을 볼 때 임베디드 리눅스 시스템의 성능을 어느정도 짐작해 볼 수 있다.

결과적으로 구현시 기대했던 시스템의 안정성과 성능을 모두 만족시켰다.

6. 결론 및 향후 연구

네트웍이 인류사회의 모든 영역을 연결시키면서 네트워킹 디바이스들은 하루가 다르게 새로워지며 강력해지고 있다. 그리고 기존의 전자장비들도 시대의 변화에 의해 점점 네트워킹 기능을 필요로 하고 있다. 이런 추세에 의해 산업계, 학계에서 네트워킹 디바이스를 구현할 수 있는 능력이 요구되어지고 있지만 기존의 내장형 운영체계들은 개발자들이 쉽게 접근할 수 없는 요소들을 많이 가지고 있다.

이런 상황에서 오픈소스 운영체계인 리눅스가 내장형 장비를 위한 운영체계로서 하나의 대안으로 등장하였고 그 자체의 우수성과 수많은 오픈소스 애플리케이션들로 인해 점점 그 사용이 증가하게 되었다. 그에 따라 이제는 임베디드 리눅스라는 내장형 운영체계로서의 하나의 조류를 형성하게 되었다.

임베디드 리눅스는 내장형 장비업계들로부터 크게 각광받고 있다. 현재 임베디드 리눅스 배포본을 판매하거나 포팅 서비스를 해주고 있는 회사들이 전세계적으로 수십개 사에 달하며 임베디드 리눅스를 운영체계로 선택하고 있는 장비 회사들도 점점 늘어나고 있는 추세이다.

본 논문은 이런 변화의 흐름속에 네트워킹 디바이스를 위한 운영체계로서 보다 나은 임베디드 리눅스 시스템을 설계하고자 하는 목적을 가지고 출발하였다. 이를 위해 우선 네트워킹 디바이스의 운영체계가 가져야 할 요구조건에 대해 여러 내장형 운영체계의 경우를 통해 조사하고 이를 이용해 리눅스가 네트워킹 디바이스를 위한 내장형 운영체계로서 요구조건들을 만족함을 보였다. 그리고 리눅스 코어 커널의 서브시스템 별로 내부 컴포넌트간의 의존성을 조사하여 내장형 운영체계로서 적당한 크기를 가질 수 있도록 커널 구성을 설계하였다. 그 위에 런타임 라이브러리, 파일시스템의 구성을 내장형 시스템에 적합하도록 설계하였다. 그리고 이를 바탕으로 임베디드 리눅스 시스템을 디자

인하였으며 실제 하드웨어에 디자인한 시스템을 구현하여 네트워킹 디바이스를 제작하였다. 제작된 네트워킹 디바이스는 인터넷 공유를 주기능으로 하는 인터넷 공유 장비이다. 테스트를 통해 이 장비가 안정성과 성능을 모두 만족함을 확인하였으며 이를 통해 구현된 임베디드 리눅스 시스템이 실제로 네트워킹 디바이스를 위한 보다 나은 운영체계임을 보였다.

구현된 임베디드 리눅스 시스템이 만족스러운 결과를 보였지만 아직 개선될 사항과 연구해 보아야 할 사항이 몇가지 있다.

우선 똑 같은 하드웨어 스펙상에서 다른 내장형 운영체계들을 이용하여 같은 기능을 하는 네트워킹 디바이스를 구현한 후 전체적인 속도와 안정성에 대한 객관적인 비교를 통해 구현한 임베디드 리눅스 시스템의 구체적인 성능평가를 해야한다.

그리고 디자인한 임베디드 리눅스 시스템의 구조를 다른 임베디드 리눅스 배포본과의 비교를 통해 디자인 구조상의 차이점을 파악하여 평가하여 보아야 한다.

시스템의 일차적인 목적을 속도로 하다보니 리소스의 낭비가 있다. 파일시스템과 런타임 라이브러리가 가장 낭비가 큰 부분이며 이를 해결하기 위해 속도와 리소스 요구량에 대하여 여러가지 대안들을 조합하여 시스템을 구성하고 최적화된 시스템 설계를 찾아내야 한다.

현재 임베디드 리눅스에 실시간성을 부여하는 프로젝트가 많이 존재한다. 이들 프로젝트들에 대해 조사해보고 그 결과를 이용해 네트워킹 디바이스에 특화할 수 있는 요소가 있는지 혹은 성능을 최적화할 수 있는 요소가 있는지 알아보는 것이 필요하다.

그리고 MPC850DE 이외의 다른 프로세서로의 포팅을 통해 구현대상을 넓혀 디자인상의 문제점을 파악하여 시스템의 디자인을 향상시키는 작업도 필요하다.

마지막으로 리눅스 커널 내부의 구조를 좀더 깊이 파악하여 네트워킹 디바이스를 위한 내장형 운영체계로서 적절하지 못한 부분을 찾아내어 이를 개선시키는 연구도 그 의미가 크다고 할 수 있다. 즉, 리눅스 커널의 세부적인 동작 특징을 명세하여 현재보다 좀더 리눅스 커널을 세분화 할 수 있는 여지를 제공하는 일도 큰 의미가 있다.

참고 문헌

[1] Emmanuel Marty, "The TUNES Project for a Reflective Computing System – Review of Operating Systems", April 2000, http://www.tunes.org/Review/OSes.html#Embedded_Systems.

[2] Rick Lehrbaum, "Using Linux in Embedded and Real-time Systems", February 2000, http://www.linuxdevices.com/articles/AT3611822672.html.

[3] Embedded Systems Programming, "2001 Buyer's Guide", http://www.embedded.com/bg/2001_bg.htm.

[4] Rick Lehrbaum, "What is Linux?", July 2000, http://www.linuxdevices.com/articles/AT5383333597.html.

[5] Linux Documentation Project, http://www.linuxdoc.org.

[6] Korean Linux Documentation Project, http://kldp.org.

[7] Rick Lehrbaum, "Embedded Linux – one year later", September 2000, http://www.linuxdevices.com/articles/AT3595894022.html.

[8] University of Canberra, Australia, "The Xinu Page", October 1998, http://willow.canberra.edu.au/~chrisc/xinu.html.

[9] AI Lab Zurich, "Embedded and Real-Time Systems", October 2000, http://www.ifi.unizh.ch/groups/ailab/links/embedded.html.

[10] Micrium, Inc., "Home of uC/OS-II, The Real-Time Kernel", December 2000, http://www.ucos-ii.com/.

[11] RedHat, "eCOS", http://www.redhat.com/embedded/technologies/ecos/.

[12] TRON Association, "ITRON Project Home Page", 1999, http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html.

[13] Real-Time Magazine, "Comparison between VxWorks/x86 5.3.1, QNX 4.25 and pSOSystem/x86 2.2.6", April 1999.

[14] Wind River, "pSOSystem", http://www.wrs.com/products/html/psosystem.html.

[15] QNX, "QNX Products", http://www.qnx.com/products/index.html.

[16] Mentor Graphics, "VRTX Real Time Operating System", http://www.mentor.com/embedded/vrtxos/index.html.

[17] Wind River, "VxWorks 5.4",

http://www.windriver.com/products/html/vxwks54.html.

[18] Accelerated Technology, "Nucleus real-time, multitasking kernel", http://www.atinucleus.com/nucprods/kernels.htm.

[19] Microware, "OS-9 Real-Time Operating System", http://www.microware.com/Products/Software/OS9-overview.html.

[20] Cisco Systems, "Cisco IOS Software", http://www.cisco.com/warp/public/732/jump.shtml.

[21] 3Com, "Enterprise OS User Guides", http://infodeli.3com.com/infodeli/tools/bridrout/u_guides/html/index.htm.

[22] Linux Devices, "The Embedded Linux Market", May 2000, http://www.linuxdevices.com/files/article002/.

[23] Lineo, "Developer Info : White Papers", http://www.lineo.com/file_index/w/white_papers/index.html.

[24] Jim Ready and Bill Weinberg, "Leveraging Linux for Embedded Applications", January 2000, http://www.linuxdevices.com/articles/AT9844090738.html.

[25] Rick Lehrbaum, "Real-time Linux – what is it, why do you want it, how do you do it?", September 2000, http://www.linuxdevices.com/articles/AT9837719278.html.

[26] Heurikon Corporation, "Choosing an Operating System for Embedded Real-Time Applications", 1996, http://pats.crane.navy.mil/pubdoc/choosos.htm.

[27] IEEE Standards Association, "Portable Operating System Interface (POSIX)", http://standards.ieee.org/catalog/posix.html.

[28] Real-Time Korea, "An Introduction to the Real-Time OS & Nucleus PLUS Training Guide".

[29] uClinux, "Embedded Linux Microcontroller Project", http://www.uclinux.org/.

[30] Linux's Ports, "Current ports of Linux OS", July 2000, http://perso.wanadoo.es/xose/linux/linux_ports.html.

[31] Linux on IA-64, "The Official Home Page of the IA-64 Linux Project", http://www.linuxia64.org/.

[32] SourceForge, "SourceForge Statistics", http://sourceforge.net.

[33] Freshmeat, "Application Index", http://freshmeat.net/appindex/.

[34] Paul Dunne, “An Introduction to Linux”, 2000, http://members.linuxstart.com/~dunne//what_is_linux.html.

[35] Juancarlo Añez, “The true meaning behind open source licenses”, October 1999, http://community.borland.com/devnews/article/1,1714,10335,00.html.

[36] Eric Steven Raymond, “Software Release Practice HOWTO”, August 2000, http://www.linuxdoc.org/HOWTO/Software-Release-Practice-HOWTO/index.html.

[37] Open Source Initiative, “The Approved Licenses”, http://www.opensource.org/licenses/.

[38] Meyer C. Tanuan, “An Introduction to the Linux Operating System Architecture”, January 1998, http://www.grad.math.uwaterloo.ca/~mctanuan/cs746g/LinuxCA.html.

[39] Ivan Bowman, Saheem Siddiqi, Meyer C. Tanuan, “Concrete Architecture of the Linux Kernel”, February 1998, http://plg.uwaterloo.ca/~itbowman/CS746G/a2/.

[40] Graham Stoney, “Linux for PowerPC Embedded Systems HOWTO”, December 2000, http://penguinppc.org/usr/embedded/howto/PowerPC-Embedded-HOWTO.html.

[41] K. Egevang, Cray Communications, P. Francis, NTT, “The IP Address Translator (NAT)”, May 1994, RFC 1631, http://sunsite.cnlab-switch.ch/ftp/doc/standard/rfc/16xx/1631.

[42] VicomSoft, “Network Address Translation”, 2000, http://www.vicomsoft.com/knowledge/reference/nat.html.

[43] Craig A. Haller, “The ZEN of BDM”, 1997, http://www.macraigor.com/zenofbdm.pdf.

[44] Abatron AG, “BDM/JTAG Debug Interfaces”, http://www.abatron.ch/BDI/bdimain.html.

[45] Andrew S. Tanenbaum, “Operating Systems: Design and Implementation”, Prentice-Hall, 1987.

[46] David A. Rusling, “The Linux Kernel”, 1999, http://www.linuxdoc.org/LDP/tlk/.

[47] LynuxWorks, “LynxOS”, http://www.lynuxworks.com/products/whatislos.html.

[48] Joshua Drake, “Linux Networking HOWTO – DocBook Rev. 02”, 2000, http://www.linuxdoc.org/HOWTO/Net-HOWTO/index.html.

[49] Embedded Systems Conference, “Embedded Systems Conference Papers”, 2000, http://www.esconline.com/papers.htm.

[50] Jeff Child and Charles J. Murray, “Linux deals rock the RTOS status quo”, October 2000, http://www.eet.com/story/OEG20000929S0036.