사용자 수준의 단말기 구동기 모델

이 형 주† 임 성 락††

요 약

기존의 유닉스 시스템에서는 모든 장치의 구동기가 커널 내부에 구현되어 있다. 따라서, 새로운 장치를 지원하기 위한 구동기를 추가하거나 기존의 구동기를 변경할 경우 커널 내부의 수정 작업이 불가피하다. 일반적으로 유닉스 시스템에서는 커널 내부의 수정 작업이 매우 어렵다. 본 논문에서는 이러한 어려움을 극복하기 위한 방법으로써 사용자 수준의 단말기 구동기 모델을 제시한다. 제시한 모델의 기본 개념은 단말기 구동기를 사용자 수준의 서버로 구현함으로써 새로운 단말기 구동기의 동적 재구성을 제공하는 것이다. 제시한 모델의 타당성을 검증하기 위해 사용자 수준의 단말기 구동기를 SunOS 와 Linux 환경에서 구현하고, 그 성능을 평가하였다.

TTY Device Driver Model of User-Level

Hyung Ju Lee† Seung Rak Rim††

ABSTRACT

In the conventional UNIX system, the all device driver realized in the kernel. Hence, whenever we want to add a new device driver or change the existing device driver, the modification of kernel is unavoidable. Generally, it is very difficult to modify the kernel codes.

As a method of overcoming this difficulty, a TTY device driver model of user-level is presented in this paper. The basic concept of this model is providing a dynamic reconfiguration of TTY device driver by realizing a user-level server process for TTY device driver. In order to verify the property of this model, a prototype of TTY device driver has been realized in the SunOS and Linux environments and evaluated its performance.

1. 서 론

유닉스 시스템에서 입출력 장치를 포함하여 모든 주변 장치를 제어하기 위한 장치 구동기(device driver)루틴은 커널 내부에 구현되어 있다. 따라서 새로운 유형의 장치를 첨가하거나 기존 구동기의 기능을 수정하기 위해서는 커널 내부를 수정해야만 한다[5]. 예를 들어, 한글을 지원하기 위한 코드 체계는 여러가지가 있으며, 이 모든 한글 코드 체계를 지원하기 위해서는 각각의 코드 체계에 대한 장치 구동기가 요구된다. 따라서 다양한 한글 코드 체계를 지원하기 위해서는 커널 내부의 수정이 불가피하다. 그러나, 일반적으로 커널 내부를 수정하는 작업은 매우 어렵다.

이러한 문제점을 해결하기 위한 연구가 계속 진행되어 오고 있다. 일반적으로 유닉스 시스템에서 제공하고 있는 여러가지 코드 체계는 특정 언어에 대해서만 정의되어 있거나 최적화 되어 있는데, 이는 이식성을 떨어뜨릴 뿐만 아니라 새로운 언어를 지원하기 위해서는 많은 개발 시간을 필요로 한다. 따라서 입출력에 관련된 유틸리티는 특정 언어와는 독립적으로 작성되어야 한다. AT&T를 포함한 여러 연구진들은 한 바이트 문자나 한글, 중국어 등과 같은 다중 바이트 문자를 처리하기 위해서 스트림에 기반한 단말기 부 시스템을 개발하였다[3]. 입출력 처리기 중 스트림(STREAMS)에 기반한 단말기 부 시스템

† 정 회 원 : 호서대학교 컴퓨터공학과 석사과정
†† 정 회 원 : 호서대학교 컴퓨터공학과 조교수
논문접수 : 1995년 6월 28일, 심사완료 : 1995년 11월 25일

은 커널 내부의 회선 규범 모듈(line discipline)을 이용하여 단말기 구동기와 사용자 프로세스 사이에 유연한 인터페이스를 제공해 준다. 회선 규범 모듈은 전체 단말기 부 시스템에 영향을 주지 않으면서, 특정 목적을 위해서 만들어질 수 있다. 그러므로, 스트림에 기반한 단말기 부 시스템은 다양한 단말기 구동기를 지원하는데 있어서 유연성을 제공해 준다[3]. 개발된 단말기 부 시스템은 한 바이트 데이타를 처리하는 "raw 구동기"(하드웨어 의존 함수를 수행)와 4바이트까지의 추가 코드 체계를 다룰 수 있는 회선 규범 모듈로 구성되어 있다. 이렇게 일반화된 문자 처리 기법은 대상 언어에 대한 특정 스트림 처리 모듈만을 첨가함으로써 여러 아시아 지역의 다중 바이트 언어를 효율적으로 지원해 줄 수 있다. 그러나, 이 기법들도 새롭게 구현된 회선 규범 모듈및 스트림 모듈을 첨가하기 위해서는 결국 커널 내부의 수정 작업이 불가피하다. 따라서, 본 논문에서는 커널 내부의 수정 없이 사용자 수준에서 단말기 인터페이스 기능을 제공할 수 있는 사용자 수준의 장치 구동기 모델을 제시하고자 한다.

사용자 수준의 장치 구동기 관리에 대한 기존 연구로서, Alessandro Forin이 Mach 3.0상에서 ethernet와 SCSI 디스크 구동기를 사용자 수준에서 작성하여 그 성능을 평가하였으며, 이들 사용자 수준의 장치 구동기가 좋은 성능을 나타낼 수 있다는 사실을 발표하였다[1]. 그 후에 CMU와 IBM에서는 Alessandro Forin의 연구 내용을 개선시켜 더욱 완성된 형태의 사용자 수준의 장치 구동기 모델을 제시하였다[2]. 이 모델에서는 장치 구동기가 동일한 장치를 상호 보완적으로 관리함으로써, 장치에 대한 접근과 장치에 대한 연산을 분리시켰다. 이렇게 함으로써, 여러 장치 구동기가 동일한 하드웨어 자원을 공유할 수 있으며, 시스템의 이식성이 향상되었다. 결국, 단말기 구동기에 대한 최근의 연구들은 다국적 언어의 효율적인 지원과 이식성의 향상 그리고 시스템의 성능 향상에 초점을 두고 있다.

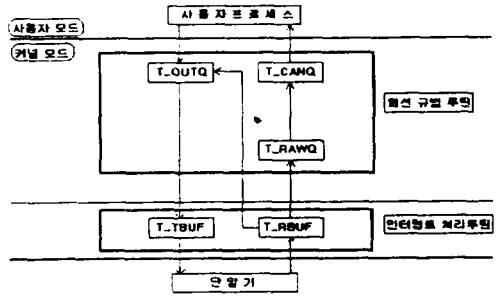
본 논문에서는 기존 단말기 구동기와 비교하여 큰 성능 차이가 없으면서도, 새로운 언어 체계를 효율적으로 지원해 주고 단말기 구동기의 이식성을 향상시킬 수 있는 사용자 수준의 단말기 구동기 모델을 제시하고자 한다. 본 논문의 구성은, 2장에서 사용자 수준의 단말기 구동기 모델을 설계하고, 3장에서 설계한 모델을 기반으로 프로토타입의 단말기 서버를 구현하고, 4장에서 구현한 단말기의 입출력 데이타 성능을 평가하고, 5장에서 결론을 기술한다.

2. 단말기 서버 모델의 설계

2.1 배경

유닉스 시스템에서 단말기의 표준 문자 입출력 처리와 버퍼링은 커널 내부의 회선 규범 모듈에 의해서 처리된다[5]. 커널 내부에 위치한 회선 규범 모듈은 단말기와 사용자 프로세스 사이의 인터페이스를 제공해 준다. 예를 들어, 단말기로부터 입력된 백스페이스 문자의 처리, 문자의 반향(echo), 탭 확장, CR(carriage return)/NL(new line) 사상(mapping)등과 같은 작업은 모든 직렬 장치 구동기에서 공통적으로 사용된다. 이와같이 공통적으로 사용되는 표준 문자 처리 루틴을 포함하고 있는 회선 규범 모듈은 사용자 프로세스로부터의 어떤 입출력 요청이 발생하면, 이 처리 루틴을 수행하게 된다. (그림 1)에서는 입출력 데이타 처리를 위해, 회선 규범 모듈과 구동기 인터럽트 루틴이 사용하는 버퍼들을 나타내고 있다.

본 논문에서는 단말기 구동기에 관련된 루틴을 다음과 같이 두개의 그룹으로 분류하였다. 하나는 단말기 장치에 관련된 인터럽트 처리 루틴



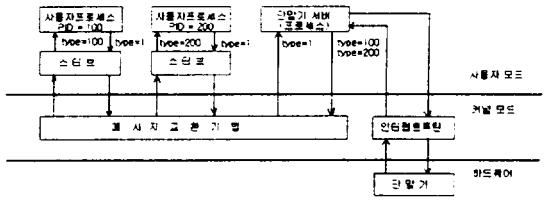
(그림 1) 단말기 인터페이스에서 사용하는 버퍼들
(Fig. 1) Buffers for TTY interface

(drvrint, drvxint)이고, 다른 하나는 장치 독립 루틴(ttopen, ttclose, ttin, ttout, ttread, ttwrite, ttioctl)이다. 인터럽트 처리 루틴 중 drvrint는 구동기의 수신 인터럽트 루틴으로 키보드 사상과 입력 흐름 제어, 키보드 신호 등을 처리한다. 그리고, 수신 인터럽트 버퍼(T RBUF)를 비울 필요가 있을 때는 ttin을 호출하여 T RBUF로부터 회선 규범 버퍼(T RAWQ)로 문자를 전송하게 한다. Drvxint는 구동기의 전송 인터럽트 루틴으로 전송 인터럽트 버퍼(T TBUF)를 비우고 난 다음, ttout을 호출하여 출력 문자들을 회선 규범 버퍼(T OUTQ)로부터 받아들이게 한다.

Ttin, ttout 루틴 외에 구동기를 열고(ttopen) 닫는(ttclose)루틴, 회선 규범 버퍼(T CANQ)에서 사용자 프로세스 영역으로 문자를 옮기거나(ttread) 사용자 프로세스 영역으로부터 회선 규범 버퍼(T OUTQ)로 문자를 옮기는(ttwrite) 루틴, 회선 규범 모듈의 변경이 요구될 때 호출되는 ttioctl 루틴 등은 장치와는 독립적으로 작성될 수 있다. 이들 장치 독립 루틴의 경우, 표준 모드(canonical mode)에서 특수 문자들(backspace, erase, kill 등)을 처리한다. 또한, 유닉스가 지원해야 하는 코드 체계와도 밀접한 관련이 있다. 예를 들어, 한 문자를 지우기 위해서 백스페이스 문자는 내부 큐들 사이에서 해당 코드 체계에 의존하여 처리 되야 한다. 그런데, 장치 독립 루틴이 커널 내부에 있으므로 새로운 코드 체계를 개발할 때마다 커널 내부의 수정 작업이 불가피하다. 이를 해결하기 위해서 코드 체계와 밀접한 관련이 있는 장치 독립 루틴을 사용자 수준에서 구현한 단말기 구동기 모델을 제시하고자 한다.

2.2 단말기 서버 모델

다른 코드 체계를 쉽게 지원하기 위해서, 장치 독립 루틴을 사용자 수준의 서버 프로세스로 대체시킨다. 또한, 이 서버 프로세스는 동일 단말기에서 수행되는 모든 사용자 프로세스들에 의해 공유된다. 인터럽트 처리 루틴은 시간 제약 조건과 하드웨어 의존성을 가지기 때문에 커널 내부에 구현되어야 하며, 장치와 서버 사이의 문자 전송을 직접 수행한다. 서버 프로세스는 기존 장치 구동기의 하드웨어 독립적인 부분을 구현하고 일련의 특수 문자들을 처리하도록 한다. (그림 2)는 본 논문에서 제안한 단말기 서버 모델을 나타내고 있다.



(그림 2) 단말기 서버 모델
(Fig. 2) TTY server model

제시된 모델에서 사용자 프로세스는 메시지 교환 기법을 이용하므로써 서버 프로세스에 입출력 수행을 요청한다[6, 7]. 서버 프로세스를 공유하기 위해서 서버 프로세스의 메시지 형(message type)은 모든 사용자 프로세스가 알아야 한다. 그리고, 사용자 프로세스의 메시지 형(message type)은 프로세스 식별자(PID)로 명시된다. 입출력을 요청한 사용자 프로세스의 식별자는 사용자 프로세스와 연결되는 스터브라는 입출력 라이브러리에 정의된 함수들에 의해서 서버 프로세스로 전달되며, 서버는 그 식별자를 메시지 형(message type)으로 사용하여, 응답에 대한 메시지를 사용자 프로세스로 전송한다. 이렇게 하므로써, 서버 프로세스를 여러 사용자 프로세스가 공유할 수 있게 된다. 그림 2에서는 여러 사용자 프로세스들에 의해서 하나의 메시지 큐가 공유되는 단말기 서버 모델이 나타나 있다. 또한, 단말기마다 서버 프로세스를 두지 않고 호스트에 하나의 서버 프로세스를 두어 모든 사용자 프로세스의 단말기에 대한 입출력 서비스를 처리한다. 이런 경우에, 입출력 연산을 수행하는 사용자 프로세스의 스터브는 프로세스 식별자뿐만 아니라 자신이 작업하는 단말기 식별자를 메시지에 포함하여 서버 프로세스에게 전송하도록 한다. 서버 프로세스는 먼저 해당 단말기에 대응하는 화일 디스크립터(file descriptor)를 열어(open) 관리하면서 쓰기 혹은 읽기 연산을 수행할 때마다 그 화일 디스크립터를 이용하여 해당 단말기로 입력을 받거나 출력을 한다. 본 논문에서 제시된 모델은 Stream 기법과 동적으로 적재

가능한 커널 모듈 기능을 지원하는 System V 뿐만 아니라 이를 지원하지 않는 다른 유닉스 시스템, 특히 마이크로 커널 기법을 채택하고 있는 유닉스 시스템 상에서도 구현될 수 있는 기법이다. 그리고, 단말기 인터페이스의 대부분을 사용자 수준에서 구현하는 것은 장치 구동기도 사용자 프로세스라는 의미에서 일관된 프로그래밍 기법을 제시해 준다. 또한, 장치 구동기를 개발하는 시간을 단축 시켜줄 뿐만 아니라, 테스트와 디버깅도 용이하게 해준다.

3. 구 현

2장에서 제시한 모델의 타당성을 검증하기 위해서 메시지를 기반으로 한 프로세스간 통신 기법(IPC)을 이용하여 프로토타입 단말기 구동기 서버를 구현한다[6, 7]. 단말기 서버 구현시 커널 변경을 필요로하는 부분은 기존의 입출력 시스템 호출을 이용하였다. 구현된 단말기 인터페이스에서 단말기 서버는 raw모드와 표준모드를 가진다. Raw모드에서 단말기 서버는 각 문자가 단말기로부터 입력되자마자 문자를 복귀시킨다. 표준모드에서는 단말기 서버는 NL(New Line) 문자나 화일-끝(EOF)를 수신 할 때까지 입력 바이트를 버퍼링시킨다. 버퍼링하면서 erase키나 kill키를 처리한다. 제시된 단말기 구동기 서버에서 수행되는 입출력 과정은 2장에서 제시된 모델과 같도록 한다. 사용자 수준에서 구현된 단말기 구동기 모델은 사용자 프로세스와 연결되는 스터브(stub)와 특수 문자 등을 처리하는 서버 프로세스로 구성된다.

3.1 스터브(stub)

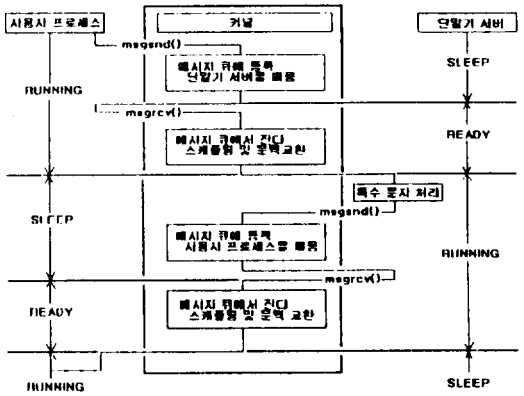
스터브는 입출력 라이브러리로 사용자 프로세스와 연결된다. 스터브는 커널 내부의 회선 규범 모듈에 있는 함수(ttread, ttwrite, ttopen, ttclose)들을 우회하기 위해서 ioctl시스템 호출을 사용하여 단말기 모드를 raw 모드와 no echo모드뿐만 아니라 no break모드, no signal모드, no crnl모드 등과 같이 사용자가 입력한 모든 데이타에 대해서 어떠한 처리도 하지 않도록 설정한다. 또한, 문자 입력도 한 바이트씩 입력하도록 설정한다(topen). 그런 다음 사용자 프로세스가 입출력을 요청할 때마다 회선 규범 모듈의 표준 문자 처리 루틴을 수행하지 않고, 스터브를 통해 필요한 제어 정보와 입출력 데이타를 메시지로 만들어 서버 프로세스에 전달해 준다. 시스템에 하나의 단말기 서버만을 둘 경우에는 해당 입출력 단말기 식별자도 서버 프로세스에게 전달한다. 스터브는 입출력 처리후 단말기 모드를 raw 모드에서 canonical 모드로 복구시켜 주는 역할도 한다(tclose). 스터브에서 메시지 전송에 쓰이는 데이타 구조(data structure)는 다음과 같다.

```
struct {
        long mesg type;
        int mesg pid;
        char mesg cmd;
        char mesg data[1024];
}
```

Mesg-type는 공유 메시지 큐에 여러 사용자 프로세스들이 메시지 전송을 할 경우에 어떤 사용자 프로세스가 메시지를 전송했는지 식별하기 위한 변수이고, mesg pid는 입출력을 요청한 사용자 프로세스 식별자(PID)이다. 또한, mesg cmd는 read, write, open, close인지를 식별하기 위한 변수이며, mesg data는 메시지로 주고받을 데이타가 저장될 버퍼다.

3.2 단말기 구동기 서버

서버 프로세스는 사용자 프로세스에서 스터브



(그림 3) 메시지 교환및 동기화 과정
(Fig. 3) Procedure of message exchange and synchronization

를 통하여 만들어진 메시지를 전송 받아 메시지를 분석하고 처리한다. 단말기 서버 프로세스는 호스트 컴퓨터에서 백그라운드 프로세스로 수행되고 사용자 프로세스에 의해서 공유 메시지 큐에 메시지가 전송되면 깨어나서 일련의 처리 루틴을 수행한다. 또한, 일련의 처리 과정이 끝나면 메시지 큐에 처리한 결과를 전송하고 다시 sleep상태로 돌아가게 된다. (그림 3)에는 사용자 프로세스와 단말기 서버 프로세스 사이의 메시지 교환및 동기화 과정을 나타내고 있다.

서버 프로세스는 사용자 프로세스의 읽기/쓰기 요청에 따라 다음 연산을 수행한다.

1) Read( ) 호출에 대한 입력 연산
   ① 입력된 데이타를 raw 큐에 쓴다. 문자를 반향(echo)시키기 위해서 입력 데이타를 출력 큐에 쓴다.
   ② Raw 큐로부터 한 바이트를 가져온다.
   ③ 일련의 특수 문자(backspace, erase, kill 등)를 지정된 방식으로 처리하여 표준 큐에 넣는다.
   ④ 지정한 문자 수 만큼 ①, ②, ③ 과정을 반복한다.
   ⑤ 입력 문자들은 메시지로 만들어져 사용자 프로세스로 전송된다.
2) Write( ) 호출에 대한 출력 연산
   ① 사용자 프로세스의 출력 데이타를 출력 큐로 넘겨받는다.
   ② 특수 문자가 처리되면서 출력 데이타를 출력 큐로 보내진다.
   ③ 지정한 문자 수 만큼 ①, ②를 반복한다.

제시된 단말기 서버에서 단말기마다 하나의 서버 프로세스를 두는 경우, 혹은 호스트 시스템에 단 하나의 서버 프로세스만 두는 경우에 기존 입출력 연산과 비교하여 사용자 프로세스와 서버 프로세스로의 두번의 문맥 교환이 부가적으로 필요하게 된다. 기존에는 사용자 프로세스가 입출력 연산을 수행하면 사용자 프로세스는 사용자 모드에서 커널 모드로 전이되고, 커널이 해당 테이타를 U BLOCK 정보를 이용하여 처리해 주었다. 반면, 본 논문에서 구현된 단말기 서버는 먼저 사용자 프로세스가 단말기 서버를 통하여 입출력 처리를 요청하기 때문에 입출력 요청시 사용자 프로세스로부터 서버로, 입출력 완료시 서버로부터 사용자 프로세스로의 문맥 교환이 더 필요하다. 그리고, 사용자 프로세스가 입출력 데이타를 스터브로부터 서버에게 전달해 주면 서버가 데이타를 처리하여 해당 단말기로 출력한다.

3.3 예 : 입력 연산에서의 백스페이스 처리

유닉스 시스템의 표준 모드에서 "백스페이스" 특수 문자는 백스페이스, 공백, 백스페이스로 반향(echo)되고, 입력 큐로는 한 문자가 삭제되어 입력된다. 그러나, 두 바이트 한글 코드 체계를 효율적으로 지원하기 위해서는 표준 모드의 백스페이스 처리 루틴를 수행하면 안된다. 왜냐하면, 2-바이트 한글 코드 체계에서의 한 문자는 두개의 칸(column)을 차지하기 때문에 백스페이스가 입력되면 두 바이트가 삭제 되야 한다. 또한, n (n≥2)바이트를 사용하는 또 다른 한글 코드 체계의 입력 연산 일 경우에는 백스페이스 처리시 n개의 바이트를 삭제하도록 해야 한다. 따라서, 제시된 서버 프로세스에서는 백스페이스 문자 처리 루틴을 구현하기 위하여 (그림 4)와 같이 두 바이트 한글 코드에서 백스페이스 문자 처리를 지원하도록 한다.

```
/*여기서 ff는 해당 단말기에 대한 화일 디스크립터다 */
read(ff,&r,1);                          /*raw 큐로부터 한 바이트 가져옴*/
if(r == BACK_SPACE) {
  write(ff,&r,1);                       /* 커서 왼쪽으로 이동 */
    r = ' ';
    write(ff,&r,1);                     /* 백스페이스 반향(echo) 시킴 */
    r = BACK_SPACE;
    write(ff,&r,1);                     /* 커서 왼쪽으로 이동 */
    bp--;                               /* 내부 버퍼에서 한 바이트 삭제 */
    if(buffer[bp] & 0x80) {             /* 한글 코드인가 ?*/
        r = BACK_SPACE;
        write(ff,&r,1);
        r = ' ';
        write(ff,&r,1);
        r = BACK_SPACE;
        write(ff,&r,1);
        bp--;                           /* 내부 버퍼에서 한 바이트 삭제 */
    }
}else {
  write(ff,&r,1);                       /* 문자 반향(echo) 시킴 */
  buffer[bp++] = r;                     /* 내부 버퍼에 한 바이트 삽입 */
}
```

(그림 4) 2 바이트 한글 코드에서 백스페이스 문자 처리
(Fig. 4) A process for backspace character in 2 byte KLCS

4. 성능 평가

이 장에서는 본 논문에서 구현한 단말기 인터페이스와 기존 단말기 인터페이스와의 성능 차이를 측정한다. 사용자가 단말기로 입력한 문자를

처리하여 반향(echo)시키는데 소요되는 시간은 사용자의 단말기 입력 속도에 비해 매우 빠르므로 단말기 서버를 통하여 발생할 수 있는 입력 연산의 시간 차이는 무시할 수 있지만 출력 연산은 무시할 수 없다. 이를 평가하기 위해서, 단말기 서버를 이용한 출력과 write시스템 호출에 의한 기존 단말기 출력과의 성능 차이를 측정하였다. 실험 환경으로는 SunOS와 Linux 시스템을 택했다. SunOS 5.3이 구동되는 Sparc워크스테이션에 ethernet으로 연결된 단말기와 Linux 1.2.8이 구동되는 펜티엄 PC에 직렬포트와 ethernet을 각각 연결한 386PC에서 측정하였다.

4.1 SunOS에 ethernet 으로 연결된 단말기에서의 성능

기존 시스템과 비교하여, 본 논문에서 구현한 단말기 서버는 메시지 교환 기법에 의한 데이타 교환과 동기화로 인해 성능이 저하되었다. 사용자 프로세스가 단말기 서버에게 16바이트에서 512바이트 크기의 데이타를 전송하였다. Write 시스템 호출을 이용한 출력과 서버를 이용한 출력의 시간 차이는 0.57 ~ 2.32 μsec이다. 그 이유는 사용자 프로세스와 단말기 서버 사이의 메시지 전송과 스케쥴링, 문맥 교환에 걸리는 시간이 포함되어 있기 때문이다. 이 수치는 1000번 측정된 평균값을 산출한 것이다. 〈표 1〉에 나타난 실험 결과는 ethernet의 전송 속도와 전송되는 패킷의 갯수등에 따라서 영향을 받을 수 있다. 이와같은 차이는 사용자의 반응 속도에 비추어 볼 때 사실상 크지 않으며 사용자 수준에서 구현한 단말기 서버의 타당성을 보여준다.

〈표 1〉 출력 처리 시간(μ/buffer)
〈Table 1〉 Output processing time

(바이트수/버퍼)	Write( )시스템 호출에 의함	단말기 서버에 의함
16	2.08	2.76
32	6.31	6.88
64	11.89	12.69
128	23.21	24.37
256	48.13	49.11
512	95.55	97.87

4.2 Linux에 ethernet으로 연결된 단말기에서의 성능

이번 실험은 Linux 1.2.8이 구동되는 펜티엄 PC에 ethernet으로 연결된 386PC에서 출력 시간을 측정하였다. Write시스템 호출과 단말기 서버을 500번씩 수행하여 얻은 결과를 평균내어 〈표 2〉에 나타내었다. Write시스템 호출과 단말기 서버사이의 출력 시간 차이는 0.086μsec ~ 1.214μsec이다. 이 시간 차이는 메시지 전송과 스케쥴링, 문맥 교환에 걸리는 시간이 포함되어 있기 때문이며, 사용자의 반응 속도를 비추어 볼 때 사실상 크지 않은 차이다. 〈표 2〉에 나타난 실험 결과는 ethernet의 전송 속도와 전송되는 패킷 수 등에 따라서 영향을 받을 수 있다.

〈표 2〉 출력 처리 시간(μ/buffer)
〈Table 2〉 Output processing time

(바이트수/버퍼)	Write( )시스템 호출에 의함	단말기 서버에 의함
16	0.99	1.84
32	1.548	2.762
64	2.592	2.838
128	17.554	18.244
256	46.694	46.780
512	104.036	104.326
1024	216.89	217.984

4.3 직렬 포트로 연결된 단말기에서의 성능

이번 실험은 Linux 1.2.8이 구동되는 펜티엄 PC에 9600BPS(Bit Per Second)의 전송 속도를 갖는 직렬포트로 연결한 386PC에서 출력 시간을 측정하였다. Write 시스템 호출과 단말기 서버를 이용해서 출력하는 경우에 소요되는 시간을 각각 500번씩 수행하여 얻은 결과를 평균내어 〈표 3〉에 나타내었다.

Write 시스템 호출과 단말기 서버를 이용한 출력 시간 차이는 0.038 ~ 41.976 μsec이다. 적은 양의 데이타일 경우에는 단말기 서버와 write시스템 호출을 이용하여 출력되는 시간은 많은 차이를 나타내었다. 그러나, 많은 양의 데이타 크기를 출력 할 경우에는 차이가 거의 나질 않았다. 그 이유는 UART(Universal Asynchronous Receiver Transmitter)의 전송 속도가 한정되어

있으므로 적은 양의 데이타일 경우에는 데이타 병목이 생기지 않지만 많은 양의 데이타가 전송될 때는 데이타 병목이 발생할 수 있기 때문이다. 〈표 3〉의 실험결과는 UART(Universal Asynchronous Receiver Transmitter)의 전송 속도와 전송되는 패킷의 갯수, 교통량, 출력 버퍼의 크기 등에 따라서 영향을 받을 수 있으므로 상황에 따라 달라질 수 있다.

〈표 3〉 출력 처리 시간(μ/buffer)
〈Table 3〉 Output processing time

(바이트수/버퍼)	Write( )시스템 호출에 의함	단말기 서버에 의함
16	0.228	2.704
32	2.458	2.896
64	2.81	3.21
128	3.606	27.970
256	5.252	47.288
512	6.552	6.560
1024	13.13	13.168

이상의 결과에서 사용자 수준에서 구현한 단말기 서버를 이용하여 출력할 경우, 약간의 성능 저하가 있다. 최소 성능 차이는 수 μsec이며, 이는 두번의 문맥 교환과 메시지 전송에 걸리는 시간이다.

5. 결 론

제안한 사용자 수준의 단말기 구동기 모델은 기존 유닉스 시스템의 커널 내부에 있는 단말기 구동기 루틴을 수정하지 않고도 다중 바이트 코드 체계를 효과적으로 지원할 수 있도록 해준다. 또한, 단말기 구동기 모듈을 두개의 그룹으로 분류하여 시스템 설계자로 하여금 단말기 구동기를 효율적으로 개발하게 해주었으며, 커널의 이식성을 높여 주었다.

성능 평가를 통해, 단말기의 출력 연산에 대한 응답 시간이 UART(Universal Asynchronous Receiver Transmitter)전송율에 제한 받는 다는 것을 보였다. 또한, ethernet에 연결되어 있는 경우도 기존 write시스템 호출과 비교할 때 수 μ sec 정도의 성능 차이를 보였다. 이는 사용자의 반응 속도에 비추어 볼 때 심각하지 않다. 그러므로, 단말기 구동기를 사용자 수준의 서버 프로세스로 설계하는 것은 타당하다고 본다. 이 기법은 유닉스 시스템이 실행되고 있는 동안에도 단말기 구동기를 동적으로 첨가할 수 있다. 또한, 유연성을 제공하기 위해서 다른 장치 구동기에도 본 논문에서 제안한 기법을 적용할 수 있을 것이다. 향후, 메시지 전송으로 인한 성능 저하를 없애기 위한 연구가 요구된다.

참 고 문 헌

[1] Alessandro Forin, David Golub, Brian Bershad, "An I/O System for Mach 3.0," Proceedings of the USENIX Mach Symposium, 1991, PP.163-176.

[2] David B.Golub, Guy G. Sotomayor, Jr, Freeman L.Lawson III, "An Architecture for Device Drivers Executing as User-Level Tasks," Proceedings of the USENIX Mach III Symposium, 1993,PP. 153-171.

[3] Hiromichi K. and Richard M. "A UNIX System V STREAMS TTY implementation for Multiple Language Processing," Proceedings of the Summer 1987 USENIX Conference, 1987, PP.323-336.

[4] Mark D. Compbell and Tracy R. Edmonds, "TOWER STREAM-Based TTY : Architecture and Implementation," Proceedings of the Summer 1989 USENIX Conference, 1989, PP. 15-27.

[5] Maurice J.Bach, "The Design of the UNIX Operating System," Prentice-Hall, international Editions, 1992.

[6] W.Richard Stevens, "Unix network programming," Prentice Hall, 1990.

[7] Keith Haviland and Ben Salama, "Unix system programmming," 1987.

[8] 임성락, "동적 재구성이 가능한 운영 체제의 기본 요소", 박사 학위 논문, 서울 대학교, 1992.

이 형 주

1994년 호서대학교 컴퓨터공학과 졸업
1994년~현재 호서대학교 컴퓨터공학과 석사과정
관심분야 : 운영체제, 시스템 소프트웨어, 컴퓨터네트워크

임 성 락

1979년 서강대학교 전자공학과 졸업
1983년 서울대학교 컴퓨터공학과 석사
1983년~90년 금성 반도체(주) 연구소
1992년 서울대학교 컴퓨터공학과 박사
1992년~현재 호서대학교 컴퓨터공학과 조교수
1995년~현재 호서대학교 전자계산소 소장
관심분야 : 운영체제, 실시간처리시스템, 분산처리시스템