1766년 11월 6일

간단한 유닉스 계열의 교육용 운영 체제

러스 콕스
프란스 카슈크
로버트 모리스

xv6-book@pdos.csail.mit.edu

2018년 9월 4일 기준 초안

내용물

서문 및 감사의 말

이것은 운영 체제 수업을 위한 초안 텍스트입니다. xv6이라는 예제 커널을 연구하여 운영 체제의 주요 개념을 설명합니다. xv6은 Dennis Ritchie와 Ken Thompson의 Unix Version 6(v6)을 다시 구현한 것입니다. xv6은 v6의 구조와 스타일을 느슨하게 따르지만 x86-

기반 멀티프로세서.

텍스트는 xv6의 소스 코드와 함께 읽어야 합니다. 이 접근 방식은 영감을 받았습니다.
John Lions의 UNIX 6판 주석(Peer to Peer Communications; IS-BN: 1-57398-013-7; 1판(2000년 6월 14일)) 참조). https://pdos.csail.mit.edu/6.828 참조
xv6를 사용한 여러 가지 실습 숙제를 포함하여 v6 및 xv6에 대한 온라인 리소스에 대한 포인터입니다.

우리는 이 텍스트를 MIT의 운영 체제 수업인 6.828에서 사용했습니다. 우리는 xv6에 직간접적으로 기여한 6.828의 교수진, 조교, 학생들에게 감사드립니다. 특히 Austin Clements와 Nickolai에게 감사드립니다.

Zeldovich. 마지막으로, 텍스트에 버그를 이메일로 보내주신 분들께 감사드리고 싶습니다.
개선을 위한 제안: Abutalib Aghayev, Sebastian Boehm, Anton Burtsev,
라파엘 카르발류, 라싯 에스키초글루, 컬러 퍼지, 주세페, 타오 궈, 로버트 힐더맨, 볼프강 켈러, 오스틴 리우, 파반 마담세티, 야체크 마시울라니에츠, 마이클
맥콘빌, 미겔그비에이라, 마크 모리시, 해리 팬, 아스카르 사핀, 살만 샤, 루스란 사브첸코, 파벨 슈추르코, 워렌 투미, 티프크다, 주창웨이.

오류를 발견하거나 개선에 대한 제안 사항이 있는 경우 Frans에게 이메일을 보내주십시오.
Kaashoek 및 Robert Morris(kaashoek,rtm@csail.mit.edu).

챕터 0

운영 체제 인터페이스

운영 체제의 역할은 여러 프로그램 간에 컴퓨터를 공유하는 것입니다.
그리고 하드웨어만으로 지원하는 것보다 더 유용한 서비스 세트를 제공합니다.
운영 체제는 저수준 하드웨어를 관리하고 추상화하므로 예를 들어,
워드 프로세서는 어떤 유형의 디스크 하드웨어가 사용되고 있는지에 대해 걱정할 필요가 없습니다.
사용됩니다. 또한 여러 프로그램 간에 하드웨어를 공유하여 동시에 실행(또는 실행되는 것처럼 보임)할
수 있습니다. 마지막으로 운영 체제는 다음을 위한 제어된 방법을 제공합니다.
프로그램 간에 상호 작용이 가능하여 데이터를 공유하거나 함께 작업할 수 있습니다.

운영체제는 인터페이스를 통해 사용자 프로그램에 서비스를 제공합니다.
좋은 인터페이스를 디자인하는 것은 어려운 일로 밝혀졌습니다. 한편으로는 우리는
인터페이스는 간단하고 좁아야 합니다. 그렇게 하면 구현을 올바르게 하는 것이 더 쉬워지기 때문입니다.
반면에 우리는 많은 정교한
기능을 애플리케이션에 적용합니다. 이러한 긴장을 해결하는 비결은 인터페이스를 설계하는 것입니다.
많은 일반성을 제공하기 위해 결합할 수 있는 몇 가지 메커니즘에 의존합니다.

이 책은 단일 운영 체제를 구체적인 예로 사용하여 운영 체제 개념을 설명합니다. 해당 운영 체제
xv6는 Ken Thompson과 Dennis Ritchie의 Unix 운영 체제에서 도입한 기본 인터페이스를 제공하
고 Unix의 내부 디자인을 모방합니다. Unix는 메커니즘이

잘 결합되어 놀라운 수준의 일반성을 제공합니다. 이 인터페이스는 너무
현대 운영 체제(BSD, Linux, Mac OS X, Solaris 등)의 성공은
덜한 정도로, Microsoft Windows는 Unix와 유사한 인터페이스를 가지고 있습니다. xv6 이해
는 이러한 시스템과 다른 여러 시스템을 이해하는 데 좋은 시작이 될 것입니다.

그림 0-1에서 보듯이, xv6는 실행 중인 프로그램에 서비스를 제공하는 특수 프로그램인 커널 의
전통적인 형태를 취합니다. 프로세스 라고 하는 각 실행 중인 프로그램은 명령어, 데이터, 스택을 포함하
는 메모리를 갖습니다. 명령어는 프로그램의 계산을 구현합니다. 데이터는 계산이 작동하는 변수입니다.
스택은 프로그램의 프로시저 호출을 구성합니다.

프로세스가 커널 서비스를 호출해야 하는 경우 프로시저 호출을 호출합니다.
운영 체제 인터페이스. 이러한 절차를 시스템 호출이라고 합니다. 시스템
호출이 커널에 들어간다. 커널은 서비스를 수행하고 반환한다. 따라서 프로세스는 사용자 공간 과 커널
공간 에서 번갈아 실행된다 .

커널은 CPU의 하드웨어 보호 메커니즘을 사용하여 각
사용자 공간에서 실행되는 프로세스는 자체 메모리에만 액세스할 수 있습니다. 커널은 다음을 실행합니다.
이러한 보호 기능을 구현하는 데 필요한 하드웨어 권한이 있는 사용자 프로그램
해당 권한 없이 실행하십시오. 사용자 프로그램이 시스템 호출을 호출하면
하드웨어는 권한 수준을 높이고 미리 준비된 기능을 실행하기 시작합니다.
핵심.

커널이 제공하는 시스템 호출의 컬렉션은 사용자 프로그램이 보는 인터페이스입니다. xv6 커널은
Unix가 제공하는 서비스와 시스템 호출의 하위 집합을 제공합니다.
커널은 전통적으로 다음을 제공합니다. 그림 0-2는 xv6의 모든 시스템 호출을 나열합니다.

프로세스

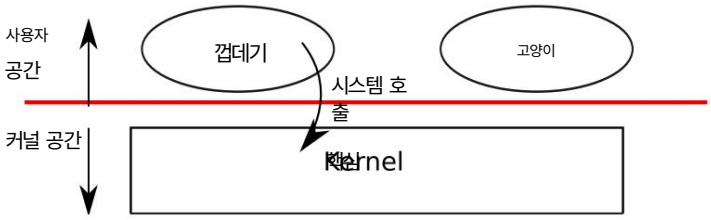


그림 0-1. 커널과 두 개의 사용자 프로세스.

껍데기
타임쉐어
pid+code
fork+code 자
식 프로세스 부모 프
로세스 fork+code
종료+코드 대
가+코드

이 장의 나머지 부분에서는 xv6의 서비스(프로세스, 메모리, 파일 설명자, 파이프, 파일 시스템)를 개략적으로 설명하고 코드 조각과 기존 Unix 유사 시스템의 기본 사용자 인터페이스인 셸 이 이를 사용하는 방법에 대한 논의를 통해 이를 설명합니다. 셸이 시스템 호출을 사용하는 방식은 이들이 얼마나 신중하게 설계되었는지를 보여줍니다.

쉘은 사용자로부터 명령을 읽어서 실행하는 일반적인 프로그램입니다. 쉘이 커널의 일부가 아니라 사용자 프로그램이라는 사실은 시스템 호출 인터페이스의 힘을 보여줍니다. 쉘에는 특별한 것이 없습니다. 또한 쉘을 쉽게 교체할 수 있다는 것을 의미합니다. 결과적으로 현대의 Unix 시스템은 다양한 쉘을 선택할 수 있으며 각각 고유한 사용자 인터페이스와 스크립팅 기능이 있습니다. xv6 쉘은 Unix Bourne 쉘의 본질을 간단하게 구현한 것입니다. 구현은 (8550)줄에서 찾을 수 있습니다.

프로세스와 메모리

xv6 프로세스는 사용자 공간 메모리(명령어, 데이터 및 스택)와 커널에 비공개인 프로세스별 상태로 구성됩니다. Xv6는 프로세스를 시간 공유 할 수 있습니다 . 실행을 기다리는 프로세스 집합 간에 사용 가능한 CPU를 투명하게 전환합니다. 프로세스가 실행 중이 아닐 때 xv6는 CPU 레지스터를 저장하여 다음에 프로세스를 실행할 때 복원합니다. 커널은 각 프로세스에 프로세스 식별자 또는 pid를 연결합니다.

프로세스는 fork 시스템 호출을 사용하여 새 프로세스를 만들 수 있습니다. Fork는 호출 프로세스인 부모 프로세스와 정확히 동일한 메모리 내용을 가진 자식 프로세스라는 새 프로세스를 만듭니다. Fork는 부모와 자식 모두에서 반환합니다.
부모에서 fork는 자식의 pid를 반환하고 자식에서 0을 반환합니다. 예를 들어, 다음 프로그램 조각을 고려하세요.

```
int pid = fork(); if(pid > 0)
{ printf("부모: 자식
    =%d\n", pid); pid = wait(); printf("자식 %d가 완료되었
    습니다\n", pid); }
    else if(pid == 0){ printf("자식: 종료\n"); exit(); } else
{ printf("포크 오류\n");




}
```

종료 시스템 호출은 호출 프로세스가 실행을 중지하고 메모리 및 열린 파일과 같은 리소스를 해제하도록 합니다. 대기 시스템 호출은 pid를 반환합니다.

시스템 호출	설명
포크()	프로세스를 생성하다
출구()	현재 프로세스를 종료합니다
기다리다()	자식 프로세스가 종료될 때까지 기다리세요
kill(pid)	프로세스 pid 종료
getpid()	현재 프로세스의 pid를 반환합니다.
수면(n)	n 시계 똑딱거림 동안 잠을 자다
exec(파일 이름, *argv)	파일을 로드하고 실행합니다
sbrk(명)	프로세스의 메모리를 n바이트만큼 늘립니다.
open(파일 이름, 플래그)	파일을 엽니다. 플래그는 읽기/쓰기를 나타냅니다.
읽기(fd, 버퍼, n)	열려 있는 파일에서 n바이트를 buf로 읽습니다.
쓰기(fd, buf, n)	열려 있는 파일에 n바이트를 씁니다.
닫기(fd)	열려있는 파일 fd를 릴리스합니다
중복(fd)	중복 fd
파이프(p)	파이프를 생성하고 p에서 fd를 반환합니다.
chdir(디렉토리이름)	현재 디렉토리 변경
mkdir(디렉토리이름)	새로운 디렉토리를 생성하세요
mknod(name, major, minor)	장치 파일을 생성합니다.
fstat(fd)	열려 있는 파일에 대한 정보 반환
링크(f1, f2)	파일 f1에 대해 다른 이름(f2)을 만듭니다.
unlink(파일 이름)	파일 제거

그림 0-2. Xv6 시스템 호출

현재 프로세스의 종료된 자식; 호출자의 자식 중 어느 것도 종료되지 않은 경우 대기 대기+코드
그렇게 할 때까지 기다립니다. 예에서 출력 줄은 다음과 같습니다.

```
부모: 자식=1234
아이: 나가다
```

printf+코드
대기+코드
실행+코드
실행+코드

부모 또는 자식 중 누가 목적지에 도달하는지에 따라 어느 순서로든 나올 수 있습니다.
먼저 printf 호출을 합니다. 자식이 종료된 후 부모의 대기가 반환되어 부모가
인쇄

```
부모 : 자식 1234 완료
```

자식은 처음에는 부모와 동일한 기억 내용을 가지고 있지만 부모는
그리고 자식은 다른 메모리와 다른 레지스터로 실행되고 있습니다. 한 쪽에서 변수를 변경해도 다른 쪽에는 영향을 미치지 않습니다. 예를 들어, wait의 반환 값이 다음과 같은 경우
부모 프로세스의 pid에 저장되므로 자식 프로세스의 pid 변수는 변경되지 않습니다.
자식의 pid 값은 여전히 0입니다.

exec 시스템 호출은 호출 프로세스의 메모리를 새 메모리로 교체합니다.
파일 시스템에 저장된 파일에서 로드된 이미지. 파일에는 특정 형식이 있어야 하며, 이는 파일의 어느 부분이 명령어를 보관하고 어느 부분이 데이터인지를 지정합니다.
시작할 명령어 등 xv6은 2장에서 논의하는 ELF 형식을 사용합니다.
더 자세히. exec가 성공하면 호출 프로그램으로 돌아가지 않습니다. 대신,
파일에서 로드된 명령어는 선언된 진입점에서 실행을 시작합니다.
ELF 헤더. Exec는 두 개의 인수를 취합니다. 실행 파일이 들어 있는 파일의 이름
그리고 문자열 인수의 배열. 예를 들어:

```
char *argv[3];

argv[0] = "에코"; argv[1]
= "안녕하세요"; argv[2] = 0;
exec("/bin/
echo", argv); printf("exec 오류\n");
```


이 조각은 호출하는 프로그램을 인수 목록 echo hello로 실행되는 프로그램 /bin/echo의 인스턴스로 대체합니다. 대부분의 프로그램은 일반적으로 프로그램 이름인 첫 번째 인수를 무시합니다.

xv6 셸은 위의 호출을 사용하여 사용자를 대신하여 프로그램을 실행합니다. 셸의 기본 구조는 간단합니다. main (8701)을 참조하세요. 메인 루프는 getcmd를 사용하여 사용자로부터 한 줄의 입력을 읽습니다. 그런 다음 fork를 호출하여 셸 프로세스의 사본을 만듭니다. 부모는 wait를 호출하는 반면 자식은 명령을 실행합니다. 예를 들어, 사용자가 셸에 ''echo hello''를 입력한 경우 runcmd는 인수로 ''echo hello''를 사용하여 호출됩니다. runcmd (8606) 는 실제 명령을 실행합니다. ''echo hello''의 경우 exec (8626) 를 호출합니다 . exec가 성공하면 자식은 runcmd 대신 echo에서 명령어를 실행합니다. 어느 시점에서 echo는 exit를 호출하여 부모가 main (8701)의 wait에서 반환되도록 합니다. fork와 exec가 단일 호출에서 결합되지 않는 이유가 궁금할 수 있습니다. 나중에 프로세스를 만들고 프로그램을 로드하기 위한 별도의 호출이 영리한 설계라는 것을 알게 될 것입니다.

Xv6는 대부분의 사용자 공간 메모리를 암묵적으로 할당합니다. fork는 부모 메모리의 자식 복사본에 필요한 메모리를 할당하고 exec는 실행 파일을 보관하기에 충분한 메모리를 할당합니다. 런타임에 더 많은 메모리가 필요한 프로세스(아마도 malloc의 경우)는 sbrk(n)을 호출하여 데이터 메모리를 n바이트만큼 늘릴 수 있습니다. sbrk는 새 메모리의 위치를 반환합니다.

Xv6는 사용자 개념이나 한 사용자를 다른 사용자로부터 보호하는 개념을 제공하지 않습니다.
유닉스 용어로, 모든 xv6 프로세스는 루트로 실행됩니다.

I/O 및 파일 설명자 파일 설명자는 프로세스

가 읽거나 쓸 수 있는 커널 관리 객체를 나타내는 작은 정수입니다. 프로세스는 파일, 디렉토리 또는 장치를 열거나 파이프를 만들거나 기존 설명자를 복제하여 파일 설명자를 얻을 수 있습니다. 단순화를 위해 파일 설명자가 참조하는 객체를 종종 ''파일''이라고 합니다. 파일 설명자 인터페이스는 파일, 파이프 및 장치 간의 차이점을 추상화하여 모두 바이트 스트림처럼 보이게 합니다.

내부적으로 xv6 커널은 파일 설명자를 프로세스별 테이블의 인덱스로 사용하므로 모든 프로세스는 0에서 시작하는 파일 설명자의 개인 공간을 갖습니다. 관례에 따라 프로세스는 파일 설명자 0(표준 입력)에서 읽고, 파일 설명자 1(표준 출력)에 출력을 쓰고, 파일 설명자 2(표준 오류)에 오류 메시지를 씁니다. 알다시피, 셸은 이 관례를 활용하여 I/O 리디렉션과 파이프라인을 구현합니다. 셸은 항상 세 개의 파일 설명자 (8707)를 열어두도록 보장하는데, 이는 기본적으로 콘솔의 파일 설명자입니다.

읽기 및 쓰기 시스템 호출은 파일 설명자로 명명된 열린 파일에서 바이트를 읽고 바이트를 씁니다. 호출 read(fd, buf, n)은 최대 n바이트를 읽습니다.

파일 설명자 fd는 buf에 복사하고 읽은 바이트 수를 반환합니다. 파일을 참조하는 각 파일 설명자에는 연관된 오프셋이 있습니다. Read는 현재 파일 오프셋에서 데이터를 읽고 그 오프셋을 읽은 바이트 수만큼 앞으로 이동합니다. 후속 읽기는 첫 번째 읽기에서 반환된 바이트 다음의 바이트를 반환합니다.

더 이상 읽을 바이트가 없으면 read는 0을 반환하여 파일의 끝을 알립니다.

write(fd, buf, n) 호출은 buf에서 파일 설명자 fd로 n바이트를 쓰고 쓰여진 바이트 수를 반환합니다. n바이트 미만은 오류가 발생할 때만 쓰여집니다. read와 마찬가지로 write는 현재 파일 오프셋에 데이터를 쓰고 그 오프셋을 쓰여진 바이트 수만큼 앞으로 이동합니다. 각 쓰기는 이전 쓰기가 중단된 곳에서 시작합니다.

다음 프로그램 조각(cat의 본질을 형성함)은 표준 입력에서 표준 출력으로 데이터를 복사합니다. 오류가 발생하면 표준 오류에 메시지를 씁니다.

```
char 버퍼[512]; int n;


for(;;){ n =
    read(0, buf, sizeof buf); if(n == 0) break; if(n
    < 0){ fprintf(2,
        "읽기 오
    류\n"); exit();


    }
    if(write(1, buf, n) != n){
        fprintf(2, "쓰기 오류\n"); exit();

    }
}
```

코드 조각에서 주의해야 할 중요한 점은 cat이 파일, 콘솔 또는 파이프에서 읽고 있는지 모른다는 것입니다. 마찬가지로 cat은 콘솔, 파일 또는 다른 곳에 인쇄하고 있는지 모릅니다. 파일 설명자를 사용하고 파일 설명자 0이 입력이고 파일 설명자 1이 출력이라는 관례를 사용하면 cat을 간단하게 구현할 수 있습니다.

close 시스템 호출은 파일 디스크립터를 해제하여 미래의 open, pipe 또는 dup 시스템 호출(아래 참조)에서 재사용할 수 있도록 합니다. 새로 할당된 파일 디스크립터는 항상 현재 프로세스의 가장 낮은 번호의 사용되지 않은 디스크립터입니다.

파일 설명자와 포크가 상호 작용하여 I/O 리디렉션을 쉽게 구현할 수 있습니다.
Fork는 부모의 파일 설명자 테이블을 메모리와 함께 복사하여 자식이 부모와 정확히 같은 열린 파일로 시작합니다. 시스템 호출 exec는 호출 프로세스의 메모리를 대체하지만 파일 테이블은 보존합니다. 이 동작을 통해 셸은 포킹하고 선택한 파일 설명자를 다시 연 다음 새 프로그램을 exec하여 I/O 리디렉션을 구현할 수 있습니다. 다음은 셸이 cat < input.txt 명령에 대해 실행하는 코드의 단순화된 버전입니다.

```
char *argv[2];

argv[0] = "고양이";
argv[1] = 0; fork()
== 0인 경우 { 0을 닫습니다.
    "입력.txt"를 엽
    니다(O_RDONLY); exec("고양이", argv);

}
```

자식이 파일 설명자 0을 닫은 후, open은 새로 열린 input.txt에 대해 해당 파일 설명자를 사용하도록 보장됩니다. 0은 사용 가능한 가장 작은 파일 설명자입니다. 그런 다음 Cat은 input.txt를 참조하는 파일 설명자 0(표준 입력)으로 실행합니다.

xv6 셸에서 I/O 리디렉션을 위한 코드는 정확히 이런 방식으로 작동합니다 (8630). 이 코드에서 셸이 이미 자식 셸을 포크했고 runcmd가 exec를 호출하여 새 프로그램을 로드한다는 점을 다시 상기해 보세요. 이제 fork와 exec가 별도의 호출인 것이 왜 좋은지 분명해졌을 것입니다. 왜냐하면 별개라면 셸이 자식을 포크하고 자식에서 open, close, dup를 사용하여 표준 입력 및 출력 파일 설명자를 변경한 다음 exec를 수행할 수 있기 때문입니다. exec되는 프로그램(예에서는 cat)을 변경할 필요가 없습니다. fork와 exec가 단일 시스템 호출로 결합된 경우 셸이 표준 입력 및 출력을 리디렉션하려면 다른(아마도 더 복잡한) 체계가 필요하거나 프로그램 자체가 I/O를 리디렉션하는 방법을 이해해야 합니다.

fork가 파일 디스크립터 테이블을 복사하지만, 각 기본 파일 오프셋은 부모와 자식 간에 공유됩니다. 다음 예를 고려하세요.

```
fork() == 0이면 { write(1,
    "안녕하세요 ", 6); 종료(); } 그렇지 않으
    면 { wait();
write(1, "세
    계\n", 6);

}
```

이 조각의 끝에서 파일 설명자 1에 첨부된 파일에는 데이터 hello world가 포함됩니다. 부모의 쓰기(wait 덕분에 자식이 완료된 후에만 실행됨)는 자식의 쓰기가 중단된 곳에서 시작합니다. 이 동작은 (echo hello; echo world) >output.txt와 같은 셸 명령 시퀀스에서 순차적인 출력을 생성하는 데 도움이 됩니다.

dup 시스템 호출은 기존 파일 디스크립터를 복제하여 동일한 기본 I/O 객체를 참조하는 새 디스크립터를 반환합니다. 두 파일 디스크립터는 fork로 복제된 파일 디스크립터와 마찬가지로 오프셋을 공유합니다. 이것은 hello world를 파일에 쓰는 또 다른 방법입니다.

```
fd = dup(1);
write(1, "안녕하세요 ", 6); write(fd,
"세계\n", 6);
```

두 파일 설명자는 fork 및 dup 호출 시퀀스에 의해 동일한 원본 파일 설명자에서 파생된 경우 오프셋을 공유합니다. 그렇지 않으면 파일 설명자는 동일한 파일에 대한 open 호출에서 발생한 경우에도 오프셋을 공유하지 않습니다. Dup는 셸을 허용합니다.

다음과 같은 명령을 구현하려면: ls 기존 파일 비기존 파일 > tmp1 2>&1. 2>&1은 셸에 명령에 디스크립터 1의 중복인 파일 디스크립터 2를 제공하라고 지시합니다. 기존 파일의 이름과 존재하지 않는 파일의 오류 메시지는 모두 tmp1 파일에 표시됩니다. xv6 셸은 오류 파일 디스크립터에 대한 I/O 리디렉션을 지원하지 않지만 이제 구현 방법을 알게 되었습니다.

파일 설명자는 연결된 대상의 세부 정보를 숨기기 때문에 강력한 추상화입니다. 파일 설명자 1에 쓰는 프로세스는 파일, 콘솔과 같은 장치 또는 파이프에 쓰는 것일 수 있습니다.

파이프

파이프 는 프로세스에 노출된 작은 커널 버퍼로, 하나는 읽기용이고 다른 하나는 쓰기용인 파일 설명자 쌍입니다. 파이프의 한쪽 끝에 데이터를 쓰면 파이프의 다른 쪽 끝에서 해당 데이터를 읽을 수 있습니다. 파이프는 프로세스가 통신할 수 있는 방법을 제공합니다.

다음 예제 코드는 표준 입력을 파이프의 읽기 쪽에 연결하여 wc 프로그램을 실행합니다.

```
int p[2]; char
*argv[2];

argv[0] = "wc"; argv[1]
= 0;

파이프(p);
fork() == 0인 경우 { 0을 닫습
    니다.
    dup(p[0]); p[0]
    을 닫습니다. p[1]을
    닫습니다. exec("/
    bin/wc", argv); } 그렇지 않은 경우
{ p[0]을 닫습
    니다. p[1], "안녕하
    세요 세계\n", 12를 씁니다. p[1]을 닫습니다.

}
```

이 프로그램은 파이프를 호출하는데, 파이프는 새로운 파이프를 만들고 배열 p에 읽기 및 쓰기 파일 기술자를 기록합니다. fork 후, 부모와 자식 모두 파이프를 참조하는 파일 기술자를 갖습니다. 자식은 읽기 끝을 파일 기술자 0에 복제하고, p에 있는 파일 기술자를 닫고, wc를 실행합니다. wc가 표준 입력에서 읽을 때 파이프에서 읽습니다. 부모는 파이프의 읽기 측면을 닫고, 파이프에 쓰고, 쓰기 측면을 닫습니다.

데이터가 없으면 파이프에서 읽기는 데이터가 쓰여지거나 쓰기 끝을 참조하는 모든 파일 설명자가 닫힐 때까지 기다립니다. 후자의 경우, 데이터 파일의 끝에 도달한 것처럼 read는 0을 반환합니다. 새 데이터가 도착할 수 없을 때까지 read가 차단된다는 사실은 자식이 위의 wc를 실행하기 전에 파이프의 쓰기 끝을 닫는 것이 중요한 한 가지 이유입니다. wc의 파일 설명자 중 하나가 파이프의 쓰기 끝을 참조하는 경우 wc는 결코 파일 끝을 보지 못할 것입니다.

뿌리
경로
현재 디렉토리

xv6 셸은 위의 코드 (8650)와 비슷한 방식으로 grep fork sh.c | wc -l과 같은 파이프라인을 구현합니다. 자식 프로세스는 파이프라인의 왼쪽 끝과 오른쪽 끝을 연결하는 파이프를 만듭니다. 그런 다음 파이프라인의 왼쪽 끝에 대해 fork와 runcmd를 호출하고 오른쪽 끝에 대해 fork와 runcmd를 호출하고 둘 다 완료될 때까지 기다립니다. 파이프라인의 오른쪽 끝은 파이프를 포함하는 명령(예: a | b | c)일 수 있으며, 이 명령은 두 개의 새 자식 프로세스(하나는 b에 대해, 다른 하나는 c에 대해)를 포크합니다. 따라서 셸은 프로세스 트리를 만들 수 있습니다. 이 트리의 잎은 명령이고 내부 노드는 왼쪽과 오른쪽 자식이 완료될 때까지 기다리는 프로세스입니다. 원칙적으로 내부 노드가 파이프라인의 왼쪽 끝을 실행하도록 할 수 있지만 올바르게 하면 구현이 복잡해집니다.

파이프는 임시 파일보다 더 강력하지 않은 것처럼 보일 수 있습니다. 파이프라인

```
에코 헬로 월드 | wc
```

파이프 없이도 구현할 수 있습니다.

```
echo hello world >/tmp/xyz; wc </tmp/xyz
```

파이프는 이 상황에서 임시 파일에 비해 적어도 네 가지 장점이 있습니다. 첫째, 파이프는 자동으로 스스로를 정리합니다. 파일 리디렉션을 사용하면 셸은 작업이 끝나면 /tmp/xyz를 조심해서 제거해야 합니다. 둘째, 파이프는 임의로 긴 데이터 스트림을 전달할 수 있는 반면, 파일 리디렉션은 모든 데이터를 저장할 디스크에 충분한 여유 공간이 필요합니다. 셋째, 파이프는 파이프라인 단계의 병렬 실행을 허용하는 반면, 파일 접근 방식은 두 번째 프로그램이 시작되기 전에 첫 번째 프로그램이 완료되어야 합니다. 넷째, 프로세스 간 통신을 구현하는 경우 파이프의 차단 읽기 및 쓰기는 파일의 비차단 의미론보다 효율적입니다.

파일 시스템

xv6 파일 시스템은 해석되지 않은 바이트 배열인 데이터 파일과 데이터 파일 및 다른 디렉토리에 대한 명명된 참조를 포함하는 디렉토리를 제공합니다. 디렉토리는 루트라는 특수 디렉토리에서 시작하여 트리를 형성합니다 . /a/b/c와 같은 경로 는 루트 디렉토리 /에 있는 a라는 디렉토리 내의 b라는 디렉토리 내에 있는 c라는 파일 또는 디렉토리를 참조합니다. /로 시작하지 않는 경로는 호출 프로세스의 현재 디렉토리를 기준으로 평가되며, chdir 시스템 호출로 변경할 수 있습니다. 이 두 코드 조각은 모두 동일한 파일을 엽니다(관련된 모든 디렉토리가 존재한다고 가정).

```
chdir("/a");
chdir("b");
open("c", O_RDONLY);

("/a/b/c", O_RDONLY)를 엽니다.
```

첫 번째 조각은 프로세스의 현재 디렉토리를 /a/b로 변경합니다. 두 번째 조각은 프로세스의 현재 디렉토리를 참조하지도 변경하지도 않습니다.

새 파일이나 디렉토리를 만드는 데는 여러 시스템 호출이 있습니다. mkdir은 새 디렉토리를 만들고, O_CREATE 플래그로 open은 새 데이터 파일을 만들고, mknod는 새 장치 파일을 만듭니다. 이 예는 세 가지 모두를 보여줍니다.

```
mkdir("/dir"); fd =
open("/dir/file", O_CREATE|O_WRONLY); fd를 닫습니다. mknod("/
console", 1, 1);
```

Mknod는 파일 시스템에 파일을 생성하지만 파일에는 내용이 없습니다. 대신 파일의 메타데이터는 이를 장치 파일로 표시하고 주요 및 부차 장치 번호(mknod에 대한 두 인수)를 기록하는데, 이는 커널 장치를 고유하게 식별합니다. 나중에 프로세스가 파일을 열면 커널은 읽기 및 쓰기 시스템 호출을 파일 시스템에 전달하는 대신 커널 장치 구현으로 전환합니다. fstat는 파일 설명자가 참조하는 객체에 대한 정보를 검색합니다. stat.h에 다음과 같이 정의된 구조체 stat를 채웁니다.

```
#define T_DIR 1 // 디렉토리 #define T_FILE 2 // 파
일 #define T_DEV 3 // 장치


구조체 통계 {
    short type; // 파일 유형 int dev; uint ino; short
    nlink; // 파일              // 파일 시스템의 디스크 장치
    에 대한 링크 수             // 아이노드 번호
    uint size; // 파일 크기(바이트) };
```

파일 이름은 파일 자체와 다릅니다. in-ode 라고 하는 동일한 기본 파일은 링크 라고 하는 여러 이름을 가질 수 있습니다 . 링크 시스템 호출은 기존 파일과 동일한 inode를 참조하는 또 다른 파일 시스템 이름을 만듭니다. 이 조각은 a와 b라는 이름의 새 파일을 만듭니다.

```
("a", O_CREATE|O_WRONLY)를 엽니다. ("a",
"b")를 연결합니다.
```

a에서 읽거나 쓰는 것은 b에서 읽거나 쓰는 것과 같습니다. 각 inode는 고유한 inode 번호로 식별됩니다. 위의 코드 시퀀스 후에 fstat의 결과를 검사하여 a와 b가 동일한 기본 콘텐츠를 참조한다는 것을 확인할 수 있습니다. 둘 다 동일한 inode 번호(ino)를 반환하고 nlink 카운트는 2로 설정됩니다.

unlink 시스템 호출은 파일 시스템에서 이름을 제거합니다. 파일의 inode와 해당 내용을 보관하는 디스크 공간은 파일의 링크 수가 0이고 해당 파일을 참조하는 파일 설명자가 없을 때만 해제됩니다. 따라서 다음을 추가합니다.

```
unlink("a");
```

마지막 코드 시퀀스로 넘어가면 inode와 파일 내용이 b로 접근 가능합니다. 더 나아가-
더,

```
fd = open("/tmp/xyz", O_CREATE|O_RDWR); unlink("/tmp/
xyz");
```

프로세스가 fd를 닫거나 종료될 때 정리될 임시 inode를 만드는 관용적인 방법입니다.

파일 시스템 작업을 위한 셸 명령은 mkdir, ln, rm 등과 같은 사용자 수준 프로그램으로 구현됩니다. 이 설계를 통해 누구나 셸을 확장할 수 있습니다.

새로운 사용자 레벨 프로그램을 추가하기만 하면 새로운 사용자 명령이 가능합니다. 이 계획은 돌이켜보면 당연한 것처럼 들리지만, 유닉스 당시 설계된 다른 시스템들은 종종 이런 명령을 셸에 내장했습니다(그리고 셸을 커널에 내장했습니다).

한 가지 예외는 쉘 (8716) 에 내장된 cd입니다 . cd는 쉘 자체의 현재 작업 디렉토리를 변경해야 합니다. cd를 일반 명령으로 실행하면
쉘은 자식 프로세스를 포크하고 자식 프로세스는 cd를 실행하고 cd는
자식의 작업 디렉토리를 변경합니다 . 부모(즉, 쉘)의 작업 디렉토리
변하지 않을 것이다.

현실 세계

''표준'' 파일 기술자, 파이프 및 편리한 셸을 결합한 Unix
이들에 대한 연산을 위한 구문은 범용 쓰기에 있어서 중요한 발전이었습니다.
재사용 가능한 프로그램. 이 아이디어는 Unix의 힘과 인기의 대부분을 담당한 ''소프트웨어 도구''의 전체 문화를 촉발했으며, 셸은 최초의 소위
''스크립팅 언어.'' Unix 시스템 호출 인터페이스는 오늘날 BSD와 같은 시스템에서 지속됩니다.
리눅스, Mac OS X.

Unix 시스템 호출 인터페이스는 Portable Operating System Interface(POSIX) 표준을 통해 표준화되었습니다. Xv6는 POSIX와 호환되지 않습니다 . 시스템 호출(lseek와 같은 기본 호출 포함)이 누락되고 시스템 호출을 부분적으로만 구현합니다. xv6에 대한 우리의 주요 목표는 단순성과 명확성이며 동시에 간단한

UNIX와 유사한 시스템 호출 인터페이스. 여러 사람이 xv6를 몇 가지 더 확장했습니다.
기본 시스템 호출과 간단한 C 라이브러리가 있어 기본적인 Unix 프로그램을 실행할 수 있습니다.
그러나 최신 커널은 더 많은 시스템 호출과 더 많은 종류의 기능을 제공합니다.
커널 서비스, xv6보다. 예를 들어, 네트워킹, 윈도잉 시스템을 지원합니다.
사용자 수준 스레드, 많은 장치의 드라이버 등. 최신 커널은 지속적이고 빠르게 진화하며 POSIX를 넘어서는 많은 기능을 제공합니다.

대부분의 현대 유닉스 기반 운영 체제는 다음을 따르지 않았습니다.
초기 유닉스 모델은 위에서 논의한 콘솔 장치 파일과 같은 특수 파일로 장치를 노출합니다. 유닉스의 저자는 네트워크, 그래픽 및 기타 리소스를 파일 또는 파일 트리로 표현하는 ''재자원은 파일이다''라는 개념을 현대 시설에 적용한 Plan 9를 구축했습니다.

파일 시스템 추상화는 강력한 아이디어였습니다. 그럼에도 불구하고 다른
운영 체제 인터페이스에 대한 모델. Unix의 전신인 Multics는 추상화된 파일
기억처럼 보이는 방식으로 저장하여 매우 다른 맛을 만들어냅니다.
인터페이스. Multics 디자인의 복잡성은 디자이너에게 직접적인 영향을 미쳤습니다.
유닉스를 만들려고 노력한 사람입니다.

이 책에서는 xv6가 Unix와 유사한 인터페이스를 어떻게 구현하는지 살펴보지만 아이디어와
개념은 Unix에만 적용되는 것이 아닙니다. 모든 운영 체제는 기본 하드웨어에 프로세스를 멀티플렉싱하고, 프로세스를 서로 분리하고, 제어된 프로세스 간 통신을 위한 메커니즘을 제공해야 합니다. xv6을 공부한 후에는

다른 더 복잡한 운영 체제를 살펴보고 그 기본 개념을 파악할 수 있습니다.
해당 시스템에도 xv6이 있습니다.

제1장

운영 체제 조직

운영체제의 핵심 요구 사항 중 하나는 여러 활동을 동시에 지원하는 것입니다.
예를 들어, 0장에서 설명한 시스템 호출 인터페이스를 사용하면 프로세스가 fork로 새 프로세스를 시작할 수 있습니다.운영 체제는 이러한 프로세스 간에 컴퓨터 리소스를 시간 공유 해야 합니다.예를 들어, 하드웨어 프로세서보다 프로세스가 많더라도 운영 체제는 모든 프로세스가 진행되도록 해야 합니다.운영 체제는 또한 프로세스 간의 격리 를 마련해야 합니다 .즉, 한 프로세스에 버그가 있어 실패하더라도 실패한 프로세스에 의존하지 않는 프로세스에는 영향을 미치지 않아야 합니다.그러나 완전한 격리는 프로세스가 상호 작용할 수 있어야 하기 때문에 너무 강력합니다.파이프라인이 그 예입니다.따라서 운영 체제는 멀티플렉싱, 격리 및 상호 작용이라는 세 가지 요구 사항을 충족해야 합니다.

이 장에서는 운영 체제가 이러한 3가지 요구 사항을 달성하기 위해 어떻게 구성되어 있는지에 대한 개요를 제공합니다. 그렇게 하는 방법은 여러 가지가 있지만, 이 텍스트는 많은 Unix 운영 체제에서 사용되는 모놀리식 커널을 중심으로 한 주류 설계에 초점을 맞춥니다. 이 장에서는 xv6가 실행되기 시작할 때 첫 번째 프로세스가 생성되는 과정을 추적하여 xv6의 설계를 소개합니다. 이를 통해 텍스트는 xv6가 제공하는 모든 주요 추상화의 구현, 상호 작용 방식, 멀티플렉싱, 격리, 상호 작용의 세 가지 요구 사항이 충족되는 방식을 간략하게 살펴봅니다. 대부분의 xv6는 첫 번째 프로세스를 특수 케이스화하지 않고 대신 xv6가 표준 작업을 위해 제공해야 하는 코드를 재사용합니다. 이후 장에서는 각 추상화를 더 자세히 살펴봅니다.

Xv6는 PC 플랫폼에서 Intel 80386 이상(''x86'') 프로세서에서 실행되며, 저수준 기능의 대부분(예: 프로세스 구현)은 x86에만 해당됩니다.
이 책은 독자가 어떤 아키텍처에 대한 약간의 머신 레벨 프로그래밍을 했다고 가정하고, x86 특정 아이디어가 떠오르면 소개할 것입니다. 부록 A는 PC 플랫폼을 간략하게 설명합니다.

물리적 자원 추상화

운영 체제를 접했을 때 가장 먼저 떠오르는 질문은 왜 그것을 가지고 있는가? 즉, 그림 0-2의 시스템 호출을 라이브러리로 구현하여 애플리케이션이 링크할 수 있습니다. 이 계획에서 각 애플리케이션은 필요에 맞게 조정된 자체 라이브러리를 가질 수도 있습니다. 애플리케이션은 하드웨어 리소스와 직접 상호 작용하고 해당 리소스를 애플리케이션에 가장 적합한 방식으로 사용할 수 있습니다(예: 높거나 예측 가능한 성능을 달성하기 위해). 임베디드 장치 또는 실시간 시스템을 위한 일부 운영 체제는 이런 방식으로 구성됩니다.

이 라이브러리 접근 방식의 단점은 두 개 이상의 애플리케이션이 있는 경우입니다.

실행 중이려면 애플리케이션이 제대로 작동해야 합니다. 예를 들어, 각 애플리케이션
다른 애플리케이션이 실행될 수 있도록 주기적으로 프로세서를 포기해야 합니다. 이러한 공동 특권
모든 애플리케이션이 서로를 신뢰하고 이를 충족하는 경우 작동 시간 공유 방식이 적합할 수 있습니다.
버그가 없습니다. 애플리케이션이 서로를 신뢰하지 않고 버그가 있는 것이 더 일반적이므로
사람들은 종종 협동 계획이 제공하는 것보다 더 강력한 고립을 원합니다.

커널 모드
사용자 모드

지침

강력한 격리를 달성하려면 애플리케이션이 중요한 하드웨어 리소스에 직접 액세스하는 것을 금지하고, 대신 리소스를 서비스로 추상화하는 것이 좋습니다.
예를 들어, 애플리케이션은 열기, 읽기, 쓰기를 통해서만 파일 시스템과 상호 작용합니다.
원시 디스크 섹터를 읽고 쓰는 대신 시스템 호출을 닫습니다. 이것은 다음을 제공합니다.
경로명의 편의성을 갖춘 응용 프로그램이며 운영 체제(예:
디스크를 관리하는 인터페이스의 구현자입니다.

마찬가지로 Unix는 필요에 따라 하드웨어 프로세서를 프로세스 간에 투명하게 전환하여 레지스터 상태를 저장하고 복원하므로 애플리케이션이 이를 인식할 필요가 없습니다.
시간 공유. 이 투명성을 통해 운영 체제는 프로세서를 공유할 수 있습니다.
일부 애플리케이션이 무한 루프에 있는 경우에도 마찬가지입니다.

또 다른 예로, Unix 프로세스는 exec를 사용하여 메모리 이미지를 구축합니다.
물리적 메모리와 직접 상호 작용하는 대신. 이를 통해 운영 체제가
프로세스를 메모리에 어디에 배치할지 결정합니다. 메모리가 부족하면 운영 체제가 프로세스의 일부 데이터를 디스크에 저장할 수도 있습니다. Exec은 또한 사용자에게 다음을 제공합니다.
실행 가능한 프로그램 이미지를 저장하는 파일 시스템의 편리함.

Unix 프로세스 간의 많은 형태의 상호 작용은 파일 기술자를 통해 발생합니다.
파일 설명자는 많은 세부 사항(예: 파이프나 파일의 데이터가 있는 위치)을 추상화합니다.
저장됨), 또한 상호 작용을 단순화하는 방식으로 정의됩니다. 예를 들어, 하나의 경우
파이프라인의 애플리케이션이 실패하면 커널은 다음 프로세스에 대해 파일 끝을 생성합니다.
파이프라인.

그림 0-2의 시스템 호출 인터페이스는 다음과 같은 사항을 고려하여 신중하게 설계되었습니다.
프로그래머의 편의성과 강력한 격리 가능성을 모두 제공합니다.
Unix 인터페이스는 리소스를 추상화하는 유일한 방법은 아니지만 매우 효과적인 것으로 입증되었습니다.
좋은 것.

사용자 모드, 커널 모드 및 시스템 호출

강력한 격리에는 애플리케이션과 운영 시스템 간에 엄격한 경계가 필요합니다.
시스템. 애플리케이션이 실수를 하면 운영 체제가 실패하는 것을 원하지 않습니다.
또는 다른 응용 프로그램이 실패할 수 있습니다. 대신 운영 체제는 정리할 수 있어야 합니다.
실패한 애플리케이션을 계속 실행하고 다른 애플리케이션을 계속 실행합니다. 강력한 격리를 달성하려면 운영 체제에서 애플리케이션이 수정(또는 심지어
운영 체제의 데이터 구조와 명령어를 읽고, 애플리케이션이 다른 프로세스의 메모리에 접근할 수 없다는 것을 의미합니다.

프로세서는 강력한 격리를 위한 하드웨어 지원을 제공합니다. 예를 들어, x86
프로세서는 다른 많은 프로세서와 마찬가지로 프로세서가 명령을 실행할 수 있는 두 가지 모드, 즉 커널 모드 와 사용자 모드를 가지고 있습니다. 커널 모드에서 프로세서는 다음을 수행할 수 있습니다.
특권 명령을 실행합니다 . 예를 들어, 디스크(또는 모든
다른 I/O 장치)에는 특권 명령이 포함됩니다. 사용자 모드의 애플리케이션이-

특권 명령을 실행하려고 시도하면 프로세서는 명령을 실행하지 않고 커널 모드로 전환하여 커널 모드의 소프트웨어가 정리할 수 있도록 합니다.

응용 프로그램을 실행하지 않은 것은 응용 프로그램이 하지 말아야 할 일을 했기 때문입니다. 0장의 그림 0-1은 이러한 구성을 보여줍니다. 응용 프로그램은 사용자 모드 명령(예: 숫자 추가 등)만 실행할 수 있으며 사용자 공간에서 실행 중이라고 합니다 .

커널 모드의 소프트웨어는 특권 명령을 실행할 수도 있으며 커널 공간에서 실행 중이라고 합니다 . 커널 공간(또는 커널 모드)에서 실행되는 소프트웨어는
커널 이라고 부릅니다 .

디스크에 있는 파일을 읽거나 쓰려는 애플리케이션은 다음으로 전환해야 합니다.
커널이 그렇게 하도록 하는 이유는 애플리케이션 자체가 I/O 명령어를 실행할 수 없기 때문입니다. 프로세서는 프로세서를 사용자 모드에서 커널 모드로 전환하는 특수 명령어를 제공합니다.
모드로 전환하고 커널에서 지정한 진입점에서 커널에 들어갑니다. (x86 프로세서는 이 목적을 위해 int 명령어를 제공합니다.) 프로세서가 전환되면
커널 모드에서는 커널이 시스템 호출의 인수를 검증하고 결정할 수 있습니다.
애플리케이션이 요청된 작업을 수행할 수 있는지 여부를 확인한 다음 거부합니다.
그것을 실행하거나 커널이 전환을 위한 진입점을 설정하는 것이 중요합니다.
커널 모드; 애플리케이션이 커널 진입점을 결정할 수 있다면 악성 애플리케이션은 인수 등의 검증이 필요한 지점에서 커널에 진입할 수 있습니다.

건너뛰었습니다.

커널 조직

핵심적인 설계 질문은 운영 체제의 어떤 부분이 커널에서 실행되어야 하는가입니다.
모드. 한 가지 가능성은 전체 운영 체제가 커널에 상주한다는 것입니다.
모든 시스템 호출의 구현은 커널 모드에서 실행됩니다. 이 조직을 다음과 같이 부릅니다.
단일체 커널 .

이 조직에서는 전체 운영 체제가 전체 하드웨어 권한으로 실행됩니다.
이러한 구성은 OS 설계자가 어떤 것을 선택할지 결정할 필요가 없기 때문에 편리합니다.
운영 체제의 일부에는 전체 하드웨어 권한이 필요하지 않습니다. 게다가 쉽습니다.
운영 체제의 다른 부분이 협력할 수 있도록 합니다. 예를 들어, 운영 체제
시스템에는 파일 시스템과 모두 공유할 수 있는 버퍼 캐시가 있을 수 있습니다.
가상 메모리 시스템.

모놀리식 조직의 단점은 서로 다른 조직 간의 인터페이스가
운영 체제의 일부는 종종 복잡합니다(이 텍스트의 나머지 부분에서 볼 수 있듯이).
따라서 운영 체제 개발자가 실수를 하기 쉽습니다.
모놀리식 커널의 경우 실수는 치명적입니다. 커널 모드에서 오류가 발생하는 경우가 많기 때문입니다.
커널에서 실패합니다. 커널이 실패하면 컴퓨터가 작동을 멈추고 모든 애플리케이션도 실패합니다. 컴퓨터를 재부팅해야 다시 시작할 수 있습니다.

커널에서의 실수 위험을 줄이기 위해 OS 설계자는 다음을 최소화할 수 있습니다.
커널 모드에서 실행되는 운영 체제 코드의 양과 대량의 실행
사용자 모드의 운영 체제. 이 커널 조직을 마이크로커널이라고 합니다.

그림 1-1은 이 마이크로커널 설계를 보여줍니다. 그림에서 파일 시스템은 다음과 같이 실행됩니다.
사용자 수준 프로세스. 프로세스로 실행되는 OS 서비스를 서버라고 합니다. 애플리케이션이 파일 서버와 상호 작용할 수 있도록 커널은 한 사용자 모드 프로세스에서 다른 사용자 모드 프로세스로 메시지를 보내는 프로세스 간 통신 메커니즘을 제공합니다.

마이크로커널

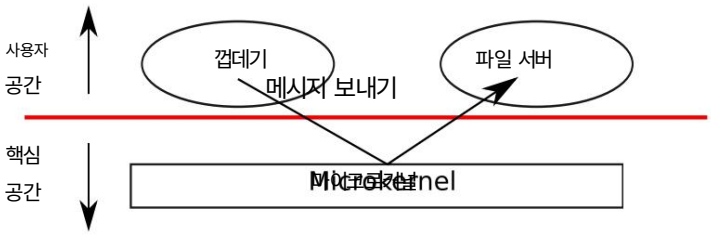


그림 1-1. 파일 시스템 서버를 갖춘 마이크로커널

프로세스
주소 공간
가상 주소
물리적 주소
사용자 메모리

예를 들어, 셸과 같은 애플리케이션이 파일을 읽거나 쓰려면 메시지를 보냅니다.
파일 서버로 전송하여 응답을 기다립니다.

마이크로커널에서 커널 인터페이스는 몇 가지 저수준 기능으로 구성됩니다.
응용 프로그램 시작, 메시지 전송, 장치 하드웨어 액세스 등. 이 조직을 사용하면 대부분의 운영 체제가 상주하므로 커널이 비교적 간단해집니다.
사용자 수준 서버에서.

Xv6는 대부분의 Unix 운영 체제를 따르는 모놀리식 커널로 구현됩니다. 따라서 xv6에서 커널 인터페이스는 운영 체제 인터페이스에 해당합니다.
그리고 커널은 완전한 운영 체제를 구현합니다. xv6는 제공하지 않기 때문에
많은 서비스의 경우 커널은 일부 마이크로 커널보다 작습니다.

프로세스 개요

xv6의 격리 단위(다른 Unix 운영 체제와 마찬가지로)는 프로세스 입니다 .
프로세스 추상화는 한 프로세스가 다른 프로세스를 파괴하거나 감시하는 것을 방지합니다.
메모리, CPU, 파일 기술자 등. 또한 프로세스가 커널을 파괴하는 것을 방지합니다.
그 자체로, 프로세스가 커널의 격리 메커니즘을 파괴할 수 없도록 합니다. 커널
버그가 있거나 악의적인 애플리케이션이 커널이나 하드웨어를 속여 나쁜 일을 할 수 있으므로 프로세스 추상화를 신중하게 구현해야 합니다(예: 우회).
강제 격리). 커널이 프로세스를 구현하는 데 사용하는 메커니즘에는 사용자/커널 모드 플래그, 주소 공간, 스레드의 시간 분할이 포함됩니다.

격리를 강화하기 위해 프로세스 추상화는 프로그램에 자체 개인 머신이 있다는 환상을 제공합니다. 프로세스는 다른 프로세스가 할 수 없는 개인 메모리 시스템 또는 주소 공간 처럼 보이는 것을 프로그램에 제공합니다.

읽거나 쓸 수 있습니다. 프로세스는 또한 프로그램에 자체적으로 보이는 것을 제공합니다.
CPU는 프로그램의 명령을 실행합니다.

Xv6는 각 프로세스에 페이지 테이블(하드웨어로 구현됨)을 제공합니다.
자체 주소 공간입니다. x86 페이지 테이블은 가상 주소 (
x86 명령어가 조작하는 주소)를 물리적 주소 (x86 명령어가 조작하는 주소)로 변환합니다.
프로세서 칩이 주 메모리로 전송함).

Xv6는 프로세스별로 해당 프로세스를 정의하는 별도의 페이지 테이블을 유지합니다.
주소 공간. 그림 1-2에서 설명한 대로 주소 공간에는 프로세스의 사용자
가상 주소 0에서 시작하는 메모리 . 명령어가 먼저 나오고 그 다음에 글로벌 명령어가 나옵니다.
변수, 스택, 마지막으로 프로세스가 필요에 따라 확장할 수 있는 ''힙' 영역(malloc용)이 있습니다.

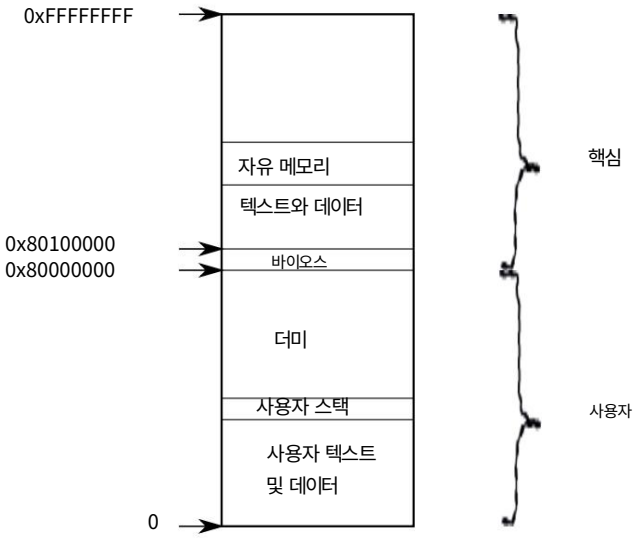


그림 1-2. 가상 주소 공간의 레이아웃

구조체 proc+코드 p->xxx+코드 스레드 p->kstack+코드

각 프로세스의 주소 공간은 커널의 명령어와 데이터, 그리고 사용자 프로그램의 메모리를 매핑합니다. 프로세스가 시스템 호출을 호출하면 시스템 호출은 프로세스 주소 공간의 커널 매핑에서 실행됩니다. 이러한 배열은 커널의 시스템 호출 코드가 사용자 메모리를 직접 참조할 수 있도록 존재합니다. 사용자 메모리에 충분한 공간을 남겨두기 위해 xv6의 주소 공간은 0x80100000에서 시작하여 높은 주소에서 커널을 매핑합니다.

xv6 커널은 각 프로세스에 대한 많은 상태 조각을 유지 관리하고 이를 struct proc (2337)로 수집합니다. 프로세스의 가장 중요한 커널 상태 조각은 페이지 테이블, 커널 스택 및 실행 상태입니다. proc 구조의 요소를 참조하기 위해 p->xxx 표기법을 사용합니다.

각 프로세스에는 프로세스의 명령을 실행하는 실행 스레드(또는 줄여서 스레드)가 있습니다. 스레드는 일시 중단되었다가 나중에 재개될 수 있습니다. 프로세스 간에 투명하게 전환하기 위해 커널은 현재 실행 중인 스레드를 일시 중단하고 다른 프로세스의 스레드를 재개합니다. 스레드의 상태(로컬 변수, 함수 호출 반환 주소)의 대부분은 스레드의 스택에 저장됩니다. 각 프로세스에는 사용자 스택과 커널 스택(p->kstack)의 두 스택이 있습니다. 프로세스가 사용자 명령을 실행할 때는 사용자 스택만 사용 중이고 커널 스택은 비어 있습니다. 프로세스가 커널에 들어가면(시스템 호출 또는 인터럽트의 경우) 커널 코드는 프로세스의 커널 스택에서 실행됩니다. 프로세스가 커널에 있는 동안 사용자 스택에는 여전히 저장된 데이터가 있지만 적극적으로 사용되지는 않습니다. 프로세스의 스레드는 사용자 스택과 커널 스택을 적극적으로 번갈아 사용합니다. 커널 스택은 분리되어 있고(사용자 코드로부터 보호됨) 프로세스가 사용자 스택을 망가뜨렸더라도 커널이 실행될 수 있습니다.

프로세스가 시스템 호출을 할 때 프로세서는 커널 스택으로 전환하고 하드웨어 권한 수준을 높이고 시스템 호출을 구현하는 커널 명령어를 실행하기 시작합니다. 시스템 호출이 완료되면 커널은 사용자 공간으로 돌아갑니다. 하드웨어는 권한 수준을 낮추고 사용자 스택으로 다시 전환하고 시스템 호출 명령어 바로 다음에 사용자 명령어 실행을 재개합니다. 프로세스의 스레드는 커널에서 ''차단''하여 I/O를 기다리고,

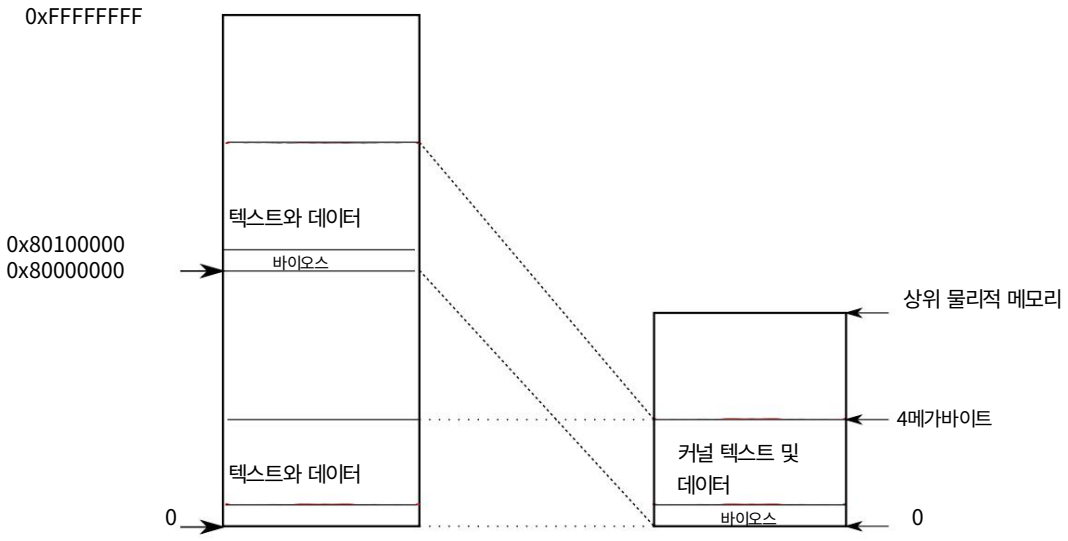


그림 1-3. 가상 주소 공간의 레이아웃

I/O가 완료되었습니다.

p->state+code p->pgdir+code 부트로더 항목+코드

p->state는 프로세스가 할당되었는지, 실행할 준비가 되었는지, 실행 중인지, I/O를 기다리는지, 종료 중인지를 나타냅니다. p->pgdir은 x86 하드웨어가 예상하는 형식으로 프로세스의 페이지 테이블을 보관합니다. xv6는 페이징 하드웨어가 해당 프로세스를 실행할 때 프로세스의 p->pgdir을 사용하도록 합니다. 프로세스의 페이지 테이블은 프로세스의 메모리를 저장하기 위해 할당된 물리적 페이지의 주소 레코드 역할도 합니다.

코드: 첫 번째 주소 공간

xv6 조직을 더 구체적으로 만들기 위해 커널이 첫 번째 주소 공간(자체)을 만드는 방법, 커널이 첫 번째 프로세스를 만들고 시작하는 방법, 해당 프로세스가 첫 번째 시스템 호출을 수행하는 방법을 살펴보겠습니다. 이러한 작업을 추적하면 xv6가 프로세스에 강력한 격리를 제공하는 방법을 자세히 알 수 있습니다. 강력한 격리를 제공하는 첫 번째 단계는 커널이 자체 주소 공간에서 실행되도록 설정하는 것입니다.

PC가 켜지면 자체를 초기화한 다음 디스크에서 부트 로더를 메모리로 로드하여 실행합니다. 부록 B에서 자세한 내용을 설명합니다. Xv6의 부트 로더는 디스크에서 xv6 커널을 로드하여 엔트리 (1044)에서 시작합니다. 커널이 시작될 때 x86 페이징 하드웨어는 활성화되지 않습니다. 가상 주소는 물리적 주소에 직접 매핑됩니다.

부트로더는 xv6 커널을 물리 주소 0x100000의 메모리에 로드합니다.
커널이 명령어와 데이터를 찾을 것으로 예상하는 0x80100000에 커널을 로드하지 않는 이유는 작은 머신에서 그렇게 높은 주소에 물리적 메모리가 없을 수 있기 때문입니다. 커널을 0x0이 아닌 0x100000에 배치하는 이유는 주소 범위 0xa0000:0x100000에 I/O 장치가 포함되어 있기 때문입니다.

나머지 커널이 실행될 수 있도록 항목은 가상 머신을 매핑하는 페이지 테이블을 설정합니다.

0x80000000에서 시작하는 모든 주소(KERNBASE (0207)라고 함) 에서 0x0에서 시작하는 물리적 주소까지(그림 1-2 참조). 두 범위의 가상 주소 설정
동일한 물리적 메모리 범위는 페이지 테이블의 일반적인 사용이며 우리는 더 많은 것을 볼 것입니다.
이런 예가 있습니다.

엔트리 페이지 테이블은 main.c (1306)에 정의되어 있습니다. 2장에서 페이지 테이블의 세부 사항을 살펴보겠지만, 간단히 말해서 엔트리 0은 가상 주소를 매핑합니다.
0:0x400000에서 물리적 주소 0:0x400000으로 매핑합니다. 이 매핑은 다음 경우에 필요합니다.
항목이 낮은 주소에서 실행되지만 결국 제거됩니다.

항목 512는 가상 주소 KERNBASE:KERNBASE+0x400000을 물리적 주소 0:0x400000에 매핑합니다.
이 항목은 항목이 완료된 후 커널에서 사용됩니다.
커널이 명령어를 찾을 것으로 예상하는 높은 가상 주소를 매핑합니다.
부트로더가 로드한 하위 물리적 주소로 데이터를 전송합니다. 이 매핑
커널 명령어와 데이터를 4MB로 제한합니다.

entry로 돌아가서 entrypgdir의 물리적 주소를 제어 레지스터 %cr3에 로드합니다. %cr3의 값은 물리적 주소여야 합니다. 이것은 의미가 없습니다.
%cr3는 페이징 하드웨어가 없기 때문에 entrypgdir의 가상 주소를 유지합니다.
아직 가상 주소를 변환하는 방법을 모릅니다. 아직 페이지 테이블이 없습니다. 심볼 엔트리 pgdir은 하이 메모리의 주소를 참조하고 매크로 V2P_WO (0213)
KERNBASE를 빼서 물리 주소를 찾습니다. 페이징 하드웨어를 활성화하기 위해 xv6는 제어 레지스터 %cr0에 플래그 CR0_PG를 설정합니다.

페이징이 활성화된 후에도 프로세서는 여전히 낮은 주소에서 명령어를 실행하고 있습니다. entrypgdir이 낮은 주소를 매핑하기 때문에 작동합니다. xv6에서 항목 0을 생략했다면
entrypgdir에서, 페이징을 활성화하는 명령어 다음에 명령어를 실행하려고 하면 컴퓨터가 충돌했을 것입니다.

이제 항목을 커널의 C 코드로 전송하고 고메모리에서 실행해야 합니다.
먼저 스택 포인터 %esp가 스택으로 사용될 메모리를 가리키도록 합니다 (1058). 모두
스택을 포함하여 심볼에는 높은 주소가 있으므로 스택은 여전히 유효합니다.
낮은 매핑이 제거되면 마지막으로 엔트리가 메인으로 점프합니다. 메인도 마찬가지입니다.
높은 주소. 간접 점프가 필요한 이유는 어셈블러가 그렇지 않으면
저메모리 버전을 실행하는 PC 관련 직접 점프를 생성합니다.
main. Main은 스택에 리턴 PC가 없기 때문에 리턴할 수 없습니다. 이제 커널
main 함수에서 높은 주소 (1217)에서 실행 중입니다.

코드: 첫 번째 프로세스 생성

이제 커널이 사용자 수준 프로세스를 생성하고 이를 보장하는 방법을 살펴보겠습니다.
그들은 강하게 고립되어 있습니다.

main (1217) 이 여러 장치와 하위 시스템을 초기화한 후 userinit (2520) 을 호출하여 첫 번째 프로세스를 생성합니다 . Userinit의 첫 번째 동작은 allocproc를 호출하는 것입니다. 작업
allocproc (2473) 의 목적은 프로세스 테이블에 슬롯(구조체 proc)을 할당하는 것입니다.
커널 스레드가 실행되기 위해 필요한 프로세스 상태의 부분을 초기화합니다. al-locproc는 각 새 프로세스에 대해 호출되는 반면 userinit는 맨 처음에만 호출됩니다.
프로세스. Allocproc는 UNUSED (2480-2482) 상태의 슬롯에 대한 proc 테이블을 스캔합니다 .
사용되지 않는 슬롯을 찾으면 allocproc는 상태를 EMBRYO로 설정하여 사용됨으로 표시합니다.
프로세스에 고유한 pid (2469-2489)를 제공합니다. 다음으로 커널 스택을 할당하려고 합니다.

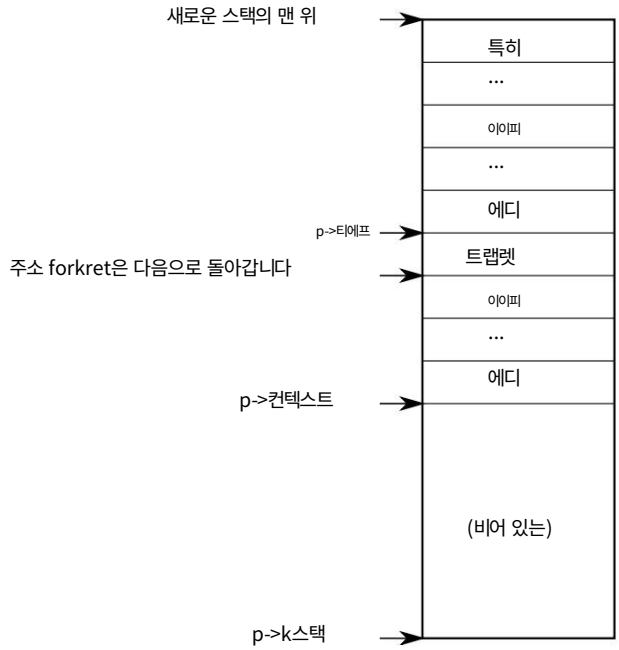


그림 1-4. 새로운 커널 스택.

프로세스의 커널 스레드. 메모리 할당이 실패하면 allocproc는 상태를 UNUSED로 다시 변경하고 실패 신호를 보내기 위해 0을 반환합니다.

포크렛+코드 트랩렛 +코드 p->컨텍스트 +코드 포크렛+코드

이제 allocproc는 새 프로세스의 커널 스택을 설정해야 합니다. allocproc는 fork에서 사용할 수 있을 뿐만 아니라 첫 번째 프로세스를 생성할 때도 사용할 수 있도록 작성되었습니다. allocproc는 특별히 준비된 커널 스택과 처음 실행될 때 사용자 공간으로 ''돌아가도록'' 하는 커널 레지스터 세트로 새 프로세스를 설정합니다. 준비된 커널 스택의 레이아웃은 그림 1-4와 같습니다. allocproc는 새 프로세스의 커널 스레드가 먼저 forkret에서 실행되고 그다음 trapret에서 실행되도록 하는 리턴 프로그램 카운터 값을 설정하여 이 작업의 일부를 수행합니다 (2507-2512). 커널 스레드는 p->context에서 복사된 레지스터 내용으로 실행을 시작합니다. 따라서 p->context->eip를 forkret으로 설정하면 커널 스레드가 forkret의 시작 부분에서 실행됩니다 (2853). 이 함수는 스택 맨 아래에 있는 주소로 돌아갑니다. 컨텍스트 스위치 코드 (3059)는 스택 포인터를 p->context의 끝 바로 너머를 가리키도록 설정합니다. allocproc는 p->context를 스택에 놓고, 바로 위에 trapret에 대한 포인터를 놓습니다. 여기가 forkret이 반환할 곳입니다. trapret은 커널 스택 맨 위에 저장된 값에서 사용자 레지스터를 복원하고 프로세스로 점프합니다 (3324).

트랩렛+코드 포크렛+코드 트랩렛+코드 사용자 초기화+코드

이 설정은 일반적인 포크와 첫 번째 프로세스를 생성하는 경우 모두 동일하지만, 후자의 경우 프로세스는 포크에서 반환된 지점이 아닌 사용자 공간 위치 0에서 실행을 시작합니다.

3장에서 살펴보겠지만, 제어가 사용자 소프트웨어에서 커널로 전송되는 방식은 시스템 호출, 인터럽트 및 예외에서 사용되는 인터럽트 메커니즘을 통한 것입니다. 프로세스가 실행되는 동안 제어가 커널로 전송될 때마다 하드웨어 및 xv6 트랩 항목 코드는 프로세스의 커널 스택에 사용자 레지스터를 저장합니다. userinit은 새 스택의 맨 위에 값을 쓰는데, 이 값은

프로세스가 인터럽트 (2533-2539) 를 통해 커널에 진입한 경우 해당 프로세스의 사용자 코드로 커널에서 복귀하기 위한 일반 코드가 작동할 것입니다.
이 값들은 사용자 레지스터를 저장하는 struct trapframe입니다. 이제 새로운 프로세스의 커널 스택은 그림 1-4에 표시된 대로 완전히 준비되었습니다.

첫 번째 프로세스는 작은 프로그램(initcode.S; (8400)) 을 실행하려고 합니다 . 프로세스는 이 프로그램을 저장할 실제 메모리가 필요하고, 프로그램은 해당 메모리에 복사되어야 하며, 프로세스는 사용자 공간 주소를 해당 메모리에 매핑하는 페이지 테이블이 필요합니다. userinit는 setupkvm (1818) 을 호출하여 (처음에는) 커널이 사용하는 메모리에 대한 매

핑만 있는 프로세스의 페이지 테이블을 만듭니다. 이 함수는 2장에서 자세히 살펴보겠지만, 높은 수준에서 setupkvm과 userinit는 그림 1-2에 표시된 대로 주소 공간을 만듭니다.

첫 번째 프로세스의 사용자 공간 메모리의 초기 내용은 initcode.S의 컴파일된 형태입니다. 커널 빌드 프로세스의 일부로 링커는 해당 바이너리를 커널에 임베드하고 바이너리의 위치와 크기를 나타내는 두 개의 특수 심볼인 _binary_initcode_start와 _binary_initcode_size를 정의합니다. Userinit는 inituvm을 호출하여 해당 바이너리를 새 프로세스의 메모리에 복사합니다. inituvm은 물리적 메모리의 한 페이지를 할당하고 가상 주소 0을 해당 메모리에 매핑하고 바이너리를 해당 페이지에 복사합니다 (1886).

그런 다음 userinit는 초기 사용자 모드 상태로 트랩 프레임 (0602) 을 설정합니다 . %cs 레지스터는 권한 수준 DPL_USER(즉, 커널 모드가 아닌 사용자 모드)에서 실행되는 SEG_UCODE 세그먼트에 대한 세그먼트 선택기를 포함하고 마찬가지로 %ds, %es 및 %ss는 권한 DPL_USER와 함께 SEG_UDATA를 사용합니다. %eflags FL_IF 비트는 하드웨어 인터럽트를 허용하도록 설정됩니다. 3장에서 이를 다시 살펴보겠습니다.

스택 포인터 %esp는 프로세스의 가장 큰 유효 가상 주소 p->sz로 설정됩니다.
명령어 포인터는 initcode의 진입점인 주소 0으로 설정됩니다.

함수 userinit는 주로 디버깅을 위해 p->name을 initcode로 설정합니다. p->cwd를 설정하면 프로세스의 현재 작업 디렉토리가 설정됩니다. 6장에서 namei를 자세히 살펴보겠습니다.

프로세스가 초기화되면 userinit은 설정에 따라 스케줄링을 위해 사용 가능하다고 표시합니다.
p->state를 RUNNABLE로 전환합니다.

코드: 첫 번째 프로세스 실행 이제 첫 번째 프로세스의 상

태가 준비되었으므로 실행할 시간입니다. main이 userinit를 호출한 후 mpmain이 scheduler를 호출하여 프로세스 실행을 시작합니다 (1257). Scheduler (2758) 는 p->state가 RUNNABLE로 설정된 프로세스를 찾고, 그 프로세스는 initproc 하나뿐입니다. CPU당 변수 proc를 찾은 프로세스로 설정하고 switchuvm을 호출하여 하드웨어에 대상 프로세스의 페이지 테이블을 사용하도록 지시합니다 (1879). 커널에서 실행하는 동안 페이지 테이블을 변경하는 것은 setupkvm이 모든 프로세스의 페이지 테이블이 커널 코드와 데이터에 대해 동일한 매핑을 갖도록 하기 때문에 작동합니다. switchuvm은 또한 하드웨어에 프로세스의 커널 스택에서 시스템 호출과 인터럽트를 실행하도록 지시하는 작업 상태 세그먼트 SEG_TSS를 설정합니다. 3장에서 작업 상태 세그먼트를 다시 살펴보겠습니다. scheduler는 이제 p->state를 RUNNING으로 설정하고 swtch (3059) 를 호출하여 다음을 수행합니다.

대상 프로세스의 커널 스레드로 컨텍스트 전환. swtch는 먼저 현재 레지스터를 저장합니다. 현재 컨텍스트는 프로세스가 아니라 특수한 CPU별 스케줄러 컨텍스트이므로 스케줄러는 swtch에 현재 하드웨어 레지스터를 모든 프로세스의 커널 스레드 컨텍스트가 아닌 CPU별 저장소(cpu->scheduler)에 저장하라고 지시합니다. 그런 다음 swtch는 대상 커널 스레드(p->context)의 저장된 레지스터를 스택 포인터와 명령어 포인터를 포함하여 x86 하드웨어 레지스터에 로드합니다. 5장에서 swtch를 더 자세히 살펴보겠습니다. 마지막 ret 명령어 (3078) 는 스택에서 대상 프로세스의 %eip를 팝하여 컨텍스트 전환을 완료합니다. 이제 프로세서는 프로세스 p의 커널 스택에서 실행됩니다.

cpu-
> 스케줄러+코드 스위치+코드 ret+코드 포크
렛+코드
ret+코드

포크렛+코드 포크렛
+코드 메안+코드 p-
>컨텍스트+코드

트랩렛+코드 스위치+코드

Allocproc는 이전에 initproc의 p->context->eip를 forkret으로 설정했으므로 ret은 forkret을 실행하기 시작합니다. 첫 번째 호출(바로 이 호출)에서 forkret (2853) 은 자체 커널 스택이 있는 일반 프로세스의 컨텍스트에서 실행해야 하기 때문에 main에서 실행할 수 없는 초기화 함수를 실행합니다. 그런 다음 forkret이 반환됩니다.

popal+코드
popl+코드
addl+코드
iret+코드

initproc+코드
initcode.S+코드
allocuvm+코드

PTE_U+코드
userinit+코드
실행+코드

SYS_exec+코드
T_SYSCALL+코드 실행+코드

Allocproc은 p->context가 팝된 후 스택의 맨 위에 있는 단어를 trapret으로 설정했으므로 이제 trapret이 실행되고 %esp가 p->tf로 설정됩니다.
Trapret (3324)는 swtch가 커널 컨텍스트에서 한 것처럼 트랩 프레임 (0602) 에서 레지스터를 복원하기 위해 pop 명령어를 사용합니다 . popal은 일반 레지스터를 복원한 다음 popl 명령어는 %gs, %fs, %es, %ds를 복원합니다. addl은 두 필드 trapno와 errcode를 건너뜁니다. 마지막으로 iret 명령어는 스택에서 %cs, %eip, %flags, %esp, %ss를 팝합니다. 트랩 프레임의 내용이 CPU 상태로 전송되었으므로 프로세서는 트랩 프레임에 지정된 %eip에서 계속합니다. init-proc의 경우 이는 initcode.S의 첫 번째 명령어인 가상 주소 0을 의미합니다.

이 시점에서 %eip는 0을 보유하고 %esp는 4096을 보유합니다. 이것들은 프로세스의 주소 공간에 있는 가상 주소입니다. 프로세서의 페이징 하드웨어는 이를 물리적 주소로 변환합니다. allocuvm은 프로세스의 페이지 테이블을 설정하여 가상 주소 0이 이 프로세스에 할당된 물리적 메모리를 참조하고, 페이징 하드웨어에 사용자 코드가 해당 메모리에 액세스할 수 있도록 허용하는 플래그(PTE_U)를 설정합니다. userinit (2533) 가 CPL=3에서 프로세스의 사용자 코드를 실행하기 위해 %cs의 하위 비트를 설정했다는 사실은 사용자 코드가 PTE_U가 설정된 페이지만 사용할 수 있고 %cr3과 같은 민감한 하드웨어 레지스터를 수정할 수 없다는 것을 의미합니다. 따라서 프로세스는 자체 메모리만 사용하도록 제한됩니다.

첫 번째 시스템 호출: exec

이제 커널이 프로세스에 대해 강력한 격리를 어떻게 제공하는지 살펴보았으므로, 사용자 수준 프로세스가 스스로 수행할 수 없는 서비스를 요청하기 위해 커널에 다시 들어가는 방법을 살펴보겠습니다.

initcode.S의 첫 번째 동작은 exec 시스템 호출을 호출하는 것입니다. 0장에서 보았듯이 exec는 현재 프로세스의 메모리와 레지스터를 새 프로그램으로 대체하지만 파일 설명자, 프로세스 ID 및 부모 프로세스는 변경되지 않습니다.

Initcode.S (8409) 는 스택에 세 개의 값($argv, $init, $0)을 푸시하는 것으로 시작한 다음 %eax를 SYS_exec로 설정하고 int T_SYSCALL을 실행합니다. 커널에 exec 시스템 호출을 실행하도록 요청합니다. 모든 것이 잘 진행되면 exec는 결코 반환하지 않습니다. $init으로 명명된 프로그램을 실행하기 시작하는데, 이는 NUL로 끝나는 문자열 /init (8422-8424)을 가리키는 포인터입니다. 다른 인수는 명령줄 인수의 argv 배열입니다.

배열의 끝에 있는 0은 끝을 표시합니다. exec가 실패하고 반환하면 init-code는 종료 시스템 호출을 호출하는 루프를 돌며, 이는 확실히 반환되어서는 안 됩니다 (8416-8420).

종료+코드
/init+코드
init코드+코드 /
init+코드

이 코드는 3장에서 살펴볼 일반적인 시스템 호출처럼 보이도록 첫 번째 시스템 호출을 수동으로 제작합니다. 이전과 마찬가지로 이 설정은 첫 번째 프로세스(이 경우 첫 번째 시스템 호출)를 특수 케이스로 처리하지 않고 대신 xv6가 표준 작업을 위해 제공해야 하는 코드를 재사용합니다.

2장에서는 exec의 구현을 자세히 다루겠지만, 높은 수준에서는 initcode를 파일 시스템에서 로드된 /init 바이너리로 대체합니다. 이제 init-code (8400) 가 완료되었고 프로세스는 대신 /init을 실행합니다. Init (8510)은 필요한 경우 새 콘솔 장치 파일을 만든 다음 파일 설명자 0, 1, 2로 엽니다. 그런 다음 루프를 실행하여 콘솔 셸을 시작하고 셸이 종료될 때까지 고아 좀비를 처리한 다음 다시 반복합니다. 시스템이 가동됩니다.

현실 세계

실제 세계에서는 모놀리식 커널과 마이크로커널을 모두 찾을 수 있습니다. 많은 유닉스 커널이 모놀리식입니다. 예를 들어, 리눅스는 모놀리식 커널을 가지고 있지만, 일부 OS 기능은 사용자 수준 서버(예: 윈도잉 시스템)로 실행됩니다. L4, Minix, QNX와 같은 커널은 서버가 있는 마이크로커널로 구성되었으며, 임베디드 설정에서 널리 배포되었습니다.

대부분의 운영 체제는 프로세스 개념을 채택했으며, 대부분의 프로세스는 xv6의 프로세스와 유사합니다. 실제 운영 체제는 allocproc의 선형 시간 검색 대신 상수 시간에 명시적 자유 목록을 사용하여 자유 proc 구조를 찾습니다. xv6는 단순성을 위해 선형 스캔(첫 번째)을 사용합니다.

수업 과정

1. swtch에 중단점을 설정합니다. gdb의 stepi를 사용하여 ret에서 forkret까지 단일 단계를 수행한 다음 gdb의 finish를 사용하여 trapret으로 진행한 다음 가상 주소 0에서 initcode에 도달할 때까지 stepi를 수행합니다.

2. KERNBASE는 단일 프로세스가 사용할 수 있는 메모리 양을 제한하는데, 이는 4GB의 RAM이 있는 머신에서는 짜증스러울 수 있습니다. KERNBASE를 높이면 프로세스가 더 많은 메모리를 사용할 수 있을까요?

2장

페이지 테이블

페이지 테이블은 운영 체제가 메모리 주소의 의미를 제어하는 메커니즘입니다. 이를 통해 xv6는 여러 프로세스의 주소 공간을 단일 물리적 메모리로 멀티플렉싱하고 여러 프로세스의 메모리를 보호할 수 있습니다. 페이지 테이블이 제공하는 간접성 수준은 많은 멋진 트릭을 허용합니다. xv6는 주로 페이지 테이블을 사용하여 주소 공간을 멀티플렉싱하고 메모리를 보호합니다. 또한 몇 가지 간단한 페이지 테이블 트릭을 사용합니다. 여러 주소 공간에 동일한 메모리(커널)를 매핑하고, 한 주소 공간에 동일한 메모리를 두 번 이상 매핑하고(각 사용자 페이지는 커널의 메모리 물리적 뷰에도 매핑됨), 매핑되지 않은 페이지로 사용자 스택을 보호합니다. 이 장의 나머지 부분에서는 x86 하드웨어가 제공하는 페이지 테이블과 xv6에서 이를 사용하는 방법에 대해 설명합니다. 실제 운영 체제와 비교했을 때 xv6의 디자인은 제한적이지만 핵심 아이디어를 보여줍니다.

페이징 하드웨어 상기시켜드리

자면, x86 명령어(사용자 및 커널 모두)는 가상 주소를 조작합니다.
머신의 RAM 또는 물리적 메모리는 물리적 주소로 색인화됩니다. x86 페이지 테이블 하드웨어는 각 가상 주소를 물리적 주소에 매핑하여 이 두 종류의 주소를 연결합니다.

x86 페이지 테이블은 논리적으로 2^20(1,048,576)개의 페이지 테이블 항목(PTE) 배열입니다 . 각 PTE에는 20비트의 물리적 페이지 번호(PPN)와 몇 개의 플래그가 들어 있습니다. 페이징 하드웨어는 가상 주소를 변환하는데, 가상 주소의 상위 20비트를 사용하여 페이지 테이블을 인덱싱하여 PTE를 찾고, 주소의 상위 20비트를 PTE의 PPN으로 바꿉니다. 페이징 하드웨어는 가상 주소에서 변환된 물리적 주소로 하위 12비트를 변경하지 않고 복사합니다. 따라서 페이지 테이블은 운영 체제가 4096(2^12)바이트의 정렬된 청크 단위로 가상에서 물리적 주소로의 변환을 제어할 수 있도록 합니다. 이러한 청크를 페이지라고 합니다 .

그림 2-1에서 볼 수 있듯이 실제 변환은 두 단계로 이루어집니다. 페이지 테이블은 2단계 트리로 물리적 메모리에 저장됩니다. 트리의 루트는 1024개의 PTE 유사 페이지 테이블 페이지 참조를 포함하는 4096바이트 페이지 디렉토리 입니다. 각 페이지 테이블 페이지는 1024개의 32비트 PTE 배열입니다. 페이징 하드웨어는 가상 주소의 상위 10비트를 사용하여 페이지 디렉토리 항목을 선택합니다. 페이지 디렉토리 항목이 있는 경우 페이징 하드웨어는 가상 주소의 다음 10비트를 사용하여 페이지 디렉토리 항목이 참조하는 페이지 테이블 페이지에서 PTE를 선택합니다. 페이지 디렉토리 항목이나 PTE가 없는 경우 페이징 하드웨어는 오류를 발생시킵니다. 이 2단계 구조를 사용하면 페이지 테이블에서 광범위한 가상 주소에 매핑이 없는 일반적인 경우 전체 페이지 테이블 페이지를 생략할 수 있습니다.

각 PTE에는 페이징 하드웨어에 연관된 가상 주소를 사용할 수 있는 방법을 알려주는 플래그 비트가 포함되어 있습니다. PTE_P는 PTE가 있는지 여부를 나타냅니다.

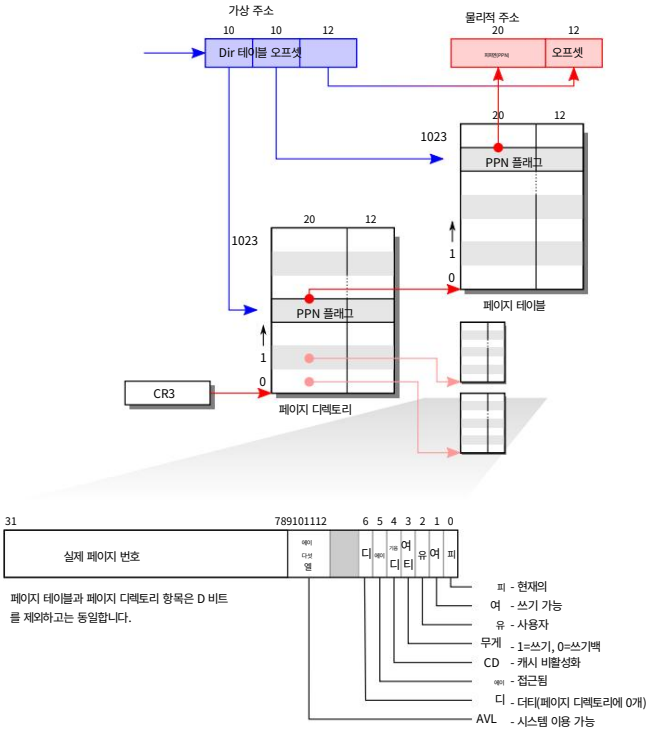


그림 2-1. x86 페이지 테이블 하드웨어.

PTE_W+코드
PTE_U+코드
kvmalloc+코드

설정되지 않은 경우, 페이지에 대한 참조가 오류를 발생시킵니다(즉, 허용되지 않음). PTE_W는 명령어가 페이지에 쓰기를 발행할 수 있는지 여부를 제어합니다. 설정되지 않은 경우 읽기 및 명령어 페치만 허용됩니다. PTE_U는 사용자 프로그램이 페이지를 사용할 수 있는지 여부를 제어합니다. 지우면 커널만 페이지를 사용할 수 있습니다. 그림 2-1은 모든 작동 방식을 보여줍니다. 플래그와 다른 모든 페이지 하드웨어 관련 구조는 mmu.h (0700)에 정의되어 있습니다.

용어에 대한 몇 가지 참고 사항. 물리적 메모리는 DRAM의 저장 셀을 말합니다. 물리적 메모리의 바이트는 물리적 주소라고 하는 주소를 갖습니다. 명령어는 가상 주소만 사용하는데, 페이징 하드웨어가 이를 물리적 주소로 변환한 다음 DRAM 하드웨어로 보내 스토리지를 읽거나 씁니다. 이 수준의 논의에서는 가상 메모리라는 것은 없고 가상 주소만 있습니다.

프로세스 주소 공간

entry가 만든 페이지 테이블에는 커널의 C 코드가 실행을 시작할 수 있을 만큼 충분한 매핑이 있습니다. 그러나 main은 kvmalloc (1840)을 호출하여 즉시 새 페이지 테이블로 변경합니다. 커널은 프로세스 주소 공간을 설명하는 데 더 정교한 계획을 가지고 있기 때문입니다.

각 프로세스는 별도의 페이지 테이블을 가지고 있으며, xv6는 xv6가 프로세스 간에 전환할 때 페이지 테이블 하드웨어에 페이지 테이블을 전환하라고 지시합니다. 그림 2-2에서 볼 수 있듯이 프로세스의 사용자 메모리는 가상 주소 0에서 시작하여 KERNBASE까지 커질 수 있으므로 프로세스는 최대 2기가바이트의 메모리를 주소 지정할 수 있습니다. 파일 memlayout.h (0200)

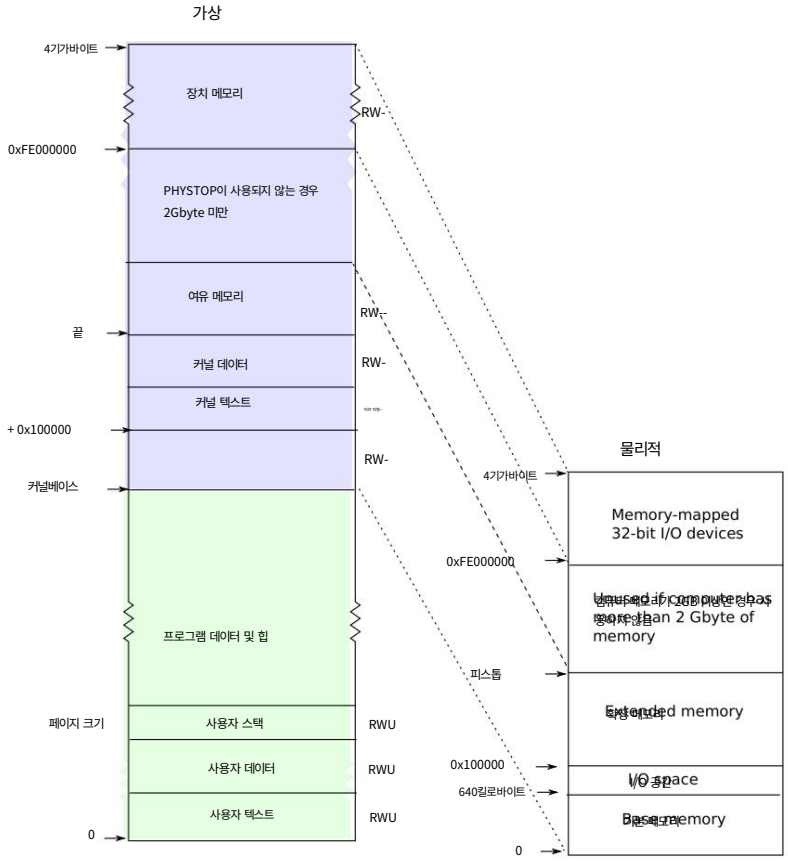


그림 2-2. 프로세스의 가상 주소 공간 레이아웃과 물리 주소 공간 레이아웃
공간. 머신에 2GB 이상의 실제 메모리가 있는 경우 xv6는 해당 메모리만 사용할 수 있습니다.
KERNBASE와 0xFE00000 사이에 들어맞습니다.

---

xv6의 메모리 레이아웃에 대한 상수와 가상 주소를 실제 주소로 변환하는 매크로를 선언합니다.

프로세스가 xv6에 더 많은 메모리를 요청하면 xv6는 먼저 사용 가능한 물리적 페이지를 찾습니다.
저장소를 제공한 다음 프로세스의 페이지 테이블에 해당 PTE를 추가합니다.
새로운 물리적 페이지. xv6는 이러한 PTE에서 PTE_U, PTE_W 및 PTE_P 플래그를 설정합니다. 대부분
프로세스는 전체 사용자 주소 공간을 사용하지 않습니다. xv6는 사용되지 않는 PTE_P를 비워 둡니다.
PTE. 다른 프로세스의 페이지 테이블은 사용자 주소를 다른 페이지로 변환합니다.
물리적 메모리, 즉 각 프로세스가 개인 사용자 메모리를 갖도록 합니다.

Xv6에는 모든 프로세스의 페이지 테이블에서 커널이 실행되는 데 필요한 모든 매핑이 포함되어 있습니다. 이러한
매핑은 모두 KERNBASE 위에 나타납니다. 가상 주소 KERN-BASE:KERNBASE+PHYSTOP을 0:PHYSTOP에 매
핑합니다. 이 매핑의 한 가지 이유는 다음과 같습니다.
커널은 자체 명령어와 데이터를 사용할 수 있습니다. 또 다른 이유는 커널이 일부
times는 예를 들어 주어진 물리적 메모리 페이지를 쓸 수 있어야 합니다.
페이지 테이블 페이지 생성; 모든 물리적 페이지가 예측 가능한 가상 위치에 표시됨
주소가 이것을 편리하게 만듭니다. 이 배열의 결함은 xv6가 만들 수 없다는 것입니다.
커널 주소 공간 부분이 2기가바이트이기 때문에 2기가바이트 이상의 물리적 메모리를 사용합니다. 따라서 xv6에서는 컴
퓨터에 2기가바이트 이상의 물리적 메모리가 있더라도 PHYSTOP이 2기가바이트보다 작아야 합니다.

메모리 매핑 I/O를 사용하는 일부 장치는 0xFE000000에서 시작하는 물리 주소에 나타나므로, xv6 페이지 테이블에는 직접 매핑이 포함됩니다. 따라서 PHYSTOP은 2기가바이트(16메가바이트(장치 메모리))보다 작아야 합니다.

Xv6는 KERNBASE 위의 PTE에서 PTE_U 플래그를 설정하지 않으므로 커널만 사용할 수 있습니다.

PTE_U+코드
PTE_U+코드
main+코드
kvmalloc+코드
setupkvm+코드
mappages+코드
kmap+코드 모든

프로세스의 페이지 테이블에 사용자 메모리와 PHYSTOP+코드 mappages+code 에 대한 매핑이 포함되어 있으면 walkpgdir+code 시스템 호출 및 인터럽트 중에 사용자 코드에서 커널 코드로 전환할 때 이러한 전환에는 페이지 테이블 전환이 필요하지 않습니다. walkpgdir+code 대부분의 경우 커널에는 자체 페이지 테이블이 없습니다. 거의 항상 PHYSTOP+code 일부 프로세스의 페이지 전체 커널이 편리합니다 . 테이블을 빌립니다 .

검토를 위해 xv6는 각 프로세스가 자체 메모리만 사용할 수 있도록 합니다. 그리고 각 프로세스는 메모리가 0에서 시작하는 연속된 가상 주소를 갖는 것으로 보는 반면, 프로세스의 물리적 메모리는 비연속적일 수 있습니다. xv6는 프로세스 자체 메모리를 참조하는 가상 주소의 PTE에만 PTE_U 비트를 설정하여 첫 번째를 구현합니다. 페이지 테이블이 연속된 가상 주소를 프로세스에 할당된 물리적 페이지로 변환하는 기능을 사용하여 두 번째를 구현합니다.

코드: 주소 공간 생성

main은 kvmalloc (1840) 을 호출하여 커널이 실행되는 데 필요한 KERNBASE 이상의 매핑을 사용하여 페이지 테이블을 만들고 전환합니다. 대부분의 작업은 setup-kvm (1818)에서 발생합니다. 먼저 페이지 디렉토리를 보관하기 위해 메모리 페이지를 할당합니다. 그런 다음 mappages를 호출하여 커널에 필요한 변환을 설치합니다. 이는 kmap (1809) 배열에 설명되어 있습니다. 변환에는 커널의 명령어와 데이터, PHYSTOP까지의 실제 메모리, 실제로는 I/O 장치인 메모리 범위가 포함됩니다. setup-kvm은 사용자 메모리에 대한 매핑을 설치하지 않습니다. 이는 나중에 수행됩니다.

mappages (1760) 는 가상 주소 범위에 대한 매핑을 해당 물리적 주소 범위에 대한 페이지 테이블에 설치합니다. 범위 내의 각 가상 주소에 대해 페이지 간격으로 별도로 이를 수행합니다. 매핑할 각 가상 주소에 대해 map-pages는 walkpgdir을 호출하여 해당 주소에 대한 PTE의 주소를 찾습니다. 그런 다음 PTE를 초기화하여 관련 물리적 페이지 번호, 원하는 권한(PTE_W 및/또는 PTE_U) 및 PTE_P를 보관하여 PTE를 유효로 표시합니다 (1772).

walkpgdir (1735) 는 가상 주소에 대한 PTE를 찾을 때 x86 페이징 하드웨어의 동작을 모방합니다(그림 2-1 참조). walkpgdir는 가상 주소의 상위 10비트를 사용하여 페이지 디렉토리 항목 (1740)을 찾습니다. 페이지 디렉토리 항목이 없으면 필요한 페이지 테이블 페이지가 아직 할당되지 않은 것입니다. alloc 인수가 설정된 경우 walkpgdir는 이를 할당하고 해당 물리 주소를 페이지 디렉토리에 넣습니다. 마지막으로 가상 주소의 다음 10비트를 사용하여 페이지 테이블 페이지에서 PTE의 주소를 찾습니다 (1753).

물리적 메모리 할당

커널은 페이지 테이블, 프로세스 사용자 메모리, 커널 스택 및 파이프 버퍼를 위해 런타임에 물리적 메모리를 할당하고 해제해야 합니다. xv6은 커널 끝과 PHYSTOP 사이의 물리적
메모리를 사용합니다.

런타임 할당. 한 번에 4096바이트 페이지를 모두 할당하고 해제합니다. 연결 목록을 페이지 자체에 스레딩하여 어떤 페이지가 해제되었는지 추적합니다.
할당은 연결 리스트에서 페이지를 제거하는 것으로 이루어지고, 해제는 해제된 페이지를 리스트에 추가하는 것으로 이루어진다.

구조체 실행+코드
메인+코드
kinit1+코드
kinit2+코드
PHYSTOP+코드 프리
레인자+코드 케이프리
+코드
PGROUNDUP+코드
타입 캐스트

부트스트랩 문제가 있습니다. 할당자가 자유 목록을 초기화하려면 모든 물리적 메모리를 매핑해야 하지만, 이러한 매핑으로 페이지 테이블을 만드는 것은 페이지 테이블 페이지를 할당하는 것을 포함합니다. xv6는 진입하는 동안 별도의 페이지 할당자를 사용하여 이 문제를 해결합니다. 이 페이지 할당자는 커널의 데이터 세그먼트가 끝난 직후에 메모리를 할당합니다. 이 할당자는 해제를 지원하지 않으며 진입 pgdir의 4MB 매핑으로 제한되지만, 첫 번째 커널 페이지 테이블을 할당하기에 충분합니다.

코드: 물리적 메모리 할당자 할당자의 데이터 구조는 할당 가능한 물리

적 메모리 페이지의 자유 목록 입니다. 각 자유 페이지의 목록 요소는 struct run (3115)입니다. 할당자는 해당 데이터 구조를 보관할 메모리를 어디에서 얻습니까? 자유 페이지 자체에 각 자유 페이지의 run 구조를 저장합니다. 거기에는 다른 것이 저장되어 있지 않기 때문입니다. 자유 목록은 스핀 잠금 (3119-3123)으로 보호됩니다. 목록과 잠금은 잠금이 구조체의 필드를 보호한다는 것을 명확히 하기 위해 구조체로 래핑됩니다. 지금은 잠금과 acquire 및 release에 대한 호출을 무시합니다. 4장에서 잠금을 자세히 살펴보겠습니다.

함수 main은 kinit1과 kinit2를 호출하여 할당자를 초기화합니다 (3131). 두 개의 호출이 있는 이유는 main의 대부분에서 4메가바이트 이상의 잠금이나 메모리를 사용할 수 없기 때문입니다. kinit1에 대한 호출은 처음 4메가바이트에서 잠금 없는 할당을 설정하고 kinit2에 대한 호출은 잠금을 활성화하고 더 많은 메모리를 할당할 수 있도록 합니다. main은 사용 가능한 실제 메모리 양을 결정해야 하지만 x86에서는 어려운 것으로 밝혀졌습니다. 대신 머신에 224가 있다고 가정합니다.

메가바이트(PHYSTOP)의 실제 메모리를 사용하고 커널 끝과 PHYSTOP 사이의 모든 메모리를 초기 여유 메모리 풀로 사용합니다. kinit1과 kinit2는 kfree에 대한 페이지별 호출을 통해 메모리를 여유 목록에 추가하기 위해 freerange를 호출합니다. PTE는 4096바이트 경계에 정렬된 실제 주소만 참조할 수 있으므로(4096의 배수임) freerange는 PGROUNDUP을 사용하여 정렬된 실제 주소만 해제합니다. 할당자는 메모리 없이 시작합니다. kfree에 대한 이러한 호출은 관리할 메모리를 제공합니다.

할당자는 물리적 주소가 아닌 하이 메모리에 매핑된 가상 주소로 물리적 페이지를 참조합니다. 이것이 kinit이 P2V(PHYSTOP)를 사용하여 PHYSTOP(물리적 주소)을 가상 주소로 변환하는 이유입니다. 할당자는 때때로 주소를 정수로 처리하여 산술 연산을 수행하고(예: kinit의 모든 페이지 탐색) 때때로 주소를 포인터로 사용하여 메모리를 읽고 씁니다(예: 각 페이지에 저장된 런 구조 조작). 이러한 주소의 이중 사용은 할당자 코드가 C 유형 캐스트로 가득 찬 주된 이유입니다. 또 다른 이유는 해제와 할당이 본질적으로 메모리의 유형을 변경하기 때문입니다.

kfree (3164) 함수는 해제되는 메모리의 모든 바이트를 값 1로 설정하는 것으로 시작합니다. 이렇게 하면 해제한 후 메모리를 사용하는 코드(''dangling references'' 사용)가 이전의 유효한 내용 대신 가비지를 읽게 됩니다. 바라건대 이런 코드가 더 빨리 중단되기를 바랍니다. 그런 다음 kfree는 v를 struct run에 대한 포인터로 캐스팅하고 다음을 기록합니다.

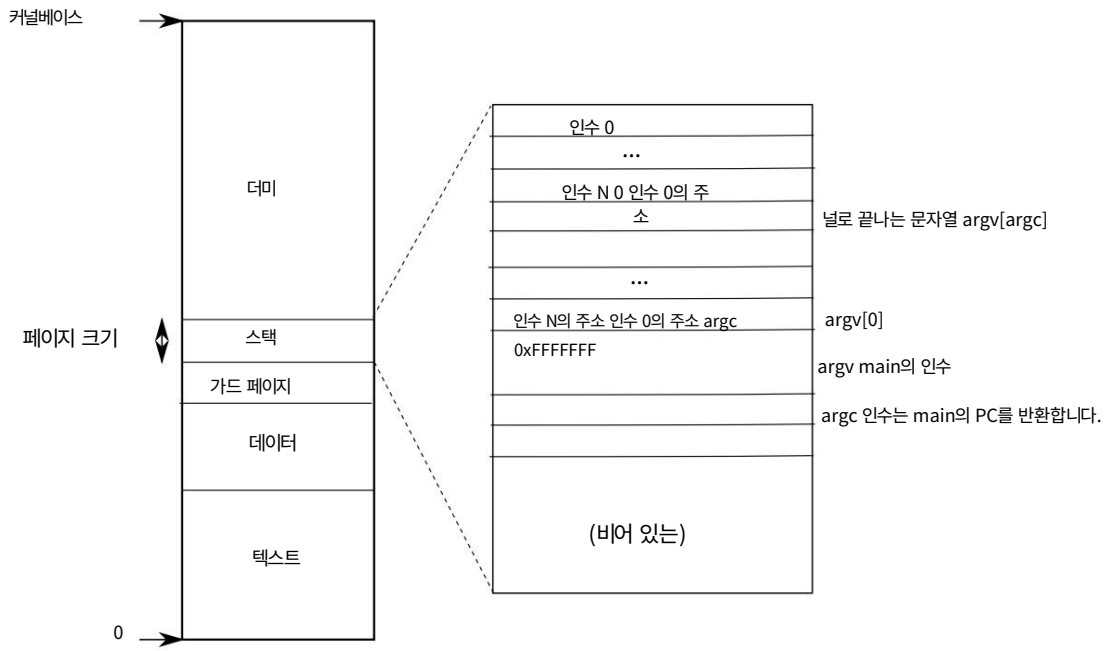


그림 2-3. 초기 스택이 있는 사용자 프로세스의 메모리 레이아웃.

---

r->next에서 자유 목록의 이전 시작을 반환하고, 자유 목록을 r과 동일하게 설정합니다. kalloc은 자유 목록의 첫 번째 요소를 제거하고 반환합니다.

칼로크+코드
sbrk+코드

주소 공간의 사용자 부분

그림 2-3은 xv6에서 실행 중인 프로세스의 사용자 메모리 레이아웃을 보여줍니다.
각 사용자 프로세스는 주소 0에서 시작합니다. 주소 공간의 맨 아래에는 사용자 프로그램의 텍스트, 데이터, 스택이 들어 있습니다. 힙은 스택 위에 있으므로 프로세스가 sbrk를 호출할 때 힙이 확장될 수 있습니다. 텍스트, 데이터, 스택 섹션은 프로세스의 주소 공간에 연속적으로 배치되지만 xv6는 해당 섹션에 비연속적인 물리적 페이지를 자유롭게 사용할 수 있습니다. 예를 들어, xv6가 프로세스의 힙을 확장할 때 새 가상 페이지에 사용 가능한 물리적 페이지를 사용한 다음 페이지 테이블 하드웨어를 프로그래밍하여 가상 페이지를 할당된 물리적 페이지에 매핑할 수 있습니다.

이러한 유연성은 페이징 하드웨어를 사용하는 주요 장점입니다.

스택은 단일 페이지이며 ex-ec에서 생성된 초기 내용과 함께 표시됩니다. 명령줄 인수를 포함하는 문자열과 해당 인수에 대한 포인터 배열은 스택의 맨 위에 있습니다. 바로 아래에는 함수 호출 main(argc, argv)가 방금 시작된 것처럼 프로그램이 main에서 시작할 수 있도록 하는 값이 있습니다. 스택 페이지에서 확장되는 스택을 보호하기 위해 xv6는 스택 바로 아래에 가드 페이지를 배치합니다. 가드 페이지는 매핑되지 않으므로 스택이 스택 페이지에서 실행되면 하드웨어가 오류 주소를 변환할 수 없기 때문에 예외를 생성합니다. 실제 운영 체제는 스택에 더 많은 공간을 할당하여 한 페이지 이상 확장될 수 있습니다.

코드: sbrk

Sbrk는 프로세스가 메모리를 축소하거나 확장하기 위한 시스템 호출입니다. 이 시스템 호출은 growproc (2558) 함수에 의해 구현됩니다. n이 양수이면 growproc는 하나 이상의 물리적 페이지를 할당하고 프로세스 주소 공간의 맨 위에 매핑합니다. n이 음수이면 growproc는 하나 이상의 페이지를 프로세스 주소 공간에서 매핑 해제하고 해당 물리적 페이지를 해제합니다. 이러한 변경을 하기 위해 xv6는 프로세스의 페이지 테이블을 수정합니다. 프로세스의 페이지 테이블은 메모리에 저장되므로 커널은 일반 할당 명령문으로 테이블을 업데이트할 수 있으며, 이는 allocuvm과 deallocuvm이 하는 일입니다. x86 하드웨어는 TLB(Translation Look-aside Buffer) 에 페이지 테이블 항목을 캐시 하고 xv6가 페이지 테이블을 변경하면 캐시된 항목을 무효화해야 합니다. 캐시된 항목을 무효화하지 않았다면 나중에 어느 시점에서 TLB가 오래된 매핑을 사용하여 그동안 다른 프로세스에 할당된 실제 페이지를 가리키고, 결과적으로 프로세스가 다른 프로세스의 메모리에 낙서할 수 있습니다. Xv6는 현재 페이지 테이블의 주소를 보관하는 레지스터인 cr3를 다시 로드하여 오래된 캐시된 항목을 무효화합니다.

코드: exec

Exec는 주소 공간의 사용자 부분을 만드는 시스템 호출입니다. 파일 시스템에 저장된 파일에서 주소 공간의 사용자 부분을 초기화합니다. Exec (6610)는 6장에서 설명하는 namei (6623)를 사용하여 명명된 바이너리 경로를 엽니다 . 그런 다음 ELF 헤더를 읽습니다. Xv6 애플리케이션은 elf.h에 정의된 널리 사용되는 ELF 형식으로 설명됩니다. ELF 바이너리는 ELF 헤더, struct elfhdr (0905), 그 뒤에 일련의 프로그램 섹션 헤더, struct proghdr (0924)로 구성됩니다. 각 progh-dr은 메모리에 로드해야 하는 애플리케이션 섹션을 설명합니다. xv6 프로그램에는 프로그램 섹션 헤더가 하나만 있지만 다른 시스템에는 명령어와 데이터에 대한 별도의 섹션이 있을 수 있습니다.

첫 번째 단계는 파일에 ELF 바이너리가 들어 있는지 빠르게 확인하는 것입니다. ELF 바이너리는 4바이트 ''매직 넘버'' 0x7F, 'E', 'L', 'F' 또는 ELF_MAGIC (0902)로 시작합니다. ELF 헤더에 올바른 매직 넘버가 있으면 exec는 바이너리가 잘 형성되었다고 가정합니다.

Exec은 setupkvm(6637)을 사용하여 사용자 매핑이 없는 새 페이지 테이블을 할당하고 , allocuvm (6651) 을 사용하여 각 ELF 세그먼트에 메모리를 할당하고 , loaduvm (6655) 을 사용하여 각 세그먼트를 메모리에 로드합니다 . allocuvm은 요청된 가상 주소가 KERNBASE 아래에 있는지 확인합니다. loaduvm (1903) 은 walkpgdir을 사용하여 ELF 세그먼트의 각 페이지를 쓸 할당된 메모리의 물리적 주소를 찾고, readi를 사용하여 파일에서 읽습니다.

exec로 생성된 첫 번째 사용자 프로그램인 /init의 프로그램 섹션 헤더
이렇게 보입니다:

```
# objdump -p _init

_초기화:     파일 형식 elf32-i386

프로그램 헤더:
    LOAD off 0x00000054 vaddr 0x00000000 paddr 0x00000000 정렬 2**2
```

filesz 0x000008c0 memsz 0x000008cc 플래그 rwx

allocuvm+코드
exec+코드
ustack+코드
argv+코드
argc+코드
copyout+코드

프로그램 섹션 헤더의 filesz는 memsz보다 작을 수 있는데, 이는 그 사이의 간격을 파일에서 읽는 대신 0으로 채워야 함을 나타냅니다(C 전역 변수의 경우). /init의 경우 filesz는 2240바이트이고 memsz는 2252바이트이므로 allocuvm은 2252바이트를 보관할 만큼 충분한 물리적 메모리를 할당하지만 파일 /init에서 2240바이트만 읽습니다.

이제 exec는 사용자 스택을 할당하고 초기화합니다. 스택 페이지 하나만 할당합니다.
Exec은 인수 문자열을 스택 맨 위에 한 번에 하나씩 복사하고, ustack에 해당 문자열에 대한 포인터를 기록합니다. main에 전달되는 argv 목록의 끝에 널 포인터를 배치합니다. ustack의 처음 세 항목은 가짜 리턴 PC, argc, argv 포인터입니다.

Exec는 스택 페이지 바로 아래에 접근할 수 없는 페이지를 배치하여, 여러 페이지를 사용하려는 프로그램이 오류를 발생시킵니다. 이 접근할 수 없는 페이지는 또한 exec가 너무 큰 인수를 처리할 수 있도록 합니다. 이 상황에서 exec가 인수를 스택에 복사하는 데 사용하는 copyout (2118) 함수는 대상 페이지에 접근할 수 없다는 것을 알아차리고 -1을 반환합니다.

새 메모리 이미지를 준비하는 동안 exec가 잘못된 프로그램 세그먼트와 같은 오류를 감지하면 레이블 bad로 점프하여 새 이미지를 해제하고 -1을 다시 반환합니다. Exec는 시스템 호출이 성공할 때까지 이전 이미지를 해제하기 위해 기다려야 합니다. 이전 이미지가 없어지면 시스템 호출은 -1을 반환할 수 없습니다. exec에서 유일한 오류 사례는 이미지 생성 중에 발생합니다. 이미지가 완료되면 exec는 새 이미지를 설치하고 (6701) 이전 이미지를 해제 할 수 있습니다 (6702). 마지막으로 exec는 0을 반환합니다.

Exec는 ELF 파일에서 바이트를 ELF 파일에 지정된 주소의 메모리로 로드합니다. 사용자나 프로세스는 원하는 주소를 ELF 파일에 넣을 수 있습니다. 따라서 exec는 위험합니다. ELF 파일의 주소가 실수로 또는 의도적으로 커널을 참조할 수 있기 때문입니다. 부주의한 커널에 대한 결과는 충돌에서 커널 격리 메커니즘의 악의적인 전복(즉, 보안 악용)에 이르기까지 다양할 수 있습니다. xv6는 이러한 위험을 피하기 위해 여러 가지 검사를 수행합니다. 이러한 검사의 중요성을 이해하려면 xv6가 if(ph.vaddr + ph.memsz < ph.vaddr)를 검사하지 않으면 어떤 일이 일어날 수 있는지 생각해 보세요. 이는 합계가 32비트 정수를 오버플로하는지 여부를 검사하는 것입니다. 위험한 점은 사용자가 커널을 가리키는 ph.vaddr와 합계가 0x1000으로 오버플로될 만큼 큰 ph.memsz를 사용하여 ELF 바이너리를 구성할 수 있다는 것입니다.

합계가 작으므로 allocuvm에서 if(newsz >= KERNBASE) 검사를 통과합니다.
이후 loaduvm을 호출하면 ph.memsz를 추가하지 않고 ph.vaddr을 KERNBASE와 비교하지 않고 ph.vaddr만 전달하여 ELF 바이너리에서 커널로 데이터를 복사합니다. 이는 사용자 프로그램에서 커널 권한으로 임의의 사용자 코드를 실행하는 데 악용될 수 있습니다. 이 예에서 알 수 있듯이 인수 검사는 매우 신중하게 수행해야 합니다. 커널 개발자가 중요한 검사를 생략하기 쉽고 실제 커널은 사용자 프로그램이 커널 권한을 얻기 위해 이러한 검사를 누락한 오랜 이력이 있습니다. xv6가 커널에 제공된 사용자 수준 데이터의 유효성을 완벽하게 검사하지 못할 가능성이 높으며, 악의적인 사용자 프로그램이 이를 악용하여 xv6의 격리를 우회할 수 있습니다.

현실 세계

seginit+코드
CR_PSE+코드

대부분의 운영 체제와 마찬가지로 xv6는 메모리 보호 및 매핑을 위해 페이징 하드웨어를 사용합니다. 대부분의 운영 체제는 x86의 64비트 페이징 하드웨어(3단계 변환)를 사용합니다. 64비트 주소 공간은 xv6보다 덜 제한적인 메모리 레이아웃을 허용합니다. 예를 들어, xv6의 물리적 메모리에 대한 2기가바이트 제한을 제거하는 것은 쉽습니다. 대부분의 운영 체제는 xv6보다 페이징을 훨씬 더 정교하게 사용합니다. 예를 들어, xv6는 디스크에서의 요구 페이징, 복사-쓰기 포크, 공유 메모리, 지연 할당 페이지 및 자동 확장 스택이 없습니다. x86은 세그먼테이션을 사용한 주소 변환을 지원하지만(부록 B 참조), xv6는 고정 주소에 있지만 다른 CPU에서 다른 값을 갖는 proc와 같은 CPU별 변수를 구현하는 일반적인 트릭에만 세그먼트를 사용합니다(seginit 참조). 세그먼트화되지 않은 아키텍처에서 CPU당(또는 스레드당) 저장소를 구현하려면 CPU당 데이터 영역에 대한 포인터를 보관하는 레지스터를 전담해야 하지만, x86에는 일반 레지스터가 너무 적어서 세그먼테이션을 사용하기 위해 필요한 추가적인 노력이 가치가 있습니다.

Xv6는 각 사용자 프로세스의 주소 공간에 커널을 매핑하지만 프로세서가 사용자 모드에 있을 때 주소 공간의 커널 부분에 액세스할 수 없도록 설정합니다. 이 설정은 프로세스가 사용자 공간에서 커널 공간으로 전환한 후 커널이 메모리 위치를 직접 읽어서 사용자 메모리에 쉽게 액세스할 수 있기 때문에 편리합니다. 그러나 보안을 위해서는 커널에 대한 별도의 페이지 테이블을 두고 사용자 모드에서 커널에 들어갈 때 해당 페이지 테이블로 전환하는 것이 더 나을 것입니다. 이렇게 하면 커널과 사용자 프로세스가 서로 더 분리됩니다. 예를 들어, 이 설계는 Meltdown 취약성에 노출되어 사용자 프로세스가 임의의 커널 메모리를 읽을 수 있는 사이드 채널을 완화하는 데 도움이 됩니다.

메모리가 많은 머신에서는 x86의 4메가바이트 ''슈퍼 페이지''를 사용하는 것이 합리적일 수 있습니다. 물리적 메모리가 작을 때 작은 페이지는 디스크에 할당 및 페이지 아웃을 미세하게 허용하기 위해 합리적입니다. 예를 들어, 프로그램이 8킬로바이트의 메모리만 사용하는 경우 4메가바이트의 물리적 페이지를 제공하는 것은 낭비입니다.
더 큰 페이지는 RAM이 많은 머신에서 의미가 있으며, 페이지 테이블 조작에 대한 오버헤드를 줄일 수 있습니다. Xv6는 한 곳에서 슈퍼 페이지를 사용합니다: 초기 페이지 테이블 (1306). 배열 초기화는 1024개 PDE 중 두 개를 인덱스 0과 512(KERNBASE>>PDXSHIFT)에 설정하고 다른 PDE는 0으로 둡니다. Xv6는 이 두 PDE에서 PTE_PS 비트를 설정하여 슈퍼 페이지로 표시합니다. 커널은 또한 %cr4에서 CR_PSE 비트(페이지 크기 확장)를 설정하여 페이징 하드웨어에 슈퍼 페이지를 허용하도록 지시합니다.

Xv6는 224MB를 가정하는 대신 실제 RAM 구성을 결정해야 합니다. x86에는 적어도 세 가지 일반적인 알고리즘이 있습니다. 첫 번째는 메모리처럼 동작하는 영역을 찾아 물리적 주소 공간을 조사하여 해당 영역에 기록된 값을 보존하는 것입니다. 두 번째는 PC의 비휘발성 RAM에 있는 알려진 16비트 위치에서 메모리의 킬로바이트 수를 읽는 것입니다. 세 번째는 멀티프로세서 테이블의 일부로 남아 있는 메모리 레이아웃 테이블을 BIOS 메모리에서 찾는 것입니다. 메모리 레이아웃 테이블을 읽는 것은 복잡합니다.

메모리 할당은 오래전부터 화제가 된 문제였는데, 그 기본적인 문제는 제한된 메모리의 효율적인 사용과 알 수 없는 미래 요청에 대비하는 것이었습니다. Knuth를 참조하세요.
오늘날 사람들은 공간 효율성보다 속도를 더 중시합니다. 게다가, 더 정교한 커널은 (xv6에서처럼) 4096바이트 블록만 할당하는 것이 아니라, 여러 가지 다른 크기의 작은 블록을 할당할 가능성이 높습니다. 실제 커널 할당자는 작은 할당을 처리해야 합니다.

대형뿐만 아니라 소형도 가능합니다.

수업 과정

1. 실제 운영 체제를 살펴보고 메모리 크기를 어떻게 지정하는지 확인하세요.
2. xv6가 슈퍼 페이지를 사용하지 않았다면 en-trypgdir에 대한 올바른 선언은 무엇일까요?

3. 다음을 호출하여 주소 공간을 1바이트로 늘리는 사용자 프로그램을 작성하세요.
sbrk(1). 프로그램을 실행하고 프로그램의 페이지 테이블을 조사합니다.
sbrk를 호출하고 sbrk를 호출한 후. 커널은 얼마나 많은 공간을 할당했습니까?
새로운 메모리의 PTE에는 무엇이 들어있나요?
4. 커널의 페이지가 프로세스 간에 공유되도록 xv6를 수정합니다.
메모리 소모를 줄입니다.
5. 사용자 프로그램이 널 포인터를 역참조할 때 오류를 수신하도록 xv6를 수정합니다. 즉, 가상 주소 0이 사용자 프로그램에 매핑되지 않도록 xv6를 수정합니다.

6. exec의 Unix 구현은 전통적으로 셸에 대한 특수 처리를 포함합니다.
스크립트. 실행할 파일이 #! 텍스트로 시작하면 첫 번째 줄은 #!로 간주됩니다.
파일을 해석하기 위해 실행할 프로그램입니다. 예를 들어, exec가 호출되어 myprog를 실행하면
arg1과 myprog의 첫 번째 줄은 #!/interp이고 exec는 명령과 함께 /interp를 실행합니다.
line /interp myprog arg1. xv6에서 이 규칙에 대한 지원을 구현합니다.
7. exec.c에서 if(ph.vaddr + ph.memsz < ph.vaddr) 검사를 삭제하고 검사가 누락되었다는 점을 악용하는 사용자 프로그램을 구성합니다.
8. 사용자 프로세스가 커널의 최소 부분으로만 실행되도록 xv6를 변경합니다.
매핑되어 커널이 해당 페이지 테이블을 포함하지 않는 자체 페이지 테이블로 실행되도록 합니다.
사용자 프로세스.
9. xv6가 64비트에서 실행되는 경우 xv6의 메모리 레이아웃을 어떻게 개선하시겠습니까?
프로세서?

3장

트랩, 인터럽트 및 드라이버

프로세스를 실행할 때 CPU는 일반적인 프로세서 루프를 실행합니다. 즉, 명령어를 읽고, 프로그램 카운터를 전진시키고, 명령어를 실행하고, 반복합니다. 그러나 사용자 프로그램의 제어를 다음 명령어를 실행하는 대신 커널로 다시 전송해야 하는 이벤트가 있습니다. 이러한 이벤트에는 주의가 필요하다는 신호를 보내는 장치, 불법적인 작업을 하는 사용자 프로그램(예: 페이지 테이블 항목이 없는 가상 주소를 참조하는 경우) 또는 시스템 호출을 통해 커널에 서비스를 요청하는 사용자 프로그램이 포함됩니다. 이러한 이벤트를 처리하는 데는 세 가지 주요 과제가 있습니다. 1) 커널은 프로세서가 사용자 모드에서 커널 모드로(그리고 다시) 전환되도록 해야 합니다. 2) 커널과 장치는 병렬 활동을 조정해야 합니다. 3) 커널은 장치의 인터페이스를 이해해야 합니다. 이러한 세 가지 과제를 해결하려면 하드웨어에 대한 자세한 이해와 신중한 프로그래밍이 필요하며, 불투명한 커널 코드가 생성될 수 있습니다. 이 장에서는 xv6가 이러한 세 가지 과제를 어떻게 해결하는지 설명합니다.

시스템 호출, 예외 및 인터럽트

제어를 사용자 프로그램에서 커널로 이전해야 하는 경우는 세 가지가 있습니다. 첫째, 시스템 호출: 사용자 프로그램이 운영 체제 서비스를 요청할 때, 이는 지난 장의 마지막에서 살펴본 바와 같습니다. 둘째, 예외 : 프로그램이 불법적인 동작을 수행할 때입니다. 불법적인 동작의 예로는 0으로 나누기, 존재하지 않는 페이지 테이블 항목에 대한 메모리 액세스를 시도하는 것 등이 있습니다. 셋째, 인터럽트 : 장치가 운영 체제의 주의가 필요하다는 신호를 생성할 때입니다. 예를 들어, 클록 칩은 커널이 시간 공유를 구현할 수 있도록 100밀리초마다 인터럽트를 생성할 수 있습니다. 또 다른 예로, 디스크가 디스크에서 블록을 읽을 때, 블록을 검색할 준비가 되었다는 것을 운영 체제에 알리기 위해 인터럽트를 생성합니다.

대부분의 경우 커널만이 필요한 권한과 상태를 가지고 있기 때문에 프로세스가 인터럽트를 처리하는 것이 아니라 커널이 모든 인터럽트를 처리합니다. 예를 들어, 클록 인터럽트에 응답하여 프로세스 간에 타임 슬라이스를 하려면 비협조적인 프로세스가 프로세서를 양보하도록 강제하기 위해서라도 커널이 개입해야 합니다.

세 가지 경우 모두 운영 체제 설계는 다음이 발생하도록 해야 합니다. 시스템은 미래의 투명한 재개를 위해 프로세서의 레지스터를 저장해야 합니다.
시스템은 커널에서 실행되도록 설정되어야 합니다. 시스템은 커널이 실행을 시작할 장소를 선택해야 합니다. 커널은 이벤트에 대한 정보(예: 시스템 호출 인수)를 검색할 수 있어야 합니다. 이 모든 것이 안전하게 수행되어야 합니다. 시스템은 사용자 프로세스와 커널의 격리를 유지해야 합니다.

이 목표를 달성하려면 운영 체제는 하드웨어가 시스템 호출, 예외 및 인터럽트를 처리하는 방법에 대한 세부 정보를 알고 있어야 합니다. 대부분의 프로세서에서 이 세 가지 이벤트는 단일 하드웨어 메커니즘으로 처리됩니다. 예를 들어, x86에서

프로그램은 int 명령어를 사용하여 인터럽트를 생성하여 시스템 호출을 호출합니다.
마찬가지로 예외도 인터럽트를 생성합니다. 따라서 운영 체제에 인터럽트 처리 계획이 있는 경우 운영 체제는 시스템 호출과 예외도 처리할 수 있습니다.

int+코드
인터럽트 핸들러 트랩
커널
모드 사용자 모드

기본 계획은 다음과 같습니다. 인터럽트는 일반 프로세서 루프를 중지하고 인터럽트 핸들러라는 새로운 시퀀스를 실행하기 시작합니다. 인터럽트 핸들러를 시작하기 전에 프로세서는 레지스터를 저장하여 운영 체제가 인터럽트에서 돌아올 때 이를 복원할 수 있도록 합니다. 인터럽트 핸들러로 전환하는 데 있어 어려운 점은 프로세서가 사용자 모드에서 커널 모드로 전환하고 다시 돌아와야 한다는 것입니다.

용어에 대한 설명: 공식 x86 용어는 예외이지만, xv6는 트랩이라는 용어를 사용합니다. 주로 PDP11/40에서 사용했고 따라서 일반적인 Unix 용어이기 때문입니다. 또한 이 장에서는 트랩과 인터럽트라는 용어를 서로 바꿔 사용하지만, 트랩은 프로세서에서 실행 중인 현재 프로세스에 의해 발생하고(예: 프로세스가 시스템 호출을 하고 결과적으로 트랩을 생성함) 인터럽트는 장치에 의해 발생하며 현재 실행 중인 프로세스와 관련이 없을 수 있다는 점을 기억하는 것이 중요합니다. 예를 들어, 디스크는 한 프로세스의 블록을 검색하는 것을 마쳤을 때 인터럽트를 생성할 수 있지만, 인터럽트가 발생하는 시점에 다른 프로세스가 실행 중일 수 있습니다. 인터럽트의 이러한 속성은 인터럽트가 다른 활동과 동시에 발생하기 때문에 트랩에 대해 생각하는 것보다 인터럽트에 대해 생각하는 것을 더 어렵게 만듭니다. 그러나 둘 다 사용자 모드와 커널 모드 간에 제어를 안전하게 전송하기 위해 동일한 하드웨어 메커니즘에 의존하는데, 이에 대해서는 다음에 논의하겠습니다.

X86 보호

x86에는 0(최대 권한)에서 3(최소 권한)까지 4가지 보호 수준이 있습니다.
실제로 대부분의 운영 체제는 0과 3의 두 가지 레벨만 사용하는데, 이를 각각 커널 모드 와 사용자 모드 라고 합니다 . x86이 명령어를 실행하는 현재 권한 레벨은 CPL 필드의 %cs 레지스터에 저장됩니다.

x86에서는 인터럽트 핸들러가 인터럽트 설명자 테이블(IDT)에 정의되어 있습니다.
IDT에는 256개의 항목이 있으며, 각 항목은 해당 인터럽트를 처리할 때 사용할 %cs와 %eip를 제공합니다.

x86에서 시스템 호출을 하려면 프로그램이 int n 명령어를 호출합니다.
n은 IDT에 대한 인덱스를 지정합니다. int 명령어는 다음 단계를 수행합니다. IDT에서 n번째 설명자를 가져옵

- 니다. 여기서 n 은 int의 인수입니다. • %cs의 CPL이 DPL보다 작거나 같은지 확인합니다.

여기서 DPL은 설명자의 권한 수준입니다.
필사자.

- 대상 세그먼트 선택기의 PL < CPL인 경우에만 CPU 내부 레지스터에 %esp 및 %ss를 저장합니다.

• 작업 세그먼트 설명자에서 %ss 및 %esp를 로드합니다. • %ss를 푸시합니다.

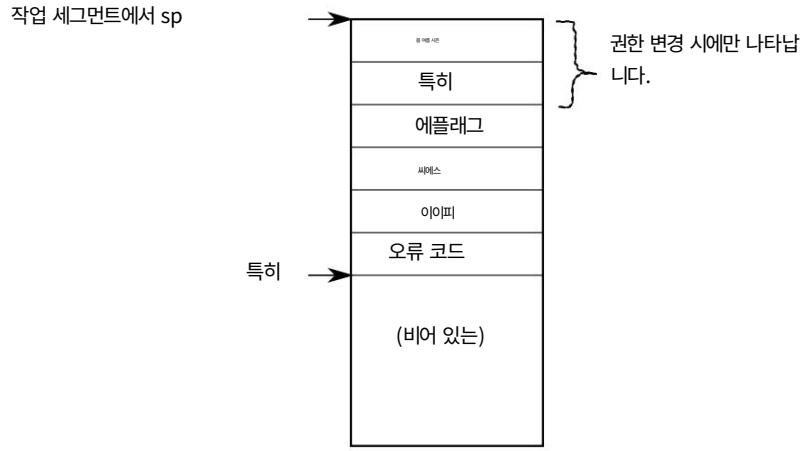


그림 3-1. int 명령어 이후의 커널 스택.

---

int+코드
iret+코드
int+코드
initcode.S+코드

- %esp를 푸시합니다.
- %eflags를 푸시합니다.
- %cs를 푸시합니다.
- %eip를 누르세요.
- %eflags의 IF 비트를 지우지만, 인터럽트에서만 그렇습니다.
- %cs와 %eip를 설명자의 값으로 설정합니다.

int 명령어는 복잡한 명령어이며, 이러한 모든 동작이 필요한지 궁금할 수 있습니다. 예를 들어, check CPL <= DPL 은 커널이 장치 인터럽트 루틴과 같은 부적절한 IDT 항목에 대한 int 호출을 금지할 수 있도록 합니다. 사용자 프로그램이 int를 실행하려면 IDT 항목의 DPL이 3이어야 합니다. 사용자 프로그램에 적절한 권한이 없으면 int는 int 13을 생성하는 데, 이는 일반 보호 오류입니다. 또 다른 예로, int 명령어는 프로세스에 유효한 스택 포인터가 없을 수 있으므로 사용자 스택을 사용하여 값을 저장할 수 없습니다. 대신 하드웨어는 커널에서 설정한 작업 세그먼트에 지정된 스택을 사용합니다.

그림 3-1은 int 명령어가 완료되고 권한 수준이 변경된 후의 스택을 보여줍니다(설명자의 권한 수준이 CPL보다 낮음). int 명령어가 권한 수준 변경을 필요로 하지 않으면 x86은 %ss와 %esp를 저장하지 않습니다. 두 경우 모두 %eip는 설명자 테이블에 지정된 주소를 가리키고 해당 주소의 명령어는 실행될 다음 명령어이며 int n에 대한 핸들러의 첫 번째 명령어입니다. 이러한 핸들러를 구현하는 것은 운영 체제의 역할이며 아래에서 xv6이 무엇을 하는지 살펴보겠습니다.

운영 체제는 iret 명령어를 사용하여 int 명령어에서 돌아올 수 있습니다. int 명령어 중에 저장된 값을 스택에서 팝하고 저장된 %eip에서 실행을 재개합니다.

코드: 첫 번째 시스템 호출

1장은 initcode.S가 시스템 호출을 호출하면서 끝났습니다. 다시 살펴보겠습니다.

(8414). 프로세스는 프로세스의 스택에 exec 호출에 대한 인수를 푸시하고, 시스템 호출 번호를 %eax에 넣었습니다. 시스템 호출 번호는 함수 포인터 테이블인 syscalls 배열의 항목과 일치합니다 (3672). int 명령어가 프로세서를 사용자 모드에서 커널 모드로 전환하고, 커널이 올바른 커널 함수(예: sys_exec)를 호출하고, 커널이 sys_exec 에 대한 인수를 검색할 수 있도록 해야 합니다. 다음 몇 개의 하위 섹션에서는 xv6가 시스템 호출에 대해 이를 어떻게 정렬하는지 설명하고, 그런 다음 인터럽트와 예외에 대해 동일한 코드를 재사용할 수 있다는 것을 알게 될 것입니다.

실행+코드
sys_exec+코드
int+코드
tvinit
메인+코드
idt+코드
벡터[i]+코드
T_SYSCALL+코드
IF+코드
DPL_USER+코드
switchuvm+코드
올트랩스+코드

코드: 어셈블리 트랩 핸들러

Xv6는 프로세서가 트랩을 생성하도록 하는 int 명령어를 만나면 x86 하드웨어가 합리적인 작업을 하도록 설정해야 합니다. x86은 256개의 다른 인터럽트를 허용합니다. 인터럽트 0-31은 나누기 오류나 잘못된 메모리 주소에 액세스하려는 시도와 같은 소프트웨어 예외에 대해 정의됩니다. Xv6는 32개의 하드웨어 인터럽트를 32-63 범위에 매핑하고 인터럽트 64를 시스템 호출 인터럽트로 사용합니다.

main에서 호출된 Tvinit (3367)는 idt 테이블에 256개 항목을 설정합니다. 인터럽트 i는 vectors[i]의 주소에 있는 코드에서 처리합니다. 각 진입점은 x86이 인터럽트 핸들러에 트랩 번호를 제공하지 않기 때문에 다릅니다. 256개의 다른 핸들러를 사용하는 것이 256개 케이스를 구별하는 유일한 방법입니다.

Tvinit은 특히 사용자 시스템 호출 트랩인 T_SYSCALL을 처리합니다. 두 번째 인수로 1 값을 전달하여 게이트가 ''트랩'' 유형임을 지정합니다. 트랩 게이트는 IF 플래그를 지우지 않으므로 시스템 호출 핸들러 중에 다른 인터럽트를 허용합니다.

커널은 또한 시스템 호출 게이트 권한을 DPL_USER로 설정하는데, 이를 통해 사용자 프로그램이 명시적 int 명령어로 트랩을 생성할 수 있습니다. xv6에서는 프로세스가 int로 다른 인터럽트(예: 장치 인터럽트)를 발생시키는 것을 허용하지 않습니다. 프로세스가 이를 시도하면 일반 보호 예외가 발생하고, 이 예외는 벡터 13으로 이동합니다.

보호 수준을 사용자 모드에서 커널 모드로 변경할 때 커널은 유효하지 않을 수 있으므로 사용자 프로세스의 스택을 사용해서는 안 됩니다. 사용자 프로세스가 악성이거나 오류가 있어서 사용자 %esp가 프로세스의 사용자 메모리에 속하지 않는 주소를 포함하게 될 수 있습니다. Xv6는 하드웨어가 스택 세그먼트 선택기와 %esp에 대한 새 값을 로드하는 작업 세그먼트 설명자를 설정하여 트랩에서 스택 스위치를 수행하도록 x86 하드웨어를 프로그래밍합니다. switchuvm (1860) 함수는 사용자 프로세스의 커널 스택 맨 위 주소를 작업 세그먼트 설명자에 저장합니다.

트랩이 발생하면 프로세서 하드웨어는 다음을 수행합니다. 프로세서가 사용자 모드에서 실행 중이었다면 작업 세그먼트 설명자에서 %esp 및 %ss를 로드하고 이전 사용자 %ss 및 %esp를 새 스택에 푸시합니다. 프로세서가 커널 모드에서 실행 중이었다면 위의 어떤 일도 일어나지 않습니다. 그런 다음 프로세서는 %eflags, %cs 및 %eip 레지스터를 푸시합니다. 일부 트랩(예: 페이지 오류)의 경우 프로세서는 오류 단어도 푸시합니다. 그런 다음 프로세서는 관련 IDT 항목에서 %eip 및 %cs를 로드합니다.

xv6는 Perl 스크립트 (3250) 를 사용하여 IDT 항목이 가리키는 진입점을 생성합니다. 각 항목은 프로세서가 그렇지 않은 경우 오류 코드를 푸시하고, 인터럽트 번호를 푸시한 다음 alltraps로 점프합니다.

Alltraps (3304) 는 프로세서 레지스터를 계속 저장합니다. %ds, %es, %fs를 푸시합니다.

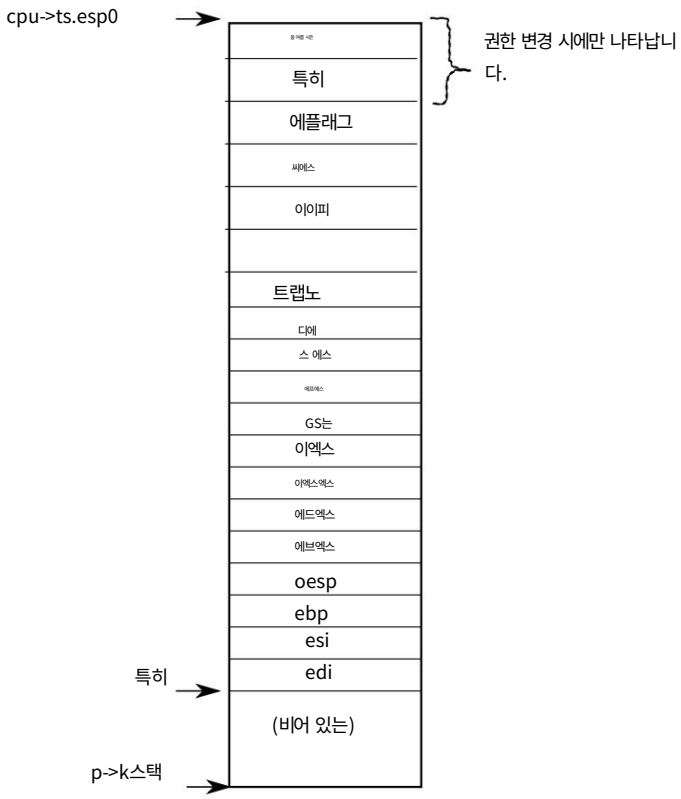


그림 3-2. 커널 스택의 트랩프레임

%gs 및 범용 레지스터 (3305-3310). 이러한 노력의 결과로 커널 스택에는 트랩 시점의 프로세서 레지스터를 포함하는 struct trapframe (0602) 이 포함됩니다 (그림 3-2 참조). 프로세서는 %ss, %esp, %eflags, %cs 및 %eip를 푸시합니다. 프로세서 또는 트랩 벡터는 오류 번호를 푸시하고 alltraps는 나머지를 푸시합니다. 트랩 프레임에는 커널이 현재 프로세스로 돌아갈 때 사용자 모드 프로세서 레지스터를 복원하는 데 필요한 모든 정보가 포함되어 있어 프로세서가 트랩이 시작되었을 때와 정확히 동일하게 계속할 수 있습니다.

올트랩+코드
올트랩+코드
올트랩+코드
트랩+코드
올트랩+코드

2장에서 userinit가 이 목표를 달성하기 위해 수동으로 트랩프레임을 구축했다는 점을 기억하세요(그림 1-4 참조).

첫 번째 시스템 호출의 경우 저장된 %eip는 int 명령어 바로 뒤에 있는 명령어의 주소입니다. %cs는 사용자 코드 세그먼트 선택기입니다. %eflags는 int 명령어를 실행할 때의 %eflags 레지스터의 내용입니다. 범용 레지스터를 저장하는 일환으로 alltraps는 커널이 나중에 검사할 수 있도록 시스템 호출 번호가 들어 있는 %eax도 저장합니다.

이제 사용자 모드 프로세서 레지스터가 저장되었으므로 alltraps는 커널 C 코드를 실행하기 위해 프로세서 설정을 완료할 수 있습니다. 프로세서는 핸들러에 들어가기 전에 선택자 %cs 및 %ss를 설정합니다. alltraps는 %ds 및 %es (3313-3315)를 설정합니다.

세그먼트가 제대로 설정되면 alltraps는 C 트랩 핸들러 트랩을 호출할 수 있습니다. 방금 구성한 트랩 프레임을 가리키는 %esp를 트랩에 대한 인수로 스택에 푸시합니다 (3318). 그런 다음 트랩을 호출합니다 (3319). 트랩이 반환된 후 alltraps는 스택 포인터에 추가하여 (3320) 스택에서 인수를 팝한 다음 실행을 시작합니다.

레이블 trapret에서 코드를 ing합니다. 우리는 2장에서 첫 번째 사용자 프로세스가 사용자 공간으로 나가기 위해 실행했을 때 이 코드를 추적했습니다. 여기에서도 동일한 시퀀스가 발생합니다. 트랩 프레임을 통해 팝핑하면 사용자 모드 레지스터가 복원되고 iret이 사용자 공간으로 다시 점프합니다.

지금까지의 논의는 사용자 모드에서 발생하는 트랩에 대해 이야기했지만, 커널이 실행되는 동안에도 트랩이 발생할 수 있습니다. 이 경우 하드웨어는 스택을 전환하거나 스택 포인터 또는 스택 세그먼트 선택기를 저장하지 않습니다. 그렇지 않으면 사용자 모드에서 발생하는 트랩과 동일한 단계가 발생하고 동일한 xv6 트랩 처리 코드가 실행됩니다.

나중에 iret이 커널 모드 %cs를 복원하면 프로세서는 커널 모드에서 계속 실행됩니다.

트랩렛+코드 아이렛+코드
트랩+코드 tf->트랩없음+코드
T_SYSCALL+코드 시스템 호출+코드
proc->killed+코드 트랩+코드 패닉+코드 트랩+코드 시스템 호출+코드
SYS_exec+코드
cp->tf+코드
syscall+코드

코드: C 트랩 핸들러

우리는 마지막 섹션에서 각 핸들러가 트랩 프레임을 설정한 다음 C 함수 트랩을 호출하는 것을 보았습니다. 트랩(3401) 은 하드웨어 트랩 번호 tf->trapno를 보고 왜 호출되었는지와 무엇을 해야 하는지 결정합니다. 트랩이 T_SYSCALL이면 트랩은 시스템 호출 핸들러 syscall을 호출합니다. 우리는 5장에서 proc->killed 검사를 다시 살펴볼 것입니다.

시스템 호출을 확인한 후, 트랩은 하드웨어 인터럽트를 찾습니다(아래에서 논의). 예상되는 하드웨어 장치 외에도, 트랩은 불필요한 인터럽트, 즉 원치 않는 하드웨어 인터럽트로 인해 발생할 수 있습니다.

트랩이 시스템 호출이 아니고 주의를 끌기 위한 하드웨어 장치가 아니라면, 트랩은 트랩이 트랩 전에 실행 중이던 코드의 일부로서 잘못된 동작(예: 0으로 나누기)으로 인해 발생했다고 가정합니다. 트랩을 발생시킨 코드가 사용자 프로그램인 경우, xv6는 세부 정보를 출력한 다음 proc->killed를 설정하여 사용자 프로세스를 정리하도록 기억합니다. 5장에서 xv6가 이 정리를 어떻게 수행하는지 살펴보겠습니다.

커널이 실행 중이었다면 커널 버그가 있어야 합니다. trap이 세부 정보를 인쇄합니다.
놀란 후에 당황감을 느낀다.

코드: 시스템 호출

시스템 호출의 경우 trap은 syscall (3701)을 호출합니다. Syscall은 저장된 %eax를 포함하는 trap 프레임에서 시스템 호출 번호를 로드하고 시스템 호출 테이블에 인덱싱합니다. 첫 번째 시스템 호출의 경우 %eax는 SYS_exec (3507) 값을 포함하고 syscall은 sys_exec를 호출하는 것과 같은 시스템 호출 테이블의 SYS_exec번째 항목을 호출합니다.

Syscall은 %eax에 시스템 호출 함수의 반환 값을 기록합니다. 트랩이 사용자 공간으로 돌아오면 cp->tf의 값을 머신 레지스터에 로드합니다. 따라서 exec가 돌아오면 시스템 호출 핸들러가 반환한 값 (3708)을 반환합니다. 시스템 호출은 일반적으로 오류를 나타낼 때는 음수를 반환하고 성공을 나타낼 때는 양수를 반환합니다. 시스템 호출 번호가 잘못되었으면 syscall은 오류를 인쇄하고 -1을 반환합니다.

이후 장에서는 특정 시스템 호출의 구현을 검토합니다. 이 장에서는 시스템 호출의 메커니즘에 대해 다룹니다. 한 가지 메커니즘이 남았습니다. 시스템 호출 인수를 찾는 것입니다. 도우미 함수 argint, argptr, argstr,

argfd는 n번째 시스템 호출 인수를 정수, 포인터, 문자열 또는 파일 설명자로 검색합니다. argint는 사용자 공간 %esp 레지스터를 사용하여 n번째 인수를 찾습니다. %esp는 시스템 호출 스텁의 반환 주소를 가리킵니다. 인수는 바로 위, %esp+4에 있습니다. 그런 다음 n번째 인수는 %esp+4+4*n에 있습니다.

argint+코드
fetchint+코드
p->sz+코드
argptr+코드
argstr+코드
argfd+코드

argint는 fetchint를 호출하여 사용자 메모리에서 해당 주소의 값을 읽고 *ip에 씁니다. fetchint는 사용자와 커널이 동일한 페이지 테이블을 공유하기 때문에 주소를 포인터로 간단히 캐스팅할 수 있지만 커널은 포인터가 주소 공간의 사용자 부분에 있는지 확인해야 합니다. 커널은 프로세스가 로컬 개인 메모리 외부의 메모리에 액세스할 수 없도록 페이지 테이블 하드웨어를 설정했습니다. 사용자 프로그램이 p->sz 이상의 주소에서 메모리를 읽거나 쓰려고 하면 프로세서가 세그먼테이션 트랩을 발생시키고 트랩은 위에서 본 것처럼 프로세스를 종료합니다. 그러나 커널은 사용자가 전달했을 수 있는 모든 주소를 역참조할 수 있으므로 주소가 p->sz 아래에 있는지 명시적으로 확인해야 합니다.

argptr는 n번째 시스템 호출 인수를 페치 하고 이 인수가 유효한 사용자 공간 포인터인지 확인합니다. argptr를 호출하는 동안 두 가지 확인이 발생한다는 점에 유의하세요. 먼저 인수를 페치하는 동안 사용자 스택 포인터를 확인합니다. 그런 다음 인수 자체인 사용자 포인터를 확인합니다.

argstr은 n번째 인수를 포인터로 해석합니다. 포인터가 NUL로 끝나는 문자열을 가리키고 전체 문자열이 주소 공간의 사용자 부분 끝 아래에 위치하도록 합니다.

마지막으로 argfd (6071) 는 argint를 사용하여 파일 기술자 번호를 검색하고 해당 번호가 맞는지 확인합니다. 유효한 파일 설명자이며 해당 구조체 파일을 반환합니다.

시스템 호출 구현(예: sysproc.c 및 sysfile.c)은 일반적으로 래퍼입니다. argint, argptr 및 argstr을 사용하여 인수를 디코딩한 다음 실제 구현을 호출합니다. 2장에서 sys_exec는 이러한 함수를 사용하여 인수를 가져옵니다.

코드: 인터럽트

마더보드의 장치는 인터럽트를 생성할 수 있으며, xv6는 이러한 인터럽트를 처리하기 위해 하드웨어를 설정해야 합니다. 장치는 일반적으로 I/O 완료와 같은 일부 하드웨어 이벤트가 발생했음을 커널에 알리기 위해 인터럽트를 발생시킵니다. 인터럽트는 일반적으로 커널이 대신 주기적으로 장치 하드웨어를 검사(또는 "폴링")하여 새 이벤트를 확인할 수 있다는 의미에서 선택 사항입니다. 이벤트가 비교적 드물기 때문에 폴링이 CPU 시간을 낭비할 경우 인터럽트를 폴링하는 것이 더 바람직합니다. 인터럽트 처리에서는 시스템 호출 및 예외에 이미 필요한 일부 코드를 공유합니다.

인터럽트는 시스템 호출과 비슷하지만, 장치가 언제든지 인터럽트를 생성한다는 점이 다릅니다.
마더보드에는 장치에 주의가 필요할 때(예: 사용자가 키보드에서 문자를 입력한 경우) CPU에 신호를 보내는 하드웨어가 있습니다. 우리는 장치가 인터럽트를 생성하도록 프로그래밍하고 CPU가 인터럽트를 수신하도록 해야 합니다.

타이머 장치와 타이머 인터럽트를 살펴보겠습니다. 타이머 하드웨어가 초당 100회 인터럽트를 생성하여 커널이 시간 경과를 추적하고 커널이 여러 실행 중인 프로세스 간에 타임 슬라이스를 수행할 수 있도록 하려고 합니다. 초당 100회를 선택하면 프로세서가 인터럽트를 처리하면서 압도당하지 않으면서도 적절한 대화형 성능을 얻을 수 있습니다.

x86 프로세서 자체와 마찬가지로 PC 마더보드도 진화했고 인터럽트를 제공하는 방식도 진화했습니다. 초기 보드에는 간단한 프로그래밍 가능 인터럽트 컨트롤러(PIC라고 함)가 있었습니다. 멀티프로세서 PC 보드가 등장하면서 인터럽트를 처리하는 새로운 방법이 필요했습니다. 각 CPU에 인터럽트 컨트롤러가 필요하여 전송된 인터럽트를 처리하고 프로세서에 인터럽트를 라우팅하는 방법이 있어야 하기 때문입니다. 이 방법은 두 부분으로 구성됩니다. I/O 시스템(IO APIC, ioapic.c)에 있는 부분과 각 프로세서에 연결된 부분(로컬 APIC, lapic.c)입니다. Xv6는 여러 프로세서가 있는 보드에 맞게 설계되었습니다. PIC에서 인터럽트를 무시하고 IOAPIC 및 로컬 APIC를 구성합니다.

라피니트+코드
IRQ_TIMER,+코드
IF+코드
cli+코드
sti+코드
switchuvm+코드
아이디티닛+코드
트랩+코드 웨
이크업+코드 드라
이버

IO APIC에는 테이블이 있으며 프로세서는 메모리 매핑된 I/O를 통해 테이블에 있는 항목을 프로그래밍할 수 있습니다. 초기화하는 동안 xv6는 인터럽트 0을 IRQ 0에 매핑하도록 프로그래밍하지만 모두 비활성화합니다. 특정 장치는 특정 인터럽트를 활성화하고 인터럽트를 어느 프로세서로 라우팅해야 하는지 알려줍니다. 예를 들어, xv6는 키보드 인터럽트를 프로세서 0 (8274)으로 라우팅합니다. Xv6는 디스크 인터럽트를 시스템에서 가장 높은 번호의 프로세서로 라우팅합니다(아래에서 볼 수 있음).

타이머 칩은 LAPIC 내부에 있으므로 각 프로세서가 타이머 인터럽트를 독립적으로 수신할 수 있습니다. Xv6는 lapicinit (7408)에서 이를 설정합니다. 핵심 라인은 타이머 (7421) 를 프로그래밍하는 라인입니다 . 이 라인은 LAPIC에 IRQ_TIMER(IRQ 0)에서 인터럽트를 주기적으로 생성하라고 지시합니다. 라인 (7451) 은 CPU의 LAPIC에서 인터럽트를 활성화하여 로컬 프로세서에 인터럽트를 전달합니다.

프로세서는 %eflags 레지스터의 IF 플래그를 통해 인터럽트를 수신할지 여부를 제어할 수 있습니다. 명령어 cli는 IF를 지워 프로세서에서 인터럽트를 비활성화하고 sti는 프로세서에서 인터럽트를 활성화합니다. Xv6는 메인 CPU (9112) 와 다른 프로세서 (1124)를 부팅하는 동안 인터럽트를 비활성화합니다. 각 프로세서의 스케줄러는 인터럽트를 활성화합니다 (2766). 특정 코드 조각이 중단되지 않도록 제어하기 위해 xv6는 이러한 코드 조각 동안 인터럽트를 비활성화합니다(예: switchuvm (1860) 참조).

타이머는 벡터 32(xv6가 IRQ 0을 처리하도록 선택)를 통해 인터럽트를 발생시키고, xv6는 idtinit (1255)에서 이를 설정합니다. 벡터 32와 벡터 64(시스템 호출용)의 유일한 차이점은 벡터 32가 트랩 게이트가 아닌 인터럽트 게이트라는 것입니다. 인터럽트 게이트는 IF를 지우므로 인터럽트된 프로세서는 현재 인터럽트를 처리하는 동안 인터럽트를 수신하지 않습니다. 여기에서 트랩까지 인터럽트는 시스템 호출 및 예외와 동일한 코드 경로를 따라 트랩 프레임을 구축합니다.

타이머 인터럽트에 대한 트랩은 두 가지 일만 합니다. ticks 변수 (3417) 를 증가시키고 , wakeup을 호출합니다. 후자는 5장에서 볼 수 있듯이 인터럽트가 다른 프로세스에서 반환되도록 할 수 있습니다.

운전자

드라이버 는 특정 장치를 관리하는 운영 체제의 코드입니다. 장치 하드웨어에 작업을 수행하도록 지시하고, 완료되면 장치가 인터럽트를 생성하도록 구성하고, 결과 인터럽트를 처리합니다. 드라이버 코드는 드라이버가 관리하는 장치와 동시에 실행되기 때문에 작성하기 어려울 수 있습니다. 또한 드라이버는 장치의 인터페이스(예: 어떤 I/O 포트가 무엇을 하는지)를 이해해야 하며, 해당 인터페이스는 복잡하고 제대로 문서화되지 않을 수 있습니다.

디스크 드라이버는 좋은 예를 제공합니다. 디스크 드라이버는 데이터를 복사합니다.

디스크로 돌아갑니다. 디스크 하드웨어는 전통적으로 디스크의 데이터를 512바이트 블록 (섹터 라고도 함)의 번호가 매겨진 시퀀스로 표시합니다. 섹터 0은 처음 512바이트이고 섹터 1은 그 다음입니다. 운영 체제가 파일 시스템에 사용하는 블록 크기는 디스크가 사용하는 섹터 크기와 다를 수 있지만 일반적으로 블록 크기는 섹터 크기의 배수입니다. Xv6의 블록 크기는 디스크의 섹터 크기와 동일합니다. 블록을 나타내기 위해 xv6에는 struct buf (3850) 구조가 있습니다. 이 구조에 저장된 데이터는 종종 디스크와 동기화되지 않습니다. 아직 디스크에서 읽히지 않았을 수도 있고(디스크가 작업 중이지만 아직 섹터의 내용을 반환하지 않음) 업데이트되었지만 아직 쓰여지지 않았을 수도 있습니다. 드라이버는 구조가 디스크와 동기화되지 않을 때 나머지 xv6가 혼동되지 않도록 해야 합니다.

코드: 디스크 드라이버

IDE 장치는 PC 표준 IDE 컨트롤러에 연결된 디스크에 대한 액세스를 제공합니다. IDE는 이제 SCSI와 SATA에 밀려 유행에서 벗어나고 있지만, 인터페이스가 간단하고 특정 하드웨어의 세부 사항 대신 드라이버의 전체 구조에 집중할 수 있습니다.

Xv6는 struct buf (3850)를 사용하여 파일 시스템 블록을 나타냅니다. BSIZE (4055) 는 IDE의 섹터 크기와 동일하므로 각 버퍼는 특정 디스크 장치의 한 섹터의 내용을 나타냅니다. dev 및 sector 필드는 장치 및 섹터 번호를 제공하고 data 필드는 디스크 섹터의 메모리 내 복사본입니다. xv6 파일 시스템은 BSIZE를 IDE의 섹터 크기와 동일하게 선택하지만 드라이버는 섹터 크기의 배수인 BSIZE를 처리할 수 있습니다. 운영 체제는 종종 더 높은 디스크 처리량을 얻기 위해 512바이트보다 큰 블록을 사용합니다.

플래그는 메모리와 디스크 간의 관계를 추적합니다. B_VALID 플래그는 데이터가 읽혀졌음을 의미하고, B_DIRTY 플래그는 데이터를 써야 함을 의미합니다.

커널은 부팅 시 main(1232)에서 ideinit (4251)을 호출하여 디스크 드라이버를 초기화합니다 . Ideinit는 ioapicenable을 호출하여 IDE_IRQ 인터럽트 (4256)를 활성화합니다. ioapicenable에 대한 호출은 마지막 CPU(ncpu-1)에서만 인터럽트를 활성화합니다. 2개 프로세서 시스템에서 CPU 1은 디스크 인터럽트를 처리합니다.

다음으로 ideinit는 디스크 하드웨어를 조사합니다. 디스크가 명령을 수락할 수 있을 때까지 기다리기 위해 idewait (4257)를 호출하는 것으로 시작합니다. PC 마더보드는 I/O 포트 0x1f7에서 디스크 하드웨어의 상태 비트를 표시합니다. Idewait (4238)는 바쁜 비트(IDE_BSY)가 지워지고 준비 비트(IDE_DRDY)가 설정될 때까지 상태 비트를 폴링합니다.

이제 디스크 컨트롤러가 준비되었으므로 ideinit는 현재 디스크가 몇 개 있는지 확인할 수 있습니다. 부트 로더와 커널이 모두 디스크 0에서 로드되었기 때문에 디스크 0이 있다고 가정하지만 디스크 1을 확인해야 합니다. 디스크 1을 선택하기 위해 I/O 포트 0x1f6에 쓰고 디스크가 준비되었음을 나타내는 상태 비트 (4259-4266)가 표시될 때까지 잠시 기다립니다. 그렇지 않으면 ideinit는 디스크가 없다고 가정합니다.

ideinit 후, 디스크는 버퍼 캐시가 iderw를 호출할 때까지 다시 사용되지 않으며, iderw는 플래그에 표시된 대로 잠긴 버퍼를 업데이트합니다. B_DIRTY가 설정된 경우 iderw는 버퍼를 디스크에 씁니다. B_VALID가 설정되지 않은 경우 iderw는 디스크에서 버퍼를 읽습니다.

디스크 액세스는 일반적으로 밀리초가 걸리며 프로세서에게는 긴 시간입니다. 부팅

로더는 디스크 읽기 명령을 내리고 데이터가 준비될 때까지 상태 비트를 반복적으로 읽습니다(부록 B 참조). 이 폴링 또는 바쁜 대기는 더 나은 방법이 없는 부트 로더에서는 괜찮습니다. 그러나 운영 체제에서는 다른 프로세스가 CPU에서 실행되도록 하고 디스크 작업이 완료되면 인터럽트를 받도록 하는 것이 더 효율적입니다. Iderw는 후자의 접근 방식을 취하여 보류 중인 디스크 요청 목록을 대기열에 보관하고 인터럽트를 사용하여 각 요청이 완료된 시점을 확인합니다. iderw는 요청 대기열을 유지 관리하지만 간단한 IDE 디스크 컨트롤러는 한 번에 하나의 작업만 처리할 수 있습니다. 디스크 드라이버는 대기열의 앞에 있는 버퍼를 디스크 하드웨어로 보냈다는 불변성을 유지합니다. 다른 프로세스는 단순히 차례를 기다리고 있습니다.

폴링 바
쁜 대기
iderw+code
idestart+code
idestart+code
iderw+code
트랩+코드
ideintr+코드
insl+코드
B_VALID+코드
B_DIRTY+코드

Iderw (4354) 는 버퍼 b를 큐의 끝에 추가합니다 (4367-4371). 버퍼가 큐의 앞에 있는 경우 iderw는 idestart (4326-4328)를 호출하여 디스크 하드웨어로 보내야 합니다 . 그렇지 않으면 버퍼는 앞에 있는 버퍼가 처리되면 시작됩니다.

Idestart (4274) 는 플래그에 따라 버퍼의 장치와 섹터에 대한 읽기 또는 쓰기를 실행합니다. 작업이 쓰기인 경우 idestart는 지금 데이터를 제공해야 합니다 (4296). idestart는 outsl in-명령을 사용하여 디스크 컨트롤러의 버퍼로 데이터를 이동합니다. CPU 명령을 사용하여 장치 하드웨어로/장치 하드웨어에서 데이터를 이동하는 것을 프로그래밍된 I/O라고 합니다. 결국 디스크 하드웨어는 데이터가 디스크에 쓰여졌다는 신호를 보내기 위해 인터럽트를 발생시킵니다. 작업이 읽기인 경우 인터럽트는 데이터가 준비되었다는 신호를 보내고 핸들러는 이를 읽습니다. idestart는 IDE 장치에 대한 자세한 지식을 가지고 있으며 올바른 포트에 올바른 값을 씁니다. 이러한 outb 문 중 하나라도 잘못되면 IDE는 우리가 원하는 것과 다른 작업을 수행합니다. 이러한 세부 정보를 올바르게 얻는 것이 장치 드라이버를 작성하는 것이 어려운 이유 중 하나입니다.

요청을 큐에 추가하고 필요한 경우 시작한 후 iderw는 결과를 기다려야 합니다. 위에서 논의했듯이 폴링은 CPU를 효율적으로 사용하지 않습니다. 대신 iderw는 다른 프로세스에 CPU를 양보하여 휴면 상태로 만들고 인터럽트 핸들러가 버퍼의 플래그에 작업이 완료되었음을 기록할 때까지 기다립니다 (4378-4379).

이 프로세스가 절전 모드에 있는 동안 xv6는 다른 프로세스를 예약하여 CPU를 바쁘게 유지합니다.

결국 디스크는 작업을 마치고 인터럽트를 트리거합니다. trap은 ideintr를 호출하여 이를 처리합니다 (3424). ideintr (4304) 는 대기열의 첫 번째 버퍼를 참조하여 어떤 작업이 진행 중인지 알아냅니다. 버퍼가 읽히고 디스크 컨트롤러에 대기 중인 데이터가 있는 경우 ideintr는 insl (4317-4319)을 사용하여 디스크 컨트롤러의 버퍼에서 메모리로 데이터를 읽습니다. 이제 버퍼가 준비되었습니다. ideintr는 B_VALID를 설정하고 B_DIRTY를 지우고 버퍼에서 잠자고 있는 모든 프로세스를 깨웁니다 (4321-4324). 마지막으로 ideintr는 다음 대기 버퍼를 디스크로 전달해야 합니다 (4326-4328).

현실 세계

PC 마더보드의 모든 장치를 온전히 지원하는 것은 많은 작업입니다. 장치가 많고, 장치가 많은 기능을 가지고 있으며, 장치와 드라이버 간의 프로토콜이 복잡할 수 있기 때문입니다. 많은 운영 체제에서 드라이버가 합쳐지면 코어 커널보다 운영 체제에서 더 많은 코드를 차지합니다.

실제 장치 드라이버는 이 장의 디스크 드라이버보다 훨씬 더 복잡합니다.

하지만 기본 아이디어는 동일합니다. 일반적으로 장치는 CPU보다 느리므로 하드웨어는 인터럽트를 사용 일괄
하여 운영 체제에 상태 변경을 알립니다. 최신 디스크
컨트롤러는 일반적으로 한 번에 여러 개의 디스크 요청을 수락하고 이를 다시 정렬하기도 합니다 .
디스크 암을 가장 효율적으로 활용하세요. 디스크가 더 간단했을 때 운영 체제
요청 대기열을 스스로 재정렬하는 경우가 많습니다.

많은 운영 체제에는 솔리드 스테이트 디스크용 드라이버가 있습니다.
데이터에 대한 훨씬 빠른 액세스. 그러나 솔리드 스테이트 디스크는 다음과 매우 다르게 작동합니다.
기존의 기계적 디스크와 마찬가지로 두 장치 모두 블록 기반 인터페이스를 제공하며 솔리드 스테이트 디
스크에서 블록을 읽고 쓰는 것은 여전히 읽고 쓰는 것보다 비쌉니다.
숫양.

다른 하드웨어는 놀랍게도 디스크와 유사합니다. 네트워크 장치 버퍼는 패킷을 보관하고 오디오 장
치 버퍼는 사운드 샘플을 보관하고 그래픽 카드 버퍼는 비디오 데이터를 보관합니다.
명령 시퀀스. 고대역폭 장치 - 디스크, 그래픽 카드 및 네트워크
카드는 종종 프로그래밍된 I/O 대신 DMA(직접 메모리 액세스)를 사용합니다.
outsl). DMA는 장치가 물리적 메모리에 직접 액세스할 수 있도록 합니다. 드라이버는 다음을 제공합니다.
장치 버퍼 데이터의 물리적 주소와 장치가 직접 복사하거나
주 메모리에서 복사가 완료되면 중단합니다. DMA는 더 빠르고 더 많습니다.
프로그래밍된 I/O보다 효율적이며 CPU 메모리 캐시에 대한 부담도 적습니다.

일부 드라이버는 폴링과 인터럽트 사이를 동적으로 전환합니다.
인터럽트는 비용이 많이 들 수 있지만 폴링을 사용하면 드라이버가 이벤트를 처리할 때까지 지연이 발생
할 수 있습니다. 예를 들어, 패킷 버스트를 수신하는 네트워크 드라이버는
더 많은 패킷을 처리해야 한다는 사실을 알고 있으므로 인터럽트에서 폴링으로 전환합니다.
그리고 폴링을 사용하여 처리하는 것이 더 저렴합니다. 더 이상 패킷이 필요하지 않으면
패킷이 처리되면 드라이버가 인터럽트로 다시 전환하여 새로운 패킷이 도착하면 즉시 알림을 받을 수 있
습니다.

IDE 드라이버는 인터럽트를 특정 프로세서에 정적으로 라우팅합니다. 일부 드라이버는
인터럽트를 여러 프로세서로 라우팅하여 분산시키도록 IO APIC를 구성합니다.
패킷 처리 작업. 예를 들어, 네트워크 드라이버는 다음을 전달하도록 준비할 수 있습니다.
프로세서가 관리하는 한 네트워크 연결의 패킷에 대한 인터럽트
그 연결은 다른 연결의 패킷에 대한 인터럽트가 다른 프로세서로 전달되는 동안 이루어집니다. 이 라우팅
은 매우 복잡해질 수 있습니다. 예를 들어, 어떤 네트워크가
연결은 수명이 짧지만 다른 연결은 수명이 길며 운영 체제가 원합니다.
높은 처리량을 달성하기 위해 모든 프로세서를 바쁘게 유지합니다.

프로그램이 파일을 읽으면 해당 파일의 데이터가 두 번 복사됩니다. 먼저 복사됩니다.
드라이버에 의해 디스크에서 커널 메모리로 복사되고 나중에 커널에서 복사됩니다.
읽기 시스템 호출을 통해 사용자 공간으로 공간을 전송합니다. 그런 다음 프로그램이 데이터를 전송하면
네트워크에서 데이터는 사용자 공간에서 커널 공간으로, 그리고
네트워크 장치에 커널 공간을 제공합니다. 효율성이 중요한 애플리케이션(예: 웹에서 인기 있는 이미지 제
공)을 지원하기 위해 운영 체제는 특수 코드를 사용합니다.
복사를 피하기 위한 경로. 예를 들어, 실제 운영 체제에서 버퍼는 일반적으로 하드웨어 페이지 크기와 일
치하므로 읽기 전용 복사본은 복사 없이 페이징 하드웨어를 사용하여 프로세스의 주소 공간에 매핑될
수 있습니다.

수업 과정

1. syscall의 첫 번째 명령어에 중단점을 설정하여 첫 번째 시스템 호출(예: br syscall)을 잡습니다. 이 지점에서 스택에 어떤 값이 있습니까? 각 값이 무엇인지 레이블이 지정된 해당 중단점에서 x/37x $esp의 출력을 설명합니다(예: saved
(트랩의 경우 %ebp, trapframe.eip, 스크래치 공간 등).

2. 현재 UTC 시간을 가져와 사용자에게 반환하는 새로운 시스템 호출을 추가합니다.
프로그램. 실제를 읽으려면 도우미 함수인 cmostime (7552) 을 사용할 수 있습니다 .
시간 시계. 파일 date.h에는 struct rtcdate (0950) 의 정의가 포함되어 있습니다 .
cmostime에 포인터를 인수로 제공합니다.

3. SATA 표준을 지원하는 디스크용 드라이버를 작성합니다(SATA 검색).
웹). IDE와 달리 SATA는 더 이상 사용되지 않습니다. SATA의 태그가 지정된 명령 대기열을 사용하여
디스크가 내부적으로 명령을 재정렬할 수 있도록 디스크에 많은 명령을 내립니다.
높은 성과를 얻습니다.

4. 이더넷 카드에 대한 간단한 드라이버를 추가합니다.

4장

잠금

Xv6는 멀티프로세서에서 실행됩니다. 즉, 여러 CPU가 독립적으로 실행되는 컴퓨터입니다. 이러한 여러 CPU는 물리적 RAM을 공유하고 xv6는 공유를 활용하여
모든 CPU가 읽고 쓰는 데이터 구조를 유지합니다. 이 공유는 한 CPU가 데이터 구조를 읽는 동안 다른 CPU가 데이터 구조를 업데이트하는 중간 단계일 가능성이 높거나, 심지어 여러 CPU가 동시에 같은 데이터를 업데이트할 가능성이 높습니다. 신중하게 설계하지 않으면 이러한 병렬 액세스는 잘못된 결과나 깨진 데이터 구조를 생성할 가능성이 높습니다. 단일 프로세서에서도 인터럽트 불가능한 코드와 같은 데이터를 사용하는 인터럽트 루틴은 인터럽트가 잘못된 시간에 발생하면 데이터를 손상시킬 수 있습니다.

공유 데이터에 동시에 액세스하는 모든 코드는 동시성에도 불구하고 정확성을 유지하기 위한 전략이 있어야 합니다. 동시성은 다음에 의해 액세스될 수 있습니다.
여러 코어, 또는 여러 스레드, 또는 인터럽트 코드. xv6는 소수의 간단한 동시성 제어 전략을 사용하며, 훨씬 더 정교해질 수 있습니다. 이 장에서는
xv6 및 기타 여러 시스템에서 광범위하게 사용되는 전략 중 하나에 초점을 맞춥니다.
잠그다.

잠금은 상호 배제를 제공하여 한 번에 하나의 CPU만 보유할 수 있도록 보장합니다.
잠금. 잠금이 각 공유 데이터 항목과 연관되어 있고 코드가 항상 유지되는 경우
주어진 항목을 사용할 때 연관된 잠금이 있으면 항목이 사용되었는지 확인할 수 있습니다.
한 번에 하나의 CPU에서만 가능합니다. 이 상황에서 우리는 잠금이 데이터를 보호한다고 말합니다.
목.

이 장의 나머지 부분에서는 xv6에 잠금이 필요한 이유와 xv6가 잠금을 구현하는 방법을 설명합니다.
그리고 그것을 어떻게 사용하는지. 중요한 관찰은 xv6의 일부 코드를 살펴보면,
다른 프로세서(또는 인터럽트)가 의도한 것을 변경할 수 있는지 스스로에게 물어봐야 합니다.
데이터(또는 하드웨어 리소스)를 수정하여 코드의 동작을 변경합니다.
단일 C 명령문은 여러 개의 기계 명령어일 수 있다는 점을 명심해야 합니다.
따라서 다른 프로세서나 인터럽트가 C 문장의 중간에 어지럽혀질 수 있습니다. 페이지의 코드 줄이 원자적으로 실행된다고 가정할 수 없습니다.
동시성은 정확성에 대한 추론을 훨씬 어렵게 만듭니다.

경쟁 조건

잠금이 필요한 이유의 예로, 단일 잠금을 공유하는 여러 프로세서를 생각해 보세요.
디스크, 예를 들어 xv6의 IDE 디스크. 디스크 드라이버는 미처리 디스크 요청 (4226) 의 연결 목록을 유지 관리하고
프로세서는 목록에 새 요청을 동시에 추가할 수 있습니다 (4354). 동시 요청이 없는 경우 연결 목록을 구현할 수 있습니다.
다음과 같습니다:

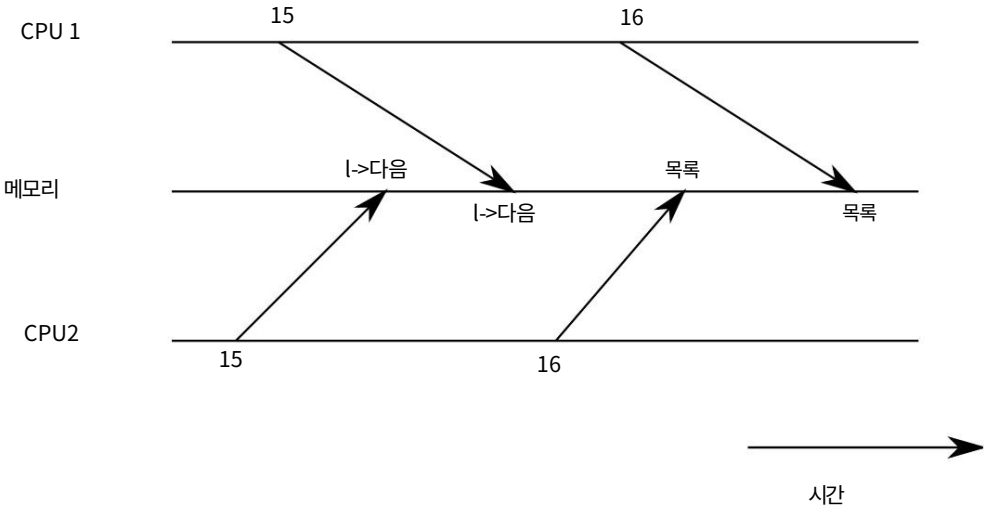


그림 4-1. 예시 레이스

경쟁 조건

```
1   구조체 리스트 {
2       int 데이터;
3       구조체 리스트 *next;
4   };
5
6   구조체 리스트 *리스트 = 0;
7
8   무효의
9   삽입(int 데이터)
10  {
11      구조체 리스트 *l;
12
13      l = malloc(크기 *l);
14      l->데이터 = 데이터;
15      l->다음 = 목록;
16      리스트 = l;
17  }
```

이 구현은 격리되어 실행되면 정확합니다. 그러나 두 개 이상의 사본이 동시에 실행되면 코드가 정확하지 않습니다. 두 개의 CPU가 삽입을 실행하는 경우
동시에, 둘 다 16번째 줄을 실행하기 전에 15번째 줄을 실행할 수도 있습니다(참조).
그림 4-1). 이런 일이 발생하면 이제 next가 list의 이전 값으로 설정된 두 개의 목록 노드가 생깁니다. 목록에 대한 두 가지 할당이 16번째 줄에서 발생하면 두 번째
하나는 첫 번째 것을 덮어쓰게 되고, 첫 번째 할당에 관련된 노드는 손실됩니다.

16번째 줄에서 손실된 업데이트는 경쟁 조건의 예입니다 . 경쟁 조건은
메모리 위치가 동시에 액세스되고 적어도 하나의 액세스가 발생하는 상황
쓰기입니다. 경쟁은 종종 버그의 신호이거나 업데이트 손실(액세스가 있는 경우)입니다.
쓰기) 또는 불완전하게 업데이트된 데이터 구조의 읽기. 레이스의 결과
관련된 두 CPU의 정확한 타이밍과 메모리 시스템에서 메모리 작업이 어떻게 정렬되는지에 따라 달라지므로 경쟁으로 인한 오류가 발생하기 어려울 수 있습니다.
재현하고 디버깅합니다. 예를 들어, 디버깅하는 동안 인쇄 명령문을 추가합니다.

sert는 경쟁이 사라질 만큼 실행 타이밍을 바꿀 수도 있습니다.

경쟁을 피하는 일반적인 방법은 잠금 장치를 사용하는 것입니다. 잠금 장치는 상호 배제를 보장하므로 한 번에 하나의 CPU만 삽입을 실행할 수 있다는 점입니다. 이는 위의 시나리오를 불가능하게 만듭니다. 위 코드의 올바르게 잠긴 버전은 몇 줄만 추가합니다(
번호가 매겨짐):

```
6       구조체 리스트 *리스트 = 0;
        구조체 잠금 목록 잠금;
7
8       무효의
9       삽입(int 데이터)
10      {
11          구조체 리스트 *l;
12          l = malloc(크기 *l);
13          l->데이터 = 데이터;
14
            획득(&listlock);
15          l->다음 = 목록;
16          리스트 = l;
            해제(&listlock);
17      }
```

획득과 해제 사이의 명령 시퀀스를 종종 중요 명령이라고 합니다.
섹션 과 잠금장치는 목록을 보호합니다.

잠금 장치가 데이터를 보호한다고 말할 때 실제로는 잠금 장치가 일부 데이터를 보호한다는 의미입니다.
데이터에 적용되는 불변식의 모음. 불변식은 연산 간에 유지되는 데이터 구조의 속성입니다. 일반적으로 연산의 올바른 동작
작업이 시작될 때 불변식이 참인지에 따라 달라집니다. 작업은
일시적으로 불변성을 위반하지만 완료하기 전에 다시 설정해야 합니다. 예를 들어, 연결 리스트의 경우 불변성은 리스트가 첫 번째 노드를 가리킨다는 것입니다.
목록과 각 노드의 다음 필드가 다음 노드를 가리킨다는 것입니다. 구현
insert는 이 불변성을 일시적으로 위반합니다. 15번째 줄에서 l은 다음 목록 요소를 가리킵니다.
그러나 목록은 아직 l을 가리키지 않습니다(16번째 줄에서 다시 설정됨). 위에서 살펴본 경쟁 조건은 두 번째 CPU가 l에 의존하는 코드를 실행했기 때문에 발생했습니다.
(일시적으로) 위반된 동안 불변식을 나열합니다. 잠금을 적절히 사용하면
한 번에 하나의 CPU만이 중요 섹션의 데이터 구조에서 작동할 수 있으므로
데이터 구조의 불변성이 유지되지 않으면 어떤 CPU도 데이터 구조 연산을 실행하지 않습니다.

잠금은 동시 중요 섹션을 직렬화 하여 실행되도록 생각할 수 있습니다 .
한 번에 하나씩, 따라서 불변량이 보존됩니다(단독적으로 옳다고 가정할 경우).
또한 중요 섹션은 서로에 대해 원자적이라고 생각할 수도 있습니다.
나중에 잠금을 획득하는 중요 섹션은 변경 사항의 전체 세트만 볼 수 있습니다.
이전의 중요 섹션에서 업데이트가 이루어지지 않았고, 부분적으로 완료된 업데이트는 전혀 표시되지 않았습니다.

insert에서 acquire를 더 일찍 이동하는 것도 옳을 것입니다.
예를 들어, 호출을 12번째 줄 이전까지 이동해도 됩니다. 이렇게 하면
병렬 처리가 필요한 이유는 malloc에 대한 호출도 직렬화되기 때문입니다. "사용하기" 섹션
아래의 "잠금"에서는 획득 및 해제 호출을 삽입할 위치에 대한 몇 가지 지침을 제공합니다.

코드: 잠금

Xv6에는 두 가지 유형의 잠금이 있습니다. 스핀 잠금과 슬립 잠금입니다. 스핀 잠금부터 시작하겠습니다.
Xv6는 스핀 잠금을 구조체 스핀 잠금 (1501)으로 나타냅니다. 중요한 필드는 다음과 같습니다.
구조가 잠겨 있습니다. 잠금이 사용 가능한 경우 0이고 잠금이 사용 가능한 경우 0이 아닌 단어입니다.
유지됩니다. 논리적으로 xv6는 다음과 같은 코드를 실행하여 잠금을 획득해야 합니다.

```
21      무효의
22      획득(구조 스핀락 *lk)
23      {
24          을 위한(;;) {
25              if(!lk->잠김) {
26                  잠금->잠금 = 1;
27                  부서지다;
28              }
29          }
30      }
```

불행히도, 이 구현은 멀티 프로세서에서 상호 배제를 보장하지 않습니다. 두 CPU가 동시에 25번째 줄에 도달할 수 있습니다. lk-
>locked는 0이고, 그런 다음 26번째 줄을 실행하여 둘 다 잠금을 잡습니다. 이 시점에서 두
서로 다른 CPU가 잠금을 유지하는데, 이는 상호 배제 속성을 위반합니다. 오히려
경쟁 조건을 피하는 데 도움이 되는 것보다 이 획득 구현에는 고유한 것이 있습니다.
경쟁 조건. 여기서 문제는 25행과 26행이 별도의 작업으로 실행된다는 것입니다.
위의 루틴이 정확하려면 25행과 26행이 한 번에 실행되어야 합니다.
원자적 (즉, 나눌 수 없는) 단계.

이 두 줄을 원자적으로 실행하기 위해 xv6는 특수한 x86 명령어인 xchg를 사용합니다.
(0569). 하나의 원자 연산에서 xchg는 메모리의 단어를 내용과 교환합니다.
등록. acquire (1574) 함수는 이 xchg 명령어를 루프에서 반복합니다. 각 반복은 lk->locked를 원자적으로 읽고 1
로 설정합니다 (1581). 잠금이 이미 유지된 경우,
lk->locked는 이미 1이므로 xchg는 1을 반환하고 루프가 계속됩니다.
xchg는 0을 반환하지만 acquire는 잠금을 성공적으로 획득했습니다. locked는 0이었습니다.
그리고 이제 1이 되어 루프가 멈출 수 있습니다. 잠금이 획득되면 레코드를 획득합니다.
디버깅, 잠금을 획득한 CPU 및 스택 추적. 프로세스가 잠금을 다시 해제하는 것을 잊은 경우 이 정보는 범인을 식별하
는 데 도움이 될 수 있습니다. 이러한 디버깅 필드
는 잠금 장치로 보호되므로 잠금 장치를 해제한 상태에서만 편집할 수 있습니다.

기능 해제 (1602) 는 획득의 반대입니다. 디버깅을 지웁니다.
필드를 닫은 다음 잠금을 해제합니다. 이 함수는 어셈블리 명령어를 사용하여 잠금을 해제합니다.
잠겨 있습니다. 이 필드를 지우는 것은 원자적이어야 하므로 xchg 명령어가 실행되지 않습니다.
잠긴 업데이트를 유지하는 4바이트의 하위 집합을 참조하세요. x86은 32비트를 보장합니다.
movl은 4바이트를 모두 원자적으로 업데이트합니다. Xv6은 일반 C 할당을 사용할 수 없습니다.
C 언어 사양에서는 단일 할당이 원자적이라고 지정하지 않습니다.

Xv6의 스핀 잠금 구현은 x86 전용이므로 xv6는 직접적으로 사용할 수 없습니다.
다른 프로세서로 이식 가능합니다. 스핀 잠금의 이식 가능한 구현을 허용하려면
C 언어는 원자 명령어 라이브러리를 지원합니다. 이식 가능한 운영 체제입니다.
해당 지침을 사용하면 됩니다.

코드: 잠금 사용

iderw+코드
이데락+코드

Xv6는 경쟁 조건을 피하기 위해 많은 곳에서 잠금을 사용합니다. 간단한 예는 다음과 같습니다.
IDE 드라이버 (4200). 이 장의 시작 부분에서 언급했듯이 iderw (4354) 에는
디스크 요청 및 프로세서 대기열은 동시에 목록에 새 요청을 추가할 수 있습니다.
(4369). 이 목록과 드라이버의 다른 불변식을 보호하기 위해 iderw는 ide-lock (4365) 을 획득 하고 함수가 끝나면 해제합니다.

연습 1에서는 우리가 본 IDE 드라이버 경쟁 조건을 트리거하는 방법을 살펴봅니다.
대기열 조작 뒤로 획득을 이동하여 장의 시작 부분을 변경합니다.
이 운동을 시도해 보는 것은 가치가 있습니다. 그렇게 쉬운 일이 아니라는 것을 분명히 알게 될 것이기 때문입니다.
레이스를 트리거하여 레이스 조건 버그를 찾는 것이 어렵다는 것을 암시합니다. xv6에 레이스가 있을
가능성은 적지 않습니다.

잠금 장치 사용에 있어서 어려운 부분은 사용할 잠금 장치 수와 사용할 데이터를 결정하는 것입니다.
그리고 각 잠금 장치가 보호하는 불변성. 몇 가지 기본 원칙이 있습니다. 첫째, 언제든지
변수는 한 CPU가 동시에 쓸 수 있고 다른 CPU가 읽을 수도 있습니다.
이렇게 쓰려면, 두 작업이 겹치지 않도록 잠금을 도입해야 합니다.
둘째, 잠금은 불변식을 보호한다는 점을 기억하세요. 불변식에 여러 개의 항목이 포함되는 경우
일반적으로 모든 메모리 위치는 단일 잠금 장치로 보호되어야 합니다.
불변성이 유지됩니다.

위의 규칙은 잠금이 필요한 시점을 말하지만 잠금이 필요한 시점에 대해서는 아무 말도 하지 않습니다.
불필요하며 효율성을 위해 잠금을 너무 많이 하지 않는 것이 중요합니다.
병렬성을 줄이십시오. 병렬성이 중요하지 않은 경우 다음만 준비할 수 있습니다.
단일 스레드로 잠금에 대해 걱정할 필요가 없습니다. 간단한 커널은 커널에 들어갈 때 획득해야 하고 커
널을 종료할 때 다시 해제해야 하는 단일 잠금을 갖는 멀티프로세서에서 이를 수행할 수 있습니다(파이
프 읽기 또는 대기와 같은 시스템 호출은
문제가 발생합니다.) 많은 단일 프로세서 운영 체제가 실행되도록 변환되었습니다.
이 접근 방식을 사용하는 멀티프로세서는 때때로 ''거대한 커널 잠금''이라고도 하지만 이 접근 방식은
병렬성을 희생합니다. 한 번에 하나의 CPU만 커널에서 실행할 수 있습니다.
커널이 무거운 계산을 수행하는 경우 더 큰 집합을 사용하는 것이 더 효율적입니다.
더욱 세분화된 잠금 덕분에 커널이 여러 CPU에서 동시에 실행될 수 있습니다.

궁극적으로, 잠금 세분성의 선택은 병렬 프로그래밍의 연습입니다.
Xv6는 몇 가지 거친 데이터 구조 특정 잠금을 사용합니다(그림 4-2 참조). 예를 들어, xv6
전체 프로세스 테이블과 그 불변식을 보호하는 잠금 장치가 있습니다.
5장에서. 더 세분화된 접근 방식은 항목당 잠금을 갖는 것입니다.
프로세스 테이블에서 다른 항목에서 작업하는 스레드가 병렬로 진행할 수 있도록 프로세스 테이블을
만듭니다. 그러나 이는 불변성이 있는 작업을 복잡하게 만듭니다.
여러 개의 잠금을 획득해야 할 수도 있으므로 전체 프로세스 테이블입니다. 이후의 챕터에서는 xv6의
각 부분이 동시성을 처리하는 방법을 설명하며, 이를 사용하는 방법을 보여줍니다.
잠금장치.

교착 상태 및 잠금 순서

커널을 통한 코드 경로가 동시에 여러 잠금을 유지해야 하는 경우 모든 코드 경로가 동일한 순서로 잠
금을 획득하는 것이 중요합니다. 그렇지 않은 경우
교착 상태의 위험. xv6의 두 코드 경로에 잠금 A와 B가 필요하지만 코드 경로 1이 필요하다고 가정해 보겠습니다.
A, B 순서로 잠금을 획득하고 다른 경로는 B 순서로 잠금을 획득합니다.

Lock	설명
bcache.lock	블록 버퍼 캐시 항목의 할당을 보호합니다.
cons.lock	콘솔 하드웨어에 대한 액세스를 직렬화하고 혼합된 출력을 방지합니다.
ftable.lock	파일 테이블에서 구조체 파일의 할당을 직렬화합니다.
icache.lock	inode 캐시 항목 할당을 보호합니다.
idelock	디스크 하드웨어 및 디스크 대기열에 대한 액세스를 직렬화합니다.
kmem.lock	메모리 할당을 직렬화합니다
log.lock 파	트랜잭션 로그에 대한 작업을 직렬화합니다.
이프의 p->lock	각 파이프에서 작업을 직렬화합니다.
ptable.lock	컨텍스트 전환과 proc->state 및 proctable에 대한 작업을 직렬화합니다.
tickslock	tick 카운터에 대한 작업을 직렬화합니다.
inode의 ip->lock 각 inode와 그 내용에 대한 작업을 직렬화합니다.	
buf의 b->lock은 각 블록 버퍼에 대한 작업을 직렬화합니다.	

그림 4-2. xv6의 잠금 장치

그런 다음 A. 이 상황은 두 스레드가 코드 경로를 실행하는 경우 교착 상태로 이어질 수 있습니다.
동시에. 스레드 T1이 코드 경로 1을 실행하고 잠금 A를 획득하고 스레드가
T2는 코드 경로 2를 실행하고 잠금 B를 획득합니다. 그 다음 T1은 잠금 B를 획득하려고 시도합니다.
T2는 잠금 A를 획득하려고 시도합니다. 두 획득 모두 무기한 차단됩니다.
다른 스레드가 필요한 잠금을 보유하고 있고, 획득이 반환될 때까지 잠금을 해제하지 않는 경우. 이러한 교착 상태를 피
하려면 모든 코드 경로가 동일한 순서로 잠금을 획득해야 합니다.
전역 잠금 획득 순서가 필요하다는 것은 잠금이 효과적으로 다음의 일부임을 의미합니다.
각 함수의 사양: 호출자는 잠금을 발생시키는 방식으로 함수를 호출해야 합니다.
합의된 순서대로 인수됩니다.

아이디어인터+코드
웨이크업+코드
ptable+코드
틱+코드
sys_sleep+코드
tickslock+코드
iderw+코드
이데락+코드
아이디어인터+코드

Xv6에는 ptable.lock과 관련된 길이가 2인 많은 잠금 순서 체인이 있습니다.
5장에서 논의된 것처럼 수면이 작동하는 방식입니다. 예를 들어, ideintr는 다음을 보유합니다.
웨이크업을 호출하는 동안 ide 잠금을 해제하여 ptable 잠금을 획득합니다. 파일 시스템 코드
xv6의 가장 긴 잠금 체인을 포함합니다. 예를 들어, 파일을 만들려면 동시에
디렉토리에 대한 잠금, 새 파일의 inode에 대한 잠금, 디스크 블록에 대한 잠금을 유지합니다.
버퍼, idelock 및 ptable.lock. 교착 상태를 피하기 위해 파일 시스템 코드는 항상 다음을 획득합니다.
이전 문장에서 언급된 순서대로 잠금합니다.

인터럽트 핸들러

Xv6는 인터럽트와 관련된 데이터를 보호하기 위해 여러 상황에서 스핀 잠금을 사용합니다.
핸들러 및 스레드. 예를 들어, 타이머 인터럽트는 (3414) 틱을 증가시킬 수 있습니다.
커널 스레드가 sys_sleep(3823)에서 tick을 읽는 것과 거의 같은 시간입니다 . 잠금
tickslock은 두 가지 액세스를 직렬화합니다.

인터럽트는 단일 프로세서에서도 동시성을 유발할 수 있습니다. 인터럽트가 활성화되면 커널 코드는 언제든지 중
지되어 대신 인터럽트 핸들러를 실행할 수 있습니다.
iderw가 idelock을 유지한 다음 ideintr를 실행하기 위해 중단되었다고 가정합니다. Ideintr
idelock을 잠그려고 시도하고, 그것이 유지되고 있는지 확인하고, 그것이 풀릴 때까지 기다립니다. 이 상황에서
idelock은 결코 풀리지 않을 것입니다. 오직 iderw만이 그것을 풀 수 있고, iderw는 그것을 풀지 않을 것입니다.
ideintr가 반환될 때까지 계속 실행되므로 프로세서와 결국 전체가
시스템은 교착 상태에 빠지게 됩니다.

pushcli+코드
팝클라+코드
획득+코드
xchg+코드
릴리스+코드
팝클라+코드
xchg+코드

이 상황을 피하기 위해 인터럽트 핸들러에서 스핀 잠금을 사용하는 경우 프로세서
인터럽트가 활성화된 상태에서는 해당 잠금을 유지해서는 안 됩니다. Xv6은 더 보수적입니다.
프로세서가 스핀 잠금 중요 섹션에 진입하면 xv6은 항상 인터럽트가 비활성화되도록 보장합니다.
해당 프로세서에서. 인터럽트는 다른 프로세서에서 여전히 발생할 수 있으므로 인터럽트의 ac-quire는 스레드가
스핀 잠금을 해제할 때까지 기다릴 수 있습니다. 단, 동일한 프로세서에서는 기다릴 수 없습니다.

프로세서가 스핀 잠금을 유지하지 않을 때 xv6는 인터럽트를 다시 활성화합니다.
중첩된 중요 섹션을 처리하기 위한 회계. 호출 pushcli (1667) 를 획득 하고
릴리스는 현재 프로세서의 잠금 중첩 수준을 추적하기 위해 popcli (1679) 를 호출합니다.
해당 카운트가 0에 도달하면 popcli는 존재했던 인터럽트 활성화 상태를 복원합니다.
가장 바깥쪽 중요 섹션의 시작. cli 및 sti 함수는 x86을 실행합니다.
각각 인터럽트 비활성화 및 활성화 명령어입니다.

xchg가 획득할 수 있는 호출 pushcli를 획득하기 전에 호출 pushcli를 획득하는 것이 중요합니다.
잠금 (1581). 두 가지가 반대로 되면 몇 가지 명령 주기가 발생합니다.
인터럽트가 활성화된 상태에서 잠금이 유지되었고 불행히도 타이밍이 지정된 인터럽트가 발생했습니다.
시스템을 교착 상태로 만듭니다. 마찬가지로, 릴리스 호출 popcli는 다음 경우에만 중요합니다.
잠금을 해제하는 xchg (1581).

지시 및 메모리 순서

이 장에서는 코드가 프로그램에 나타나는 순서대로 실행된다고 가정했습니다. 그러나 많은 컴파일러와 프로세서
는 코드를 다음과 같이 실행합니다.
더 높은 성능을 달성하기 위해. 명령을 완료하는 데 많은 사이클이 걸리는 경우,
프로세서는 다른 명령과 겹칠 수 있도록 명령을 일찍 내리고 싶어할 수 있습니다.
지침을 따르고 프로세서 정지를 방지합니다. 예를 들어, 프로세서는 다음에서 알아차릴 수 있습니다.
명령어 A와 B의 직렬 시퀀스는 서로 종속되지 않으며 프로세서가 A를 완료할 때 완료되도록 명령어 B에서 A보다 먼저
시작합니다.
컴파일러는 실행 파일에서 명령어 A보다 명령어 B를 먼저 내보내어 유사한 재정렬을 수행할 수 있습니다. 그러나 동시성
은 이 재정렬을 노출시킬 수 있습니다.
소프트웨어는 잘못된 동작으로 이어질 수 있습니다.

예를 들어, 삽입을 위한 이 코드에서 컴파일러나
프로세서는 라인 4(또는 2 또는 5)의 효과가 다른 코어에 표시되도록 했습니다.
6번 라인의 효과:

```
1    l = malloc(크기 *l);
2    l->데이터 = 데이터;
3    획득(&listlock);
4    l->다음 = 목록;
5    리스트 = l;
6    해제(&listlock);
```

예를 들어 하드웨어나 컴파일러가 라인 4의 효과를 라인 6 이후에 보이도록 재정렬하면 다른 프로세서가 리스트 잠금을
획득하고 해당 리스트를 관찰할 수 있습니다.
l을 가리키지만 l->next가 목록의 나머지 부분으로 설정되어 있다는 것을 관찰하지 못합니다.
나머지 목록은 읽을 수 없습니다.

하드웨어와 컴파일러에 이러한 재정렬을 수행하지 말라고 알리기 위해 xv6는 다음을 사용합니다.
__sync_synchronize()는 획득 및 해제 모두에서 사용됩니다. _sync_synchronize()는
메모리 장벽: 컴파일러와 CPU에 로드나 스토어를 재정렬하지 말라고 알려줍니다.

장벽. Xv6는 획득과 해제에서만 주문에 대해 걱정합니다. 잠금 구조가 아닌 다른 데이터 구조에 대한 동시 액세스는 획득과 해제 사이에 수행되기 때문입니다.

sleep-locks 재귀적 잠금

수면 잠금

때때로 xv6 코드는 장시간 잠금을 유지해야 합니다. 예를 들어, 파일 시스템(6장)은 디스크에서 내용을 읽고 쓰는 동안 파일을 잠근 상태로 유지하며, 이러한 디스크 작업은 수십 밀리초가 걸릴 수 있습니다. 효율성을 위해 프로세서는 대기하는 동안 양보되어야 다른 스레드가 진행할 수 있으며, 이는 xv6가 컨텍스트를 넘어 유지될 때 잘 작동하는 잠금이 필요하다는 것을 의미합니다.

스위치. Xv6는 sleep-lock 형태로 이러한 잠금을 제공합니다 .

Xv6 sleep-locks는 중요 섹션 동안 프로세서를 양보하는 것을 지원합니다. 이 속성은 설계 과제를 제기합니다. 스레드 T1이 잠금 L1을 보유하고 프로세서를 양보했으며 스레드 T2가 L1을 획득하고자 하는 경우, T1이 대기하는 동안 T1이 실행될 수 있도록 해야 T1이 L1을 해제할 수 있습니다. T2는 여기서 spin-lock 획득 함수를 사용할 수 없습니다. 인터럽트가 꺼진 상태에서 회전하고, 그러면 T1이 실행되지 않습니다.

이러한 교착 상태를 피하기 위해, sleep-lock 획득 루틴(acquiresleep이라고 함)은 대기하는 동안 프로세서의 권한을 양보하고, 인터럽트를 비활성화하지 않습니다.

acquiresleep (4622) 는 5장에서 설명할 기술을 사용합니다. 높은 수준에서 sleep-lock은 spinlock으로 보호되는 잠긴 필드를 가지고 있으며 ac-quiresleep의 sleep 호출은 CPU를 원자적으로 생성하고 spin-lock을 해제합니다. 결과적으로 acquiresleep이 대기하는 동안 다른 스레드가 실행될 수 있습니다.

sleep-locks는 인터럽트를 활성화된 상태로 두기 때문에 인터럽트 핸들러에서 사용할 수 없습니다. acquiresleep이 프로세서를 양보할 수 있기 때문에 sleep-locks는 spin-lock 중요 섹션 내부에서 사용할 수 없습니다(spin-locks는 sleep-lock 중요 섹션 내부에서 사용할 수 있음).

Xv6는 오버헤드가 낮기 때문에 대부분 상황에서 스핀 잠금을 사용합니다. 파일 시스템에서만 슬립 잠금을 사용하는데, 긴 디스크 작업에서 잠금을 유지하는 것이 편리합니다.

잠금의 한계

잠금은 종종 동시성 문제를 깔끔하게 해결하지만, 어색한 경우도 있습니다. 이후의 챕터에서는 xv6에서 그러한 상황을 지적할 것입니다. 이 섹션에서는 발생하는 몇 가지 문제를 간략하게 설명합니다.

때때로 함수는 잠금으로 보호해야 하는 데이터를 사용하지만, 함수는 이미 잠금을 보유한 코드와 그렇지 않으면 잠금이 필요하지 않은 코드에서 모두 호출됩니다. 이를 처리하는 한 가지 방법은 잠금을 획득하는 함수와 호출자가 이미 잠금을 보유하고 있을 것으로 예상하는 함수의 두 가지 변형을 갖는 것입니다. 예를 들어 wakeup1을 참조하세요 (2953). 또 다른 접근 방식은 sched(2758)와 같이 호출자에게 필요 여부와 관계없이 호출자에게 잠금을 보유하도록 요구하는 것입니다 . 커널 개발자는 이러한 요구 사항을 알고 있어야 합니다.

호출자와 호출자 모두가 잠금을 필요로 하는 상황을 재귀적 잠금을 허용함으로써 단순화할 수 있을 것 같습니다 . 즉, 함수가 잠금을 보유하고 있으면 해당 함수가 잠금을 보유하고 있는 모든 함수가 잠금을 보유할 수 있습니다.

호출은 잠금을 다시 획득할 수 있습니다. 그러나 프로그래머는 그런 다음 다음을 수행해야 합니다.
호출자와 수신자의 모든 조합에 대한 이유는 더 이상 그렇지 않을 것이기 때문입니다.
데이터 구조의 불변식은 획득 후에도 항상 유지됩니다. 재귀적이든
잠금은 잠금이 필요한 함수에 대한 규칙을 사용하는 xv6보다 더 좋습니다.
held는 명확하지 않습니다. 더 큰 교훈은 (교착 상태를 피하기 위한 글로벌 잠금 순서와 마찬가지로) 잠금 요구 사항이 때때로 비공개
일 수 없고 함수와 모듈의 인터페이스에 침입한다는 것입니다.

잠금이 부족한 상황은 한 스레드가 기다려야 하는 경우입니다.
예를 들어 파이프의 리더가 대기하는 경우와 같이 다른 스레드가 데이터 구조를 업데이트하는 경우
파이프를 쓸 다른 스레드가 있습니다. 대기 스레드는 잠금을 유지할 수 없습니다.
데이터, 그렇게 하면 기다리고 있는 업데이트가 방해를 받기 때문입니다. 대신 xv6는 잠금 및 이벤트 대기
를 공동으로 관리하는 별도의 메커니즘을 제공합니다. 설명을 참조하세요.
5장에서 잠과 깨어남에 대해 설명합니다.

현실 세계

동시성 기본 요소와 병렬 프로그래밍은 활발한 연구 분야입니다.
잠금을 사용한 프로그래밍은 여전히 어렵습니다. 잠금을 기반으로 사용하는 것이 가장 좋습니다.
xv6에서는 이를 수행하지 않지만 동기화된 대기열과 같은 상위 수준 구성 요소.
잠금 장치가 있는 프로그램의 경우 경쟁 조건을 식별하려는 도구를 사용하는 것이 현명합니다.
잠금이 필요한 불변식을 놓치기 쉽기 때문입니다.

대부분의 운영 체제는 사용자가 POSIX 스레드(Pthreads)를 사용할 수 있도록 지원합니다.
여러 프로세서에서 동시에 실행되는 여러 스레드를 갖는 프로세스입니다. Pthreads
사용자 수준 잠금, 장벽 등을 지원합니다. Pthread를 지원하려면 지원이 필요합니다.
운영 체제에서. 예를 들어, 하나의 pthread가 있는 경우 다음과 같아야 합니다.
시스템 호출의 블록에서 동일한 프로세스의 다른 pthread가 실행될 수 있어야 합니다.
그 프로세서. 또 다른 예로, pthread가 프로세스의 주소 공간을 변경하는 경우
(예: 확장 또는 축소) 커널은 스레드를 실행하는 다른 프로세서를 정렬해야 합니다.
동일한 프로세스의 하드웨어 페이지 테이블을 업데이트하여 주소 공간의 변경 사항을 반영합니다. x86에
서 이는 Translation Look-aside Buffer를 격추하는 것을 포함합니다.
(TLB) 프로세서 간 인터럽트(IPI)를 사용하는 다른 프로세서.

원자적 명령어 없이 잠금을 구현하는 것은 가능하지만 비용이 많이 듭니다.
대부분의 운영체제는 원자적 명령어를 사용합니다.

많은 프로세서가 동일한 잠금을 획득하려고 시도하는 경우 잠금이 비쌀 수 있습니다.
동시에. 한 프로세서가 로컬 캐시에 잠금을 캐시하고 다른 프로세서가 잠금을 획득해야 하는 경우 캐시
라인을 업데이트하는 원자적 명령어는
잠금을 유지하려면 한 프로세서의 캐시에서 다른 프로세서의 캐시로 줄을 이동해야 하며, 아마도 캐시 줄
의 다른 사본을 무효화해야 합니다. 캐시 가져오기
다른 프로세서의 캐시에서 라인은 훨씬 더 비쌀 수 있습니다.
로컬 캐시에서 줄을 가져옵니다.

잠금과 관련된 비용을 피하기 위해 많은 운영 체제는 잠금 없는 방식을 사용합니다.
데이터 구조 및 알고리즘. 예를 들어, 연결 목록을 구현하는 것이 가능합니다.
목록 검색 중에 잠금이 필요하지 않은 장의 시작 부분과 같은 것,
그리고 목록에 항목을 삽입하기 위한 하나의 원자적 명령어. 잠금 없는 프로그래밍은 더 많습니다.
그러나 프로그래밍 잠금보다 더 복잡합니다. 예를 들어, 다음에 대해 걱정해야 합니다.

구조와 메모리 재정렬. 잠금을 사용한 프로그래밍은 이미 어렵기 때문에 xv6
잠금 없는 프로그래밍의 추가적인 복잡성을 피할 수 있습니다.

수업 과정

1. iderw에서 획득을 sleep 전으로 옮깁니다. 경주가 있나요? 왜 안 하시나요?
xv6 부팅 시 관찰하고 stressfs를 실행? 더미로 중요 섹션을 늘리세요
루프; 지금 무엇이 보이나요? 설명해주세요.

2. acquire에서 xchg를 제거하세요. xv6를 실행하면 무슨 일이 일어나는지 설명하세요?

3. 대부분 운영 체제에서 지원되는 POSIX 스레드를 사용하여 병렬 프로그램을 작성합니다. 예를 들어, 병렬 해시 테이블을 구현하고 puts/gets 수가 코어 수 증가에 따라 확장되는지 측정합니다.

4. xv6에서 Pthread의 하위 집합을 구현합니다. 즉, 사용자 수준 스레드를 구현합니다.
사용자 프로세스가 2개 이상의 스레드를 가질 수 있도록 라이브러리를 구성하고 이를 정렬합니다.
스레드는 다른 프로세서에서 병렬로 실행될 수 있습니다. 차단 시스템 호출을 하고 공유 주소를 변경하는 스레드를 올바르게 처리하는 디자인을 생각해 보세요.
공간.

5장

스케줄링

모든 운영 체제는 컴퓨터가 가지고 있는 프로세스보다 더 많은 프로세스로 실행될 가능성이 있습니다.
프로세서이므로 프로세스 간에 프로세서를 시간 공유할 계획이 필요합니다. 이상적으로는 공유가 사용자 프로세스에 투명해야 합니다. 일반적인 접근 방식은 다음과 같습니다.
각 프로세스에 하드웨어 프로세서에 프로세스를 멀티플렉싱 하여 자체 가상 프로세서가 있다는 환상을 제공합니다 . 이 장에서는 xv6
이러한 다중화를 달성합니다.

멀티플렉싱

Xv6는 각 프로세서를 두 번에 걸쳐 한 프로세스에서 다른 프로세스로 전환하여 멀티플렉스합니다.
상황. 첫째, xv6의 sleep 및 wakeup 메커니즘은 프로세스가 대기할 때 전환됩니다.
장치 또는 파이프 I/O가 완료되거나 자식이 종료될 때까지 기다리거나 sleep 시스템 호출에서 기다립니다. 둘째, xv6는 프로세스가 사용자 명령을 실행할 때 주기적으로 스위치를 강제로 실행합니다. 이 멀티플렉싱은 각 프로세스가 자체 CPU를 가지고 있다는 착각을 만듭니다.
xv6가 메모리 할당자와 하드웨어 페이지 테이블을 사용하여 환상을 만드는 것처럼
각 프로세스는 자체 메모리를 가지고 있습니다.

멀티플렉싱을 구현하는 데는 몇 가지 과제가 있습니다. 첫째, 하나에서 다른 것으로 전환하는 방법
프로세스에서 다른 프로세스로 전환하는 방법? 컨텍스트 전환이라는 아이디어는 간단하지만 구현은 xv6에서 가장 불투명한 코드 중 하나입니다. 둘째, 투명하게 전환하는 방법
사용자 프로세스? Xv6는 컨텍스트 스위치를 구동하는 표준 기술을 사용합니다.
타이머 인터럽트. 셋째, 많은 CPU가 동시에 프로세스 간에 전환할 수 있습니다.
그리고 경쟁을 피하기 위해 잠금 계획이 필요합니다. 네 번째로, 프로세스의 메모리와 다른
프로세스가 종료되면 리소스가 해제되어야 하지만 프로세스가 이 모든 것을 스스로 할 수는 없습니다. (예를 들어) 여전히 사용 중인 자체 커널 스택을 해제할 수 없기 때문입니다. 마지막으로, 각각
멀티코어 머신의 코어는 시스템 호출이 올바른 프로세스의 커널 상태에 영향을 미치도록 실행 중인 프로세스를 기억해야 합니다. Xv6는 이러한 문제를 다음과 같이 해결하려고 합니다.
가능한 한 간단하게 만들려고 했지만, 그럼에도 불구하고 그 결과로 나오는 코드는 까다로워집니다.

xv6은 프로세스가 서로 조정할 수 있는 방법을 제공해야 합니다. 예를 들어, 부모 프로세스는 자식 프로세스 중 하나가 종료될 때까지 기다려야 할 수도 있고, 프로세스
파이프를 읽으려면 다른 프로세스가 파이프를 쓸 때까지 기다려야 할 수도 있습니다.
원하는 이벤트가 있는지 반복적으로 확인하여 대기 프로세스가 CPU를 낭비하게 만듭니다.
발생했으며 xv6에서는 프로세스가 CPU를 포기하고 이벤트를 기다리며 절전 모드로 전환할 수 있습니다.
그리고 다른 프로세스가 첫 번째 프로세스를 깨울 수 있도록 합니다. 경쟁을 피하기 위해 주의가 필요합니다.
이벤트 알림이 손실되는 결과를 초래합니다. 이러한 문제의 예로는
이 장에서는 해당 솔루션을 위한 파이프의 구현을 살펴본다.

코드: 컨텍스트 전환

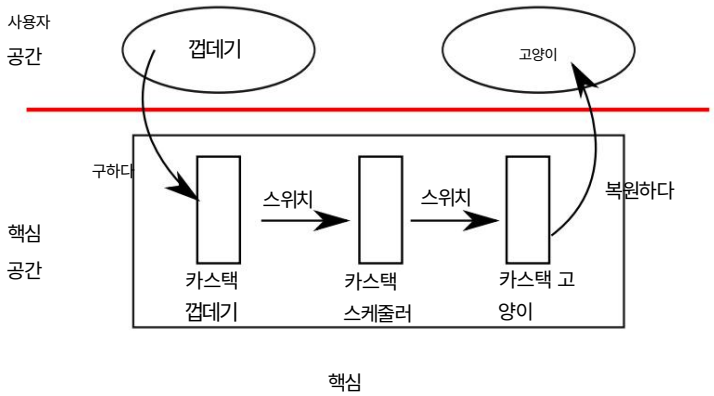


그림 5-1. 한 사용자 프로세스에서 다른 사용자 프로세스로 전환. 이 예에서 xv6는 하나의 CPU(및 따라서 하나의 스케줄러 스레드).

스위치+코드
맥락
구조 컨텍스트+코드
트랩+코드
수익률+코드
스케줄+코드
스위치+코드
씨퓨->스케줄러+코드
스위치+코드

그림 5-1은 한 사용자 프로세스에서 다른 사용자 프로세스로 전환하는 데 필요한 단계를 간략하게 설명합니다. 사용자-커널 전환(시스템 호출 또는 인터럽트)에서 이전 프로세스의 커널로 전환 스레드, 현재 CPU 스케줄러 스레드에 대한 컨텍스트 전환, 스레드에 대한 컨텍스트 전환 새로운 프로세스의 커널 스레드와 사용자 수준 프로세스로의 트랩 리턴. xv6 스케줄러는 때때로 그렇지 않기 때문에 자체 스레드(저장된 레지스터 및 스택)를 갖습니다. 모든 프로세스의 커널 스택에서 실행하기에 안전합니다. 우리는 exit에서 예를 볼 것입니다. 이 이 섹션에서는 커널 스레드와 스케줄러 스레드 간을 전환하는 메커니즘을 살펴보겠습니다.

한 스레드에서 다른 스레드로 전환하려면 이전 스레드의 CPU 레지스터를 저장하고 새 스레드의 이전에 저장된 레지스터를 복원해야 합니다. %esp 그리고 %eip가 저장되고 복원된다는 것은 CPU가 스택을 전환하고 전환한다는 것을 의미합니다. 어떤 코드를 실행하고 있는지.

swtch 함수는 스레드 스위치에 대한 저장 및 복원을 수행합니다. swtch 스레드에 대해 직접 알지 못합니다. 컨텍스트라고 하는 레지스터 세트를 저장하고 복원할 뿐입니다. 프로세스가 CPU 를 포기할 때가 되면 프로세스의 커널 스레드 swtch를 호출하여 자체 컨텍스트를 저장하고 스케줄러 컨텍스트로 돌아갑니다. 각 컨텍스트 커널에 저장된 구조체에 대한 포인터인 struct context*로 표현됩니다. 스택이 관련됨. Swtch는 struct context **old 및 struct라는 두 개의 인수를 사용합니다. 문맥 *new. 현재 레지스터를 스택에 푸시하고 스택 포인터를 *old에 저장합니다. 그런 다음 swtch는 new를 %esp 에 복사하고 이전에 저장된 레지스터를 팝하고 다시 회전.

스케줄러로의 swtch를 통한 사용자 프로세스를 따라가 봅시다. 우리는 챕터에서 보았습니다. 3 각 인터럽트의 끝에서 하나의 가능성이 트랩 호출이 양보된다는 것입니다. 양보 turn은 sched를 호출하고, sched는 swtch를 호출하여 proc->context에 현재 컨텍스트를 저장합니다.

이전에 cpu->scheduler(2822)에 저장된 스케줄러 컨텍스트로 전환합니다 .

Swtch (3052) 는 스택에서 호출자가 저장한 레지스터 %eax 및 %edx (3060-3061)로 인수를 복사하는 것으로 시작합니다. swtch는 스택 포인터를 변경하기 전에 이 작업을 수행해야 합니다 . 그리고 더 이상 %esp를 통해 인수에 액세스할 수 없습니다. 그런 다음 swtch가 레지스터를 푸시합니다. 상태, 현재 스택에 컨텍스트 구조를 생성합니다. 호출자가 저장한 레지스터만 저장해야 합니다. x86의 규칙은 %ebp, %ebx, %esi입니다.

%edi, %esp. Swtch는 처음 네 개를 명시적으로 푸시합니다 (3064-3067). 마지막을 *old에 쓰여진 struct context*로 암묵적으로 저장합니다 (3070). 중요한 레지스터가 하나 더 있습니다. 프로그램 카운터 %eip입니다. swtch를 호출한 호출 명령어에 의해 이미 스택에 저장되었습니다. 이전 컨텍스트를 저장한 swtch는 새 컨텍스트를 복원할 준비가 되었습니다. 새 컨텍스트에 대한 포인터를 스택 포인터로 옮깁니다 (3071). 새 스택은 swtch가 방금 남긴 이전 스택과 동일한 형태를 갖습니다. 새 스택은 이전에 swtch를 호출할 때의 이전 스택 이었습니다 . 따라서 swtch는 시퀀스를 반전하여 새 컨텍스트를 복원할 수 있습니다. %edi, %esi, %ebx, %ebp에 대한 값을 팝한 다음 반환합니다 (3074-3078). swtch가 스택 포인터를 변경했기 때문에 다시 저장된 값과 반환된 명령어 주소는 새 컨텍스트의 값입니다.

스위차+코드
sched+code
스위차+코드
cpu-
> 스케줄러+코드 스위차+코드
스케줄러+코드 스위치
+코드
ptable.lock+코드
sched+코드
sleep+코드 종
료+코드
sched+코드 스위
차+코드
cpu-
> 스케줄러+코드
스케줄러+코드
ptable.lock+코드 스위
차+코드
ptable.lock+코드 스위
차+코드
수익률+코드

우리의 예에서 sched는 swtch를 호출하여 CPU당 스케줄러 컨텍스트인 cpu->scheduler로 전환했습니다. 해당 컨텍스트는 스케줄러가 swtch (2781) 를 호출하여 저장되었습니다 .
추적하던 스위치가 반환되면 sched가 아니라 scheduler로 반환되고, 스택 포인터는 현재 CPU의 스케줄러 스택을 가리킵니다.

코드: 스케줄링

이전 섹션에서는 swtch의 저수준 세부 사항을 살펴보았습니다. 이제 swtch를 주어진 것으로 간주하고 스케줄러를 통해 한 프로세스에서 다른 프로세스로 전환하는 것을 살펴보겠습니다.
CPU를 포기하려는 프로세스는 프로세스 테이블 잠금 pt-able.lock을 획득하고, 보유하고 있는 다른 잠금을 해제하고, 자체 상태(proc->state)를 업데이트한 다음 sched를 호출해야 합니다. Yield (2828)는 이 규칙을 따르며, 나중에 살펴볼 sleep 및 exit도 마찬가지입니다. Sched는 이러한 조건 (2813-2818) 을 다시 확인한 다음 이러한 조건의 의미를 나타냅니다. 잠금이 유지되었으므로 CPU는 인터럽트가 비활성화된 상태에서 실행되어야 합니다. 마지막으로 sched는 swtch를 호출하여 proc->context의 현재 컨텍스트를 저장하고 cpu->scheduler의 스케줄러 컨텍스트로 전환합니다. Swtch는 스케줄러의 swtch가 반환된 것처럼 스케줄러의 스택을 다시 켭니다 (2781). 스케줄러는 for 루프를 계속 진행하고 실행할 프로세스를 찾아 전환하고 사이클이 반복됩니다.

방금 xv6가 swtch에 대한 호출에서 ptable.lock을 유지하는 것을 보았습니다.swtch의 호출자는 이미 잠금을 유지해야 하며 잠금 제어는 switched-to 코드로 전달됩니다.이 규칙은 잠금에서는 특이합니다.일반적으로 잠금을 획득한 스레드는 잠금을 해제해야 하므로 정확성에 대해 추론하기가 더 쉽습니다.컨텍스트 전환의 경우 pt-able.lock이 swtch에서 실행하는 동안 참이 아닌 프로세스의 상태 및 컨텍스트 필드에 대한 불변성을 보호하기 때문에 이 규칙을 어길 필요가 있습니다.swtch 동안 pt-able.lock을 유지하지 않으면 발생할 수 있는 문제의 한 예는 다른 CPU가 yield가 상태를 RUNNABLE로 설정한 후 swtch가 자체 커널 스택 사용을 중지시키기 전에 프로세스를 실행하기로 결정할 수 있다는 것입니다.결과적으로 두 개의 CPU가 동일한 스택에서 실행되므로 옳을 수 없습니다.

커널 스레드는 항상 sched에서 프로세서를 포기하고 스케줄러에서 항상 동일한 위치로 전환합니다. 스케줄러는 (거의) 항상 이전에 sched를 호출한 커널 스레드로 전환합니다. 따라서 xv6가 스레드를 전환하는 줄 번호를 출력하면 다음과 같은 간단한 패턴을 관찰할 수 있습니다. (2781), (2822), (2781), (2822) 등. 이 양식화된 두 가지 전환 절차

스레드가 발생하는 것을 코루틴이라고도 합니다 . 이 예에서 sched와 scheduler는 서로의 코루틴입니다.

코루틴
sched+code 스
케줄러+코드 스위치+코
드 sched+code
포크렛+코드

스케줄러의 swtch 호출이 sched로 끝나지 않는 경우가 하나 있습니다.
우리는 2장에서 이 사례를 보았습니다. 새로운 프로세스가 처음 예약되면 forkret (2853)에서 시작합니다. Forkret은 ptable.lock을 해제하기 위해 존재합니다. 그렇지 않으면 새로운 프로세스가 trapret에서 시작될 수 있습니다.

ptable.lock+code 스케
줄러+코드

스케줄러 (2758) 는 간단한 루프를 실행합니다. 실행할 프로세스를 찾아서 양보할 때까지 실행하고 반복합니다. 스케줄러는 대부분의 작업에 대해 ptable.lock을 유지하지만 외부 루프의 각 반복에서 한 번씩 잠금을 해제하고(그리고 명시적으로 인터럽트를 활성화합니다). 이는 이 CPU가 유휴 상태(RUNNABLE 프로세스를 찾을 수 없음)인 특수한 경우에 중요합니다.

ptable.lock+code
RUNNABLE+코드
switchuvm+code
스위치+코드

유휴 스케줄러가 잠금을 계속 유지한 채 루프를 돌면 프로세스를 실행 중인 다른 CPU는 컨텍스트 전환이나 프로세스 관련 시스템 호출을 수행할 수 없으며, 특히 유휴 CPU를 스케줄링 루프에서 벗어나게 하기 위해 프로세스를 RUNNABLE로 표시할 수 없습니다. 유휴 CPU에서 인터럽트를 주기적으로 활성화하는 이유는 프로세스(예: 셸)가 I/O를 기다리고 있기 때문에 RUNNABLE 프로세스가 없을 수 있기 때문입니다. 스케줄러가 항상 인터럽트를 비활성화한 상태로 두면 I/O가 도착하지 않습니다.

ptable.lock+코드 생성
+코드 실행 가능
+코드 스케줄러+코드

스케줄러는 프로세스 테이블을 순환하여 실행 가능한 프로세스, 즉 p->state == RUNNABLE인 프로세스를 찾습니다. 프로세스를 찾으면 CPU당 현재 프로세스 변수 proc를 설정하고 switchuvm으로 프로세스의 페이지 테이블로 전환하고 프로세스를 RUNNING으로 표시한 다음 swtch를 호출하여 실행을 시작합니다 (2774-2781).

p->context+code
ptable.lock+code
ptable.lock+code 종료
+code 대기
+code

스케줄링 코드의 구조에 대해 생각하는 한 가지 방법은 각 프로세스에 대한 일련의 불변식을 적용하고 이러한 불변식이 참이 아닐 때마다 ptable.lock을 유지하도록 하는 것입니다.한 가지 불변식은 프로세스가 실행 중이면 타이머 인터럽트의 yield가 프로세스에서 전환될 수 있어야 한다는 것입니다.즉, CPU 레지스터는 프로세스의 레지스터 값을 유지해야 하고(즉, 실제로 컨텍스트에 없음), %cr3은 프로세스의 페이지 테이블을 참조해야 하고, %esp는 swtch가 레지스터를 올바르게 푸시할 수 있도록 프로세스의 커널 스택을 참조해야 하고, proc는 프로세스의 proc[] 슬롯을 참조해야 합니다.또 다른 불변식은 프로세스가 실행 가능이면 유휴 CPU의 스케줄러가 이를 실행할 수 있어야 한다는 것입니다. 즉, p->context는 프로세스의 커널 스레드 변수를 보유해야 하고, 프로세스의 커널 스택에서 실행 중인 CPU가 없고, CPU의 %cr3가 프로세스의 페이지 테이블을 참조하지 않으며, CPU의 proc가 프로세스를 참조하지 않습니다.

위의 불변식을 유지하는 것이 xv6가 한 스레드(종종 yield에서)에서 ptable.lock을 획득하고 다른 스레드(스케줄러 스레드 또는 다른 다음 커널 스레드)에서 잠금을 해제하는 이유입니다.코드가 실행 중인 프로세스의 상태를 수정하여 RUNNABLE로 만들기 시작하면 불변식을 복원하는 것을 마칠 때까지 잠금을 유지해야 합니다.가장 빠른 올바른 해제 시점은 스케줄러가 프로세스의 페이지 테이블 사용을 중단하고 proc를 지운 후입니다.마찬가지로 스케줄러가 실행 가능한 프로세스를 RUNNING으로 변환하기 시작하면 커널 스레드가 완전히 실행될 때까지(swtch 이후, 예: yield에서) 잠금을 해제할 수 없습니다.ptable.lock은 프로세스 ID와 사용 가능한 프로세스 테이블 슬롯의 할당, 종료와 대기 간의 상호 작용, 웨이크업 손실을 방지하는 메커니즘(다음 섹션 참조) 및 아마도 다른 것들도 보호합니다.ptable.lock의 여러 기능을 분리할 수 있는지 생각해 볼 가치가 있을 수 있습니다.분명성을 위해, 그리고 아마도 성능을 위해.

코드: mycpu 및 myproc

xv6는 각 프로세서에 대해 struct cpu를 유지 관리하며 여기에는 프로세서에서 현재 실행 중인 프로세스(있는 경우)와 프로세서의 고유 하드웨어 식별자가 기록됩니다.
(apicid) 및 기타 정보. 함수 mycpu (2437) 는 현재
프로세서의 구조체 cpu. mycpu는 프로세서 식별자를 읽어서 이를 수행합니다.
로컬 APIC 하드웨어 및 struct cpu 배열을 통해 항목을 찾습니다.
해당 식별자. mycpu의 반환 값은 취약합니다. 타이머가 중단되고
스레드가 다른 프로세서로 이동되도록 하면 반환 값이 없습니다.
더 이상 정확하지 않습니다. 이 문제를 피하기 위해 xv6에서는 mycpu 호출자가 인터럽트를 비활성화하고 반환된 struct cpu를 사용한 후에만 활성화해야 합니다.

함수 myproc (2457) 는 프로세스에 대한 struct proc 포인터를 반환합니다.
현재 프로세서에서 실행 중입니다. myproc는 인터럽트를 비활성화하고 mycpu를 호출하고 가져옵니다.
현재 프로세스 포인터(c->proc)를 struct cpu에서 꺼낸 다음 인터럽트를 활성화합니다. 호출자가 스케줄러에서 실행 중이기 때문에 실행 중인 프로세스가 없는 경우,
myproc는 0을 반환합니다. myproc의 반환 값은 인터럽트가 발생하더라도 안전하게 사용할 수 있습니다.
활성화됨: 타이머 인터럽트가 호출 프로세스를 다른 프로세서로 이동하는 경우
struct proc 포인터는 동일하게 유지됩니다.

수면과 각성

스케줄링과 잠금은 한 프로세스의 존재를 다른 프로세스로부터 숨기는 데 도움이 되지만
지금까지 우리는 프로세스가 의도적으로 상호 작용하는 데 도움이 되는 추상화가 없습니다. 수면 및
웨이크업은 그 공백을 채워서 한 프로세스가 이벤트를 기다리며 절전 모드로 전환되고 다른 프로세스가 절전 모드로 전환되도록 합니다.
이벤트가 발생한 후 깨우기 위한 프로세스입니다. 수면과 깨우기는 종종
시퀀스 조정 또는 조건 동기화 메커니즘이 많이 있습니다.
운영 체제 관련 문헌에 나와 있는 유사한 메커니즘.

우리가 말하는 바를 설명하기 위해 간단한 생산자/소비자 대기열을 생각해 보겠습니다.
이 대기열은 프로세스에서 IDE 드라이버로 명령을 공급하는 대기열과 유사하지만(3장 참조) 모든 IDE 특정 코드를 추상화합니다. 대기열은 다음을 허용합니다.
프로세스가 0이 아닌 포인터를 다른 프로세스로 전송합니다. 보내는 사람이 한 명뿐이라면
그리고 하나의 수신기가 있고, 그것들이 다른 CPU에서 실행되었고, 컴파일러가 너무 적극적으로 최적화하지 않았다면, 이 구현은 정확할 것입니다:

```
100     구조체 q {
101         무효 *ptr;
102     };
103
104     무효의*
105     구조체 q *q, void *p를 보냅니다.
106     {
107         while(q->ptr != 0)
108             ;
109         q->ptr = p;
110     }
111
```

```
112     무효의*
113     recv(구조체 q *q)
114     {
115         무효 *p;
116
117         p = q->ptr == 0인 동안
118             ;
119         q->ptr = 0;
120         p를 반환합니다.
121     }
```

바쁘게 기다리고 있다
투표
수면+코드
웨이크업+코드
찬+코드
대기 채널

큐가 비어 있을 때까지(ptr == 0) 루프를 보낸 다음 포인터 p를 큐에 넣습니다.
큐. Recv는 큐가 비어 있지 않을 때까지 반복하고 포인터를 꺼냅니다. 실행 시
다른 프로세스에서 send와 recv는 모두 q->ptr을 수정하지만 send는 q->ptr만 씁니다.
포인터가 0일 때만 쓰고, recv는 포인터가 0이 아닐 때만 쓰기 때문에 업데이트 내용은 손실되지 않습니다.

위의 구현은 비용이 많이 듭니다. 발신자가 거의 보내지 않으면 수신자가
대부분의 시간을 while 루프에서 회전하며 포인터를 바라며 보냅니다. 수신자의 CPU는 반복적으로 폴링 하여 바쁜 대기 보다 더 생산적인 작업을 찾을 수 있습니다.
q->ptr. 바쁜 대기를 피하려면 수신기가 CPU를 양보할 수 있는 방법이 필요합니다.
send가 포인터를 전달할 때만 재개합니다.

다음과 같이 작동하는 수면 및 깨우기라는 두 가지 통화를 상상해 보겠습니다.
Sleep(chan)은 대기 채널이라고 하는 임의의 값 chan에서 잠을 잡니다 . Sleep은
호출 프로세스를 절전 모드로 전환하여 CPU를 다른 작업에 할당합니다. Wakeup(chan)이 깨어납니다.
chan에서 sleep하는 모든 프로세스(있는 경우)가 sleep 호출을 반환하도록 합니다. chan에서 기다리는 프로세스가 없으면 wakeup은 아무 작업도 수행하지 않습니다. sleep과 wakeup을 사용하도록 대기열 구현을 변경할 수 있습니다.

```
201     무효의*
202     구조체 q *q, void *p를 보냅니다.
203     {
204         while(q->ptr != 0)
205             ;
206         q->ptr = p;
207         wakeup(q); /* 웨이크 수신 */
208     }
209
210     무효의*
211     recv(구조체 q *q)
212     {
213         무효 *p;
214
215         p = q->ptr == 0인 동안
216             수면(q);
217         q->ptr = 0;
218         p를 반환합니다.
219     }
```

Recv는 이제 회전하는 대신 CPU를 포기하는데, 이는 좋습니다. 그러나
이 인터페이스를 사용하면 수면과 깨우기를 설계하는 것이 간단하지 않습니다.
''잃어버린 깨어남'' 문제로 알려진 문제로 고통받고 있습니다(그림 5-2 참조). 가정해 보겠습니다.
recv는 215번째 줄에서 q->ptr == 0을 찾습니다. recv는 215번째 줄과 216번째 줄 사이에 있습니다.

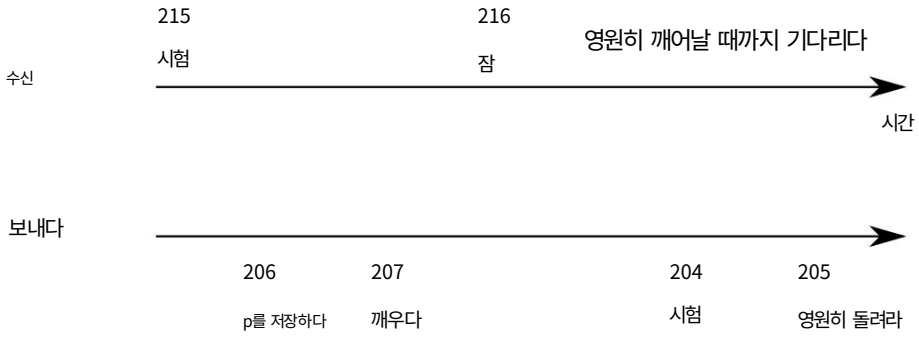


그림 5-2. 웨이크업 손실 문제의 예

교착 상태

216, send는 다른 CPU에서 실행됩니다. q->ptr을 0이 아닌 값으로 변경하고 wakeup을 호출합니다.
프로세스가 잠자고 있지 않아서 아무것도 하지 않는 것을 발견합니다. 이제 recv는 216번째 줄에서 실행을 계속합니다. sleep을 호출하고 sleep 상태로 전환됩니다. 이로 인해 문제가 발생합니다. recv는
이미 도착한 포인터를 기다리며 잠들어 있습니다. 다음 전송은 수신을 기다릴 것입니다.
큐에 있는 포인터를 소모하면 시스템은 교착 상태에 빠지게 됩니다.

이 문제의 근원은 recv가 q->ptr일 때만 sleep한다는 불변성입니다.
== 0은 잘못된 순간에 실행되는 send로 인해 위반됩니다. 잘못된 방법 중 하나
불변식을 보호하려면 recv에 대한 코드를 다음과 같이 수정해야 합니다.

```
300     구조체 q {
301         구조체 스핀락 잠금;
302         무효 *ptr;
303     };
304
305     무효의*
306     구조체 q *q, void *p를 보냅니다.
307     {
308         획득(&q->잠금);
309         while(q->ptr != 0)
310             ;
311         q->ptr = p;
312         웨이크업(q);
313         잠금 해제(&q->lock);
314     }
315
316     무효의*
317     recv(구조체 q *q)
318     {
319         무효 *p;
320
321         획득(&q->잠금);
322         p = q->ptr == 0인 동안
323             수면(q);
324         q->ptr = 0;
325         잠금 해제(&q->lock);
326         p를 반환합니다.
327     }
```

이 recv 버전이 웨이크업 손실을 피할 수 있기를 바랄 수도 있습니다.

lock은 322행과 323행 사이에서 send가 실행되는 것을 방지합니다. 그렇게 하지만 또한
교착 상태: recv는 잠자는 동안 잠금을 유지하므로 송신자는 영원히 차단되어 대기합니다.
자물쇠 때문에.

수면+코드
수면+코드
웨이크업+코드
SLEEPING+코드
스케줄+코드
실행 가능한 코드

우리는 sleep의 인터페이스를 변경하여 이전 계획을 수정할 것입니다: 호출자는 다음을 통과해야 합니다.
호출 프로세스가 잠금으로 표시된 후 잠금을 해제할 수 있도록 잠금을 해제합니다.
잠자고 있고 sleep 채널에서 기다리고 있습니다. 잠금은 동시 전송을 강제로 기다리게 합니다.
수신기가 스스로 절전 모드로 전환하는 것을 마칠 때까지 웨이크업이 다음을 찾을 수 있도록 합니다.
수면 중인 수신기를 깨우고 깨웁니다. 수신기가 다시 깨어나면 수면이 다시 획득됩니다.
반환하기 전에 잠금을 해제합니다. 우리의 새로운 올바른 계획은 다음과 같이 사용할 수 있습니다.

```
400     구조체 q {
401         구조체 스핀락 잠금;
402         무효 *ptr;
403     };
404
405     무효의*
406     구조체 q *q, void *p를 보냅니다.
407     {
408         획득(&q->잠금);
409         while(q->ptr != 0)
410             ;
411         q->ptr = p;
412         웨이크업(q);
413         잠금 해제(&q->lock);
414     }
415
416     무효의*
417     recv(구조체 q *q)
418     {
419         무효 *p;
420
421         획득(&q->잠금);
422         p = q->ptr == 0인 동안
423             sleep(q, &q->lock);
424         q->ptr = 0;
425         잠금 해제(&q->lock);
426         p를 반환합니다.
427     }
```

recv가 q->lock을 보유하고 있다는 사실은 send가 recv가 q->ptr을 체크하고 sleep을 호출하는 사이에 그것을 깨우려고 시도하는 것을 방지합니다. q->lock을 원자적으로 다시 해제하고 수신 프로세스를 sleep 상태로 만들려면 sleep이 필요합니다.

완전한 송신기/수신기 구현은 수신기가 이전 전송에서 값을 소비할 때까지 대기하는 동안 send 모드에서도 절전 모드로 전환됩니다.

코드: 잠과 깨기

sleep (2874) 과 wakeup (2953) 의 구현을 살펴보자 . 기본적인 아이디어
sleep이 현재 프로세스를 SLEEPING으로 표시한 다음 sched를 호출하여 해제하는 것입니다.
프로세서; 웨이크업은 지정된 대기 채널에서 잠자고 있는 프로세스를 찾습니다.
RUNNABLE로 표시합니다. sleep 및 wakeup 호출자는 상호 편리한 모든 것을 사용할 수 있습니다.

채널로 번호를 사용합니다. Xv6는 종종 대기에 관련된 커널 데이터 구조의 주소를 사용합니다.

ptable.lock+코드 웨이크업+코드 ptable.lock+코드 ptable.lock+코드 웨이크업1+코드 웨이크업+코드 SLEEPING+코드 chan+코드 RUNNABLE+코드

Sleep (2874)은 몇 가지 정신 건강 검사로 시작합니다. 현재 프로세스가 있어야 하고 (2878) sleep은 잠금을 통과했어야 합니다 (2881-2882). 그런 다음 sleep은 pt-able.lock (2891)을 획득합니다. 이제 sleep으로 가는 프로세스는 ptable.lock과 lk를 모두 보유합니다.
호출자(예시에서 recv)에서 lk를 유지하는 것이 필요했습니다. 다른 프로세스(예시에서 send를 실행 중)가 wakeup(chan)에 대한 호출을 시작할 수 없도록 했습니다. 이제 sleep이 ptable.lock을 유지했으므로 lk를 해제하는 것이 안전합니다. 다른 프로세스가 wakeup(chan)에 대한 호출을 시작할 수 있지만, wakeup은 ptable.lock을 획득할 때까지 실행되지 않으므로 sleep이 프로세스를 sleep 상태로 만드는 것을 마칠 때까지 기다려야 wakeup이 sleep을 놓치지 않도록 합니다.

사소한 문제가 있습니다. lk가 &ptable.lock과 같으면 sleep은 &ptable.lock으로 획득하려고 시도하다가 교착 상태에 빠진 다음 lk로 해제합니다. 이 경우 sleep은 획득과 해제를 서로 상쇄하여 완전히 건너뜁니다 (2890). 예를 들어 wait (2964) 는 &ptable.lock으로 sleep을 호출합니다.

이제 sleep은 ptable.lock만 보유하고 다른 것은 보유하지 않으므로 sleep 채널을 기록하고, 프로세스 상태를 변경하고, sched (2895-2898)를 호출하여 프로세스를 절전 모드로 전환할 수 있습니다.

어느 시점에 프로세스가 wakeup(chan)을 호출합니다. Wakeup (2964)은 pt-able.lock을 획득하고 실제 작업을 수행하는 wakeup1을 호출합니다. wakeup이 ptable.lock을 유지하는 것이 중요한 이유는 프로세스 상태를 조작하고, 방금 살펴본 것처럼 ptable.lock이 sleep과 wakeup이 서로를 놓치지 않도록 보장하기 때문입니다.

Wakeup1은 별도의 함수인데, 스케줄러가 이미 ptable.lock을 보유한 경우에도 웨이크업을 실행해야 하는 경우가 있기 때문입니다. 이에 대한 예를 나중에 살펴보겠습니다.
Wakeup1 (2953) 은 프로세스 테이블을 반복합니다. 일치하는 chan을 가진 SLEEPING 상태의 프로세스를 찾으면 해당 프로세스의 상태를 RUNNABLE로 변경합니다. 다음에 스케줄러가 실행되면 프로세스가 실행될 준비가 되었음을 알 수 있습니다.

Xv6 코드는 항상 sleep 조건을 보호하는 잠금을 유지하는 동안 wakeup을 호출합니다. 위의 예에서 해당 잠금은 q->lock입니다. 엄밀히 말해서 wakeup이 항상 acquire 다음에 오면 충분합니다(즉, release 후에 wakeup을 호출할 수 있습니다).
sleep 및 wakeup에 대한 잠금 규칙이 sleep 중인 프로세스가 필요한 wakeup을 놓치지 않도록 보장하는 이유는 무엇일까요? sleep 중인 프로세스는 조건을 확인하기 전 지점부터 sleep으로 표시된 후 지점까지 조건에 대한 잠금이나 ptable.lock 또는 둘 다를 보유합니다. 동시 스레드가 조건을 참으로 만드는 경우 해당 스레드는 sleep 중인 스레드가 조건을 획득하기 전이나 sleep 중인 스레드가 조건을 해제한 후에 조건에 대한 잠금을 보유해야 합니다. 그 전에라면 sleep 중인 스레드는 새 조건 값을 보고 어쨌든 sleep하기로 결정했을 것이므로 wakeup을 놓치더라도 문제가 없습니다. 그 이후라면 waker가 조건에 대한 잠금을 획득할 수 있는 가장 빠른 시기는 sleep이 ptable.lock을 획득한 후이므로 wakeup이 pt-able.lock을 획득하려면 sleep이 sleeper를 완전히 sleep 상태로 전환할 때까지 기다려야 합니다.

그러면 웨이크업이 잠자는 과정을 보고 깨웁니다(다른 것이 먼저 깨우지 않는 한).

여러 프로세스가 같은 채널에서 sleep하는 경우가 있습니다. 예를 들어, 파이프에서 읽는 프로세스가 두 개 이상인 경우입니다. wakeup에 대한 단일 호출로 모든 프로세스가 깨어납니다. 그중 하나가 먼저 실행되어 sleep이 호출된 잠금을 획득하고(파이프의 경우) 파이프에서 대기 중인 모든 데이터를 읽습니다.

다른 프로세스는 깨어났음에도 읽을 데이터가 없다는 것을 알게 됩니다.
그들의 관점에서 보면 깨어남은 ''거짓''이었고, 그들은 다시 잠을 자야 합니다. 이런 이유로 sleep은 항상 조건을 확인하는 루프 내부에서 호출됩니다.

sleep/wakeup을 두 번 사용하여 실수로 같은 채널을 선택하더라도 해가 되지 않습니다. 잘못된 wakeup이 발생하지만 위에서 설명한 대로 루핑하면 이 문제가 허용됩니다. sleep/wakeup의 매력은 가볍고(sleep 채널 역할을 하는 특수 데이터 구조를 만들 필요가 없음) 간접 계층을 제공한다는 것입니다(호출자는 자신이 상호 작용하는 특정 프로세스를 알 필요가 없음).

코드: 파이프 이 장의

앞부분에서 사용한 간단한 큐는 장난감이었지만, xv6에는 sleep과 wakeup을 사용하여 리더와 라이터를 동기화하는 두 개의 실제 큐가 들어 있습니다. 하나는 IDE 드라이버에 있습니다. 프로세스가 디스크 요청을 큐에 추가한 다음 sleep을 호출합니다. IDE 인터럽트 핸들러는 wakeup을 사용하여 프로세스에 요청이 완료되었음을 알립니다.

더 복잡한 예는 파이프 구현입니다. 우리는 0장에서 파이프 인터페이스를 보았습니다. 파이프의 한쪽 끝에 쓰여진 바이트는 커널 내부 버퍼에 복사된 다음 파이프의 다른 쪽 끝에서 읽을 수 있습니다. 이후의 장에서는 파이프를 둘러싼 파일 디스크립터 지원을 살펴보겠지만, 지금은 pipewrite와 piperead의 구현을 살펴보겠습니다.

각 파이프는 잠금과 데이터 버퍼를 포함하는 구조체 파이프로 표현됩니다. nread 및 nwrite 필드는 버퍼에서 읽고 버퍼에 쓴 바이트 수를 계산합니다. 버퍼는 래핑됩니다. buf[PIPESIZE-1] 다음에 쓰여진 다음 바이트는 buf[0]입니다. 카운트는 래핑되지 않습니다. 이 규칙은 구현이 전체 버퍼(nwrite == nread+PIPESIZE)와 빈 버퍼(nwrite == nread)를 구별할 수 있게 하지만, 버퍼에 대한 인덱싱은 buf[nread] 대신 buf[nread % PIPESIZE]를 사용해야 함을 의미합니다(nwrite의 경우도 마찬가지입니다). piperead 및 pipewrite에 대한 호출이 두 개의 다른 CPU에서 동시에 발생한다고 가정해 보겠습니다.

Pipewrite (6830) 는 파이프의 잠금을 획득하는 것으로 시작하는데, 이는 카운트, 데이터 및 연관된 불변식을 보호합니다. 그런 다음 Piperead (6851) 도 잠금을 획득하려고 하지만 획득할 수 없습니다. 잠금을 기다리며 acquire (1574) 에서 회전합니다 . Piperead가 기다리는 동안 Pipewrite는 쓰여지는 바이트(addr[0], addr[1], ..., addr[n-1])를 반복하면서 각각을 차례로 파이프에 추가합니다 (6844). 이 루프 중에 버퍼가 채워질 수 있습니다 (6836). 이 경우 Pipewrite는 wakeup을 호출하여 버퍼에서 대기 중인 데이터가 있다는 사실을 잠자고 있는 모든 판독자에게 경고한 다음 &p->nwrite에서 잠자면서 판독자가 버퍼에서 바이트를 꺼낼 때까지 기다립니다. Sleep은 Pipewrite의 프로세스를 잠자게 하는 일부로 p->lock을 해제합니다.

이제 p->lock을 사용할 수 있으므로, piperead는 이를 획득하고 중요한 섹션으로 들어갑니다. p->nread != p->nwrite (6856) 를 찾습니다 (pipewrite가 절전 모드로 전환된 이유는 p->nwrite == p->nread+PIPESIZE (6836) 때문입니다). 따라서 for 루프로 넘어가 파이프에서 데이터를 복사하고 (6863-6867) 복사된 바이트 수만큼 nread를 증가시킵니다. 이제 쓰기에 사용할 수 있는 바이트 수가 그만큼이므로, piperead는 호출자에게 돌아가기 전에 절전 모드에 있는 모든 작성자를 깨우기 위해 wakeup (6868) 을 호출합니다 . Wakeup은 &p->nwrite에서 절전 모드에 있는 프로세스를 찾습니다. 이 프로세스는 pipewrite를 실행 중이었지만 버퍼가 채워졌을 때 멈췄습니다. 해당 프로세스를 RUNNABLE로 표시합니다.

파이프 코드는 리더와 라이터에 대해 별도의 슬립 채널(p->nread 및 p->nwrite)을 사용합니다. 이는 많은 리더와 라이터가 같은 파이프를 기다리는 경우 시스템의 효율성을 높일 수 있습니다. 파이프 코드는 슬립 조건을 확인하는 루프 내부에서 슬립합니다. 여러 리더나 라이터가 있는 경우 깨어나는 첫 번째 프로세스가 아닌 모든 프로세스는 조건이 여전히 거짓임을 보고 다시 슬립합니다.

대기+코드
좀비+코드
구조체 proc+코드 종료+코드
sched+코드 p->kstack+코드 p->pgdir+코드 스위치+코드

코드: 대기, 종료 및 종료

Sleep과 wakeup은 여러 종류의 대기에 사용될 수 있습니다. 0장에서 볼 수 있는 흥미로운 예는 부모 프로세스가 자식이 종료될 때까지 기다리는 데 사용하는 wait 시스템 호출입니다. 자식이 종료되면 즉시 죽지 않습니다. 대신 부모가 wait을 호출하여 종료를 알 때까지 ZOMBIE 프로세스 상태로 전환됩니다. 그런 다음 부모는 프로세스와 관련된 메모리를 해제하고 재사용을 위해 struct proc를 준비할 책임이 있습니다. 부모가 자식보다 먼저 종료되면 init 프로세스가 자식을 채택하고 기다리므로 모든 자식은 부모가 정리할 수 있습니다.

구현 과제는 부모와 자식 wait과 exit, 그리고 exit과 exit 간의 경쟁 가능성입니다. Wait은 ptable.lock을 획득하는 것으로 시작합니다.
그런 다음 프로세스 테이블을 스캔하여 자식을 찾습니다. wait이 현재 프로세스에 자식이 있지만 아무도 종료되지 않았다는 것을 발견하면 sleep을 호출하여 자식 중 하나가 종료될 때까지 기다린 다음 (2707) 다시 스캔합니다. 여기서 sleep에서 해제되는 잠금은 ptable.lock이며, 위에서 본 특별한 경우입니다.

Exit는 ptable.lock을 획득한 다음 현재 프로세스의 부모 proc (2651)과 같은 대기 채널에서 잠자고 있는 모든 프로세스를 깨웁니다. 그러한 프로세스가 있는 경우 대기 중인 부모가 됩니다. 이는 exit가 아직 현재 프로세스를 ZOMBIE로 표시하지 않았기 때문에 성급하게 보일 수 있지만 안전합니다. wakeup으로 인해 부모가 실행될 수 있지만 wait의 루프는 exit가 sched를 호출하여 스케줄러에 들어가 ptable.lock을 해제할 때까지 실행할 수 없으므로 wait은 exit가 상태를 ZOMBIE로 설정할 때까지 종료 프로세스를 볼 수 없습니다 (2663). exit가 프로세서를 양보하기 전에 종료 프로세스의 모든 자식을 다시 부모로 지정하여 initproc에 전달합니다 (2653-2660). 마지막으로 exit는 sched를 호출하여 CPU를 포기합니다.

부모 프로세스가 대기 중이면 스케줄러가 결국 해당 프로세스를 실행하게 됩니다.
sleep에 대한 호출은 holding ptable.lock을 반환합니다. wait은 프로세스 테이블을 다시 스캔하고 state == ZOMBIE인 종료된 자식을 찾습니다. (2657). 자식의 pid를 기록한 다음 struct proc를 정리하여 프로세스와 관련된 메모리를 해제합니다 (2687-2694).

자식 프로세스는 종료 중에 대부분의 정리를 할 수 있지만, 부모 프로세스가 p->kstack과 p->pgdir을 해제하는 것이 중요합니다. 자식이 종료를 실행하면 스택은 p->kstack으로 할당된 메모리에 저장되고 자체 페이지 테이블을 사용합니다. 자식 프로세스가 마지막으로 실행을 완료한 후에 swtch(sched를 통해)를 호출하여 해제할 수 있습니다. 이것이 스케줄러 프로시저가 sched를 호출한 스레드의 스택이 아닌 자체 스택에서 실행되는 한 가지 이유입니다.

exit가 프로세스가 스스로 종료할 수 있도록 허용하는 반면, kill (2975) 은 한 프로세스가 다른 프로세스가 종료되도록 요청할 수 있도록 합니다. kill이 피해자 프로세스를 직접 파괴하기에는 너무 복잡할 것입니다. 피해자가 다른 CPU에서 실행 중이거나 커널 데이터 구조를 업데이트하는 중간에 절전 모드에 있을 수 있기 때문입니다. 이러한 과제를 해결하기 위해 kill은 거의 아무것도 하지 않습니다. 피해자의 p->killed를 설정하고 절전 모드에 있는 경우

이를 깨웁니다. 결국 피해자는 커널에 들어가거나 나가게 되고, 이때 p->killed가 설정되어 있으면 trap의 코드가 exit를 호출합니다. 피해자가 사용자 공간에서 실행 중이면 시스템 호출을 하거나 타이머(또는 다른 장치)가 인터럽트를 걸어 곧 커널에 들어갑니다.

피해자 프로세스가 sleep 상태인 경우 wakeup 호출로 인해 피해자 프로세스가 sleep에서 복귀합니다. wait-ing for 조건이 참이 아닐 수 있으므로 잠재적으로 위험합니다. 그러나 xv6 sleep 호출은 sleep이 복귀한 후 조건을 다시 테스트하는 while 루프에 항상 래핑됩니다. sleep에 대한 일부 호출은 루프에서 p- >killed도 테스트하고 현재 활동이 설정되어 있으면 중단합니다. 이는 이러한 중단이 올바를 때만 수행됩니다. 예를 들어, 파이프 읽기 및 쓰기 코드 (6837) 는 killed 플래그가 설정되어 있으면 복귀합니다. 결국 코드는 trap으로 돌아가서 다시 플래그를 확인하고 종료합니다.

일부 xv6 sleep 루프는 p->killed를 확인하지 않습니다. 그 이유는 코드가 원자적이어야 하는 다단계 시스템 호출의 중간에 있기 때문입니다. IDE 드라이버 (4379) 는 한 예입니다. 디스크 작업이 파일 시스템을 올바른 상태로 유지하기 위해 필요한 쓰기 세트 중 하나일 수 있기 때문에 p->killed를 확인하지 않습니다. 부분 작업 후 정리하는 복잡성을 피하기 위해 xv6는 IDE 드라이버에 있는 프로세스의 종료를 프로세스를 종료하기 쉬운 시점(예: 전체 파일 시스템 작업이 완료되고 프로세스가 사용자 공간으로 돌아가려고 할 때)까지 지연합니다.

현실 세계

xv6 스케줄러는 각 프로세스를 차례로 실행하는 간단한 스케줄링 정책을 구현합니다. 이 정책을 라운드 로빈이라고 합니다. 실제 운영 체제는 예를 들어 프로세스에 우선순위를 부여하는 것과 같은 보다 정교한 정책을 구현합니다. 아이디어는 스케줄러가 실행 가능한 우선순위가 높은 프로세스를 우선순위가 낮은 실행 가능한 프로세스보다 선호한다는 것입니다. 이러한 정책은 종종 상충되는 목표가 있기 때문에 빠르게 복잡해질 수 있습니다. 예를 들어 운영 체제는 공정성과 높은 처리량을 보장하고자 할 수도 있습니다. 또한 복잡한 정책은 우선순위 역전 및 호송과 같은 의도치 않은 상호 작용으로 이어질 수 있습니다. 우선순위 역전은 우선순위가 낮은 프로세스와 우선순위가 높은 프로세스가 잠금을 공유할 때 발생할 수 있으며, 우선순위가 낮은 프로세스가 잠금을 획득하면 우선순위가 높은 프로세스가 진행되지 않을 수 있습니다.

긴 호송대는 많은 높은 우선순위 프로세스가 공유 잠금을 획득하는 낮은 우선순위 프로세스를 기다릴 때 형성될 수 있습니다. 호송대가 형성되면 오랫동안 지속될 수 있습니다. 이런 종류의 문제를 피하려면 정교한 스케줄러에 추가 메커니즘이 필요합니다.

Sleep과 wakeup은 간단하고 효과적인 동기화 방법이지만, 그 외에도 많은 방법이 있습니다. 이 모든 방법에서 첫 번째 과제는 이 장의 시작 부분에서 본 ''잃어버린 wakeups'' 문제를 피하는 것입니다. 원래 Unix 커널의 sleep은 단순히 인터럽트를 비활성화했는데, Unix가 단일 CPU 시스템에서 실행되었기 때문에 충분했습니다. xv6은 멀티프로세서에서 실행되기 때문에 sleep에 명시적 잠금을 추가합니다. FreeBSD의 msleep도 같은 접근 방식을 취합니다. Plan 9의 sleep은 sleep에 들어가기 직전에 스케줄링 잠금을 유지한 채로 실행되는 콜백 함수를 사용합니다. 이 함수는 sleep 상태에 대한 마지막 순간 확인 역할을 하여 잃어버린 wakeups를 방지합니다. Linux 커널의 sleep은

대기 채널 대신 명시적 프로세스 대기열을 사용합니다. 대기열에는 자체 내부 잠금이 있습니다.

웨이크업에서 일치하는 채널이 있는 프로세스에 대한 전체 프로세스 목록을 스캔하는 것은 다음과 같습니다.
비효율적입니다. 더 나은 해결책은 sleep과 wakeup 모두에서 chan을 다음과 같이 대체하는 것입니다.
해당 구조에서 잠자는 프로세스 목록을 보관하는 데이터 구조. Plan 9의 잠자기
그리고 랑데부 지점이나 랑데즈를 구성하는 웨이크업 콜. 많은 스레드 라이브러리는 조건 변수와 동일한 구조
를 참조합니다. 그 맥락에서 작업 sleep
그리고 웨이크업은 대기와 신호라고 불립니다. 이러한 모든 메커니즘은 동일한 풍미를 공유합니다. 즉, 수면 조
건은 원자적으로 삭제된 어떤 종류의 잠금으로 보호됩니다.
잠.

웨이크업 구현은 특정 채널에서 대기 중인 모든 프로세스를 깨우며, 많은 프로세스가 해당 특정 채널을
기다리고 있을 수 있습니다. 운영 체제는 이러한 모든 프로세스를 스케줄링하고

수면 상태를 확인하기 위한 경쟁입니다. 이런 식으로 동작하는 프로세스는 때때로
천둥치는 무리 라고 불리며 , 피하는 것이 가장 좋습니다. 대부분의 조건 변수에는 두 가지가 있습니다.
웨이크업을 위한 기본 요소: 하나의 프로세스를 깨우는 신호(signal)와 브로드캐스트(broadcast)
대기 중인 모든 프로세스를 깨웁니다.

세마포어는 또 다른 일반적인 조정 메커니즘입니다. 세마포어는
두 가지 연산, 증가 및 감소(또는 위아래)를 갖는 정수 값입니다.
세마포어를 증가시키는 것은 항상 가능하지만 세마포어 값은 허용되지 않습니다.
0 이하로 떨어짐: 0 세마포어의 감소는 다른 프로세스가 세마포어를 증가시킬 때까지 휴면 상태가 되고, 그런
다음 두 연산은 취소됩니다. 정수 값
일반적으로 파이프에서 사용 가능한 바이트 수와 같은 실제 카운트에 해당합니다.
버퍼 또는 프로세스가 가지고 있는 좀비 자식의 수. 명시적 카운트를 사용하여
추상화의 일부는 ''잃어버린 웨이크업'' 문제를 피합니다. 명시적인 계산이 있습니다.
발생한 웨이크업의 수. 이 카운트는 또한 허위 웨이크업과 천둥치는 무리 문제를 피합니다.

xv6에서는 프로세스를 종료하고 정리하는 작업이 매우 복잡해졌습니다.
대부분의 운영 체제에서는 예를 들어 피해자가 다음과 같은 이유로 더욱 복잡합니다.
프로세스는 커널 내부에서 깊이 잠들어 있을 수 있으며 스택을 풀려면 많은 시간이 필요합니다.
신중한 프로그래밍. 많은 운영 체제는 longjmp와 같은 예외 처리를 위한 명시적 메커니즘을 사용하여 스택을
풀어줍니다. 게다가 다른 이벤트도 있습니다.
이는 대기 중인 이벤트가 있음에도 불구하고 잠자는 프로세스가 깨어날 수 있도록 합니다.
for는 아직 발생하지 않았습니다. 예를 들어, Unix 프로세스가 휴면 상태일 때 다른 프로세스가 신호를 보낼
수 있습니다. 이 경우 프로세스는 값 -1과 오류 코드가 EINTR로 설정된 인터럽트된 시스템 호출에서 반환됩니
다. 응용 프로그램
이러한 값을 확인하고 무엇을 할지 결정할 수 있습니다. Xv6는 신호를 지원하지 않으며
복잡성은 발생하지 않습니다.

Xv6의 kill 지원은 전적으로 만족스럽지 않습니다. sleep 루프가 있습니다.
아마도 p->killed를 확인해야 할 것입니다. 관련 문제는 sleep 루프의 경우에도 마찬가지입니다.
p->killed를 확인하면 sleep과 kill 사이에 경쟁이 발생합니다. 후자는 p-를 설정할 수 있습니다.
> 피해자의 루프가 p->killed를 확인한 직후에 피해자를 죽이고 깨우려고 시도합니다.
하지만 잠들기 전에. 이 문제가 발생하면 피해자는 p-를 알아차리지 못할 것입니다.
>기다리는 조건이 발생할 때까지 죽임을 당합니다. 이는 꽤 늦을 수 있습니다(예:
IDE 드라이버가 피해자가 기다리고 있는 디스크 블록을 반환하는 경우(예:) 또는 전혀 반환하지 않는 경우(예:
피해자가 콘솔에서 입력을 기다리고 있지만 사용자가 아무 것도 입력하지 않는 경우

놓다).

수업 과정

1. Sleep은 교착 상태를 피하기 위해 lk != &ptable.lock을 확인해야 합니다 (2890-2893). 특수한 경우가 제거되었다고 가정합니다.

```
if(lk != &ptable.lock){ 획득
    (&ptable.lock); 해제(lk);

}
```

~와 함께

```
해제(lk); 획득
(&ptable.lock);
```

이렇게 하면 잠이 깨질 거야. 어떻게?

2. 대부분 프로세스 정리는 종료 또는 대기로 수행할 수 있지만, 위에서 종료는 p->stack을 해제해서는 안 된다는 것을 보았습니다. 종료는 열린 파일을 닫는 것이어야 합니다. 왜 그럴까요? 답은 파이프와 관련이 있습니다.

3. xv6에서 세마포어를 구현합니다. 뮤텍스는 사용할 수 있지만 sleep과 wakeup은 사용하지 마세요. xv6에서 sleep과 wakeup의 사용을 세마포어로 바꾸세요. 결과를 판단하세요.

4. 위에서 언급한 kill과 sleep 간의 경쟁을 수정하여 피해자의 sleep 루프가 p->killed를 확인한 후 sleep을 호출하기 전에 kill이 발생하면 피해자가 현재 시스템 호출을 중단하게 됩니다.

5. 모든 sleep 루프가 p->killed를 확인하도록 계획을 설계하여, 예를 들어 IDE 드라이버에 있는 프로세스가 다른 프로세스가 해당 프로세스를 죽이더라도 while 루프에서 빠르게 돌아올 수 있도록 합니다.

6. 한 사용자 프로세스에서 다른 사용자 프로세스로 전환할 때 하나의 컨텍스트 전환만 사용하는 계획을 설계합니다. 이 계획에는 전용 스케줄러 스택 대신 사용자 프로세스의 커널 스택에서 스케줄러 프로시저를 실행하는 것이 포함됩니다. 가장 큰 과제는 사용자 프로세스를 올바르게 정리하는 것입니다. 하나의 컨텍스트 전환을 피하는 것의 성능 이점을 측정합니다.

7. xv6를 수정하여 스케줄러에서 유휴 상태이고 루프에서 회전할 때 프로세서를 끕니다. (힌트: x86 HLT 명령어를 살펴보세요.)

8. 잠금 p->lock은 많은 불변성을 보호하고, p->lock으로 보호되는 특정 xv6 코드를 볼 때 어떤 불변성이 적용되는지 알아내기 어려울 수 있습니다. p->lock을 여러 잠금으로 분할하여 더 깔끔한 계획을 설계하세요.

6장

파일 시스템

파일 시스템의 목적은 데이터를 구성하고 저장하는 것입니다. 파일 시스템은 일반적으로
사용자와 애플리케이션 간 데이터 공유를 지원하고 지속성을 유지합니다 .
재부팅 후에도 데이터를 계속 사용할 수 있습니다.

xv6 파일 시스템은 Unix와 유사한 파일, 디렉토리 및 경로 이름을 제공하고(0장 참조) 지속성을 위해 IDE 디스크에 데이터를 저장합니다(3장 참조). 이 파일 시스템은 여러 가지 과제를 해결합니다.

- 파일 시스템에는 명명된 디렉토리와 파일의 트리를 표현하고, 각 파일의 내용을 보관하는 블록의 ID를 기록하고, 디스크의 어느 영역이 비어 있는지 기록하기 위한 디스크상 데이터 구조가 필요합니다.

- 파일 시스템은 충돌 복구를 지원해야 합니다. 즉, 충돌(예: 정전)이 발생하면
  발생하면 파일 시스템은 재시작 후에도 여전히 올바르게 작동해야 합니다. 위험은 다음과 같습니다.
  충돌로 인해 업데이트 시퀀스가 중단되고 디스크 데이터가 일관되지 않을 수 있습니다.
  구조(예: 파일에서 사용되면서도 사용 가능으로 표시된 블록).
- 여러 프로세스가 동시에 파일 시스템에서 작동할 수 있으므로 파일
  시스템 코드는 불변성을 유지하기 위해 조정되어야 합니다.
- 디스크에 액세스하는 것은 메모리에 액세스하는 것보다 훨씬 느리므로 파일
  시스템은 인기 있는 블록의 메모리 내 캐시를 유지해야 합니다.

이 장의 나머지 부분에서는 xv6가 이러한 과제를 어떻게 해결하는지 설명합니다.

개요

xv6 파일 시스템 구현은 그림에 표시된 7개 계층으로 구성됩니다.
6-1. 디스크 계층은 IDE 하드 드라이브의 블록을 읽고 씁니다. 버퍼 캐시
레이어는 디스크 블록을 캐시하고 이에 대한 액세스를 동기화하여 하나만 유지되도록 합니다.
한 번에 커널 프로세스가 특정 블록에 저장된 데이터를 수정할 수 있습니다. 로깅 계층은 상위 계층이 트랜잭션에서 여러 블록에 대한 업데이트를 래핑할 수 있도록 허용 합니다 .
충돌이 발생하더라도 블록이 원자적으로 업데이트되도록 보장합니다(즉, 모든 블록이 업데이트됨)
업데이트되거나 없음). inode 계층은 각각이 다음과 같이 표현되는 개별 파일을 제공합니다.
고유한 i-번호와 파일 데이터를 보관하는 일부 블록이 있는 inode . 디렉토리
레이어는 각 디렉토리를 내용이 시퀀스인 특수한 종류의 inode로 구현합니다.
디렉토리 항목의 각각에는 파일 이름과 i-번호가 포함됩니다. 경로명
레이어는 /usr/rtm/xv6/fs.c와 같은 계층적 경로 이름을 제공하고 이를 해결합니다.
재귀적 조회를 통해 파일 설명자 계층은 많은 Unix 리소스를 추상화합니다(예:
파일 시스템 인터페이스를 사용하여 파이프, 장치, 파일 등을 관리함으로써 애플리케이션 프로그래머의 삶을 단순화합니다.

파일 시스템에는 inode와 콘텐츠 블록을 저장하는 위치에 대한 계획이 있어야 합니다.
디스크. 이를 위해 xv6는 그림 6-2와 같이 디스크를 여러 섹션으로 나눕니다.
파일 시스템은 블록 0을 사용하지 않습니다(부트 섹터를 보유합니다). 블록 1은 다음과 같이 불립니다.

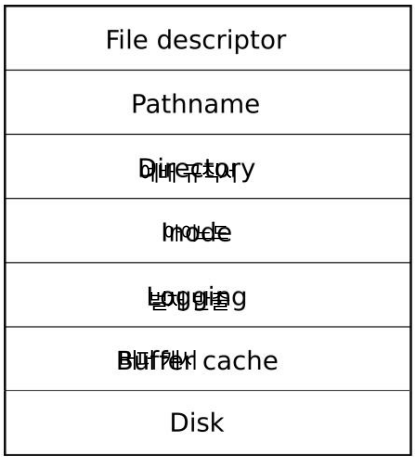


그림 6-1. xv6 파일 시스템의 계층.

슈퍼블록; 여기에는 파일 시스템에 대한 메타데이터(블록 단위의 파일 시스템 크기, 데이터 블록 수, inode 수, 로그의 블록 수)가 포함됩니다.
2에서 시작하는 블록은 로그를 보관합니다. 로그 다음에는 블록당 여러 개의 inode가 있는 inode가 있습니다. 그 다음에는 사용 중인 데이터 블록을 추적하는 비트맵 블록이 옵니다. 나머지 블록은 데이터 블록입니다. 각각은 비트맵 블록에서 비어 있는 것으로 표시되거나 파일이나 디렉토리의 내용을 보관합니다. 슈퍼블록은 초기 파일 시스템을 빌드하는 mfks라는 별도의 프로그램으로 채워집니다.

슈퍼블록
mfks+코드
빵+코드
bwrite+코드
buf
brelse+코드

이 장의 나머지 부분에서는 버퍼 캐시부터 시작하여 각 계층에 대해 설명합니다. 하위 계층에서 잘 선택된 추상화가 상위 계층의 디자인을 쉽게 만드는 상황을 살펴보세요.

버퍼 캐시 계층 버퍼 캐시에는 두 가지

작업이 있습니다. (1) 디스크 블록에 대한 액세스를 동기화하여 블록의 사본이 메모리에 하나만 있고 한 번에 하나의 커널 스레드만 해당 사본을 사용하도록 합니다. (2) 인기 있는 블록을 캐시하여 느린 디스크에서 다시 읽을 필요가 없도록 합니다. 코드는 bio.c에 있습니다.

버퍼 캐시에서 내보내는 주요 인터페이스는 bread와 bwrite로 구성됩니다. 전자는 메모리에서 읽거나 수정할 수 있는 블록의 사본을 포함하는 buf를 얻고 , 후자는 수정된 버퍼를 디스크의 해당 블록에 씁니다. 커널 스레드는 버퍼를 다 사용하면 brelse를 호출하여 버퍼를 해제해야 합니다.

버퍼 캐시는 버퍼당 슬립 잠금을 사용하여 한 번에 하나의 스레드만 각 버퍼(따라서 각 디스크 블록)를 사용하도록 보장합니다. bread는 잠긴 버퍼를 반환하고 brelse는 잠금을 해제합니다.

버퍼 캐시로 돌아가 보겠습니다. 버퍼 캐시에는 고정된 수의 버퍼가 있습니다.
디스크 블록을 보관합니다. 즉, 파일 시스템이 캐시에 없는 블록을 요청하는 경우 버퍼 캐시는 현재 다른 블록을 보관하고 있는 버퍼를 재활용해야 합니다. 버퍼 캐시는 새 블록에 대해 가장 최근에 사용되지 않은 버퍼를 재활용합니다. 가장 최근에 사용되지 않은 버퍼는 다시 사용될 가능성이 가장 낮다는 가정이 있습니다.

그림 6-2. xv6 파일 시스템의 구조. 헤더 fs.h (4050) 에는 파일 시스템의 정확한 레이아웃을 설명하는 상수와 데이터 구조가 들어 있습니다.

곧.

비닛+코드
메인+코드
NBUF+코드
bcache.head+코드
B_VALID+코드
B_DIRTY+코드
bget+코드
iderw+코드
bget+코드
bget+코드
B_VALID+코드

코드: 버퍼 캐시

버퍼 캐시는 버퍼의 이중 연결 리스트입니다. 함수 binit은 다음에 의해 호출됩니다.
main (1230)은 정적 배열 buf (4450-4459) 에 있는 NBUF 버퍼로 목록을 초기화합니다 .
버퍼 캐시에 대한 다른 모든 액세스는 bcache.head를 통한 연결 목록을 참조합니다.
버퍼 배열.

버퍼에는 두 개의 상태 비트가 연관되어 있습니다. B_VALID는 버퍼가
블록의 사본을 포함합니다. B_DIRTY는 버퍼 내용이 수정되었으며 디스크에 기록해야 함을 나타냅니다.

Bread (4502)는 bget을 호출하여 주어진 섹터(4506)에 대한 버퍼를 가져옵니다 . 버퍼가
디스크에서 읽어야 하며, bread는 버퍼를 반환하기 전에 iderw를 호출하여 해당 작업을 수행합니다.

Bget (4466)은 주어진 장치와 섹터 번호 (4472-4480)를 가진 버퍼를 버퍼 목록에서 스캔합니다. 그러한 버퍼가 있으면
bget은 버퍼에 대한 sleep-lock을 획득합니다.
그러면 bget은 잠긴 버퍼를 반환합니다.

주어진 섹터에 대해 캐시된 버퍼가 없으면 bget은 버퍼를 만들어야 합니다.
다른 섹터를 보관한 버퍼를 재사용합니다. 버퍼 목록을 두 번째로 스캔합니다.
잠겨 있지 않고 더럽혀지지 않은 버퍼를 찾습니다. 그러한 버퍼는 모두 사용할 수 있습니다.
Bget은 버퍼 메타데이터를 편집하여 새 장치와 섹터 번호를 기록하고 해당 sleep-lock을 ac-ac-ac합니다. 플래그에 대한 할당
은 B_VALID를 지워서 다음을 보장합니다.
그 빵은 버퍼의 잘못된 사용을 대신하여 디스크에서 블록 데이터를 읽을 것입니다.
이전 내용.

디스크 섹터당 최대 하나의 캐시 버퍼가 있어야 한다는 점이 중요합니다.
독자가 쓰기 내용을 보고 파일 시스템이 동기화를 위해 버퍼에 잠금을 사용하기 때문에 bget은 bache.lock을 지속적으로 유
지하여 이 불변성을 보장합니다.
첫 번째 루프는 블록이 캐시되었는지 확인하기 위해 두 번째 루프의 선언을 통해 블록이 이제 캐시되었는지 확인합니다(dev,
blockno 및 refcnt를 설정하여). 이로 인해
블록의 존재 여부 확인 및 (존재하지 않는 경우) 보관할 버퍼 지정
블록이 원자적이 되어야 합니다.

bget이 bcache.lock 외부에서 버퍼의 sleep-lock을 획득하는 것은 안전합니다.
중요 섹션은 0이 아닌 b->refcnt로 인해 버퍼가 재사용되는 것을 방지합니다.
다른 디스크 블록의 경우. sleep-lock은 블록의 읽기 및 쓰기를 보호합니다.
버퍼링된 콘텐츠인 반면 bcache.lock은 어떤 블록이 버퍼링되는지에 대한 정보를 보호합니다.
캐시됨.

모든 버퍼가 사용 중이면 너무 많은 프로세스가 동시에 실행되고 있는 것입니다.
파일 시스템 호출; bget 패닉. 더 우아한 대응은 잠자기까지일 수 있습니다.

버퍼가 비게 되었지만 교착 상태가 발생할 가능성이 있습니다.

빵+코드
bwrite+code
brelse+code가 버퍼
디스크 하드웨어와

bread가 디스크를 읽고(필요한 경우) 버퍼를 호출자에게 반환하면 iderw+code 호출자는 버퍼를 독점적으로 사용하고 데이터 바이트를 읽거나 쓸 수 있습니다. 호출자 B_DIRTY+code를 수정하는 경우 버퍼를 로그 해제하기 전에 bwrite를 호출하여 변경된 데이터를 디스크에 써야 합니다 . Bwrite (4515)는 iderw가 쓰기(읽기보다는 쓰기)를 나타내기 위해 commit B_DIRTY를 설정한 후 iderw를 호출하여 통신합니다 .

호출자가 버퍼를 다 사용하면 brelse를 호출하여 해제해야 합니다. (brelse라는 이름은 b-release의 줄임말로, 난해하지만 알아둘 가치가 있습니다. Unix에서 시작되어 BSD, Linux, Solaris에서도 사용됩니다.) Brelse (4526)는 sleep-lock을 해제하고 버퍼를 연결 목록의 앞으로 옮깁니다 (4537-4542). 버퍼를 이동하면 목록이 버퍼가 얼마나 최근에 사용되었는지에 따라 정렬됩니다(해제됨을 의미). 목록의 첫 번째 버퍼가 가장 최근에 사용되고 마지막 버퍼가 가장 최근에 사용되지 않습니다. bget의 두 루프는 이를 활용합니다. 기존 버퍼를 스캔하면 최악의 경우 전체 목록을 처리해야 하지만 가장 최근에 사용된 버퍼를 먼저 확인하면(bcache.head에서 시작하여 다음 포인터를 따름) 참조의 지역성이 좋을 때 스캔 시간이 줄어듭니다. 재사용할 버퍼를 선택하는 스캔은 뒤로 스캔하여(이전 포인터를 따름) 가장 최근에 사용되지 않은 버퍼를 선택합니다.

로깅 레이어

파일 시스템 설계에서 가장 흥미로운 문제 중 하나는 크래시 복구입니다. 이 문제는 많은 파일 시스템 작업이 디스크에 여러 번 쓰기를 포함하고 쓰기의 하위 집합 이후 크래시가 발생하면 디스크 파일 시스템이 일관되지 않은 상태가 될 수 있기 때문에 발생합니다. 예를 들어, 파일 잘라내기(파일의 길이를 0으로 설정하고 콘텐츠 블록을 해제) 중에 크래시가 발생한다고 가정해 보겠습니다. 디스크 쓰기 순서에 따라 크래시는 해제로 표시된 콘텐츠 블록을 참조하는 inode를 남기거나 할당되었지만 참조되지 않은 콘텐츠 블록을 남길 수 있습니다.

후자는 비교적 양성이지만, 해제된 블록을 참조하는 inode는 재부팅 후 심각한 문제를 일으킬 가능성이 높습니다. 재부팅 후 커널은 해당 블록을 다른 파일에 할당할 수 있으며, 이제 의도치 않게 같은 블록을 가리키는 두 개의 다른 파일이 있습니다. xv6에서 여러 사용자를 지원했다면, 이 상황은 보안 문제가 될 수 있습니다. 이전 파일의 소유자가 다른 사용자가 소유한 새 파일의 블록을 읽고 쓸 수 있기 때문입니다.

Xv6는 간단한 로깅 형식으로 파일 시스템 작업 중 충돌 문제를 해결합니다. xv6 시스템 호출은 디스크상 파일 시스템 데이터 구조를 직접 쓰지 않습니다. 대신 디스크의 로그 에 수행하려는 모든 디스크 쓰기에 대한 설명을 넣습니다 . 시스템 호출이 모든 쓰기를 로깅하면 디스크에 특별한 커밋 레코드를 기록하여 로그에 완전한 작업이 포함되어 있음을 나타냅니다. 그 시점에서 시스템 호출은 쓰기를 디스크상 파일 시스템 데이터 구조에 복사합니다.

해당 쓰기가 완료되면 시스템 호출은 디스크의 로그를 지웁니다.

시스템이 충돌하고 재부팅되면 파일 시스템 코드는 프로세스를 실행하기 전에 다음과 같이 충돌에서 복구합니다. 로그가 완전한 작업을 포함하는 것으로 표시된 경우 복구 코드는 쓰기를 디스크 파일 시스템에서 속한 위치로 복사합니다. 로그가 완전한 작업을 포함하는 것으로 표시되지 않은 경우 복구 코드는 로그를 무시합니다. 복구 코드는 로그를 지우면서 완료됩니다.

그룹 커밋 xv6의 로그 없음 복구

그가 파일 시스템 작업 중 충돌 문제를 해결하는 이유는 무엇입니까? 작업이 커밋되기 전에 충돌을 일괄 처리 하는 경우 디스크의 로그는 완료로 표시 되지 않고 복구 코드는 이를 무시하며 디스크의 상태는 작업이 시작되지 않은 것처럼 unlink+code가 됩니다 .작업이 커밋된 후에 충돌이 발생하는 경우 복구는 작업의 모든 쓰기를 재생하고 작업이 디스크상 데이터 구조에 쓰기를 시작한 경우 반복할 수 있습니다.어느 경우든 로그는 충돌과 관련하여 작업을 원자적으로 만듭니다.복구 후에 모든 작업의 쓰기가 디스크에 나타나거나 아무것도 나타나지 않습니다.

로그 디자인

로그는 슈퍼블록에 지정된 알려진 고정 위치에 있습니다. 헤더 블록과 그 뒤에 업데이트된 블록 사본(''로그된 블록'') 시퀀스로 구성됩니다.

헤더 블록에는 로깅된 블록마다 하나씩 섹터 번호 배열과 로그 블록 수가 포함됩니다. 디스크의 헤더 블록에 있는 수는 0으로, 로그에 트랜잭션이 없음을 나타내거나, 0이 아닌 경우 로그에 지정된 수의 로깅된 블록이 있는 완전한 커밋된 트랜잭션이 있음을 나타냅니다. Xv6는 트랜잭션이 커밋될 때 헤더 블록을 쓰지만, 그 전에는 쓰지 않고, 로깅된 블록을 파일 시스템에 복사한 후 카운트를 0으로 설정합니다. 따라서 트랜잭션 중간에 충돌이 발생하면 로그의 헤더 블록에 카운트가 0이 되고, 커밋 후에 충돌이 발생하면 카운트가 0이 아닌 값이 됩니다.

각 시스템 호출의 코드는 충돌과 관련하여 원자적이어야 하는 쓰기 시퀀스의 시작과 끝을 나타냅니다. 다른 프로세스가 파일 시스템 작업을 동시에 실행할 수 있도록 로깅 시스템은 여러 시스템 호출의 쓰기를 하나의 트랜잭션으로 누적할 수 있습니다. 따라서 단일 커밋에는 여러 완전한 시스템 호출의 쓰기가 포함될 수 있습니다. 시스템 호출을 트랜잭션 간에 분할하는 것을 방지하기 위해 로깅 시스템은 파일 시스템 시스템 호출이 진행 중이 아닐 때만 커밋합니다.

여러 거래를 함께 커밋하는 개념을 그룹 커밋이라고 합니다.
그룹 커밋은 커밋의 고정 비용을 여러 작업에 걸쳐 상각하기 때문에 디스크 작업 수를 줄입니다. 그룹 커밋은 또한 디스크 시스템에 동시에 더 많은 쓰기를 제공하여 디스크가 단일 디스크 회전 중에 모든 쓰기를 할 수 있도록 합니다. Xv6의 IDE 드라이버는 이러한 종류의 배칭을 지원하지 않지만 xv6의 파일 시스템 설계는 이를 허용합니다.

Xv6는 디스크에 고정된 양의 공간을 할당하여 로그를 보관합니다. 트랜잭션에서 시스템 호출이 작성한 총 블록 수는 해당 공간에 맞아야 합니다.
여기에는 두 가지 결과가 있습니다. 단일 시스템 호출은 로그의 공간보다 더 많은 개별 블록을 쓸 수 없습니다. 이는 대부분의 시스템 호출에는 문제가 되지 않지만, 두 시스템 호출은 잠재적으로 많은 블록을 쓸 수 있습니다. write와 unlink입니다. 대용량 파일 쓰기는 inode 블록뿐만 아니라 많은 데이터 블록과 많은 비트맵 블록을 쓸 수 있습니다. 대용량 파일의 unlink는 많은 비트맵 블록과 inode를 쓸 수 있습니다. Xv6의 write sys-tem 호출은 대용량 쓰기를 로그에 맞는 여러 개의 작은 쓰기로 나누고, unlink는 실제로 xv6 파일 시스템이 비트맵 블록을 하나만 사용하기 때문에 문제를 일으키지 않습니다. 제한된 로그 공간의 또 다른 결과는 로깅 시스템이 시스템 호출의 쓰기가 로그에 남은 공간에 맞는지 확신하지 않는 한 시스템 호출이 시작되도록 허용할 수 없다는 것입니다.

코드: logging

시스템 호출에서 로그를 일반적으로 사용하는 예는 다음과 같습니다.

```
시작_연산();
...
bp = 빵(...); bp->데이터
[...] = ...; log_write(bp);

...
종료_연산();
```

begin_op (4828) 는 로깅 시스템이 현재 커밋하지 않을 때까지, 그리고 이 호출에서 쓰기를 보관할 만큼 예약되지 않은 로그 공간이 충분할 때까지 기다립니다. log.outstanding은 로그 공간을 예약한 시스템 호출의 수를 계산합니다. 총 예약 공간은 log.outstanding 곱하기 MAXOPBLOCKS입니다. log.outstanding을 증가시키면 공간이 예약되고 이 시스템 호출 중에 커밋이 발생하지 않습니다. 이 코드는 각 시스템 호출이 최대 MAXOPBLOCKS개의 개별 블록을 쓸 수 있다고 보수적으로 가정합니다.

log_write (4922) 는 bwrite의 프록시 역할을 합니다. 메모리에 블록의 섹터 번호를 기록하여 디스크의 로그에 슬롯을 예약하고, 버퍼를 B_DIRTY로 표시하여 블록 캐시가 블록을 축출하지 못하도록 합니다. 블록은 커밋될 때까지 캐시에 남아 있어야 합니다. 그때까지 캐시된 사본은 수정 사항의 유일한 기록이며, 커밋 후까지 디스크의 해당 위치에 쓸 수 없습니다. 동일한 트랜잭션의 다른 읽기는 수정 사항을 확인해야 합니다. log_write는 단일 트랜잭션 중에 블록이 여러 번 쓰여지는 경우 이를 감지하고 해당 블록을 로그의 동일한 슬롯에 할당합니다. 이러한 최적화를 종종 흡수라고 합니다. 예를 들어, 여러 파일의 inode가 포함된 디스크 블록이 트랜잭션 내에서 여러 번 쓰여지는 것이 일반적입니다. 여러 디스크 쓰기를 하나로 흡수함으로써 파일 시스템은 로그 공간을 절약하고 디스크 블록의 사본을 하나만 디스크에 써야 하기 때문에 더 나은 성능을 얻을 수 있습니다.

end_op (4853) 는 먼저 미처리 시스템 호출의 카운트를 감소시킵니다. 카운트가 이제 0이면 commit()을 호출하여 현재 트랜잭션을 커밋합니다. 이 프로세스에는 4단계가 있습니다. write_log() (4885) 는 트랜잭션에서 수정된 각 블록을 버퍼 캐시에서 디스크의 로그에 있는 해당 슬롯으로 복사합니다. write_head() (4804) 는 헤더 블록을 디스크에 씁니다. 이는 커밋 지점이며 쓰기 후 충돌이 발생하면 복구가 로그에서 트랜잭션의 쓰기를 재생합니다. install_trans (4772)는 로그에서 각 블록을 읽고 파일 시스템의 적절한 위치에 씁니다. 마지막으로 end_op는 카운트를 0으로 하여 로그 헤더를 씁니다. 이는 다음 트랜잭션이 로깅된 블록을 쓰기 시작하기 전에 수행되어야 하므로 충돌로 인해 한 트랜잭션의 헤더를 후속 트랜잭션의 로깅된 블록과 함께 사용하여 복구가 수행되지 않습니다.

recover_from_log (4818) 는 첫 번째 사용자 프로세스가 실행되기 전에 부팅 중에 호출되는 initlog (4756) 에서 호출됩니다 . (2865) 로그 헤더를 읽고 헤더가 로그에 커밋된 트랜잭션이 포함되어 있음을 나타내는 경우 end_op의 동작을 모방합니다.

로그의 사용 예는 filewrite (6002)에서 발생합니다. 트랜잭션은 다음과 같습니다.

```
시작_op();
ilock(f->ip); r =
 writei(f->ip, ...); iunlock(f->ip);
end_op();
```

writei+code
balloc+code
bfree+code
inode
struct dinode+code

이 코드는 큰 쓰기를 한 번에 몇 개의 섹터에 불과한 개별 트랜잭션으로 나누는 루프로 감싸져 있어 로그가 오버플로되는 것을 방지합니다. writei를 호출하면 이 트랜잭션의 일부로 많은 블록을 씁니다. 파일의 inode, 하나 이상의 비트맵 블록, 일부 데이터 블록입니다.

코드: 블록 할당자

파일 및 디렉토리 내용은 디스크 블록에 저장되며, 이는 빈 풀에서 할당해야 합니다. xv6의 블록 할당자는 블록당 비트 1개로 디스크에 빈 비트맵을 유지합니다. 0비트는 해당 블록이 비어 있음을 나타내고, 1비트는 사용 중임을 나타냅니다. 프로그램 mkfs는 부트 섹터, 슈퍼블록, 로그 블록, inode 블록 및 비트맵 블록에 해당하는 비트를 설정합니다.

블록 할당자는 두 가지 기능을 제공합니다.balloc은 새 디스크 블록을 할당하고 bfree는 블록을 해제합니다.Balloc (5022) 의 balloc의 루프는 블록 0에서 시작하여 파일 시스템의 블록 수인 sb.size까지 모든 블록을 고려합니다.비트맵 비트가 0인 블록을 찾아 해제되었음을 나타냅니다.balloc이 그러한 블록을 찾으면 비트맵을 업데이트하고 블록을 반환합니다.효율성을 위해 루프는 두 부분으로 나뉩니다.바깥쪽 루프는 각 비트맵 비트 블록을 읽습니다.안쪽 루프는 단일 비트맵 블록의 모든 BPB 비트를 확인합니다.두 프로세스가 동시에 블록을 할당하려고 하면 발생할 수 있는 경쟁은 버퍼 캐시가 한 번에 하나의 프로세스만 하나의 비트맵 블록을 사용하도록 허용한다는 사실로 방지됩니다.

Bfree (5052) 는 올바른 비트맵 블록을 찾아 올바른 비트를 지웁니다. 다시 말해서, bread와 brelse가 암시하는 강력한 사용으로 인해 명시적인 잠금이 필요 없게 됩니다.

이 장의 나머지 부분에서 설명하는 대부분의 코드와 마찬가지로 balloc 및 bfree는 트랜잭션 내부에서 호출되어야 합니다.

아이노드 층

inode 라는 용어는 두 가지 관련 의미 중 하나를 가질 수 있습니다. 파일 크기와 데이터 블록 번호 목록을 포함하는 디스크상 데이터 구조를 나타낼 수 있습니다. 또는 ''inode''는 디스크상 inode의 사본과 커널 내에서 필요한 추가 정보를 포함하는 메모리 내 inode를 나타낼 수 있습니다.

디스크 상의 inode는 inode 블록이라고 하는 디스크의 연속된 영역에 압축됩니다. 모든 inode는 크기가 같으므로 숫자 n이 주어지면 디스크에서 n번째 inode를 찾는 것이 쉽습니다. 사실, inode 번호 또는 i-번호라고 하는 이 숫자 n은 구현에서 inode를 식별하는 방법입니다.

디스크상 inode는 struct dinode (4078)에 의해 정의됩니다. 유형 필드는 파일, 디렉토리 및 특수 파일(장치)을 구별합니다. 유형 0은 디스크상 inode가 비어 있음을 나타냅니다. nlink 필드는 디스크상 inode와 해당 데이터가 언제 인식되는지 알아보기 위해 이 inode를 참조하는 디렉토리 항목의 수를 계산합니다.

구조체 inode+코드
iget+코드
iput+코드
ilock+코드

블록은 해제되어야 합니다. 크기 필드는 파일의 내용 바이트 수를 기록합니다. addrs 배열은 파일의 내용을 보관하는 디스크 블록의 블록 번호를 기록합니다.
콘텐츠.

커널은 메모리에 활성 inode 세트를 보관합니다. struct inode (4162) 는 디스크에 있는 struct dinode의 메모리 내 사본입니다. 커널은 해당 inode를 참조하는 C 포인터가 있는 경우에만 메모리에 inode를 저장합니다. ref 필드는 메모리 내 inode를 참조하는 C 포인터의 수를 계산하고, 참조 카운트가 0으로 떨어지면 커널은 메모리에서 inode를 버립니다. iget 및 iput 함수는 inode에 대한 포인터를 획득하고 해제하여 참조 카운트를 수정합니다. inode에 대한 포인터는 파일 설명자, 현재 작업 디렉토리 및 exec와 같은 일시적인 커널 코드에서 나올 수 있습니다.

xv6의 inode 코드에는 4개의 잠금 또는 잠금과 유사한 메커니즘이 있습니다. icache.lock

inode가 캐시에 최대 한 번만 존재한다는 불변성과 캐시된 inode의 ref 필드가 캐시된 inode에 대한 메모리 내 포인터의 수를 계산한다는 불변성을 보호합니다. 각 메모리 내 inode에는 sleep-lock이 포함된 lock 필드가 있으며, 이는 inode의 필드(예: 파일 길이)와 inode의 파일 또는 디렉토리 콘텐츠 블록에 대한 배타적 액세스를 보장합니다. inode의 ref가 0보다 큰 경우 시스템은 캐시에 inode를 유지하고 다른 inode에 캐시 항목을 재사용하지 않습니다. 마지막으로 각 inode에는 파일을 참조하는 디렉토리 항목의 수를 계산하는 nlink 필드(디스크에 있고 캐시된 경우 메모리에 복사됨)가 있습니다. xv6는 링크 수가 0보다 큰 경우 inode를 해제하지 않습니다.

iget()에서 반환된 struct inode 포인터는 iput()에 대한 해당 호출이 이루어질 때까지 유효한 것으로 보장됩니다. inode는 삭제되지 않으며 포인터가 참조하는 메모리는 다른 inode에 재사용되지 않습니다. iget()은 inode에 대한 비독점적 액세스를 제공하므로 동일한 inode에 대한 포인터가 여러 개 있을 수 있습니다. 파일 시스템 코드의 많은 부분이 iget()의 이러한 동작에 의존하여 inode에 대한 장기 참조(열린 파일 및 현재 디렉토리)를 보관하고 여러 inode를 조작하는 코드(예: 경로명 조회)에서 교착 상태를 피하면서 경쟁을 방지합니다.

iget이 반환하는 struct inode에는 유용한 내용이 없을 수 있습니다. 디스크에 있는 inode의 사본을 보관하기 위해 코드는 ilock을 호출해야 합니다. 이렇게 하면 inode가 잠기고(다른 프로세스가 ilock할 수 없음) 아직 읽히지 않은 경우 디스크에서 inode를 읽습니다. iunlock은 inode의 잠금을 해제합니다. inode 포인터의 획득과 잠금을 분리하면 일부 상황(예: 디렉토리 조회 중)에서 교착 상태를 피하는 데 도움이 됩니다. 여러 프로세스가 iget이 반환한 inode에 대한 C 포인터를 보관할 수 있지만 한 번에 하나의 프로세스만 inode를 잠글 수 있습니다.

inode 캐시는 커널 코드나 데이터 구조가 C 포인터를 보유한 inode만 캐시합니다. 주요 작업은 실제로 여러 프로세스의 액세스를 동기화하는 것입니다. 캐싱은 보조적입니다. inode가 자주 사용되는 경우 inode 캐시에 보관되지 않으면 버퍼 캐시가 메모리에 보관할 가능성이 높습니다. inode 캐시는 write-through이므로 캐시된 inode를 수정하는 코드는 iupdate를 사용하여 즉시 디스크에 써야 합니다.

코드: Inodes

ialloc+코드

새로운 inode를 할당하기 위해(예를 들어, 파일을 생성할 때) xv6은 ialloc (5204)을 호출합니다. Ialloc은 balloc과 비슷합니다. 디스크의 inode 구조를 한 번에 한 iget+code 블록씩 반복하면서 free로 표시된 것을 찾습니다. 하나를 찾으면 iget+code ilock+code를 통해 디스크에 새 유형을 쓰고 ilock+code를 사용하여 inode 캐시에서 항목을 반환합니다 (iget에 대한 tail 호출 (5218)). ialloc의 올바른 작동은 한 번에 한 프로세스만 bp에 대한 참조를 보유할 수 있다는 사실에 달려 있습니다. ialloc은 다른 프로세스가 동시에 inode를 사용할 수 있음을 보고 이를 청구하려고 하지 않는다는 것을 확신할 수 있습니다.

발록+코드

아이언락+코드

iput+코드

itrunc+코드

iput+코드

Iget (5254) 은 원하는 장치와 inode 번호가 있는 활성 항목(ip->ref > 0)을 inode 캐시에서 찾습니다. 하나를 찾으면 해당 inode에 대한 새 참조를 반환합니다 (5263-5267). iget이 스캔할 때 첫 번째 빈 슬롯 (5268-5269) 의 위치를 기록하는데 , 캐시 항목을 할당해야 하는 경우 이를 사용합니다.

코드는 메타데이터나 콘텐츠를 읽거나 쓰기 전에 ilock을 사용하여 inode를 잠가야 합니다. Ilock (5303) 은 이 목적을 위해 sleep-lock을 사용합니다. ilock이 inode에 대한 배타적 액세스 권한을 얻으면 필요한 경우 디스크(더 가능성이 높은 버퍼 캐시)에서 inode를 읽습니다.
iunlock (5331) 함수는 sleep-lock을 해제하는데, 이로 인해 sleep 상태에 있는 모든 프로세스가 깨어날 수 있습니다.

Iput (5358)은 참조 카운트 (5376) 를 감소시켜 inode에 대한 C 포인터를 해제합니다 . 이것이 마지막 참조인 경우, inode 캐시의 inode 슬롯은 이제 비어 있고 다른 inode에 재사용될 수 있습니다.

iput이 inode에 대한 C 포인터 참조가 없고 inode가 링크가 없는 것을 확인하면(디렉토리에 없음) inode와 해당 데이터 블록을 해제해야 합니다. iput은 itrunc를 호출하여 파일을 0바이트로 잘라 데이터 블록을 해제하고, inode 유형을 0(할당되지 않음)으로 설정하고, inode를 디스크에 씁니다 (5366).

iput에서 inode를 해제하는 경우의 잠금 프로토콜은 자세히 살펴볼 가치가 있습니다. 한 가지 위험은 동시 스레드가 ilock에서 이 inode를 사용하기 위해 기다리고 있을 수 있다는 것입니다(예: 파일을 읽거나 디렉토리를 나열하기 위해). 그리고 in-ode가 더 이상 할당되지 않았다는 것을 알 준비가 되어 있지 않을 수 있습니다. 이는 시스템 호출이 캐시된 inode에 대한 포인터를 가져올 방법이 없고 ip->ref 가 하나이기 때문에 발생할 수 없습니다. 해당 참조는 iput을 호출하는 스레드가 소유한 참조입니다. iput이 icache.lock 중요 섹션 외부에서 참조 카운트가 1인지 확인하는 것은 사실이지만, 그 시점에서 링크 카운트는 0으로 알려져 있으므로 어떤 스레드도 새로운 참조를 획득하려고 하지 않습니다. 또 다른 주요 위험은 ialloc에 대한 동시 호출이 iput이 해제하는 동일한 inode를 선택할 수 있다는 것입니다. 이는 iupdate가 디스크를 작성하여 inode의 유형이 0이 된 후에만 발생할 수 있습니다. 이 경쟁은 양성입니다. 할당 스레드는 inode를 읽거나 쓰기 전에 inode의 sleep-lock을 획득하기 위해 정중하게 기다릴 것이고, 이때 iput은 inode를 처리합니다. iput()은 디스크에 쓸 수 있습니다. 즉, 파일 시스템을 사용하는 모든 시스템 호출은 디스크에 쓸 수 있습니다. 시스템 호출이 파일에 대한 참조를 갖는 마지막 호출일 수 있기 때문입니다. 읽기 전용인 것처럼 보이는 read()와 같은 호출조차도 결국 iput()을 호출하게 될 수 있습니다. 즉, 파일 시스템을 사용하는 경우 읽기 전용 시스템 호출도 트랜잭션에 래핑해야 합니다.

iput()과 충돌 사이에는 까다로운 상호 작용이 있습니다. iput()은 파일의 링크 수가 0으로 떨어지면 파일을 즉시 자르지 않습니다. 일부 프로세스가 메모리의 inode에 대한 참조를 계속 유지할 수 있기 때문입니다.

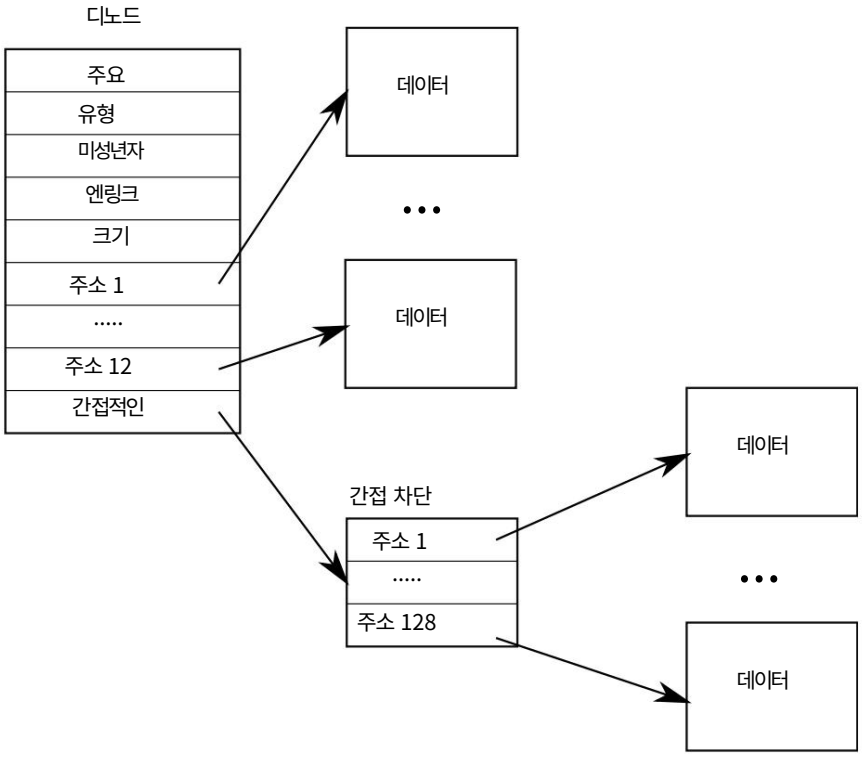


그림 6-3. 디스크에 있는 파일의 표현.

구조체 디노드+코드
NDIRECT+코드

파일을 성공적으로 열었기 때문에 파일을 읽고 쓸 수 있습니다. 하지만 마지막 프로세스가 파일의 파일 설명자를 닫기 전에 충돌이 발생하면 파일은 디스크에 할당된 것으로 표시되지만 디렉토리 항목은 이를 가리키지 않습니다.

파일 시스템은 이 경우를 두 가지 방법 중 하나로 처리합니다. 간단한 해결책은 복구 시, 재부팅 후 파일 시스템이 전체 파일 시스템을 스캔하여 할당됨으로 표시되었지만 이를 가리키는 디렉토리 항목이 없는 파일을 찾는 것입니다. 그러한 파일이 있으면 해당 파일을 해제할 수 있습니다.

두 번째 솔루션은 파일 시스템을 스캔할 필요가 없습니다. 이 솔루션에서 파일 시스템은 링크 카운트가 0으로 떨어지지만 참조 카운트가 0이 아닌 파일의 inode inumber를 디스크(예: 슈퍼 블록)에 기록합니다. 참조 카운트가 0에 도달했을 때 파일 시스템이 파일을 다시 제거하면 해당 inode를 목록에서 제거하여 디스크 목록을 업데이트합니다. 복구 시 파일 시스템은 목록에 있는 모든 파일을 해제합니다.

Xv6는 어느 솔루션도 구현하지 않으므로, inode는 더 이상 사용되지 않더라도 디스크에 할당된 것으로 표시될 수 있습니다. 즉, 시간이 지남에 따라 xv6는 디스크 공간이 부족해질 위험이 있습니다.

코드: Inode 내용

디스크상 inode 구조인 struct dinode에는 크기와 블록 번호 배열이 들어 있습니다(그림 6-3 참조). inode 데이터는 dinode의 addrs 배열에 나열된 블록에서 찾을 수 있습니다. 첫 번째 NDIRECT 데이터 블록은 첫 번째 NDIRECT에 나열됩니다.

배열의 항목; 이러한 블록을 직접 블록이라고 합니다. 다음 NINDIRECT 데이터 블록은 inode가 아니라 간접 블록이라고 하는 데이터 블록에 나열됩니다. 마지막

addrs 배열의 항목은 간접 블록의 주소를 제공합니다. 따라서 파일의 처음 6kB(NDIRECT×BSIZE) 바이트는 inode에 나열된 블록에서 로드할 수 있는 반면, 다음 64kB(NINDIRECT×BSIZE) 바이트는 간접 블록을 참조한 후에만 로드할 수 있습니다. 이는 디스크 상에서 좋은 표현이지만 클라이언트에게는 복잡한 표현입니다. bmap 함수는 곧 살펴볼 readi 및 writei와 같은 상위 수준 루틴이 표현을 관리하도록 합니다. Bmap은 inode ip에 대한 bn번째 데이터 블록의 디스크 블록 번호를 반환합니다. ip에 아직 그러한 블록이 없으면 bmap은 다음을 할당합니다.

직접 블록
NINDIRECT+코드 간접 블록
BSIZE+코드
bmap+코드
readi+코드
writei+code
비맵+코드
NDIRECT+코드
NINDIRECT+코드
itrunc+코드
읽기+코드 쓰기+코드 쓰기+코드 읽기+코드 쓰기+코드

하나.

bmap (5410) 함수는 쉬운 경우를 골라내는 것으로 시작합니다. 첫 번째 NDIRECT 블록은 inode 자체에 나열됩니다 (5415-5419). 다음 NINDIRECT 블록은 ip->addrs[NDIRECT]의 간접 블록에 나열됩니다. Bmap은 간접 블록 (5426) 을 읽은 다음 블록 내의 올바른 위치에서 블록 번호를 읽습니다 (5427). 블록 번호가 NDIRECT+NINDIRECT를 초과하면 bmap이 패닉합니다. writei에는 이런 일이 발생하지 않도록 하는 검사가 포함되어 있습니다 (5566).

readi+code
writei+code
T_DEV+코드 통계+코드 통계+코드
T_DIR+코드 구조체 dirent+코드
DIRSIZ+코드
dirlookup+코드

Bmap은 필요에 따라 블록을 할당합니다. ip->addrs[] 또는 간접 항목 0은 블록이 할당되지 않았음을 나타냅니다. Bmap이 0을 만나면 필요에 따라 할당된 새 블록의 수로 대체합니다 (5416-5417, 5424-5425). itrunc는 파일의 블록을 해제하고 inode의 크기를 0으로 재설정합니다. Itrunc (5456) 는 직접 블록 (5462-5467) 을 해제하는 것으로 시작한 다음 간접 블록 (5472-5475) 에 나열된 블록을 해제 하고 마지막으로 간접 블록 자

체 (5477-5478)를 해제합니다.

Bmap은 readi와 writei가 inode의 데이터를 쉽게 얻을 수 있도록 해줍니다. Readi (5503) 는 오프셋과 카운트가 파일 끝을 넘지 않도록 하는 것으로 시작합니다.

파일 끝을 넘어서 시작하는 읽기는 오류를 반환하고 (5514-5515), 파일 끝에서 시작하거나 끝을 넘는 읽기는 요청한 것보다 적은 바이트를 반환합니다 (5516-5517). 메인 루프는 파일의 각 블록을 처리하여 버퍼에서 dst로 데이터를 복사합니다 (5519-5524). writei (5553)는 readi와 동일하지만 세 가지 예외가 있습니다. 파일 끝에서 시작하거나 끝을 넘는 쓰기는 최대 파일 크기까지 파일을 늘립니다 (5566-5567). 루프는 데이터를 out이 아닌 버퍼에 복사합니다 (5572). 쓰기가 파일을 확장한 경우 writei는 크기를 업데이트해야 합니다 (5577-5580).

readi와 writei는 모두 ip->type == T_DEV를 확인하는 것으로 시작합니다. 이 케이스는 데이터가 파일 시스템에 없는 특수 장치를 처리합니다. 파일 설명자 계층에서 이 케이스로 돌아갑니다.

stati (5488) 함수는 inode 메타데이터를 stat 구조에 복사하고, 이 구조는 stat 시스템 호출을 통해 사용자 프로그램에 노출됩니다.

코드: 디렉토리 계층

디렉토리는 파일과 매우 비슷하게 내부적으로 구현됩니다. inode는 T_DIR 유형이고 데이터는 디렉토리 항목 시퀀스입니다. 각 항목은 이름과 inode 번호를 포함하는 struct dirent (4115) 입니다 . 이름은 최대 DIRSIZ(14) 문자입니다. 이보다 짧으면 NUL(0) 바이트로 끝납니다. inode 번호가 0인 디렉토리 항목은 비어 있습니다.

dirlookup (5611) 함수는 주어진 항목을 포함하는 디렉토리를 검색합니다.

name. 하나를 찾으면 해당 inode에 대한 포인터를 잠금 해제된 상태로 반환하고, 호출자가 편집하려는 경우 디렉토리 내 항목의 바이트 오프셋으로 *poff를 설정합니다. dirlookup이 올바른 이름의 항목을 찾으면 *poff를 업데이트하고 블록을 해제하고 iget을 통해 얻은 잠금 해제된 inode를 반환합니다. Dirlookup은 iget이 잠금 해제된 inode를 반환하는 이유입니다. 호출자가 dp를 잠갔으므로 조회가 현재 디렉토리의 별칭을 위한 것이라면 skipelem+code ilock+code를 반환하기 전에 inode를 잠그려고 하면 dp를 다시 잠그고 교착 상태가 됩니다. (여러 프로세스와 부모 디렉토리의 별칭을 포함하는 더 복잡한 교착 상태 시나리오인 nameiparent+code ios 가 있습니다 . .은 namex+code의 유일한 문제가 아닙니다.) 호출자는 dp를 잠금 해제한 다음 ip를 잠가 한 번에 하나의 잠금만 유지하도록 할 수 있습니다.

iget+코드 .+코드 ..+코드 dirlink+코드 dirlookup+코드 이름iparent+코드 이름x+코드

.,

..,

dirlookup+코드

함수 dirlink (5652) 는 주어진 이름과 inode 번호로 디렉토리 dp에 새 디렉토리 항목을 씁니다. 이름이 이미 있으면 dirlink는 오류를 반환합니다 (5658-5662). 메인 루프는 할당되지 않은 항목을 찾아 디렉토리 항목을 읽습니다. 항목을 찾으면 루프를 일찍 중지합니다 (5622-5623). off는 사용 가능한 항목의 오프셋으로 설정됩니다. 그렇지 않으면 루프는 off를 dp->size로 설정하여 끝납니다. 어느 쪽이든 dirlink는 오프셋 off(5672-5675)에 기록하여 디렉토리에 새 항목을 추가합니다 .

코드: 경로 이름

경로 이름 조회에는 각 경로 구성 요소마다 하나씩 dirlookup에 대한 호출이 연속적으로 포함됩니다. Namei (5790)는 path를 평가하고 해당 inode를 반환합니다. 함수 nameiparent는 변형입니다. 마지막 요소 앞에서 멈추고 부모 디렉터리의 inode를 반환하고 마지막 요소를 name에 복사합니다. 둘 다 일반화된 함수 namex를 호출하여 실제 작업을 수행합니다.

Namex (5755) 는 경로 평가가 시작되는 위치를 결정하는 것으로 시작합니다. 경로가 슬래시로 시작하면 평가는 루트에서 시작합니다. 그렇지 않으면 현재 디렉토리에서 시작합니다 (5759-5762). 그런 다음 skipelem을 사용하여 경로의 각 요소를 차례로 고려합니다 (5764). 루프의 각 반복은 현재 inode ip에서 name을 조회해야 합니다. 반복은 ip를 잠그고 디렉토리인지 확인하는 것으로 시작합니다. 그렇지 않으면 조회가 실패합니다 (5765-5769).

(ip 잠금이 필요한 이유는 ip->type이 변경될 수 있기 때문이 아니라(변경될 수 없음) ilock이 실행되기 전까지 ip->type이 디스크에서 로드되었는지 보장할 수 없기 때문입니다.)
호출이 nameiparent이고 이것이 마지막 경로 요소인 경우, nameiparent의 정의에 따라 루프가 일찍 멈춥니다. 마지막 경로 요소는 이미 name에 복사되었으므로 namex는 잠금 해제된 ip만 반환하면 됩니다 (5770-5774). 마지막으로, 루프는 dirlookup을 사용하여 경로 요소를 찾고 ip = next (5775-5780)를 설정하여 다음 반복을 준비합니다. 루프가 경로 요소를 모두 실행하면 ip를 반환합니다.

namex 프로시저는 완료하는 데 오랜 시간이 걸릴 수 있습니다. 경로명에서 탐색한 디렉토리의 inode와 디렉토리 블록을 읽기 위해 여러 디스크 작업이 필요할 수 있습니다(버퍼 캐시에 없는 경우). Xv6는 한 커널 스레드에서 namex를 호출하는 것이 디스크 I/O에서 차단된 경우 다른 경로명을 찾는 다른 커널 스레드가 동시에 진행될 수 있도록 신중하게 설계되었습니다. namex는 경로의 각 디렉토리를 별도로 잠그므로 다른 디렉토리에서 조회가 병렬로 진행될 수 있습니다.

이 동시성은 몇 가지 과제를 야기합니다. 예를 들어, 한 커널 스레드가 경로명을 찾는 동안 다른 커널 스레드는 디렉토리를 변경할 수 있습니다.

디렉토리를 연결 해제하여 트리를 만듭니다. 잠재적인 위험은 조회가 다른 커널 스레드에 의해 삭제된 디렉토리를 검색하고 해당 블록이 다른 디렉토리나 파일에 재사용될 수 있다는 것입니다.

Xv6는 이러한 경쟁을 피합니다. 예를 들어, namex에서 dirlookup을 실행할 때, lookup 스레드는 디렉토리에 대한 잠금을 유지하고 dirlookup은 iget을 사용하여 얻은 inode를 반환합니다. iget은 inode의 참조 카운트를 증가시킵니다. dirlookup에서 inode를 수신한 후에야 namex는 디렉토리에 대한 잠금을 해제합니다.

이제 다른 스레드가 디렉토리에서 inode의 연결을 해제할 수 있지만 xv6는 아직 inode를 삭제하지 않습니다. inode의 참조 카운트가 여전히 0보다 크기 때문입니다.

또 다른 위험은 교착 상태입니다. 예를 들어, next는 "."을 조회할 때 ip와 동일한 inode를 가리킵니다. ip에 대한 잠금을 해제하기 전에 next를 잠그면 교착 상태가 발생합니다. 이 교착 상태를 피하기 위해 namex는 next에 대한 잠금을 얻기 전에 디렉토리를 잠금 해제합니다. 여기서도 iget과 ilock을 분리하는 것이 중요한 이유를 알 수 있습니다.

파일 설명자 계층

Unix 인터페이스의 멋진 측면은 Unix의 대부분 리소스가 콘솔, 파이프, 물론 실제 파일과 같은 장치를 포함하여 파일로 표현된다는 것입니다. 파일 설명자 계층은 이러한 균일성을 달성하는 계층입니다.

Xv6는 0장에서 살펴본 것처럼 각 프로세스에 자체적인 열린 파일 또는 파일 설명자 테이블을 제공합니다. 각 열린 파일은 inode 또는 파이프를 감싸는 래퍼인 구조체 파일 (4150) 과 i/o 오프셋으로 표현됩니다. open에 대한 각 호출은 새 열린 파일(새 구조체 파일)을 만듭니다. 여러 프로세스가 동일한 파일을 독립적으로 여는 경우 다른 인스턴스는 서로 다른 i/o 오프셋을 갖습니다. 반면, 단일 열린 파일(동일한 구조체 파일)은 한 프로세스의 파일 테이블과 여러 프로세스의 파일 테이블에 여러 번 나타날 수 있습니다. 이는 한 프로세스가 open을 사용하여 파일을 연 다음 dup를 사용하여 별칭을 만들거나 fork를 사용하여 자식과 공유하는 경우에 발생합니다. 참조 카운트는 특정 열린 파일에 대한 참조 수를 추적합니다. 파일은 읽기 또는 쓰기 또는 둘 다를 위해 열릴 수 있습니다. readable 및 writable 필드는 이를 추적합니다.

시스템의 모든 열린 파일은 전역 파일 테이블인 ftable에 보관됩니다. 파일 테이블에는 파일을 할당(filealloc), 중복 참조 생성(filedup), 참조 해제(fileclose), 데이터 읽기 및 쓰기(fileread 및 filewrite) 기능이 있습니다.

처음 세 가지는 이제 익숙한 형식을 따릅니다. Filealloc (5876) 는 참조되지 않은 파일(f->ref == 0)에 대한 파일 테이블을 스캔하고 새 참조를 반환합니다. filedup (5902) 는 참조 카운트를 증가시키고 fileclose (5914) 는 감소시킵니다. 파일의 참조 카운트가 0에 도달하면 fileclose는 유형에 따라 기본 파이프나 inode를 해제합니다.

filestat, fileread 및 filewrite 함수는 파일에 대한 stat, read 및 write 작업을 구현합니다. Filestat (5952) 는 inode에서만 허용되며 stati를 호출합니다. Fileread 및 filewrite는 작업이 open 모드에서 허용되는지 확인한 다음 호출을 파이프 또는 inode 구현으로 전달합니다. 파일이 inode를 나타내는 경우 fileread 및 filewrite는 i/o 오프셋을 작업의 오프셋으로 사용한 다음 이를 진행합니다 (5975-5976, 6015-6016). 파이프에는 오프 개념이 없습니다.

트. inode 함수는 호출자가 잠금을 처리해야 한다는 점을 기억하세요 (5955-5957, 5974- sys_unlink+code 5977, 6025-6028). inode 잠금은 읽기 및 쓰기 nameiparent+code sys_link+code 세
된다는 편리한 부작용이 있어 , 같은 파일에 동시에 여러 번 쓰는 sys_link+code create+code는 서로의 데이터를 덮어쓸 수 없지만, 쓰기가 엇갈릴 수 있습니다. open+code 오프셋이 원자적으로 업데이트
mkdev+code sys_link+code create+code O_CREATE+code mkdir+code

코드: 시스템 호출

하위 계층이 제공하는 기능을 통해 대부분의 시스템을 구현할 수 있습니다.
tem 호출은 사소한 일입니다(sysfile.c 참조). 자세히 살펴볼 만한 호출이 몇 가지 있습니다. nameiparent+code
dirlookup+code

sys_link와 sys_unlink 함수는 디렉토리를 편집하여 inode에 대한 mkdir+code 참조를 만들거나 제거합니다. 이 함수는 transac- mkdev+code T_FILE+code tion을 사용하는 것의 힘을 보
여주는 또 다른 좋은 예입니다. Sys_link (6202) 는 인수인 old와 new (6207)라는 두 문자열을 가져오는 것으로 시작합니다 . ialloc+code old 가 존재하고 디렉토리가 아니라고 가정하면 (6211-6214),
sys_link는 ip- .
+code >nlink 카운트를 증가시킵니다. 그런 다음 sys_link는 nameiparent를 호출하여 부모 디렉토리와 new (6227) 의 최종 경로 요소를 찾고 old의 in- sys_link+code ode (6230) 를 가리키는 새 디
렉토리 항목을 만듭니다 . 영어: 새 부모 디렉토리는 존재해야 하며 이전 sys_open+code inode와 동일한 장치에 있어야 합니다. inode 번호는 단일 디스크에서만 고유한 의미를 갖습니다. 이와 같은
오류 sys_mkdir+code sys_mknod+code가 발생하면 sys_link는 뒤로 돌아가서 ip->nlink를 감소시켜야 합니다. open+code 트랜잭션은 여러 O_CREATE+code 디스크 블록을 업데이트해야
하기 때문에 구현
을 간소화 하지만, 이를 수행하는 순서에 대해서는 걱정할 필요가 없습니다. 모두 성공하거나 아무것도 성공하지 못할 것입니다. 예를 들어, 트랜잭션이 없으면 링크를 생성하기 전에 ip- >nlink를 업데이트하면 파일
시스템이 일시적으로 안전하지 않은 상태가 되고 그 사이에 충돌이 발생하면 엄청난 혼란이 발생할 수 있습니다. 트랜잭션이 있으면 이에 대해 걱정할 필요가 없습니다.

sys_open+코드

Sys_link는 기존 inode에 대한 새 이름을 만듭니다. create (6357) 함수는 새 inode에 대한 새 이름을 만듭니다. 이는 세 가지 파일 생성 시스템 호출을 일반화한 것입니다. O_CREATE 플래그가 있는 open은 새 일반 파일을 만들고, mkdir은 새 디렉토리를 만들고, mkdev는 새 장치 파일을 만듭니다. sys_link와 마찬가지로 create는 nameiparent를 호출하여 부모 디렉토리의 inode를 가져오는 것으로 시작합니다. 그런 다음 dirlookup을 호출하여 이름이 이미 있는지 확인합니다 (6367). 이름이 있으면 create의 동작은 사용되는 시스템 호출에 따라 달라집니다. open은 mkdir 및 mkdev와 다른 의미를 갖습니다. create가 open(type == T_FILE) 대신 사용되고 존재하는 이름 자체가 일반 파일인 경우 open은 이를 성공으로 처리하므로 create도 마찬가지입니다 (6371). 그렇지 않으면 오류입니다 (6372-6373). 이름이 이미 존재하지 않으면 create는 지금 ialloc (6376)으로 새 inode를 할당합니다. 새 inode가 디렉토리인 경우 create는 . 및 .. 항목으로 초기화합니다. 마지막으로 이제 데이터가 제대로 초기화되었으므로 create는 부모 디렉토리에 연결할 수 있습니다 (6389). sys_link와 마찬가지로 create는 ip와 dp라는 두 개의 inode 잠금을 동시에 보유합니다. inode ip가 새로 할당되었기 때문에 교착 상태가 발생할 가능성은 없습니다. 시스템의 다른 프로세스는 ip의 잠금을 유지한 다음 dp를 잠그려고 하지 않습니다.

create를 사용하면 sys_open, sys_mkdir, sys_mknod를 쉽게 구현할 수 있습니다.
Sys_open (6401) 은 가장 복잡합니다. 새 파일을 만드는 것은 할 수 있는 일의 일부에 불과하기 때문입니다. open에 O_CREATE 플래그가 전달되면 create (6414)를 호출합니다. 그렇지 않으면 namei (6420)를 호출합니다. Create는 잠긴 inode를 반환하지만 namei는 반환하지 않으므로 sys_open은 inode 자체를 잠가야 합니다. 이는 디렉토리가

sys_pipe+코드
fsck+코드

읽기 전용으로 열리고 쓰기 전용은 열리지 않습니다. inode가 한 가지 방법으로 얻어졌다고 가정합니다.
또는 다른 경우 sys_open은 파일과 파일 설명자 (6432) 를 할당한 다음 채웁니다.
파일 (6442-6446). 다른 프로세스는 부분적으로 초기화된 파일에 액세스할 수 없습니다.
현재 프로세스의 테이블에만 존재합니다.

5장에서는 파일 시스템이 생기기도 전에 파이프 구현을 살펴보았습니다. sys_pipe 함수는 파이프
쌍을 만드는 방법을 제공하여 해당 구현을 파일 시스템에 연결합니다. 인수는 두 정수에 대한 공간을 가리
키는 포인터입니다.
두 개의 새로운 파일 디스크립터를 기록할 곳입니다. 그런 다음 파이프를 할당하고 파일 디스크립터를 설치
합니다.

현실 세계

실제 운영 체제의 버퍼 캐시는 훨씬 더 복잡합니다.
xv6보다 뛰어나지만 캐싱과 액세스 동기화라는 두 가지 목적을 제공합니다.
디스크. Xv6의 버퍼 캐시는 V6와 마찬가지로 간단한 LRU(최근에 가장 적게 사용된 항목) 제거를 사용합니다.
정책; 구현할 수 있는 훨씬 더 복잡한 정책이 많이 있으며 각각은 다음과 같은 이점이 있습니다.
일부 작업 부하에는 적합하지만 다른 작업 부하에는 적합하지 않습니다. 더 효율적인 LRU 캐시는 연결 목
록을 제거하고 대신 조회에는 해시 테이블을 사용하고 LRU 제거에는 힙을 사용합니다. 최신 버퍼 캐시는 일
반적으로 가상 메모리 시스템과 통합됩니다.
메모리 맵 파일을 지원합니다.

Xv6의 로깅 시스템은 비효율적입니다. 커밋은 파일과 동시에 발생할 수 없습니다.
시스템 시스템 호출. 시스템은 블록에 몇 바이트만 있어도 전체 블록을 기록합니다.
변경됩니다. 한 번에 한 블록씩 동기 로그 쓰기를 수행하며 각각은 다음과 같습니다.
전체 디스크 회전 시간이 필요할 가능성이 있습니다. 실제 로깅 시스템은 이러한 모든 문제를 해결합니다.
문제.

로깅은 충돌 복구를 제공하는 유일한 방법이 아닙니다. 초기 파일 시스템은
재부팅 중에 모든 파일을 검사하기 위해 scavenger를 사용합니다(예: UNIX fsck 프로그램)
그리고 디렉토리와 블록 및 inode 자유 목록, 불일치를 찾고 해결합니다. 청소는 대용량 파일 시스템의 경
우 몇 시간이 걸릴 수 있으며,
원래 시스템 호출을 발생시키는 방식으로 불일치를 해결하는 것은 불가능합니다.
원자적이 되어야 합니다. 로그에서 복구하는 것이 훨씬 빠르고 시스템 호출이 원자적이 되게 합니다.
충돌에 직면해서.

Xv6는 초기 UNIX와 동일한 기본적인 디스크상 inode 및 디렉토리 레이아웃을 사용합니다.
이 계획은 수년에 걸쳐 놀라울 정도로 지속되어 왔습니다. BSD의 UFS/FFS와 Linux의
ext2/ext3는 본질적으로 동일한 데이터 구조를 사용합니다. 파일의 가장 비효율적인 부분
시스템 레이아웃은 디렉토리이며, 각 조회 중에 모든 디스크 블록에 대한 선형 스캔이 필요합니다. 디렉토리
가 몇 개의 디스크 블록에 불과할 때는 합리적이지만,
많은 파일을 보관하는 디렉토리에는 비쌉니다. Microsoft Windows의 NTFS, Mac OS X의
HFS와 Solaris의 ZFS는 몇 가지 예를 들자면, 디렉토리를 디스크에 균형 잡힌 블록 트리로 구현합니다.
이는 복잡하지만 대수 시간 디렉토리를 보장합니다.
조회.

Xv6는 디스크 오류에 대해 순진합니다. 디스크 작업이 실패하면 xv6가 패닉합니다. 이것이
합리적이라는 것은 하드웨어에 따라 달라집니다. 운영 체제가 디스크 오류를 가리기 위해 중복성을 사용하
는 특수 하드웨어 위에 있는 경우 운영 체제가 다음을 볼 수 있습니다.
실패가 너무 드물어서 당황하는 것도 괜찮습니다. 반면에 운영 체제

파일 읽기+코드
파일 쓰기 + 코드

일반 디스크를 사용하면 오류가 발생할 수 있음을 예상하고 오류를 더 우아하게 처리해야 하므로 한 파일의 블록이 손실되어도 나머지 파일 시스템의 사용에는 영향을 미치지 않습니다.

Xv6은 파일 시스템이 하나의 디스크 장치에 맞아야 하며 크기가 변경되지 않아야 합니다. 대용량 데이터베이스와 멀티미디어 파일로 인해 스토리지 요구 사항이 점점 더 높아지면서 운영 체제는 ''파일 시스템당 하나의 디스크'' 병목 현상을 제거하는 방법을 개발하고 있습니다. 기본적인 접근 방식은 여러 디스크를 단일 논리 디스크로 결합하는 것입니다. RAID와 같은 하드웨어 솔루션이 여전히 가장 인기가 있지만, 현재 추세는 가능한 한 이 논리를 소프트웨어에 구현하는 방향으로 이동하고 있습니다. 이러한 소프트웨어 구현은 일반적으로 디스크를 즉석에서 추가하거나 제거하여 논리 장치를 확장하거나 축소하는 것과 같은 풍부한 기능을 허용합니다. 물론 즉석에서 확장하거나 축소할 수 있는 스토리지 계층에는 동일한 작업을 수행할 수 있는 파일 시스템이 필요합니다. xv6에서 사용하는 고정 크기의 in-ode 블록 배열은 이러한 환경에서는 제대로 작동하지 않습니다. 디스크 관리를 파일 시스템에서 분리하는 것이 가장 깔끔한 설계일 수 있지만, 두 가지 간의 복잡한 인터페이스로 인해 Sun의 ZFS와 같은 일부 시스템은 두 가지를 결합하게 되었습니다.

Xv6의 파일 시스템은 최신 파일 시스템의 여러 기능이 부족합니다. 예를 들어, 스냅샷과 증분 백업을 지원하지 않습니다.

최신 Unix 시스템은 디스크 저장소와 동일한 시스템 호출을 통해 많은 종류의 리소스에 액세스할 수 있도록 합니다. 명명된 파이프, 네트워크 연결, 원격으로 액세스하는 네트워크 파일 시스템, /proc와 같은 모니터링 및 제어 인터페이스가 있습니다. fileread 및 filewrite의 xv6의 if 문 대신, 이러한 시스템은 일반적으로 열려 있는 각 파일에 작업당 하나씩 함수 포인터 테이블을 제공하고 함수 포인터를 호출하여 해당 inode의 호출 구현을 호출합니다. 네트워크 파일 시스템과 사용자 수준 파일 시스템은 이러한 호출을 네트워크 RPC로 전환하고 반환하기 전에 응답을 기다리는 함수를 제공합니다.

수업 과정

1. 왜 볼록에서 패닉을 하는가? xv6가 회복할 수 있을까?

2. ialloc에서 패닉이 발생하는 이유는? xv6가 복구할 수 있나요?

3. filealloc이 파일이 부족할 때 패닉을 일으키지 않는 이유는 무엇입니까? 왜 이것이 더 일반적이어서 다룰 가치가 있는가?

4. sys_link의 iunlock(ip) 및 dirlink 호출 사이에 다른 프로세스가 ip에 해당하는 파일을 연결 해제한다고 가정합니다. 링크가 올바르게 생성될까요?
왜 또는 왜 안 되는가?

6. create는 성공하는 데 필요한 네 개의 함수 호출(하나는 ialloc에, 세 개는 dirlink에)을 합니다. 성공하지 못하면 create는 패닉을 호출합니다. 이게 왜 허용되는 걸까요?
왜 그 네 가지 호출은 실패할 수 없는 걸까요?

7. sys_chdir은 iput(cp->cwd)보다 먼저 iunlock(ip)를 호출하여 cp->cwd를 잠그려고 시도할 수 있지만, iunlock(ip)를 iput 이후로 연기해도 교착 상태가 발생하지 않습니다.
왜 안 돼?

8. lseek 시스템 호출을 구현합니다. lseek를 지원하려면 lseek가 f->ip- >size를 넘어선 경우 파일의 구멍을 0으로 채우도록 filewrite를 수정해야 합니다.

9. O_TRUNC와 O_APPEND를 추가하여 >와 >> 연산자가 셸에서 작동하도록 합니다.

10. 심볼릭 링크를 지원하도록 파일 시스템을 수정합니다.

7장

요약

이 텍스트는 한 운영 체제인 xv6를 줄마다 연구하여 운영 체제의 주요 아이디어를 소개했습니다. 일부 코드 줄은 주요 아이디어의 본질을 구현하고 있습니다(예: 컨텍스트 전환, 사용자/커널 경계, 잠금 등). 각 줄이 중요합니다. 다른 코드 줄은 특정 운영 체제 아이디어를 구현하는 방법을 보여주고 다양한 방식으로 쉽게 구현할 수 있습니다 (예: 스케줄링을 위한 더 나은 알고리즘, 파일을 나타내는 더 나은 디스크 데이터 구조, 동시 트랜잭션을 허용하는 더 나은 로깅 등). 모든 아이디어는 특정하고 매우 성공적인 시스템 호출 인터페이스인 Unix 인터페이스의 맥락에서 설명되었지만, 이러한 아이디어는 다른 운영 체제의 설계로 이어집니다.

부록 A

PC 하드웨어

이 부록에서는 개인용 컴퓨터(PC) 하드웨어와 그 플랫폼을 설명합니다.
xv6가 실행됩니다.

PC는 여러 산업 표준을 준수하는 컴퓨터로, 주어진 소프트웨어가 여러 공급업체가 판매하는 PC에서 실행될 수 있도록 하는 것이 목표입니다. 이러한 표준은 시간이 지남에 따라 진화하며 1990년대의 PC는 지금의 PC처럼 보이지 않습니다. 현재 표준의 대부분은 공개되어 있으며 온라인에서 해당 표준에 대한 문서를 찾을 수 있습니다.

외부에서 보면 PC는 키보드, 화면, 다양한 장치(예: CD-ROM 등)가 있는 상자입니다. 상자 내부에는 CPU 칩, 메모리 칩, 그래픽 칩, I/O 컨트롤러 칩, 칩이 통신하는 버스가 있는 회로 기판(''마더보드'')이 있습니다. 버스는 표준 프로토콜(예: PCI 및 USB)을 준수하므로 장치가 여러 공급업체의 PC에서 작동합니다.

우리의 관점에서, 우리는 PC를 CPU, 메모리, 입출력(I/O) 장치라는 세 가지 구성 요소로 추상화할 수 있습니다. CPU는 계산을 수행하고, 메모리는 해당 계산을 위한 명령어와 데이터를 담고 있으며, 장치는 CPU가 저장, 통신 및 기타 기능을 위해 하드웨어와 상호 작용할 수 있도록 합니다.

메인 메모리는 와이어 또는 회선으로 CPU에 연결되어 있다고 생각할 수 있습니다. 일부는 주소 비트, 일부는 데이터 비트, 일부는 제어 플래그입니다. 메인 메모리에서 값을 읽으려면 CPU는 주소 회선에 1 또는 0 비트를 나타내는 높거나 낮은 전압을 보내고, ''읽기'' 회선에 1을 지정된 시간 동안 보낸 다음, 데이터 회선의 전압을 해석하여 값을 다시 읽습니다. 메인 메모리에 값을 쓰려면 CPU는 주소 및 데이터 회선에 적절한 비트를 보내고, ''쓰기'' 회선에 1을 지정된 시간 동안 보냅니다. 실제 메모리 인터페이스는 이보다 더 복잡하지만, 세부 사항은 고성능을 달성해야 하는 경우에만 중요합니다.

프로세서와 메모리

컴퓨터의 CPU(중앙 처리 장치 또는 프로세서)는 개념적으로 간단한 루프를 실행합니다. 프로그램 카운터라는 레지스터의 주소를 참조하고, 메모리의 해당 주소에서 기계 명령어를 읽고, 프로그램 카운터를 명령어 뒤로 진행하여 명령어를 실행합니다. 반복합니다. 명령어 실행이 프로그램 카운터를 수정하지 않으면 이 루프는 프로그램 카운터가 가리키는 메모리를 차례로 실행하는 기계 명령어의 시퀀스로 해석합니다. 프로그램 카운터를 변경하는 명령어에는 분기 및 함수 호출이 포함됩니다.

실행 엔진은 프로그램 데이터를 저장하고 수정할 수 있는 기능 없이는 쓸모가 없습니다. 가장 빠른 데이터 저장은 프로세서의 레지스터 세트에서 제공됩니다. 레지스터는 프로세서 자체 내부의 저장 셀로, 기계 워드 크기의 데이터를 저장할 수 있습니다.

값(일반적으로 16, 32 또는 64비트). 레지스터에 저장된 데이터는 일반적으로 읽거나
단일 CPU 사이클로 빠르게 작성되었습니다.

PC에는 x86 명령어 세트를 구현하는 프로세서가 있는데, 이는 원래 인텔에서 정의했으며 표준이 되었습니다. 여러 제조업체에서 명령어 세트를 구현하는 프로세서를 생산합니다. 다른 모든 PC 표준과 마찬가지로 이 표준은
또한 진화하고 있지만 최신 표준은 과거 표준과 역호환됩니다.
부트로더는 모든 PC 프로세서가 시작되기 때문에 이러한 진화 중 일부를 처리해야 합니다.
1981년 출시된 최초의 IBM PC에 사용된 CPU 칩인 Intel 8088을 시뮬레이션한 것입니다.
하지만 xv6의 대부분에서는 최신 x86 명령어 세트와 관련이 있을 것입니다.

최신 x86은 %eax, %ebx 등 8개의 범용 32비트 레지스터를 제공합니다.
%ecx, %edx, %edi, %esi, %ebp, %esp—그리고 프로그램 카운터 %eip(명령 포인터). 일반적인 e
접두사는 확장을 의미하는데, 이는 32비트 확장이기 때문입니다.
16비트 레지스터 %ax, %bx, %cx, %dx, %di, %si, %bp, %sp 및 %ip. 두 레지스터 세트는 별칭이
지정되어 예를 들어 %ax가 %eax의 하단 절반이 됩니다. %ax에 쓰기
%eax에 저장된 값을 변경하고 그 반대도 마찬가지입니다. 처음 네 개의 레지스터에도 다음이 있습니다.
하단 두 개의 8비트 바이트의 이름: %al 및 %ah는 하위 8비트와 상위 8비트를 나타냅니다.
%ax; %bl, %bh, %cl, %ch, %dl 및 %dh는 패턴을 계속합니다. 이러한 레지스터 외에도 x86에는 8개
의 80비트 부동 소수점 레지스터와 제어 레지스터 % cr0, %cr2, %cr3 및 %cr4와 같은 소수의 특수 용
도 레지스터, 디버그 레지스터 %dr0, %dr1, %dr2 및 %dr3, 세그먼트 레지스터 %cs, %ds, %es,
%fs, %gs 및
%ss; 그리고 글로벌 및 로컬 설명자 테이블 의사 레지스터 %gdtr 및 %ldtr.
제어 레지스터와 세그먼트 레지스터는 모든 운영 체제에 중요합니다.
부동 소수점 및 디버그 레지스터는 덜 흥미롭고 xv6에서는 사용되지 않습니다.

레지스터는 빠르지만 비쌉니다. 대부분의 프로세서는 최대 수십 개를 제공합니다.
일반 용도 레지스터. 다음 개념적 수준의 저장소는 주 랜덤 액세스 메모리(RAM)입니다. 주 메모리는 레지스
터보다 10~100배 느리지만 훨씬
더 저렴하기 때문에 더 많이 가질 수 있습니다. 주 메모리가 비교적 느린 이유 중 하나는
프로세서 칩과 물리적으로 분리되어 있습니다. x86 프로세서에는 수십 개의
레지스터이지만 오늘날 일반적인 PC에는 기가바이트의 주 메모리가 있습니다. 레지스터와 주 메모리 간의
액세스 속도와 크기 모두에서 엄청난 차이가 있기 때문에,
x86을 포함한 대부분의 프로세서는 최근 액세스한 메인 섹션의 사본을 저장합니다.
온칩 캐시 메모리의 메모리. 캐시 메모리는 액세스 시간과 크기 면에서 레지스터와 메모리의 중간 지점 역할
을 합니다. 오늘날의 x86 프로세서
일반적으로 3단계의 캐시가 있습니다. 각 코어에는 액세스가 가능한 작은 1단계 캐시가 있습니다.
프로세서의 클럭 속도와 비교적 가까운 시간과 더 큰 2차 캐시. 여러 코어가 L3 캐시를 공유합니다. 그림
A-1은 메모리 계층의 레벨을 보여줍니다.
Intel i7 Xeon 프로세서의 액세스 시간입니다.

대부분의 경우 x86 프로세서는 운영 체제에서 캐시를 숨기므로
프로세서가 레지스터와 메모리라는 두 가지 종류의 저장소만 가지고 있다고 생각할 수 있으며 메모리의 여
러 수준 간의 구별에 대해 걱정할 필요가 없습니다.
계층.

입출력

프로세서는 메모리뿐만 아니라 장치와도 통신해야 합니다. x86 프로세서

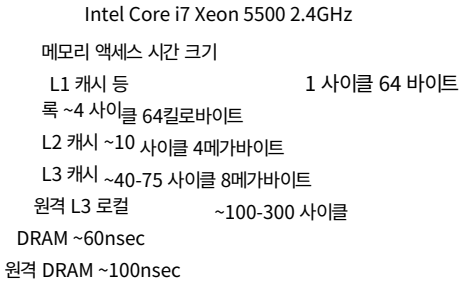


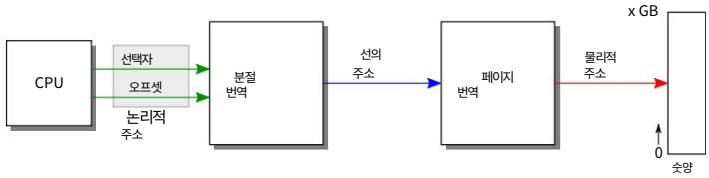
그림 A-1. http://software.intel.com에 기반한 Intel i7 Xeon 시스템의 대기 시간 수치
/sites/products/coltral/hpc/vtune/performance_analytic_guide.pdf.

I/O 포트
메모리 매핑
입출력

I/O 포트 라고 하는 장치 주소에서 값을 읽고 쓰는 특수한 입출력 명령어를 제공합니다 . 이러한 명령어의 하드웨어 구현은 본질적으로 메모리를 읽고 쓰는 것과 같습니다. 초기 x86 프로세서에는 추가 주소 줄이 있었습니다. 0은 I/O 포트에서 읽기/쓰기를 의미하고 1은 메인에서 읽기/쓰기를 의미했습니다.

메모리. 각 하드웨어 장치는 할당된 I/O 포트 범위에 대한 읽기 및 쓰기를 위해 이러한 라인을 모니터링합니다. 장치의 포트를 통해 소프트웨어는 장치를 구성하고, 상태를 검사하고, 장치가 조치를 취하도록 할 수 있습니다. 예를 들어, 소프트웨어는 I/O를 사용할 수 있습니다.
포트 읽기 및 쓰기를 통해 디스크 인터페이스 하드웨어가 섹터를 읽고 쓸 수 있도록 합니다.
디스크.

많은 컴퓨터 아키텍처에는 별도의 장치 액세스 지침이 없습니다. 대신
장치에는 고정된 메모리 주소가 있으며 프로세서는 다음과 통신합니다.
(운영 체제의 명령에 따라) 해당 주소에서 값을 읽고 쓰는 방식으로 장치를 제어합니다. 사실, 최신 x86 아키텍처는 메모리 맵 이라고 하는 이 기술을 사용합니다.
네트워크, 디스크, 그래픽 컨트롤러와 같은 대부분의 고속 장치에 대한 I/O .
그러나 이전 버전과의 호환성을 이유로 이전의 in 및 out 지침이 그대로 남아 있습니다.
IDE 디스크 컨트롤러와 같이 이를 사용하는 레거시 하드웨어 장치를 수행합니다.
용도.



논리 주소
선형 주소
물리적 주소

그림 B-1. 논리적, 선형적, 물리적 주소 간의 관계.

부트로더
실제 모드

부록 B

부트로더

x86 PC가 부팅되면 BIOS(Basic In-put/Output System)라는 프로그램을 실행하기 시작합니다. 이 프로그램은 마더보드의 비휘발성 메모리에 저장됩니다.
BIOS의 역할은 하드웨어를 준비한 다음 제어권을 운영 체제로 이전하는 것입니다. 구체적으로, 부트 섹터에서 로드된 코드로 제어권을 이전합니다.
부팅 디스크의 512바이트 섹터. 부팅 섹터에는 부트 로더가 들어 있습니다. 커널을 메모리에 로드하는 명령어입니다. BIOS는 메모리에 부트 섹터를 로드합니다.
주소 0x7c00을 지정한 다음 해당 주소로 점프합니다(프로세서의 %ip를 설정).
부트로더가 실행을 시작하고 프로세서는 Intel 8088을 시뮬레이션하고 로더의
작업은 프로세서를 보다 현대적인 운영 모드로 전환하고 xv6 커널을 로드하는 것입니다.
디스크에서 메모리로, 그리고 커널로 제어권을 넘깁니다. xv6 부트 로더는 두 개의 소스 파일로 구성되어 있으며, 하나는 16비트와 32비트 x86의 조합으로 작성되었습니다.

어셈블리(bootasm.S; (9100)) 와 C로 작성된 어셈블리(bootmain.c; (9200)).

코드: 어셈블리 부트스트랩

부트 로더의 첫 번째 명령어는 cli (9112)로, 프로세서 인터럽트를 비활성화합니다. 인터럽트는 하드웨어 장치가 인터럽트 핸들러라고 하는 운영 체제 기능을 호출하는 방법입니다. BIOS는 아주 작은 운영 체제이며,

하드웨어를 초기화하는 일부로 자체 인터럽트 핸들러를 설정합니다. 그러나
BIOS가 더 이상 실행되지 않습니다. 부트로더가 실행 중이므로 더 이상 적절하지 않거나 안전하지 않습니다.
하드웨어 장치의 인터럽트를 처리합니다. xv6이 준비되면(3장)

인터럽트를 다시 활성화합니다.

프로세서는 Intel 8088을 시뮬레이션하는 실제 모드 에 있습니다. 실제 모드에서
16비트 범용 레지스터가 8개 있지만 프로세서는 20비트의 주소를 메모리로 보냅니다. 세그먼트 레지스터 %cs, %ds, %es 및 %ss는 추가
16비트 레지스터에서 20비트 메모리 주소를 생성하는 데 필요한 비트. 프로그램이 메모리 주소를 참조할 때 프로세서는 자동으로 값의 16배를 추가합니다.
세그먼트 레지스터 중 하나; 이 레지스터는 16비트 폭입니다. 어떤 세그먼트 레지스터가 일반적으로 메모리 참조의 종류에 내재되어 있습니까? 명령어 페치는 %cs를 사용합니다.
데이터 읽기와 쓰기는 %ds를 사용하고, 스택 읽기와 쓰기는 %ss를 사용합니다.

부트로더

Xv6는 x86 명령어가 메모리 논리 주소 피연산자 에 가상 주소를 사용한다고 가장 하지만, x86 명령어는 실제로 논리 주소를 사용합니다 (그림 B-1 참조). 선형 주소 논리 주소는 세그먼트 선택기와 오프셋으로 구성되며, 때때로 가상 주소 보호 모드 에서 segment:offset 으로 작성됩니다. 더 자주, 세그먼트는 암묵적이고 프로그램은 세그먼트 설명 자만 오프셋을 직접 조작합니다. 세그먼테이션 하드웨어는 위에 설명된 표에 따라 변환을 수행하여 선형 주소를 생성합니다. 페이징 하드웨어가 활성화된 경우(2장 참조) 선형 주소를 물리 주소로 변환합니다. 그렇지 않으면 프로세서는 선형 주소를 물리 주소로 사용합니다.

gdt+코드

부트로더는 페이징 하드웨어를 활성화하지 않습니다. 사용하는 논리 주소는 세그먼테이션 하드웨어에 의해 선형 주소로 변환된 다음, 물리 주소로 직접 사용됩니다. Xv6는 세그먼테이션 하드웨어를 구성하여 변경 없이 논리 주소를 선형 주소로 변환하므로 항상 동일합니다. 역사적 이유로 프로그램에서 조작하는 주소를 나타내는 데 가상 주소라는 용어를 사용 했습니다. xv6 가상 주소는 x86 논리 주소와 동일하며 세그먼테이션 하드웨어가 매핑하는 선형 주소와 동일합니다. 페이징이 활성화되면 시스템에서 유일하게 흥미로운 주소 매핑은 선형에서 물리로 매핑됩니다.

BIOS는 %ds, %es, %ss의 내용에 대해 아무것도 보장하지 않으므로 인터럽트를 비활성화한 후 가장 먼저 해야 할 일은 %ax를 0으로 설정한 다음 해당 0을 %ds, %es, %ss (9115-9118)에 복사하는 것입니다.

가상 segment:offset은 21비트의 물리적 주소를 생성할 수 있지만, Intel 8088은 20비트의 메모리만 주소 지정할 수 있으므로, 최상위 비트를 버렸습니다: 0xffff0+0xffff = 0x10ffef, 하지만 8088의 가상 주소 0xffff:0xffff는 물리적 주소 0x0ffef를 참조합니다. 일부 초기 소프트웨어는 하드웨어가 21번째 주소 비트를 무시하는 데 의존했기 때문에, Intel이 20비트 이상의 물리적 주소를 가진 프로세서를 도입했을 때, IBM은 PC 호환 하드웨어에 필요한 호환성 해킹을 제공했습니다. 키보드 컨트롤러의 출력 포트의 두 번째 비트가 낮으면, 21번째 물리적 주소 비트는 항상 지워집니다. 높으면, 21번째 비트는 정상적으로 작동합니다. 부트 로더는 포트 0x64 및 0x60 (9120-9136)에서 키보드 컨트롤러에 대한 I/O를 사용하여 21번째 주소 비트를 활성화해야 합니다.

실제 모드의 16비트 범용 및 세그먼트 레지스터는 프로그램이 65,536바이트 이상의 메모리를 사용하는 것을 어렵게 만들고, 1메가바이트 이상을 사용하는 것은 불가능하게 만듭니다. 80286 이후의 x86 프로세서는 물리적 주소가 훨씬 더 많은 비트를 가질 수 있게 해주는 보호 모드 와 (80386 이후) 레지스터, 가상 주소 및 대부분의 정수 산술을 16비트가 아닌 32비트로 수행하게 해주는 ''32비트'' 모드를 가지고 있습니다. xv6 부팅 시퀀스는 다음과 같이 보호 모드와 32비트 모드를 활성화합니다.

보호 모드에서 세그먼트 레지스터는 세그먼트 설명자 테이블 의 인덱스입니다 (그림 B-2 참조). 각 테이블 항목은 기본 물리 주소, 제한이라고 하는 최대 가상 주소, 세그먼트에 대한 권한 비트를 지정합니다. 이러한 권한은 보호 모드에서 보호됩니다. 커널은 이를 사용하여 프로그램이 자체 메모리만 사용하도록 할 수 있습니다. xv6는 세그먼트를 거의 사용하지 않습니다. 대신 2장에서 설명한 대로 페이징 하드웨어를 사용합니다. 부트 로더는 세그먼트 설명자 테이블 gdt (9182-9185) 를 설정하여 모든 세그먼트가 기본 주소가 0이고 가능한 최대 제한(4기가바이트)을 갖도록 합니다. 테이블에는 널 항목, 실행 코드에 대한 항목 하나, en-

보호 모드

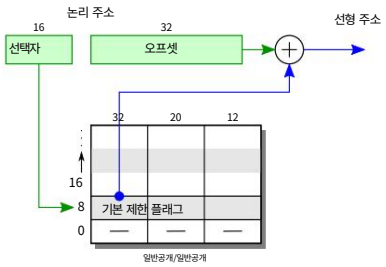


그림 B-2. 보호 모드의 세그먼트.

부트로더 글로벌 디스크립터 테이블

gdtdesc+code
gdt+code
CR0_PE+코드
gdt+코드
SEG_KDATA+코드 부트메인+코드

데이터를 시도합니다. 코드 세그먼트 설명자에는 코드가 32비트 모드 (0660)에서 실행되어야 함을 나타내는 플래그 세트가 있습니다. 이 설정으로 부트 로더가 보호 모드로 전환되면 논리 주소가 물리 주소에 일대일로 매핑됩니다.

부트로더는 lgdt 명령어 (9141) 를 실행하여 프로세서의 전역 설명자 테이블(GDT) 레지스터에 값 gdtdesc (9187-9189) 를 로드합니다 . 이 값은 테이블 gdt를 가리킵니다.

부트로더는 GDT 레지스터를 로드한 후 레지스터 %cr0 (9142-9144)에서 1비트(CR0_PE)를 설정하여 보호 모드를 활성화합니다. 보호 모드를 활성화해도 프로세서가 논리적 주소를 물리적 주소로 변환하는 방식이 즉시 변경되지는 않습니다. 프로세서가 GDT를 읽고 내부 세그먼테이션 설정을 변경하는 것은 세그먼트 레지스터에 새 값을 로드할 때뿐입니다. %cs를 직접 수정할 수 없으므로 대신 코드는 ljmp(far jump) 명령어 (9153) 를 실행하여 코드 세그먼트 선택기를 지정할 수 있습니다. 점프는 다음 줄 (9156) 에서 실행을 계속 하지만 그렇게 하면서 %cs가 gdt의 코드 설명자 항목을 참조하도록 설정합니다. 해당 설명자는 32비트 코드 세그먼트를 설명하므로 프로세서는 32비트 모드로 전환합니다. 부트로더는 8088에서 80286을 거쳐 80386으로 진화하는 동안 프로세서를 간호했습니다.

32비트 모드에서 부트 로더의 첫 번째 동작은 SEG_KDATA (9158-9161) 로 데이터 세그먼트 레지스터를 초기화하는 것입니다 . 논리 주소는 이제 물리 주소에 직접 매핑됩니다. C 코드를 실행하기 전에 남은 유일한 단계는 메모리의 사용되지 않는 영역에 스택을 설정하는 것입니다. 0xa0000에서 0x100000까지의 메모리는 일반적으로 장치 메모리 영역으로 가득 차 있으며 xv6 커널은 0x100000에 배치될 것으로 예상합니다. 부트 로더 자체는 0x7c00에서 0x7e00(512바이트)에 있습니다. 기본적으로 다른 모든 메모리 섹션은 스택에 적합한 위치가 될 것입니다. 부트 로더는 0x7c00(이 파일에서는 $start로 알려짐)을 스택의 맨 위로 선택합니다. 스택은 거기에서 부트 로더에서 멀어져 0x0000 방향으로 아래로 커집니다.

마지막으로 부트로더는 C 함수 bootmain (9168)을 호출합니다. Bootmain의 역할은 커널을 로드하고 실행하는 것입니다. 무언가 잘못되었을 때만 반환합니다. 그런 경우 코드는 포트 0x8a00 (9170-9176)에 몇 개의 출력 단어를 보냅니다. 실제 하드웨어에서는 해당 포트에 연결된 장치가 없으므로 이 코드는 아무것도 하지 않습니다. 부트로더가 PC 시뮬레이터 내부에서 실행 중이면 포트 0x8a00은 시뮬레이터 자체에 연결되어 시뮬레이터로 제어를 다시 전송할 수 있습니다. 시뮬레이터이든 아니든 코드는 무한 루프 (9177-9178)를 실행합니다. 실제 부트로더는 먼저 오류 메시지를 인쇄하려고 시도할 수 있습니다.

코드: C 부트스트랩

readseg+코드
stosb+코드
_시작+코드
입력+코드

부트로더의 C 부분인 bootmain.c (9200)는 다음 복사본을 찾을 것으로 예상합니다.
디스크에서 두 번째 섹터부터 실행 가능한 커널입니다. 커널은 2장에서 살펴본 것처럼 ELF 형식 바이너리입니다.
ELF 헤더에 액세스하려면 bootmain
ELF 바이너리의 첫 4096바이트 (9214)를 로드합니다. 메모리 내 사본을 주소 0x10000에 배치합니다.

다음 단계는 이것이 ELF 바이너리인지 아닌지 빠르게 확인하는 것입니다.
초기화되지 않은 디스크. Bootmain은 디스크 위치에서 시작하여 섹션의 내용을 읽습니다.
ELF 헤더 시작 후 바이트를 끄고 주소에서 시작하여 메모리에 씁니다.
paddr. Bootmain은 디스크에서 데이터를 로드하기 위해 readseg를 호출하고 (9238) stosb를 0으로 호출합니다.
세그먼트의 나머지 (9240). Stosb (0492) 는 x86 명령어 rep stosb를 사용합니다.
메모리 블록의 모든 바이트를 초기화합니다.

커널은 가상에서 자신을 찾을 수 있도록 컴파일되고 링크되었습니다.
0x80100000에서 시작하는 주소. 따라서 함수 호출 명령어는 0x801xxxxx처럼 보이는 대상 주소를 언급해야 합니다. kernel.asm에서 예를 볼 수 있습니다.
이 주소는 kernel.ld (9311) 에서 구성됩니다 . 0x80100000은 32비트 주소 공간의 끝부분에 있는 비교적 높은 주소입니다. 2장에서는 그 이유를 설명합니다.
이 선택. 그렇게 높은 주소에는 물리적 메모리가 없을 수도 있습니다.
커널이 실행을 시작하면 가상 주소를 매핑하기 위해 페이징 하드웨어가 설정됩니다.
0x80100000에서 시작하여 0x00100000에서 시작하는 물리적 주소까지; 커널은 이 하위 주소에 물리적 메모리가 있다고 가정합니다. 부팅의 이 지점에서
프로세스이지만 페이징은 활성화되지 않았습니다. 대신 kernel.ld는 ELF를 지정합니다.
paddr는 0x00100000에서 시작하므로 부트로더가 커널을 하위로 복사합니다.
페이징 하드웨어가 결국 가리키게 될 물리적 주소입니다.

부트로더의 마지막 단계는 커널의 진입점을 호출하는 것입니다. 진입점은 커널이 실행을 시작할 것으로 예상하는 명령어입니다. xv6의 경우 진입 주소는 다음과 같습니다.
0x10000c:

```
# objdump -f 커널

커널: 아키텍 파일 형식 elf32-i386
처: i386, 플래그 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
시작 주소 0x0010000c
```

관례에 따라 _start 심볼은 ELF 진입점을 지정합니다. 이는 다음에 정의됩니다.
파일 entry.S (1040). xv6가 아직 가상 메모리를 설정하지 않았기 때문에 xv6의 진입점은 다음과 같습니다.
진입 지점의 물리적 주소 (1044).

현실 세계

이 부록에 설명된 부트 로더는 C 코드를 컴파일할 때 사용된 최적화에 따라 약 470바이트의 머신 코드로 컴파일됩니다. 그 작은 공간에 맞추기 위해 xv6 부트 로더는 커널이 부트 디스크에 연속적으로 쓰여졌다는 주요 단순화 가정을 합니다.

섹터 1에서. 더 일반적으로 커널은 일반 파일 시스템에 저장되며 여기서
연속되지 않을 수도 있고 네트워크를 통해 로드될 수도 있습니다. 이러한 복잡성에는 다음이 필요합니다.

부트로더는 다양한 디스크 및 네트워크 컨트롤러를 구동하고 다양한 파일 시스템과 네트워크 프로토콜을 이해할 수 있어야 합니다. 즉, 부트로더 자체
작은 운영 체제여야 합니다. 이렇게 복잡한 부트 로더는 확실히
512바이트에 맞으며 대부분의 PC 운영 체제는 2단계 부팅 프로세스를 사용합니다. 먼저 이 부록에 있는 것과 같은
간단한 부트 로더는 모든 기능을 갖춘 부트 로더를 로드합니다.
알려진 디스크 위치, 종종 디스크 액세스를 위해 공간 제약이 적은 BIOS에 의존함
디스크 자체를 구동하려고 하기보다는. 그런 다음 512바이트에서 해방된 전체 로더
제한, 원하는 것을 찾고, 로드하고, 실행하는 데 필요한 복잡성을 구현할 수 있습니다.
커널. 최신 PC는 위의 복잡성 중 많은 부분을 피할 수 있습니다.
PC가 더 큰 데이터를 읽을 수 있도록 하는 UEFI(Unified Extensible Firmware Interface)
디스크에서 부트로더를 시작하고 보호 및 32비트 모드로 시작합니다.

이 부록은 전원이 켜지고 나서 일어나는 유일한 일이 다음과 같은 것처럼 작성되었습니다.
부트로더의 실행은 BIOS가 부트섹터를 로드한다는 것입니다. 사실

BIOS는 복잡한 하드웨어를 구현하기 위해 엄청난 양의 초기화를 수행합니다.
현대 컴퓨터는 전통적인 표준 PC처럼 보입니다. BIOS는 실제로 작은
컴퓨터가 설치된 후 존재하는 하드웨어에 내장된 운영 체제
부팅됨.

수업 과정

1. 섹터 세분성으로 인해 텍스트의 readseg 호출은 read-seg((uchar*)0x100000, 0xb500, 0x1000)과 동일합니다. 실제로 이 엉성한 동작은 다음과 같습니다.
문제가 되지 않습니다. 엉성한 읽기 섹션이 왜 문제를 일으키지 않습니까?
2. bootmain()이 0x200000 대신 커널을 로드하도록 하려고 한다고 가정합니다.
0x100000이고 bootmain()을 수정하여 각 va에 0x100000을 추가했습니다.
ELF 섹션. 뭔가 잘못될 거야. 뭐?
3. 부트로더가 ELF 헤더를 임의의 위치 0x10000의 메모리에 복사하는 것은 잠재적으로 위험한 듯합니다. 왜 필요한 메모리를 얻기 위해 malloc을 호출하지 않습니까?

색인