xv6: 간단한 유닉스 계열의 교육용 운영 체제

러스 콕스 프란스 카슈크 로버트 모리스

2022년 9월 5일

내용물

서문 및 감사의 말

이것은 운영 체제 수업을 위해 작성된 초안 텍스트입니다. 여기서는 xv6라는 예제 커널을 연구하여 운영 체제의 주요 개념을 설명합니다. Xv6는 Dennis Ritchie와 Ken Thompson의 Unix Version 6(v6)[17]을 모델로 합니다. Xv6는 v6의 구조와 스타일을 느슨하게 따르지만 다중 코어 RISC-V[15]를 위해 ANSI C[7]로 구현됩니다.

이 텍스트는 John Lions의 UNIX 6판 주석[11]에서 영감을 받은 접근 방식인 xv6의 소스 코드와 함께 읽어야 합니다 . xv6를 사용한 여러 랩 과제를 포함하여 v6 및 xv6에 대한 온라인 리소스에 대한 포인터는 https://pdos.csail.mit.edu/6.1810 을 참조하세요.

우리는 이 텍스트를 MIT의 운영 체제 수업인 6.828과 6.1810에서 사용했습니다. 우리는 xv6에 직간접적으로 기여한 해당 수업의 교수, 조교, 학생들에게 감사드립니다. 특히 Adam Belay, Austin Clements, Nickolai Zeldovich에게 감사드리고 싶습니다. 마지막으로, 텍스트의 버그나 개선 제안을 이메일로 보내주신 분들께 감사드립니다: Abutalib Aghayev, Sebastian Boehm, brandb97, Anton Burtsev, Raphael Carvalho, Tej Chajed, Rasit Eskicioglu, Color Fuzzy, Wojciech Gac, Giuseppe, Tao Guo, Haibo Hao, Naoki Hayama, Chris Henderson, Robert Hilderman, Eden Hochbaum, Wolfgang Keller, Henry Laih, Jin Li, Austin Liew, Pavan Maddamsetti, Jacek Masiulaniec, Michael McConville, m3hm00d, miguelgvieira, Mark Morrissey, Muhammed Mourad, Harry Pan, Harry Porter, Siyuan Qian, Askar Safin, Salman Shah, Huang Sha, Vikram Shenoy, Adeodato Simó, Ruslan Savchenko, Pawel Szczurko, Warren Toomey, tyfkda, tzerbib, Vanush Vaswani, Xi Wang, Zou Chang Wei, Sam Whitlock, LucyShawYang, Meng Zhou

오류를 발견하셨거나 개선에 대한 제안 사항이 있으시면 Frans Kaashoek과 Robert Morris(kaashoek,rtm@csail.mit.edu)에게 이메일을 보내주세요.

제1장

운영 체제 인터페이스

운영 체제의 역할은 여러 프로그램 간에 컴퓨터를 공유하고 하드웨어만으로 지원하는 것보다 더 유용한 서비스 세트를 제공하는 것입니다. 운영 체제는 저수준 하드웨어를 관리하고 추상화하므로 예를 들어 워드 프로세서는 어떤 유형의 디스크 하드웨어가 사용되고 있는지에 대해 신경 쓸 필요가 없습니다. 운영 체제는 여러 프로그램 간에 하드웨어를 공유하여 동시에 실행(또는 실행되는 것처럼 보임)합니다. 마지막으로 운영 체제는 프로그램이 상호 작용할 수 있는 제어된 방법을 제공하여 데이터를 공유하거나 함께 작업할 수 있습니다.

운영 체제는 인터페이스를 통해 사용자 프로그램에 서비스를 제공합니다. 좋은 인터페이스를 디자인하는 것은 어려운 일입니다. 한편으로는 인터페이스가 간단하고 좁기를 원합니다. 그렇게 하면 구현을 올바르게 하는 것이 더 쉬워지기 때문입니다. 다른 한편으로는 애플리케이션에 많은 정교한 기능을 제공하고 싶을 수 있습니다. 이러한 긴장을 해결하는 요령은 많은 일반성을 제공하기 위해 결합할 수 있는 몇 가지 메커니즘에 의존하는 인터페이스를 디자인하는 것입니다.

이 책은 운영 체제 개념을 설명하기 위해 단일 운영 체제를 구체적인 예로 사용합니다. 해당 운영 체제 xv6는 Ken Thompson과 Dennis Ritchie의 Unix 운영 체제[17]에서 도입한 기본 인터페이스를 제공하고 Unix의 내부 디자인을 모방합니다.

Unix는 메커니즘이 잘 결합되어 놀라울 정도로 일반성을 제공하는 좁은 인터페이스를 제공합니다. 이 인터페이스는 매우 성공적이어서 BSD, Linux, macOS, Solaris, 그리고 Microsoft Windows와 같은 현대 운영 체제는 Unix와 유사한 인터페이스를 가지고 있습니다. xv6를 이해하는 것은 이러한 시스템과 다른 많은 시스템을 이해하는 데 좋은 시작입니다.

그림 1.1에서 보듯이, xv6는 실행 중인 프로그램에 서비스를 제공하는 특수 프로그램인 커널의 전통적인 형태를 취합니다. 프로세스라고 하는 각 실행 중인 프로그램은 명령어, 데이터, 스택을 포함하는 메모리를 갖습니다. 명령어는 프로그램의 계산을 구현합니다. 데이터는 계산이 작용하는 변수입니다. 스택은 프로그램의 프로시저 호출을 구성합니다.

일반적으로 컴퓨터에는 많은 프로세스가 있지만 커널은 단 하나뿐입니다.

프로세스가 커널 서비스를 호출해야 할 때, 운영 체제 인터페이스의 호출 중 하나인 시스템 호출을 호출합니다. 시스템 호출이 커널에 들어가고, 커널은 서비스를 수행하고 반환합니다. 따라서 프로세스는 사용자 공간과 커널 공간에서 번갈아가며 실행합니다.

이후 장에서 자세히 설명했듯이 커널은 CPU1에서 제공하는 하드웨어 보호 메커니즘을 사용하여 사용자 공간에서 실행되는 각 프로세스가 액세스할 수 있는 유일한 항목만 허용합니다.

1이 텍스트는 일반적으로 CPU라는 용어로 계산을 실행하는 하드웨어 요소를 지칭합니다.

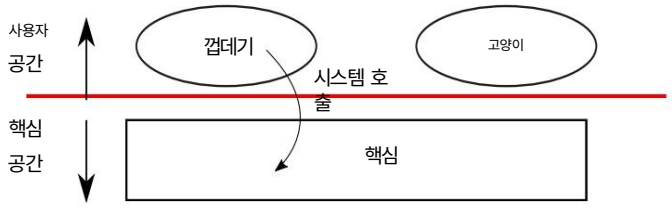

그림 1.1: 커널과 두 개의 사용자 프로세스.

자체 메모리. 커널은 이러한 보호를 구현하는 데 필요한 하드웨어 권한으로 실행되고, 사용자 프로그램은 해당 권한 없이 실행됩니다. 사용자 프로그램이 시스템 호출을 호출하면 하드웨어는 권한 수준을 높이고 커널에서 미리 준비된 함수를 실행하기 시작합니다.

커널이 제공하는 시스템 호출의 컬렉션은 사용자 프로그램이 보는 인터페이스입니다. xv6 커널은 Unix 커널이 전통적으로 제공하는 서비스와 시스템 호출의 하위 집합을 제공합니다.
그림 1.2는 xv6의 모든 시스템 호출을 나열하고 있습니다.
이 장의 나머지 부분에서는 xv6의 서비스(프로세스, 메모리, 파일 설명자, 파이프, 파일 시스템)를 개략적으로 설명하고 코드 조각과 Unix의 명령줄 사용자 인터페이스인 셸에서 이를 사용하는 방법에 대한 논의를 통해 이를 설명합니다. 셸에서 시스템 호출을 사용하는 방식은 이들이 얼마나 신중하게 설계되었는지를 보여줍니다.

셸은 사용자로부터 명령을 읽고 실행하는 일반 프로그램입니다. 셸이 커널의 일부가 아니라 사용자 프로그램이라는 사실은 시스템 호출 인터페이스의 힘을 보여줍니다. 셸에는 특별한 것이 없습니다. 또한 셸을 쉽게 교체할 수 있다는 것을 의미합니다. 결과적으로 현대 Unix 시스템은 다양한 셸을 선택할 수 있으며 각각 고유한 사용자 인터페이스와 스크립팅 기능이 있습니다. xv6 셸은 Unix Bourne 셸의 본질을 간단하게 구현한 것입니다. 구현은 (user/sh.c:1)에서 찾을 수 있습니다.

1.1 프로세스 및 메모리

xv6 프로세스는 사용자 공간 메모리(명령어, 데이터 및 스택)와 커널에 비공개인 프로세스별 상태로 구성됩니다. Xv6는 프로세스를 시간 공유합니다. 실행을 기다리는 프로세스 집합 간에 사용 가능한 CPU를 투명하게 전환합니다. 프로세스가 실행 중이 아닐 때 xv6는 프로세스의 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 새 프로세스에 호출 프로세스의 메모리, 명령어 및 데이터의 정확한 사본을 제공합니다. fork는 원래 프로세스와 새 프로세스 모두에서 반환합니다. 원래 프로세스에서 fork는 새 프로세스의 PID를 반환합니다. 새 프로세스에서 fork는 0을 반환합니다. 원래 프로세스와 새 프로세스는 종종 부모 프로세스와 자식 프로세스라고 합니다.

중앙 처리 장치에 대한 것입니다. 다른 문서(예: RISC-V 사양)도 CPU 대신 프로세서, 코어, 하트라는 단어를 사용합니다.

시스템 호출	설명
int fork()	프로세스를 생성하고 자식의 PID를 반환합니다.
int exit(int status)	현재 프로세스를 종료합니다. 상태는 wait()에 보고되었습니다. 반환 없음.
int wait(int *status) int	자식이 종료될 때까지 기다립니다. 종료 상태는 *status에 있습니다. 자식 PID를 반환합니다.
kill(int pid) int	프로세스 PID를 종료합니다. 오류의 경우 0 또는 -1을 반환합니다.
getpid() int	현재 프로세스의 PID를 반환합니다.
sleep(int n) int	n번 똑딱거리는 동안 멈춥니다.
exec(char *file, char *argv[]) 파일을 로드하고 인수와 함께 실행합니다. 오류가 발생할 경우에만 반환합니다.	
char *sbrk(int n)	프로세스의 메모리를 n바이트만큼 늘립니다. 새 메모리의 시작을 반환합니다.
int open(char *file, int flags) int	파일을 엽니다. 플래그는 읽기/쓰기를 나타냅니다. fd(파일 설명자)를 반환합니다.
write(int fd, char *buf, int n) buf에서 파일 기술자 fd로 n 바이트를 씁니다. n을 반환합니다.	
int read(int fd, char *buf, int n) n 바이트를 buf에 읽어들입니다. 읽은 숫자만큼을 반환합니다. 파일이 끝나면 0을 반환합니다.	
int 닫기(int fd)	열려 있는 파일 fd를 해제합니다.
int dup(int fd)	fd와 동일한 파일을 참조하는 새로운 파일 기술자를 반환합니다.
int 파이프(int p[])	파이프를 생성하고 p[0]과 p[1]에 읽기/쓰기 파일 기술자를 넣습니다.
int chdir(char *dir)	현재 디렉토리를 변경합니다.
int mkdir(char *dir)	새로운 디렉토리를 만듭니다.
int mknod(char *파일, int, int)	장치 파일을 생성합니다.
int fstat(int fd, 구조체 stat *st)	열려 있는 파일 정보를 *st에 넣습니다.
int stat(char *file, struct stat *st) 지정된 파일에 대한 정보를 *st에 넣습니다.	
int link(char *file1, char *file2) 파일 file1에 대한 다른 이름(file2)을 생성합니다.	
int unlink(char *파일)	파일을 제거합니다.

그림 1.2: Xv6 시스템 호출. 달리 명시되지 않은 경우 이러한 호출은 오류가 없는 경우 0을 반환하고 오류가 있는 경우 -1을 반환합니다.
오류가 있습니다.

예를 들어, C 프로그래밍 언어[7]로 작성된 다음 프로그램 조각을 고려해 보겠습니다.

```
int pid = fork();
if(pid > 0){
    printf("부모: 자식=%d\n", pid);
    pid = wait((int *) 0);
    printf("자식 %d가 완료되었습니다\n", pid);
} 그렇지 않은 경우(pid == 0){
    printf("자식: 종료\n");
    종료(0);
} 또 다른 {
    printf("포크 오류\n");
}
```

종료 시스템 호출 은 호출 프로세스가 실행을 중지하고 다음과 같은 리소스를 해제하도록 합니다.
메모리 및 열린 파일. Exit는 정수 상태 인수를 취하며, 일반적으로 성공을 나타내는 0입니다.
그리고 1은 실패를 나타냅니다. 대기 시스템 호출은 종료된(또는 종료된) 자식의 PID를 반환합니다.

현재 프로세스와 자식의 종료 상태를 wait에 전달된 주소로 복사합니다. 호출자의 자식 중 아무도 종료하지 않은 경우 wait은 종료할 때까지 기다립니다. 호출자에게 자식이 없는 경우 wait은 즉시 -1을 반환합니다. 부모가 자식의 종료 상태에 관심이 없는 경우 wait에 0 주소를 전달할 수 있습니다.

이 예에서 출력 라인은 다음과 같습니다.

```
부모: 자식=1234 자식: 종료
```

부모나 자식이 먼저 printf 호출 에 도달하는지에 따라 어느 순서로든(또는 섞여도) 나올 수 있습니다 . 자식이 종료된 후 부모의 대기가 반환되어 부모가 다음을 인쇄합니다.

```
부모 : 자식 1234 완료
```

자식은 처음에 부모와 동일한 메모리 내용을 가지고 있지만 부모와 자식은 별도의 메모리와 별도의 레지스터로 실행됩니다. 한 쪽에서 변수를 변경해도 다른 쪽에는 영향을 미치지 않습니다. 예를 들어 wait 의 반환 값이 부모 프로세스의 pid 에 저장되면 자식의 변수 pid는 변경되지 않습니다 . 자식의 pid 값은 여전히 0입니다.

exec 시스템 호출 은 호출 프로세스의 메모리를 파일 시스템에 저장된 파일에서 로드된 새 메모리 이미지로 대체합니다. 파일에는 특정 형식이 있어야 하며, 파일의 어느 부분에 명령어가 있는지, 어느 부분이 데이터인지, 어느 명령어에서 시작할지 등을 지정합니다. Xv6는 ELF 형식을 사용하는데, 3장에서 더 자세히 설명합니다. 일반적으로 파일은 프로그램의 소스 코드를 컴파일한 결과입니다. exec가 성공하면 호출 프로그램으로 돌아가지 않습니다. 대신 파일에서 로드된 명령어가 ELF 헤더에 선언된 진입점에서 실행을 시작합니다. exec는 실행 파일이 들어 있는 파일 이름과 문자열 인수 배열이라는 두 가지 인수를 사용합니다. 예를 들어:

```
char *argv[3];

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

이 조각은 호출하는 프로그램을 인수 목록 echo hello로 실행되는 프로그램 /bin/echo 의 인스턴스로 대체합니다 . 대부분의 프로그램은 인수 배열의 첫 번째 요소를 무시하는데, 이는 관례적으로 프로그램 이름입니다.

xv6 셸은 위의 호출을 사용하여 사용자를 대신하여 프로그램을 실행합니다. 셸의 주요 구조는 간단합니다. main (user/sh.c:146)을 참조하세요. 메인 루프는 getcmd로 사용자로부터 입력 줄을 읽습니다 . 그런 다음 fork를 호출하여 셸 프로세스의 사본을 만듭니다. 부모는 wait를 호출하고 자식은 명령을 실행합니다. 예를 들어, 사용자 가 셸에 "echo hello"를 입력했다면 runcmd는 인수로 "echo hello"를 사용하여 호출되었을 것입니다 . runcmd (user/sh.c:55) 실제 명령을 실행합니다. "echo hello" 의 경우 exec (user/sh.c:79) 를 호출합니다 . exec가 성공 하면 자식은 runcmd 대신 echo 에서 명령어를 실행합니다 . 어느 시점에서 echo는 exit를 호출하여 부모가 main (user/sh.c:146) 의 wait 에서 돌아오게 합니다 .

왜 fork 와 exec가 단일 호출에서 결합되지 않는지 궁금할 수 있습니다 . 나중에 셸이 I/O 리디렉션 구현에서 분리를 활용하는 것을 볼 수 있습니다. 중복 프로세스를 만든 다음 즉시 (exec로) 대체하는 낭비를 피하기 위해 운영 커널은 copy-on-write와 같은 가상 메모리 기술을 사용하여 이 사용 사례에 대한 fork 구현을 최적화합니다 (4.6절 참조).

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 n 바이트만큼 늘릴 수 있습니다. sbrk는 새 메모리의 위치를 반환합니다.

1.2 I/O 및 파일 설명자

파일 설명자는 프로세스가 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. 입력 및 출력을 I/O라고 합니다.

내부적으로 xv6 커널은 파일 설명자를 프로세스별 테이블의 인덱스로 사용하므로 모든 프로세스가 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 설명자가 열려 있도록 합니다 (user/sh.c:152). 이는 기본적으로 콘솔의 파일 설명자입니다.

읽기 및 쓰기 시스템 호출은 파일 설명자로 명명된 열린 파일에서 바이트를 읽고 바이트를 씁니다. read(fd, buf , n) 호출은 파일 설명자 fd 에서 최대 n 바이트를 읽고, 이를 buf 에 복사하고 , 읽은 바이트 수를 반환합니다. 파일을 참조하는 각 파일 설명자에는 연관된 오프셋이 있습니다. read는 현재 파일 오프셋에서 데이터를 읽고, 읽은 바이트 수만큼 오프셋을 앞으로 이동합니다. 후속 읽기는 첫 번째 읽기에서 반환된 바이트 다음의 바이트를 반환합니다 . 읽을 바이트가 더 이상 없으면 read는 파일의 끝을 나타내기 위해 0을 반환합니다.

write(fd, buf, n) 호출은 buf 에서 파일 설명자 fd 로 n 바이트를 쓰고 쓴 바이트 수를 반환합니다. 오류가 발생할 때만 n 바이트 미만이 쓰여집니다. read와 마찬가지로 write는 현재 파일 오프셋에 데이터를 쓰고 그 오프셋을 쓴 바이트 수만큼 앞으로 이동합니다. 각 쓰기는 이전 쓰기가 중단된 곳에서 시작합니다.

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

```
char 버퍼[512]; int n;

을 위한(;;){
```

```
    n = read(0, buf, sizeof buf); if(n == 0) break; if(n < 0)
    { fprintf(2, "읽기 오
        류\n");
    exit(1);



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

    }
}
```

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 향후 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 fork는 상호 작용하여 I/O 리디렉션을 쉽게 구현할 수 있도록 합니다. fork는 부모의 파일 설명자 테이블을 메모리와 함께 복사하므로 자식은 부모와 정확히 동일한 열린 파일로 시작합니다. 시스템 호출 exec는 호출 프로세스의 메모리를 대체하지만 파일 테이블은 보존합니다. 이 동작을 통해 셸은 forking 하여 자식에서 선택한 파일 설명자를 다시 열고 exec를 호출하여 새 프로그램을 실행하여 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 엽니다
      (O_RDONLY);
      exec("고양이", argv);

  }
```

자식이 파일 설명자 0을 닫은 후, open은 새로 열린 input.txt 에 대해 해당 파일 설명자를 사용하도록 보장됩니다 . 0은 사용 가능한 가장 작은 파일 설명자입니다. 그런 다음 cat은 input.txt 를 참조하는 파일 설명자 0(표준 입력)으로 실행합니다 . 부모 프로세스의 파일 설명자는 이 시퀀스에 의해 변경되지 않습니다. 왜냐하면 자식의 설명자만 수정하기 때문입니다.

xv6 셸의 I/O 리디렉션 코드는 정확히 이런 방식으로 작동합니다 (user/sh.c:83). 이 코드 지점에서 셸은 이미 자식 셸을 포크했고 runcmd 는 exec를 호출하여 새 프로그램을 로드한다는 점을 기억하세요.

open 에 대한 두 번째 인수는 비트로 표현된 플래그 집합으로 구성되며, open이 무엇을 하는지 제어합니다. 가능한 값은 파일 제어(fcntl) 헤더 (kernel/fcntl.h:1-5)에 정의되어 있습니다.

O_RDONLY, O_WRONLY, O_RDWR, O_CREATE 및 O_TRUNC 는 파일을 읽기용으로 열거나, 쓰기용으로 열거나, 읽기와 쓰기 모두를 위해 열도록 지시하고 , 파일이 없으면 파일을 생성하고, 파일의 길이를 0으로 자릅니다.

이제 fork 와 exec가 별도의 호출인 것이 왜 도움이 되는지 분명해졌을 것입니다 . 두 호출 사이에서 쉘은 메인 쉘의 I/O 설정을 방해하지 않고 자식의 I/O를 리디렉션할 기회가 있습니다. 대신 가상의 결합된 forkexec 시스템 호출을 상상할 수 있지만, 그러한 호출로 I/O 리디렉션을 수행하는 옵션은 어색해 보입니다. 쉘은 forkexec를 호출하기 전에 자체 I/O 설정을 수정할 수 있습니다 (그런 다음 해당 수정 사항을 취소할 수 있음). 또는 forkexec가 인수로 I/O 리디렉션에 대한 명령을 받을 수 있습니다. 또는 (가장 매력적이지 않은) cat 과 같은 모든 프로그램 이 자체 I/O 리디렉션을 수행하도록 교육될 수 있습니다.

fork가 파일 기술자 테이블을 복사 하더라도 각 기본 파일 오프셋은 다음과 공유됩니다.
부모와 자식. 이 예를 생각해 보세요:

```
fork() == 0이면 { write(1, "안녕하
    세요 ", 6); exit(0); } else { wait(0); write(1,
    "세계\n", 6);



}
```

이 조각의 끝에서 파일 설명자 1에 첨부된 파일에는 hello world라는 데이터가 포함됩니다.
부모의 write (wait 덕분에 자식이 끝난 후에만 실행됨) 는 자식의 write가 중단된 곳에서 시작합니다. 이 동작은 (echo hello; echo world) >output.txt 와 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 fork 로 복제된 파일 디스크립터 와 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 시퀀스를 통해 동일한 원본 파일 설명자에서 파생된 경우 오프셋을 공유합니다 . 그렇지 않으면 파일 설명자는 동일한 파일에 대한 open 호출 에서 발생한 경우에도 오프셋을 공유하지 않습니다 . dup를 사용하면 셸에서 다음과 같은 명령을 구현할 수 있습니다. ls 기존 파일 비기존 파일 > tmp1 2>&1. 2 >&1은 셸 에 설명자 1의 복제본인 파일 설명자 2를 명령에 제공하라고 지시합니다. 기존 파일의 이름과 존재하지 않는 파일의 오류 메시지는 모두 파일 tmp1에 표시됩니다. xv6 셸은 오류 파일 설명자에 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.

1.3 파이프

파이프는 프로세스에 노출된 작은 커널 버퍼로, 하나는 읽기용이고 다른 하나는 쓰기용인 파일 설명자 쌍입니다. 파이프의 한쪽 끝에 데이터를 쓰면 그 데이터를 파이프의 다른 쪽 끝에서 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 닫습니다.
    dup(p[0]); p[0]을
    닫습니다. p[1]을 닫습
    니다. exec("/bin/wc",
    argv); } 그렇지 않은 경우
    { p[0]을 닫습니다. p[1], "안녕하세요 세계\n", 12
를 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 갖게 됩니다. 자식은 close 와 dup를 호출 하여 파일 기술자 0이 파이프의 읽기 끝을 참조하도록 하고, p에서 파일 기술자를 닫고, exec를 호출하여 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, read는 데이터 파일의 끝에 도달한 것처럼 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 . wc 의 파일 설명자 중 하나가 파이프의 쓰기 끝을 참조하는 경우 wc는 결코 파일 끝을 보지 못할 것입니다.

xv6 셸은 위의 코드(user/sh.c:101)와 유사한 방식으로 grep fork sh.c | wc -l 과 같은 파이프라인을 구현합니다 . 자식 프로세스는 파이프라인의 왼쪽 끝과 오른쪽 끝을 연결하는 파이프를 만듭니다. 그런 다음 파이프라인의 왼쪽 끝에 대해 fork 와 runcmd를 호출하고 오른쪽 끝에 대해 fork 와 runcmd를 호출 하고 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 b 에 대해 , 다른 하나는 c에 대해)를 포크합니다. 따라서 셸은 프로세스 트리를 만들 수 있습니다. 잎

이 트리의 내부 노드는 명령이고, 왼쪽과 오른쪽 자식이 완료될 때까지 기다리는 프로세스입니다.

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 세 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.

1.4 파일 시스템

xv6 파일 시스템은 해석되지 않은 바이트 배열을 포함하는 데이터 파일과 데이터 파일 및 다른 디렉토리에 대한 명명된 참조를 포함하는 디렉토리를 제공합니다. 디렉토리는 루트라는 특수 디렉토리에서 시작하여 트리를 형성합니다. /a/b/c 와 같은 경로는 루트 디렉토리 /에 있는 a 라는 디렉토리 내의 b 라는 디렉토리 내에 있는 c 라는 파일 또는 디렉토리를 참조합니다 . / 로 시작하지 않는 경로는 호출 프로세스의 현재 디렉토리를 기준으로 평가되며, chdir 시스템 호출 로 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 파일과 디렉토리를 만드는 시스템 호출이 있습니다. mkdir은 새 디렉토리를 만들고, O_CREATE 플래그 로 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 장치를 참조하는 특수 파일을 만듭니다. 장치 파일과 연관된 것은 주요 및 부차 장치 번호(mknod에 대한 두 인수)이며, 이는 커널 장치를 고유하게 식별합니다.
나중에 프로세스가 장치 파일을 열면 커널은 파일 시스템에 전달하지 않고 읽기 및 쓰기 시스템 호출을 커널 장치 구현으로 전환합니다.

파일 이름은 파일 자체와 다릅니다. inode라고 하는 동일한 기본 파일은 링크라고 하는 여러 이름을 가질 수 있습니다. 각 링크는 디렉토리의 항목으로 구성됩니다. 항목에는 파일 이름과 inode에 대한 참조가 포함됩니다. inode는 파일 유형(파일 또는 디렉토리 또는 장치), 길이, 디스크에서 파일 콘텐츠의 위치, 파일에 대한 링크 수를 포함하여 파일에 대한 메타데이터를 보관합니다.

fstat 시스템 호출 은 파일 설명자가 참조하는 inode에서 정보를 검색합니다.
stat.h (kernel/stat.h) 에 정의된 구조체 stat를 채웁니다 . 처럼:

```
#T_DIR을 정의합니다.              1 // 디렉토리
T_FILE #define T_DEVICE          2 // 파일 #define
3 // 장치

구조체 통계 {
                                  // 파일 시스템의 디스크 장치 int dev;
    type; // 파일 유              // Inode 번호 uint ino; short
    형 short nlink; // 파일에 대한 링크 수 uint64 size; // 바이
    트 단위의 파일 크기 };
```

링크 시스템 호출 은 존재하는 것과 동일한 inode를 참조하는 또 다른 파일 시스템 이름을 생성합니다.
ing 파일. 이 조각은 a 와 b 라는 이름의 새 파일을 만듭니다 .

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

a 에서 읽거나 쓰는 것은 b 에서 읽거나 쓰는 것과 같습니다 . 각 inode는 고유한 inode 번호로 식별됩니다. 위의 코드 시퀀스 후에 fstat 의 결과를 검사하여 a 와 b가 동일한 기본 콘텐츠를 참조한다는 것을 확인할 수 있습니다 . 둘 다 동일한 inode 번호 (ino) 를 반환 하고 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 액세스 가능합니다. 또한,

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

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

Unix는 mkdir, ln, rm 과 같이 사용자 수준 프로그램으로 셸에서 호출할 수 있는 파일 유틸리티를 제공합니다 . 이 설계를 통해 누구나 새로운 사용자 수준 프로그램을 추가하여 명령줄 인터페이스를 확장할 수 있습니다. 지금 돌이켜보면 이 계획은 당연한 것처럼 보이지만 Unix 당시 설계된 다른 시스템은 종종 이러한 명령을 셸에 내장했습니다(그리고 셸을 커널에 내장했습니다).

한 가지 예외는 셸 (user/sh.c:161) 에 내장된 cd 입니다 . cd는 쉘 자체의 현재 작업 디렉토리를 변경해야 합니다. cd가 일반 명령으로 실행되면 쉘은

자식 프로세스를 fork하면 자식 프로세스는 cd를 실행 하고 cd는 자식의 작업 디렉토리를 변경합니다. 부모(즉, 쉘)의 작업 디렉토리는 변경되지 않습니다.

1.5 현실 세계

Unix의 "표준" 파일 설명자, 파이프, 그리고 이를 위한 편리한 셸 구문의 조합은 범용 재사용 가능 프로그램을 작성하는 데 있어 큰 진전이었습니다. 이 아이디어는 Unix의 힘과 인기의 대부분을 담당한 "소프트웨어 도구" 문화를 촉발했고, 셸은 최초의 소위 "스크립팅 언어"였습니다. Unix 시스템 호출 인터페이스는 오늘날 BSD, Linux, macOS와 같은 시스템에서 지속됩니다.

Unix 시스템 호출 인터페이스는 Portable Operating System Interface(POSIX) 표준을 통해 표준화되었습니다. Xv6는 POSIX와 호환되지 않습니다. 많은 시스템 호출(lseek와 같은 기본 호출 포함)이 누락되었고, 제공하는 많은 시스템 호출이 표준과 다릅니다. xv6에 대한 우리의 주요 목표는 단순성과 명확성을 제공하는 동시에 간단한 UNIX 유사 시스템 호출 인터페이스를 제공하는 것입니다. 여러 사람이 기본 Unix 프로그램을 실행하기 위해 몇 가지 시스템 호출과 간단한 C 라이브러리로 xv6를 확장했습니다. 그러나 최신 커널은 xv6보다 훨씬 더 많은 시스템 호출과 더 많은 종류의 커널 서비스를 제공합니다. 예를 들어 네트워킹, 윈도잉 시스템, 사용자 수준 스레드, 많은 장치의 드라이버 등을 지원합니다. 최신 커널은 지속적이고 빠르게 진화하며 POSIX를 넘어서는 많은 기능을 제공합니다.

Unix는 단일 파일 이름 및 파일 설명자 인터페이스 세트를 사용하여 여러 유형의 리소스(파일, 디렉토리 및 장치)에 대한 통합 액세스를 제공합니다. 이 아이디어는 더 많은 종류의 리소스로 확장될 수 있습니다. 좋은 예로는 "리소스는 파일이다"라는 개념을 네트워크, 그래픽 등에 적용한 Plan 9[16]가 있습니다. 그러나 대부분의 Unix 파생 운영 체제는 이 경로를 따르지 않았습니다.

파일 시스템과 파일 설명자는 강력한 추상화였습니다. 그렇더라도 운영 체제 인터페이스에 대한 다른 모델이 있습니다. Unix의 전신인 Multics는 파일 저장소를 메모리처럼 보이게 추상화하여 매우 다른 종류의 인터페이스를 만들어냈습니다. Multics 디자인의 복잡성은 Unix의 설계자에게 직접적인 영향을 미쳐 더 간단한 것을 만들고자 했습니다.

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

이 책에서는 xv6가 유닉스와 유사한 인터페이스를 구현하는 방법을 살펴보지만, 아이디어와 개념은 유닉스에만 적용되는 것이 아닙니다. 모든 운영 체제는 프로세스를 기본 하드웨어에 멀티플렉스하고, 프로세스를 서로 분리하고, 제어된 프로세스 간 통신을 위한 메커니즘을 제공해야 합니다. xv6를 공부한 후에는 다른 더 복잡한 운영 체제를 살펴보고 해당 시스템에서도 xv6의 기본 개념을 파악할 수 있어야 합니다.

1.6 연습문제

1. UNIX 시스템 호출을 사용하여 두 프로세스 간에 바이트를 "핑퐁"하는 프로그램을 작성합니다. 파이프 한 쌍을 통해 각 방향에 하나씩. 초당 교환 횟수로 프로그램의 성능을 측정합니다.

2장

운영 체제 조직

운영 체제의 핵심 요구 사항은 여러 활동을 동시에 지원하는 것입니다. 예를 들어, 1장에서 설명한 시스템 호출 인터페이스를 사용하면 프로세스가 fork로 새 프로세스를 시작할 수 있습니다.
운영 체제는 이러한 프로세스 간에 컴퓨터 리소스를 시간 공유해야 합니다. 예를 들어, 하드웨어 CPU보다 프로세스가 많더라도 운영 체제는 모든 프로세스가 실행할 기회를 얻도록 해야 합니다. 운영 체제는 또한 프로세스 간의 격리를 마련해야 합니다. 즉, 한 프로세스에 버그가 있고 오작동하더라도 버그가 있는 프로세스에 의존하지 않는 프로세스에는 영향을 미치지 않아야 합니다. 그러나 완전한 격리는 프로세스가 의도적으로 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 이 세 가지 요구 사항을 달성하기 위해 어떻게 구성되어 있는지에 대한 개요를 제공합니다. 그렇게 하는 방법은 여러 가지가 있지만, 이 텍스트에서는 많은 Unix 운영 체제에서 사용하는 모놀리식 커널을 중심으로 한 주류 설계에 초점을 맞춥니다. 이 장에서는 또한 xv6에서 격리의 단위인 xv6 프로세스와 xv6가 시작될 때 첫 번째 프로세스를 만드는 방법에 대한 개요를 제공합니다.

Xv6는 다중 코어[1] RISC-V 마이크로프로세서에서 실행되며, 저수준 기능(예: 프로세스 구현)의 대부분은 RISC-V에 특화되어 있습니다. RISC-V는 64비트 CPU이고 xv6는 "LP64" C로 작성되었습니다. 즉, C 프로그래밍 언어의 long(L)과 포인터(P)는 64비트이지만 int는 32비트입니다. 이 책에서는 독자가 일부 아키텍처에서 약간의 머신 수준 프로그래밍을 수행했다고 가정하고 RISC-V 관련 아이디어가 나오면 소개합니다. RISC-V에 대한 유용한 참고 자료는 "The RISC-V Reader: An Open Architecture Atlas"[15]입니다. 사용자 수준 ISA[2]와 권한 있는 아키텍처[3]는 공식 사양입니다.

완전한 컴퓨터의 CPU는 지원 하드웨어로 둘러싸여 있으며, 그 대부분은 I/O 인터페이스 형태입니다. Xv6는 qemu의 "-machine virt" 옵션으로 시뮬레이션된 지원 하드웨어를 위해 작성되었습니다. 여기에는 RAM, 부팅 코드가 포함된 ROM, 사용자 키보드/화면에 대한 직렬 연결 및 저장용 디스크가 포함됩니다.

1이 텍스트에서 "멀티 코어"는 메모리를 공유하지만 병렬로 실행되는 여러 CPU를 의미하며, 각각 고유한 레지스터 세트를 갖습니다. 이 텍스트에서는 멀티 코어의 동의어로 멀티 프로세서라는 용어를 사용하지만, 멀티 프로세서는 여러 개의 개별 프로세서 칩이 있는 컴퓨터를 보다 구체적으로 지칭할 수도 있습니다.

2.1 물리적 자원 추상화

운영 체제를 접했을 때 가장 먼저 떠오르는 질문은 왜 그것을 가지고 있는가? 즉, 그림 1.2의 시스템 호출을 라이브러리로 구현하여 애플리케이션이 이를 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 실행되는 경우 애플리케이션이 제대로 작동해야 한다는 것입니다. 예를 들어, 각 애플리케이션은 다른 애플리케이션이 실행될 수 있도록 주기적으로 CPU를 포기해야 합니다. 이러한 협력적 시간 공유 방식은 모든 애플리케이션이 서로를 신뢰하고 버그가 없는 경우 괜찮을 수 있습니다. 애플리케이션이 서로를 신뢰하지 않고 버그가 있는 경우가 더 일반적이므로 협력적 방식이 제공하는 것보다 더 강력한 격리를 원하는 경우가 많습니다.

강력한 격리를 달성하려면 애플리케이션이 민감한 하드웨어 리소스에 직접 액세스하는 것을 금지하고 대신 리소스를 서비스로 추상화하는 것이 좋습니다. 예를 들어, Unix 애플리케이션은 디스크를 직접 읽고 쓰는 대신 파일 시스템의 열기, 읽기, 쓰기 및 닫기 시스템 호출을 통해서만 저장소와 상호 작용합니다. 이를 통해 애플리케이션은 경로명의 편의성을 얻을 수 있으며 운영 체제(인터페이스의 구현자로서)가 디스크를 관리할 수 있습니다. 격리가 문제가 되지 않더라도 의도적으로 상호 작용하는 프로그램(또는 서로 방해하지 않으려는 프로그램)은 디스크를 직접 사용하는 것보다 파일 시스템을 더 편리한 추상화로 여길 가능성이 높습니다.

마찬가지로, Unix는 프로세스 간에 하드웨어 CPU를 투명하게 전환하여 필요에 따라 레지스터 상태를 저장하고 복원하므로 애플리케이션이 시간 공유를 알 필요가 없습니다. 이러한 투명성 덕분에 일부 애플리케이션이 무한 루프에 있더라도 운영 체제가 CPU를 공유할 수 있습니다.

또 다른 예로, 유닉스 프로세스는 물리적 메모리와 직접 상호 작용하는 대신 exec를 사용하여 메모리 이미지를 구축합니다. 이를 통해 운영 체제는 프로세스를 메모리에 배치할 위치를 결정할 수 있습니다. 메모리가 부족한 경우 운영 체제는 프로세스의 일부 데이터를 디스크에 저장할 수도 있습니다. exec는 또한 사용자에게 실행 가능한 프로그램 이미지를 저장할 수 있는 파일 시스템의 편의성을 제공합니다.

Unix 프로세스 간의 많은 형태의 상호작용은 파일 디스크립터를 통해 발생합니다. 파일 디스크립터는 많은 세부 사항(예: 파이프나 파일의 데이터가 저장된 위치)을 추상화할 뿐만 아니라 상호작용을 단순화하는 방식으로 정의됩니다. 예를 들어, 파이프라인의 한 애플리케이션이 실패하면 커널은 파이프라인의 다음 프로세스에 대한 파일 끝 신호를 생성합니다.

그림 1.2의 시스템 호출 인터페이스는 프로그래머의 편의성과 강력한 격리 가능성을 모두 제공하도록 신중하게 설계되었습니다. Unix 인터페이스는 리소스를 추상화하는 유일한 방법은 아니지만 매우 좋은 방법임이 입증되었습니다.

2.2 사용자 모드, 감독자 모드 및 시스템 호출

강력한 격리에는 애플리케이션과 운영 체제 사이에 엄격한 경계가 필요합니다. 애플리케이션이 실수를 하면 운영 체제가 실패하거나 다른 애플리케이션이 실패하는 것을 원하지 않습니다.

실패합니다. 대신 운영 체제는 실패한 애플리케이션을 정리하고 다른 애플리케이션을 계속 실행할 수 있어야 합니다. 강력한 격리를 달성하기 위해 운영 체제는 애플리케이션이 운영 체제의 데이터 구조와 명령을 수정(또는 읽기)할 수 없고 애플리케이션이 다른 프로세스의 메모리에 액세스할 수 없도록 해야 합니다.

CPU는 강력한 격리를 위해 하드웨어 지원을 제공합니다. 예를 들어, RISC-V에는 CPU가 명령을 실행할 수 있는 세 가지 모드가 있습니다. 머신 모드, 슈퍼바이저 모드, 사용자 모드입니다. 머신 모드에서 실행되는 명령은 전체 권한을 가지고 있습니다. CPU는 머신 모드에서 시작합니다. 머신 모드는 주로 컴퓨터를 구성하기 위한 것입니다. Xv6는 머신 모드에서 몇 줄을 실행한 다음 슈퍼바이저 모드로 변경합니다.

감독자 모드에서 CPU는 특권 명령을 실행할 수 있습니다.예를 들어, 인터럽트 활성화 및 비활성화, 페이지 테이블 주소를 보관하는 레지스터 읽기 및 쓰기 등입니다.사용자 모드의 애플리케이션이 특권 명령을 실행하려고 하면 CPU는 명령을 실행하지 않고 감독자 모드로 전환하여 감독자 모드 코드가 애플리케이션을 종료할 수 있도록 합니다.왜냐하면 해당 애플리케이션이 해서는 안 될 일을 했기 때문입니다.1장의 그림 1.1은 이러한 구성을 보여줍니다.애플리케이션은 사용자 모드 명령(예: 숫자 더하기 등)만 실행할 수 있으며 사용자 공간에서 실행 중이라고 하는 반면, 감독자 모드의 소프트웨어는 특권 명령을 실행할 수도 있으며 커널 공간에서 실행 중이라고 합니다.커널 공간(또는 감독자 모드)에서 실행되는 소프트웨어를 커널이라고 합니다.

커널 함수(예: xv6의 read 시스템 호출) 를 호출하려는 애플리케이션은 커널로 전환해야 합니다. 애플리케이션은 커널 함수를 직접 호출할 수 없습니다. CPU는 CPU를 사용자 모드에서 감독자 모드로 전환하고 커널에서 지정한 진입점에서 커널에 진입하는 특수 명령을 제공합니다. (RISC-V는 이 목적을 위해 ecall 명령을 제공합니다.)

CPU가 슈퍼바이저 모드로 전환되면 커널은 시스템 호출의 인수를 검증할 수 있습니다(예: 시스템 호출에 전달된 주소가 애플리케이션 메모리의 일부인지 확인). 애플리케이션이 요청된 작업을 수행할 수 있는지 여부를 결정(예: 애플리케이션이 지정된 파일을 쓸 수 있는지 확인)한 다음 거부하거나 실행할 수 있습니다. 커널이 슈퍼바이저 모드로의 전환을 위한 진입점을 제어하는 것이 중요합니다. 애플리케이션이 커널 진입점을 결정할 수 있다면 악성 애플리케이션이 인수 검증을 건너뛰는 지점에서 커널에 진입할 수 있습니다.

2.3 커널 구성

핵심 설계 질문은 운영 체제의 어떤 부분이 슈퍼바이저 모드에서 실행되어야 하는가입니다. 한 가지 가능성은 전체 운영 체제가 커널에 상주하여 모든 시스템 호출의 구현이 슈퍼바이저 모드에서 실행된다는 것입니다. 이 조직을 모놀리식 커널이라고 합니다.

이 조직에서 전체 운영 체제는 전체 하드웨어 권한으로 실행됩니다. 이 조직은 OS 설계자가 운영 체제의 어느 부분에 전체 하드웨어 권한이 필요하지 않은지 결정할 필요가 없기 때문에 편리합니다. 게다가 운영 체제의 다른 부분이 협력하기가 더 쉽습니다. 예를 들어, 운영 체제에는 파일 시스템과 가상 메모리 시스템이 모두 공유할 수 있는 버퍼 캐시가 있을 수 있습니다.

모놀리식 조직의 단점은 운영 체제의 여러 부분 간의 인터페이스가 종종 복잡하다는 것입니다(이 텍스트의 나머지 부분에서 볼 수 있듯이).

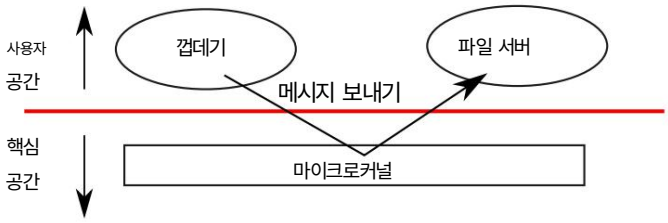

그림 2.1: 파일 시스템 서버가 있는 마이크로커널

운영 체제 개발자가 실수를 하기 쉽습니다. 모놀리식 커널에서 실수는 치명적입니다. 감독자 모드에서 오류가 발생하면 커널이 실패하는 경우가 많기 때문입니다. 커널이 실패하면 컴퓨터가 작동을 멈추고 모든 애플리케이션도 실패합니다. 컴퓨터를 재부팅해야 다시 시작할 수 있습니다.

커널에서 실수 위험을 줄이기 위해 OS 설계자는 슈퍼바이저 모드에서 실행되는 운영 체제 코드의 양을 최소화하고 운영 체제의 대부분을 사용자 모드에서 실행할 수 있습니다. 이 커널 조직을 마이크로커널이라고 합니다.

그림 2.1은 이 마이크로커널 설계를 보여줍니다. 그림에서 파일 시스템은 사용자 수준 프로세스로 실행됩니다. 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. 예를 들어, 셸과 같은 애플리케이션이 파일을 읽거나 쓰려면 파일 서버로 메시지를 보내고 응답을 기다립니다.

마이크로커널에서 커널 인터페이스는 애플리케이션 시작, 메시지 전송, 장치 하드웨어 액세스 등을 위한 몇 가지 저수준 기능으로 구성됩니다. 이러한 구성 덕분에 대부분의 운영 체제가 사용자 수준 서버에 상주하기 때문에 커널이 비교적 단순해집니다.

실제 세계에서는 모놀리식 커널과 마이크로커널이 모두 인기가 있습니다. 많은 유닉스 커널이 모놀리식입니다. 예를 들어, 리눅스는 모놀리식 커널을 가지고 있지만, 일부 OS 기능은 사용자 수준 서버(예: 윈도잉 시스템)로 실행됩니다. 리눅스는 OS 집약적 애플리케이션에 높은 성능을 제공하는데, 그 이유는 부분적으로 커널의 하위 시스템이 긴밀하게 통합될 수 있기 때문입니다.

Minix, L4, QNX와 같은 운영 체제는 서버가 있는 마이크로커널로 구성되며 임베디드 설정에서 널리 배포되었습니다. L4의 변형인 seL4는 메모리 안전 및 기타 보안 속성에 대해 검증되었을 정도로 작습니다[8].

운영 체제 개발자들 사이에서는 어느 조직이 더 나은지에 대한 많은 논쟁이 있으며, 어느 쪽이든 결정적인 증거는 없습니다. 더욱이, 그것은 "더 나은" 것이 무엇을 의미하는지에 크게 달려 있습니다: 더 빠른 성능, 더 작은 코드 크기, 커널의 안정성, 전체 운영 체제(사용자 수준 서비스 포함)의 안정성 등.

어떤 조직인지에 대한 질문보다 더 중요할 수 있는 실질적인 고려 사항도 있습니다. 일부 운영 체제는 마이크로커널을 가지고 있지만 성능상의 이유로 일부 사용자 수준 서비스를 커널 공간에서 실행합니다. 일부 운영 체제는 모놀리식 커널을 가지고 있는데, 그것이 시작 방식이기 때문이고 순수한 마이크로커널 조직으로 전환할 인센티브가 거의 없기 때문입니다. 새로운 기능이 기존 운영 체제를 마이크로커널 설계에 맞게 다시 작성하는 것보다 더 중요할 수 있기 때문입니다.

이 책의 관점에서 보면 마이크로커널과 모놀리식 운영 체제는 많은 핵심 아이디어를 공유합니다. 시스템 호출을 구현하고, 페이지 테이블을 사용하고, 인터럽트를 처리하고,

파일	설명
bio.c	파일 시스템을 위한 디스크 블록 캐시.
console.c	사용자 키보드와 화면에 연결합니다.
entry.S	첫 번째 부팅 지침입니다.
임원.c	exec() 시스템 호출.
파일.c	파일 설명자 지원.
fs.c	파일 시스템.
kalloc.c	물리적 페이지 할당자.
kernelvec.S	커널의 트랩과 타이머 인터럽트를 처리합니다.
log.c	파일 시스템 로깅 및 충돌 복구.
main.c	부팅하는 동안 다른 모듈의 초기화를 제어합니다.
파이프.c	파이프.
plic.c	RISC-V 인터럽트 컨트롤러.
printf.c	콘솔에 형식화된 출력을 표시합니다.
proc.c	프로세스 및 일정.
sleeplock.c	CPU를 양보하는 잠금입니다.
spinlock.c	CPU를 양보하지 않는 잠금입니다.
start.c	초기 머신 모드 부팅 코드.
string.c	C 문자열 및 바이트 배열 라이브러리.
swtch.S	스레드 전환.
syscall.c	시스템 호출을 처리 함수로 전송합니다.
sysfile.c	파일 관련 시스템 호출.
sysproc.c	프로세스 관련 시스템 호출.
trampoline.S	사용자와 커널 사이를 전환하는 어셈블리 코드.
trap.c	트랩과 인터럽트를 처리하고 반환하는 C 코드입니다.
유아트.c	직렬 포트 콘솔 장치 드라이버.
virtio_disk.c	디스크 장치 드라이버.
VM.C	페이지 테이블과 주소 공간을 관리합니다.

그림 2.2: Xv6 커널 소스 파일.

프로세스, 그들은 동시성 제어를 위해 잠금을 사용하고 파일 시스템을 구현합니다. 이 책은
이런 핵심 아이디어에 초점을 맞춥니다.

Xv6는 대부분의 Unix 운영 체제와 마찬가지로 모놀리식 커널로 구현됩니다. 따라서 xv6
커널 인터페이스는 운영 체제 인터페이스에 해당하며 커널은 완전한 운영 체제를 구현합니다. xv6는 많은 서비스를 제공하지 않으므로 커
널은 일부보다 작습니다.
마이크로커널이지만 개념적으로 xv6는 일체형입니다.

2.4 코드: xv6 조직

xv6 커널 소스는 kernel/ 하위 디렉토리에 있습니다. 소스는 다음과 같은 파일로 나뉩니다.
모듈성의 대략적인 개념; 그림 2.2는 파일을 나열합니다. 모듈 간 인터페이스는 다음에 정의됩니다.

그림 2.3: 프로세스의 가상 주소 공간 레이아웃

defs.h (커널/defs.h).

2.5 프로세스 개요

xv6(다른 유닉스 운영 체제와 마찬가지로)의 격리 단위는 프로세스입니다. 프로세스 추상화는 한 프로세스가 다른 프로세스의 메모리, CPU, 파일 설명자 등을 파괴하거나 감시하는 것을 방지합니다. 또한 프로세스가 커널 자체를 파괴하는 것도 방지하여 프로세스가 커널의 격리 메커니즘을 파괴할 수 없습니다. 커널은 버그가 있거나 악의적인 애플리케이션이 커널이나 하드웨어를 속여 나쁜 짓(예: 격리 우회)을 할 수 있으므로 프로세스 추상화를 신중하게 구현해야 합니다. 커널이 프로세스를 구현하는 데 사용하는 메커니즘에는 사용자/감독자 모드 플래그, 주소 공간, 스레드의 시간 분할이 포함됩니다.

격리를 강화하기 위해 프로세스 추상화는 프로그램이 자체 개인 머신을 가지고 있다는 환상을 제공합니다. 프로세스는 다른 프로세스가 읽거나 쓸 수 없는 개인 메모리 시스템 또는 주소 공간처럼 보이는 것을 프로그램에 제공합니다. 프로세스는 또한 프로그램의 명령을 실행하기 위해 자체 CPU처럼 보이는 것을 프로그램에 제공합니다.

Xv6는 페이지 테이블(하드웨어로 구현됨)을 사용하여 각 프로세스에 고유한 주소 공간을 제공합니다. RISC-V 페이지 테이블은 가상 주소(RISC-V 명령어가 조작하는 주소)를 물리적 주소(CPU 칩이 주 메모리로 보내는 주소)로 변환(또는 "맵핑")합니다.

Xv6는 각 프로세스에 대해 해당 프로세스의 주소 공간을 정의하는 별도의 페이지 테이블을 유지 관리합니다.
그림 2.3에서 설명한 대로 주소 공간에는 가상 주소 0에서 시작하는 프로세스의 사용자 메모리가 포함됩니다. 명령어가 먼저 오고, 그 다음에 전역 변수가 오고, 그 다음에 스택이 오고, 마지막으로 프로세스가 필요에 따라 확장할 수 있는 "힙" 영역(malloc용)이 옵니다. 프로세스 주소 공간의 최대 크기를 제한하는 요인에는 여러 가지가 있습니다. RISC-V의 포인터는 64비트 폭이고, 하드웨어는 페이지 테이블에서 가상 주소를 찾을 때 하위 39비트만 사용하고, xv6는 이 39비트 중 38비트만 사용합니다. 따라서 최대 주소는 2 - 1 = 0x3fffffffff입니다.

MAXVA (커널/riscv.h:363). 주소 공간의 맨 위에 xv6는 트램폴린을 위한 페이지와 프로세스의 트랩프레임을 매핑하는 페이지를 예약합니다. Xv6는 이 두 페이지를 사용하여 커널로 전환하고 다시 되돌립니다. 트램폴린 페이지에는 커널로 전환하고 커널에서 나가는 코드가 들어 있으며 트랩프레임을 매핑하는 것은 사용자 프로세스의 상태를 저장/복원하는 데 필요합니다. 4장에서 설명하겠습니다.

xv6 커널은 각 프로세스에 대한 많은 상태를 유지 관리하고 이를 struct proc 에 수집합니다.
(커널/proc.h:85). 프로세스의 가장 중요한 커널 상태는 페이지 테이블, 커널 스택, 실행 상태입니다. proc 구조 의 요소를 참조하기 위해 p->xxx 표기법을 사용합니다 . 예를 들어, p->pagetable은 프로세스의 페이지 테이블에 대한 포인터입니다.

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 두 개의 스택이 있습니다. 사용자 스택과 커널 스택 (p->kstack)입니다. 프로세스가 사용자 명령어를 실행할 때는 사용자 스택만 사용 중이고 커널 스택은 비어 있습니다. 프로세스가 커널에 들어가면(시스템 호출 또는 인터럽트를 위해) 커널 코드가 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 보호되므로 프로세스가 사용자 스택을 망가뜨린 경우에도 커널이 실행될 수 있습니다.

프로세스는 RISC-V ecall 명령어를 실행하여 시스템 호출을 할 수 있습니다 . 이 명령어는 하드웨어 권한 수준을 높이고 프로그램 카운터를 커널 정의 진입점으로 변경합니다.
진입점의 코드는 커널 스택으로 전환하고 시스템 호출을 구현하는 커널 명령어를 실행합니다. 시스템 호출이 완료되면 커널은 사용자 스택으로 다시 전환하고 sret 명령어를 호출하여 사용자 공간으로 돌아갑니다. 이 명령어는 하드웨어 권한 수준을 낮추고 시스템 호출 명령어 바로 다음에 사용자 명령어 실행을 재개합니다. 프로세스의 스레드는 커널에서 "차단"하여 I/O를 기다리고 I/O가 완료되면 중단한 곳에서 다시 시작할 수 있습니다.

p->state는 프로세스가 할당되었는지, 실행할 준비가 되었는지, 실행 중인지, I/O를 기다리고 있는지, 종료 중인지를 나타냅니
다. p->pagetable은 RISC-V 하드웨어가 예상하는 형식으로 프로세스의 페이지 테이블을 보관합니다. Xv6는 페이징 하드웨어가 사용자 공간에서 해당 프로세스를 실행할 때 프로세스의 p->pagetable을 사용하도록 합니다 . 프로세스의 페이지 테이블은 프로세스의 메모리를 저장하기 위해 할당된 물리적 페이지의 주소 레코드 역할도 합니다.

요약하자면, 프로세스는 두 가지 설계 아이디어를 묶습니다. 프로세스에 자체 메모리의 환상을 주는 주소 공간과 프로세스에 자체 CPU의 환상을 주는 스레드입니다. xv6에서 프로세스는 하나의 주소 공간과 하나의 스레드로 구성됩니다. 실제 운영 체제에서 프로세스는 여러 CPU를 활용하기 위해 두 개 이상의 스레드를 가질 수 있습니다.

2.6 코드: xv6 시작, 첫 번째 프로세스 및 시스템 호출

xv6를 더 구체적으로 만들기 위해 커널이 첫 번째 프로세스를 시작하고 실행하는 방법을 간략히 설명하겠습니다. 이후 장에서는 이 개요에 나타나는 메커니즘을 더 자세히 설명합니다.

RISC-V 컴퓨터가 켜지면 자체를 초기화하고 읽기 전용 메모리에 저장된 부트 로더를 실행합니다. 부트 로더는 xv6 커널을 메모리에 로드합니다. 그런 다음 머신 모드에서 CPU는 _entry (kernel/entry.S:7)에서 시작하여 xv6를 실행합니다. RISC-V는 페이징 하드웨어를 비활성화한 상태에서 시작합니다. 가상 주소는 물리적 주소에 직접 매핑됩니다.

로더는 xv6 커널을 물리 주소 0x80000000의 메모리에 로드합니다. 커널을 0x0이 아닌 0x80000000에 배치하는 이유는 주소 범위 0x0:0x80000000에 I/O 장치가 포함되어 있기 때문입니다.

_entry 의 명령어는 xv6가 C 코드를 실행할 수 있도록 스택을 설정합니다. Xv6는 파일 start.c(kernel/start.c:11)에서 초기 스택인 stack0에 대한 공간을 선언합니다 . _entry 의 코드는 스택 포인터 레지스터 sp를 주소 stack0+4096, 즉 스택의 맨 위로 로드합니다. RISC-V의 스택은 아래로 커지기 때문입니다. 이제 커널에 스택이 있으므로 _entry는 시작 시 C 코드 (kernel/start.c:21) 를 호출합니다 .

함수 start는 머신 모드에서만 허용되는 일부 구성을 수행한 다음, 슈퍼바이저 모드로 전환합니다. 슈퍼바이저 모드로 들어가기 위해 RISC-V는 명령어 mret를 제공합니다. 이 명령어는 슈퍼바이저 모드에서 머신 모드로 이전 호출에서 복귀하는 데 가장 많이 사용됩니다. start는 이러한 호출에서 복귀하지 않고, 마치 호출이 있었던 것처럼 설정합니다. 이전 권한 모드를 레지스터 mstatus에서 supervisor로 설정하고, main의 주소를 레지스터 mepc에 써서 복귀 주소를 main 으로 설정하고 , 페이지 테이블 레지스터 satp 에 0을 써서 슈퍼바이저 모드에서 가상 주소 변환을 비활성화하고 , 모든 인터럽트와 예외를 슈퍼바이저 모드에 위임합니다.

슈퍼바이저 모드로 점프하기 전에 start는 한 가지 작업을 더 수행합니다. 타이머 인터럽트를 생성하도록 클록 칩을 프로그래밍합니다. 이 정리 작업을 마치면 start는 mret를 호출하여 슈퍼바이저 모드로 "돌아갑니다" . 그러면 프로그램 카운터가 main (kernel/main.c:11) 으로 변경됩니다 .

메인 (kernel/main.c:11) 이후 여러 장치와 하위 시스템을 초기화하고 userinit (kernel/proc.c:233)을 호출하여 첫 번째 프로세스를 생성합니다 . 첫 번째 프로세스는 RISC-V 어셈블리로 작성된 작은 프로그램을 실행하는데, 이 프로그램은 xv6에서 첫 번째 시스템 호출을 수행합니다. initcode.S (user/initcode.S:3) exec 시스템 호출인 SYS_EXEC (kernel/syscall.h:8) 에 대한 번호를 로드합니다 . 레지스터 a7에 넣은 다음 ecall을 호출하여 커널에 다시 들어갑니다.

커널은 syscall (kernel/syscall.c:132) 의 레지스터 a7에 있는 숫자를 사용합니다. 원하는 시스템 호출을 호출합니다. 시스템 호출 테이블 (kernel/syscall.c:107) SYS_EXEC를 커널이 호출하는 sys_exec 에 매핑합니다 . 1장에서 보았듯이 exec는 현재 프로세스의 메모리와 레지스터를 새 프로그램(이 경우 /init)으로 대체합니다.

커널이 exec를 완료하면 /init 프로세스 의 사용자 공간으로 돌아갑니다 . init (user/init.c:15) 필요한 경우 새 콘솔 장치 파일을 만든 다음 파일 설명자 0, 1, 2로 엽니다. 그런 다음 콘솔에서 셸을 시작합니다. 시스템이 가동됩니다.

2.7 보안 모델

운영 체제가 버그가 있거나 악성 코드를 어떻게 처리하는지 궁금할 수 있습니다. 악의에 대처하는 것이 우연한 버그를 처리하는 것보다 엄격히 어렵기 때문에 이 주제를 보안과 관련된 것으로 보는 것이 합리적입니다. 운영 체제 설계에서 일반적인 보안 가정과 목표에 대한 고차원적 관점은 다음과 같습니다.

운영 체제는 프로세스의 사용자 수준 코드가 커널이나 다른 프로세스를 파괴하기 위해 최선을 다할 것이라고 가정해야 합니다. 사용자 코드는 허용된 주소 공간 외부의 포인터를 역참조하려고 할 수 있습니다. 사용자 코드에 의도되지 않은 RISC-V 명령어를 실행하려고 할 수 있습니다. RISC-V 제어 레지스터를 읽고 쓰려고 할 수 있습니다. 장치 하드웨어에 직접 액세스하려고 할 수 있습니다. 커널을 속여 충돌시키거나 어리석은 일을 하도록 하기 위해 시스템 호출에 영리한 값을 전달할 수 있습니다. 커널의 목표는 각 사용자 프로세스를 제한하여 자체 사용자 메모리를 읽고 쓰고 실행하고, 32개의 범용 RISC-V 레지스터를 사용하고, 시스템 호출이 허용하도록 의도된 방식으로 커널과 다른 프로세스에 영향을 미치는 것입니다. 커널은 다른 모든 동작을 방지해야 합니다. 이는 일반적으로 커널 설계에서 절대적인 요구 사항입니다.

커널 자체 코드에 대한 기대는 상당히 다릅니다. 커널 코드는 선의의 신중한 프로그래머가 작성한 것으로 가정합니다. 커널 코드는 버그가 없고, 확실히 악성 코드가 없어야 합니다. 이 가정은 커널 코드를 분석하는 방법에 영향을 미칩니다. 예를 들어, 커널 코드에서 잘못 사용하면 심각한 문제를 일으킬 수 있는 내부 커널 함수(예: 스핀 잠금)가 많이 있습니다. 특정 커널 코드를 검사할 때 올바르게 동작한다고 스스로 확신해야 합니다. 그러나 일반적으로 커널 코드는 올바르게 작성되었으며 커널 자체 함수와 데이터 구조 사용에 대한 모든 규칙을 따른다고 가정합니다. 하드웨어 수준에서 RISC-V CPU, RAM, 디스크 등은 하드웨어 버그 없이 설명서에 광고된 대로 작동한다고 가정합니다.

물론 실제 생활에서는 그렇게 간단하지 않습니다. 영리한 사용자 코드가 디스크 공간, CPU 시간, 프로세스 테이블 슬롯 등 커널에서 보호하는 리소스를 소모하여 시스템을 사용할 수 없게 만들거나 패닉을 일으키는 것을 방지하는 것은 어렵습니다. 버그 없는 코드를 작성하거나 버그 없는 하드웨어를 설계하는 것은 일반적으로 불가능합니다. 악의적인 사용자 코드 작성자가 커널 또는 하드웨어 버그를 알고 있다면 이를 악용할 것입니다. Linux와 같이 성숙하고 널리 사용되는 커널에서도 사람들은 계속해서 새로운 취약점을 발견합니다[1]. 버그가 있을 가능성에 대비하여 커널에 보호 장치를 설계하는 것이 좋습니다. 어설션, 유형 검사, 스택 가드 페이지 등이 있습니다. 마지막으로, 사용자 코드와 커널 코드의 구별은 때때로 모호합니다. 일부 권한이 있는 사용자 수준 프로세스는 필수 서비스를 제공하고 효과적으로 운영 체제의 일부가 될 수 있으며, 일부 운영 체제에서는 권한이 있는 사용자 코드가 커널에 새 코드를 삽입할 수 있습니다(Linux의 로드 가능한 커널 모듈의 경우처럼).

2.8 현실 세계

대부분의 운영 체제는 프로세스 개념을 채택했으며, 대부분의 프로세스는 xv6의 프로세스와 유사합니다. 그러나 최신 운영 체제는 프로세스 내에서 여러 스레드를 지원하여 단일 프로세스가 여러 CPU를 활용할 수 있도록 합니다. 프로세스에서 여러 스레드를 지원하려면 xv6에 없는 상당한 양의 기계가 필요합니다. 여기에는 잠재적인 인터페이스 변경(예: Linux의 복제본, fork 의 변형) 이 포함되어 프로세스 스레드가 공유하는 측면을 제어합니다.

2.9 연습문제

1. xv6에 사용 가능한 메모리 양을 반환하는 시스템 호출을 추가합니다.

3장

페이지 테이블

페이지 테이블은 운영 체제가 각 프로세스에 자체 개인 주소 공간과 메모리를 제공하는 가장 인기 있는 메커니즘입니다. 페이지 테이블은 메모리 주소가 무엇을 의미하는지, 그리고 물리적 메모리의 어떤 부분에 액세스할 수 있는지 결정합니다. 이를 통해 xv6는 다른 프로세스의 주소 공간을 분리하고 이를 단일 물리적 메모리로 멀티플렉싱할 수 있습니다. 페이지 테이블은 운영 체제가 많은 트릭을 수행할 수 있도록 하는 간접성 수준을 제공하기 때문에 인기 있는 디자인입니다. Xv6는 몇 가지 트릭을 수행합니다. 여러 주소 공간에서 동일한 메모리(트램폴린 페이지)를 매핑하고 매핑되지 않은 페이지로 커널 및 사용자 스택을 보호합니다. 이 장의 나머지 부분에서는 RISC-V 하드웨어가 제공하는 페이지 테이블과 xv6가 이를 사용하는 방법을 설명합니다.

3.1 페이징 하드웨어

상기시켜드리자면, RISC-V 명령어(사용자와 커널 모두)는 가상 주소를 조작합니다. 머신의 RAM 또는 물리적 메모리는 물리적 주소로 색인됩니다. RISC-V 페이지 테이블 하드웨어는 각 가상 주소를 물리적 주소에 매핑하여 이 두 종류의 주소를 연결합니다.

Xv6는 Sv39 RISC-V에서 실행되며, 이는 64비트 가상 주소의 하위 39비트만 사용되고 상위 25비트는 사용되지 않는다는 것을 의미합니다. 이 Sv39 구성에서 RISC-V 페이지 테이블은 논리적으로 2(134,217,728)개의 페이지 테이블 항목(PTE) 배열입니다. 각 PTE에는 44비트의 물리적 페이지 번호(PPN)와 몇 가지 플래그가 포함되어 있습니다. 페이징 하드웨어는 39비트의 상위 27비트를 사용하여 페이지 테이블을 인덱싱하여 PTE를 찾고, 상위 44비트가 PTE의 PPN에서 나오고 하위 12비트가 원래 가상 주소에서 복사된 56비트 물리적 주소를 만들어 가상 주소를 변환합니다. 그림 3.1은 페이지 테이블을 PTE의 간단한 배열로 논리적으로 보는 이 프로세스를 보여줍니다(자세한 내용은 그림 3.2 참조). 페이지 테이블은 운영 체제가 4096(2 12) 바이트의 정렬된 청크 단위로 가상-물리 주소 변환을 제어할 수 있도록 합니다 . 이러한 청크를 페이지라고 합니다.

27

Sv39 RISC-V에서 가상 주소의 상위 25비트는 변환에 사용되지 않습니다. 물리적 주소도 확장할 여지가 있습니다. PTE 형식에는 물리적 페이지 번호가 10비트 더 확장될 수 있는 공간이 있습니다. RISC-V 설계자는 기술 예측에 따라 이러한 숫자를 선택했습니다. 2바이트는 512GB로, 이는 다음을 실행하는 애플리케이션에 충분한 주소 공간이어야 합니다.

39

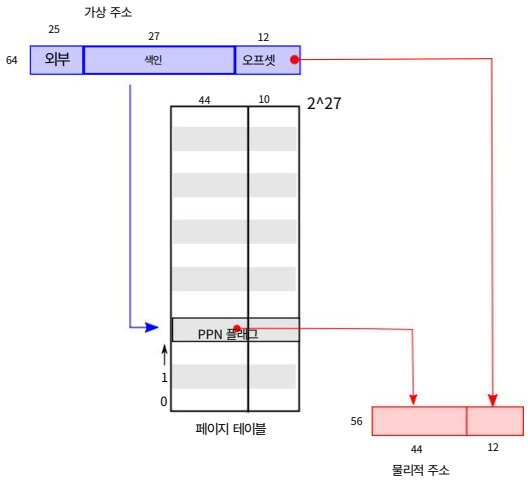

그림 3.1: 단순화된 논리적 페이지 테이블을 포함한 RISC-V 가상 및 물리 주소.

RISC-V 컴퓨터에서 2는 가까운 미래에 많은 I/O 장치와 DRAM 칩을 수용할 수 있는 충분한 물리적 메모리 공간입니다. 더 필요한 경우 RISC-V 설계자는 48비트 가상 주소[3]를 사용하여 Sv48을 정의했습니다.

그림 3.2에서 보듯이 RISC-V CPU는 3단계로 가상 주소를 물리 주소로 변환합니다.
페이지 테이블은 3단계 트리로 물리적 메모리에 저장됩니다. 트리의 루트는 512개의 PTE를 포함하는 4096바이트 페이지 테이블 페이지로, 트리의 다음 단계에 있는 페이지 테이블 페이지의 물리적 주소를 포함합니다. 각 페이지에는 트리의 마지막 단계에 대한 512개의 PTE가 포함됩니다.
페이징 하드웨어는 27비트 중 상위 9비트를 사용하여 루트 페이지 테이블 페이지에서 PTE를 선택하고, 중간 9비트를 사용하여 트리의 다음 레벨에 있는 페이지 테이블 페이지에서 PTE를 선택하고, 하위 9비트를 사용하여 최종 PTE를 선택합니다.
(Sv48 RISC-V에서 페이지 테이블은 4개 레벨로 구성되고, 가상 주소의 비트 39~47은 최상위 레벨로 색인됩니다.)

주소를 변환하는 데 필요한 세 개의 PTE 중 하나가 없으면 페이징 하드웨어
페이지 오류 예외가 발생하여 커널이 예외를 처리하게 됩니다(4장 참조).

그림 3.2의 3단계 구조는 그림 3.1의 단일 단계 설계와 비교하여 PTE를 기록하는 메모리 효율적인 방법을 허용합니다. 대규모 가상 주소에 매핑이 없는 일반적인 경우, 3단계 구조는 전체 페이지 디렉토리를 생략할 수 있습니다. 예를 들어, 애플리케이션이 주소 0에서 시작하는 몇 개의 페이지만 사용하는 경우 최상위 페이지 디렉토리의 항목 1~511은 유효하지 않으며 커널은 511개의 중간 페이지 디렉토리에 페이지를 할당할 필요가 없습니다. 더욱이 커널은 해당 511개의 중간 페이지 디렉토리에 대해 최하위 페이지 디렉토리에 대한 페이지를 할당할 필요도 없습니다. 따라서 이 예에서 3단계 설계는 중간 페이지 디렉토리에 511개의 페이지를 저장하고 최하위 페이지 디렉토리에 511×512개의 페이지를 저장합니다.

CPU는 로드 또는 저장 명령어를 실행하는 일부로 하드웨어에서 3단계 구조를 따르지만, 3단계의 잠재적인 단점은 CPU가 로드/저장 명령어의 가상 주소를 물리적 주소로 변환하기 위해 메모리에서 세 개의 PTE를 로드해야 한다는 것입니다.

물리적 메모리에서 PTE를 로드하는 비용을 피하기 위해 RISC-V CPU는 TLB(Translation Look-aside Buffer)에 페이지 테이블 항목을 캐시합니다.

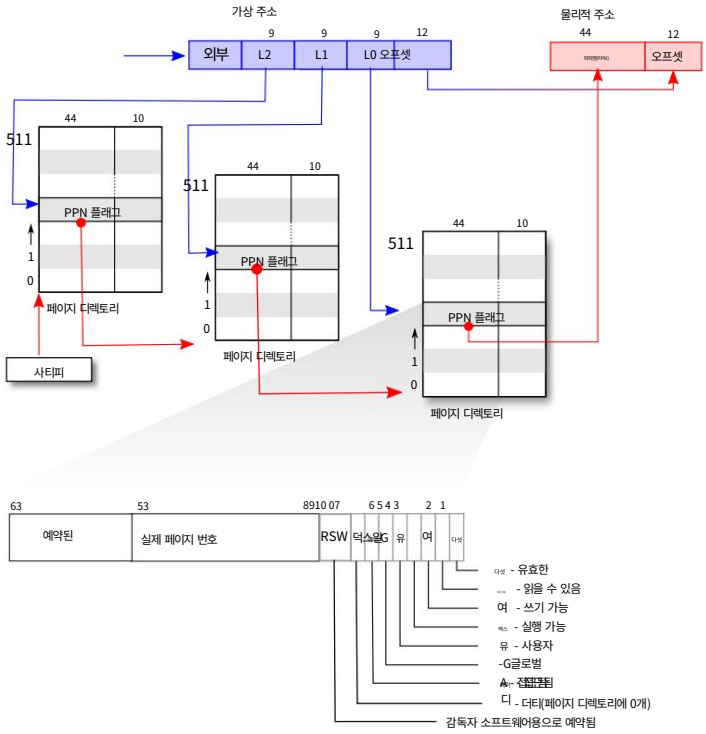

그림 3.2: RISC-V 주소 변환 세부 정보.

각 PTE에는 페이징 하드웨어에 연관된 가상 주소를 사용할 수 있는 방법을 알려주는 플래그 비트가 들어 있습니다. PTE_V는 PTE가 있는지 여부를 나타냅니다. 설정되지 않은 경우 페이지에 대한 참조가 예외를 발생시킵니다(즉, 허용되지 않음). PTE_R은 명령어가 페이지를 읽을 수 있는지 여부를 제어합니다. PTE_W는 명령어가 페이지에 쓸 수 있는지 여부를 제어합니다. PTE_X는 CPU가 페이지의 내용을 명령어로 해석하여 실행할 수 있는지 여부를 제어합니다.

PTE_U는 사용자 모드의 명령어가 페이지에 액세스할 수 있는지 여부를 제어합니다. PTE_U가 설정되지 않은 경우 PTE는 감독자 모드에서만 사용할 수 있습니다. 그림 3.2는 모든 작동 방식을 보여줍니다. 플래그와 다른 모든 페이지 하드웨어 관련 구조는 (kernel/riscv.h) 에 정의되어 있습니다.

CPU에 페이지 테이블을 사용하도록 지시하려면 커널이 루트 페이지 테이블 페이지의 물리적 주소를 satp 레지스터에 써야 합니다. CPU는 자체 satp가 가리키는 페이지 테이블을 사용하여 후속 명령어에서 생성된 모든 주소를 변환합니다. 각 CPU는 자체 satp를 가지고 있으므로 다른 CPU가 자체 페이지 테이블로 설명된 개인 주소 공간을 가진 다른 프로세스를 실행할 수 있습니다.

일반적으로 커널은 모든 물리적 메모리를 페이지 테이블에 매핑하여 로드/저장 명령을 사용하여 물리적 메모리의 모든 위치를 읽고 쓸 수 있습니다. 페이지 디렉토리가 물리적 메모리에 있으므로 커널은 표준 저장 명령을 사용하여 PTE의 가상 주소에 쓰면서 페이지 디렉토리에 있는 PTE의 내용을 프로그래밍할 수 있습니다.

용어에 대한 몇 가지 참고 사항. 물리적 메모리는 DRAM의 저장 셀을 말합니다. 물리적 메모리의 바이트에는 물리적 주소라고 하는 주소가 있습니다. 명령어는 가상 주소만 사용하는데, 페이징 하드웨어가 이를 물리적 주소로 변환한 다음 DRAM 하드웨어로 보내서 읽습니다.

또는 쓰기 저장소입니다. 실제 메모리 및 가상 주소와 달리 가상 메모리는 실제가 아닙니다.
객체이지만 커널이 관리하기 위해 제공하는 추상화 및 메커니즘의 컬렉션을 말합니다.
실제 메모리와 가상 주소.

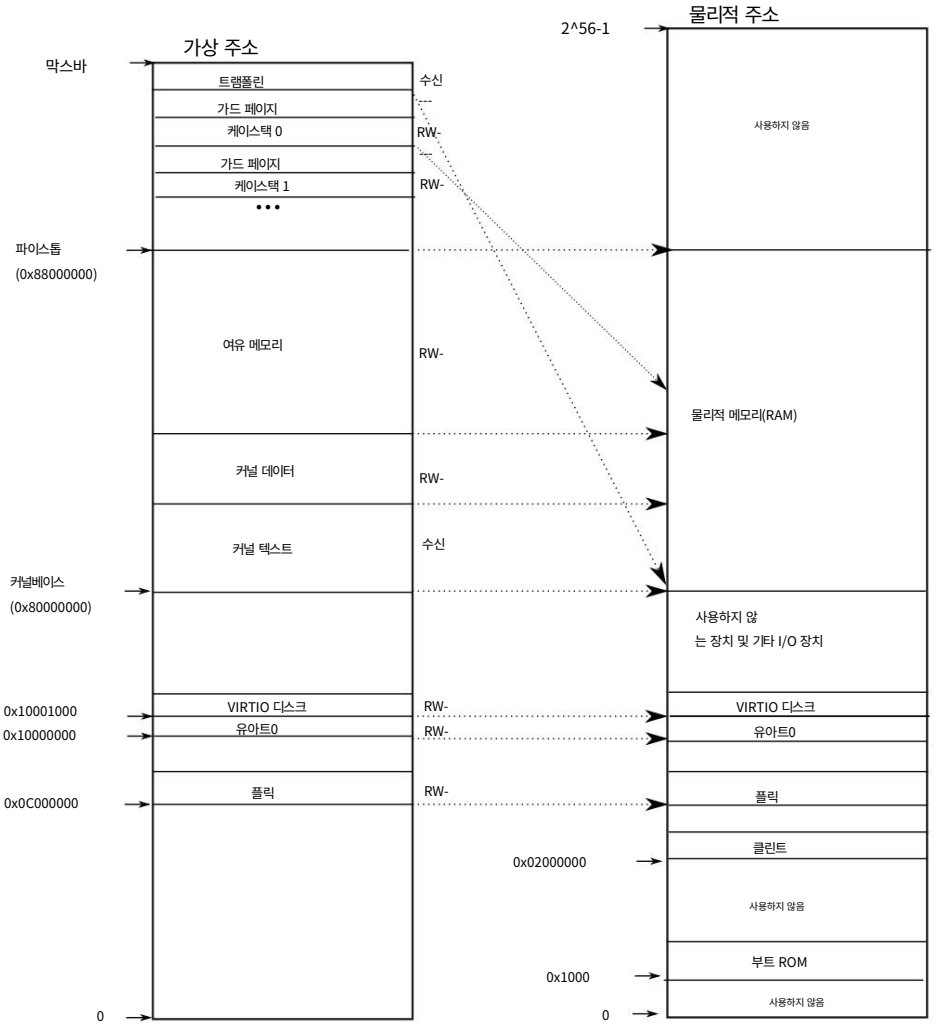

그림 3.3: 왼쪽은 xv6의 커널 주소 공간입니다. RWX는 PTE 읽기, 쓰기 및 실행을 나타냅니다.
권한. 오른쪽에는 xv6가 볼 것으로 예상하는 RISC-V 물리적 주소 공간이 있습니다.

3.2 커널 주소 공간

Xv6는 프로세스당 하나의 페이지 테이블을 유지 관리하여 각 프로세스의 사용자 주소 공간과 커널의 주소 공간을 설명하는 단일 페이지 테이블을 설명합니다. 커널은 주소 공간의 레이아웃을 구성하여 예측 가능한 속도로 물리적 메모리와 다양한 하드웨어 리소스에 액세스할 수 있도록 합니다.
가상 주소. 그림 3.3은 이 레이아웃이 커널 가상 주소를 물리적 주소에 매핑하는 방식을 보여줍니다. 파일 (kernel/memlayout.h) xv6의 커널 메모리 레이아웃에 대한 상수를 선언합니다.

QEMU는 물리 주소 0x80000000에서 시작하여 적어도 0x88000000까지 계속되는 RAM(물리적 메모리)이 포함된 컴퓨터를 시뮬레이션합니다. xv6에서는 이를 PHYSTOP이라고 합니다.
QEMU 시뮬레이션에는 디스크 인터페이스와 같은 I/O 장치도 포함됩니다. QEMU는 장치 인터페이스를 소프트웨어에 물리적 주소 공간의 0x80000000 아래에 있는 메모리 매핑 제어 레지스터로 노출합니다. 커널은 이러한 특수 물리적 주소를 읽고 쓰는 방식으로 장치와 상호 작용할 수 있습니다. 이러한 읽기 및 쓰기는 RAM이 아닌 장치 하드웨어와 통신합니다. 4장에서는 xv6가 장치와 상호 작용하는 방식을 설명합니다.

커널은 "직접 매핑"을 사용하여 RAM 및 메모리 매핑된 장치 레지스터에 접근합니다. 즉, 리소스를 물리적 주소와 동일한 가상 주소에 매핑합니다. 예를 들어, 커널 자체는 가상 주소 공간과 물리적 메모리 모두에서 KERNBASE=0x80000000에 있습니다 . 직접 매핑은 물리적 메모리를 읽거나 쓰는 커널 코드를 간소화합니다. 예를 들어, fork가 자식 프로세스에 사용자 메모리를 할당하면 할당자는 해당 메모리의 물리적 주소를 반환합니다. fork는 부모의 사용자 메모리를 자식에 복사할 때 해당 주소를 가상 주소로 직접 사용합니다.

직접 매핑되지 않는 커널 가상 주소는 몇 가지 있습니다.

- 트램펄린 페이지. 가상 주소 공간의 맨 위에 매핑됩니다. 사용자 페이지 테이블도 이와 동일한 매핑을 갖습니다. 4장에서는 트램펄린 페이지의 역할에 대해 설명하지만, 여기서는 페이지 테이블의 흥미로운 사용 사례를 볼 수 있습니다. 물리적 페이지(트램펄린 코드를 보유)는 커널의 가상 주소 공간에 두 번 매핑됩니다. 한 번은 가상 주소 공간의 맨 위에, 한 번은 직접 매핑으로 매핑됩니다.

- 커널 스택 페이지. 각 프로세스는 자체 커널 스택을 가지고 있으며, 이 스택은 높게 매핑되어 그 아래에 xv6가 매핑되지 않은 가드 페이지를 남길 수 있습니다. 가드 페이지의 PTE는 유효하지 않습니다(즉, PTE_V가 설정되지 않음). 따라서 커널이 커널 스택을 오버플로하면 예외가 발생하고 커널이 패닉 상태가 됩니다. 가드 페이지가 없으면 오버플로 스택이 다른 커널 메모리를 덮어쓰게 되어 잘못된 작업이 발생합니다. 패닉 크래시가 바람직합니다.

커널이 고메모리 매핑을 통해 스택을 사용하는 반면, 커널은 직접 매핑된 주소를 통해 스택에 액세스할 수도 있습니다. 다른 설계는 직접 매핑만 있고 직접 매핑된 주소에서 스택을 사용할 수 있습니다. 그러나 그러한 배열에서 가드 페이지를 제공하려면 그렇지 않으면 물리적 메모리를 참조하는 가상 주소를 매핑 해제해야 하며, 그러면 사용하기 어려울 것입니다.

커널은 트램폴린과 커널 텍스트에 대한 페이지를 PTE_R 및 PTE_X 권한으로 매핑합니다. 커널은 이러한 페이지에서 명령어를 읽고 실행합니다. 커널은 다른 페이지를 PTE_R 및 PTE_W 권한으로 매핑하여 해당 페이지의 메모리를 읽고 쓸 수 있습니다. 가드 페이지에 대한 매핑은 유효하지 않습니다.

3.3 코드: 주소 공간 생성

주소 공간과 페이지 테이블을 조작하기 위한 xv6 코드의 대부분은 vm.c (kernel /vm.c:1)에 있습니다. 중앙 데이터 구조는 pagetable_t이며 이는 실제로 RISC-V에 대한 포인터입니다.

루트 페이지 테이블 페이지; pagetable_t는 커널 페이지 테이블이거나 프로세스별 페이지 테이블 중 하나일 수 있습니다. 중심 함수는 가상 주소에 대한 PTE를 찾는 walk와 새 매핑에 대한 PTE를 설치하는 mappages입니다. kvm으로 시작하는 함수는 커널 페이지 테이블을 조작합니다. uvm으로 시작하는 함수는 사용자 페이지 테이블을 조작합니다. 다른 함수는 둘 다에 사용됩니다. copyout과 copyin은 시스템 호출 인수로 제공된 사용자 가상 주소에서 데이터를 복사합니다. 이러한 함수는 해당 실제 메모리를 찾기 위해 해당 주소를 명시적으로 변환해야 하기 때문에 vm.c에 있습니다.

부팅 시퀀스 초반에 main은 kvminit (kernel/vm.c:54) 를 호출합니다 . kvmmake를 사용하여 커널의 페이지 테이블을 생성합니다 (kernel/vm.c:20). 이 호출은 xv6가 RISC-V에서 페이징을 활성화하기 전에 발생하므로 주소는 물리적 메모리를 직접 참조합니다. kvmmake 는 먼저 루트 페이지 테이블 페이지를 보관하기 위해 물리적 메모리 페이지를 할당합니다. 그런 다음 kvmmap을 호출하여 커널에 필요한 변환을 설치합니다. 변환에는 커널의 명령어와 데이터, PHYSTOP 까지의 물리적 메모리, 실제로는 장치인 메모리 범위가 포함됩니다 . proc_mapstacks (kernel/proc.c:33) 각 프로세스에 커널 스택을 할당합니다. KSTACK 에서 생성된 가상 주소에 각 스택을 매핑하기 위해 kvmmap을 호출하여 잘못된 스택 가드 페이지에 대한 공간을 남겨둡니다.

kvmmap (커널/vm.c:132) 맵 페이지 호출 (커널/vm.c:143) 가상 주소 범위에 대한 매핑을 해당 물리적 주소 범위에 대한 페이지 테이블에 설치하는 것입니다. 범위 내의 각 가상 주소에 대해 페이지 간격으로 별도로 이를 수행합니다. 매핑할 각 가상 주소에 대해 mappages는 walk를 호출하여 해당 주소에 대한 PTE의 주소를 찾습니다. 그런 다음 PTE를 초기화하여 관련 물리적 페이지 번호, 원하는 권한 (PTE_W, PTE_X 및/또는 PTE_R) 및 PTE_V를 보관하여 PTE를 유효로 표시합니다 (kernel/vm.c:158).

걷기 (커널/vm.c:86) 가상 주소에 대한 PTE를 찾는 RISC-V 페이징 하드웨어를 모방합니다(그림 3.2 참조). walk는 3단계 페이지 테이블을 한 번에 9비트씩 내립니다. 각 단계의 9비트 가상 주소를 사용하여 다음 단계 페이지 테이블이나 최종 페이지 (kernel/vm.c:92)의 PTE를 찾습니다 . PTE가 유효하지 않으면 필요한 페이지가 아직 할당되지 않은 것입니다. alloc 인수가 설정된 경우 walk는 새 페이지 테이블 페이지를 할당하고 해당 물리 주소를 PTE에 넣습니다. 트리의 가장 낮은 계층에 있는 PTE의 주소를 반환합니다 (kernel/vm.c:102).

위의 코드는 물리적 메모리가 커널 가상 주소 공간에 직접 매핑되는 데 의존합니다. 예를 들어, walk가 페이지 테이블의 레벨을 내려갈 때 PTE(kernel/vm.c:94)에서 다음 레벨 아래 페이지 테이블의 (물리적) 주소를 가져옵니다 . 그런 다음 해당 주소를 가상 주소로 사용하여 바로 아래 레벨의 PTE를 가져옵니다 (kernel/vm.c:92).

메인 호출 kvminithart (kernel/vm.c:62) 커널 페이지 테이블을 설치합니다. 루트 페이지 테이블 페이지의 물리적 주소를 레지스터 satp에 씁니다. 그 후 CPU는 커널 페이지 테이블을 사용하여 주소를 변환합니다. 커널은 ID 매핑을 사용하므로 다음 명령어의 현재 가상 주소는 올바른 물리적 메모리 주소에 매핑됩니다.

각 RISC-V CPU는 페이지 테이블 항목을 변환 룩어사이드 버퍼(TLB)에 캐시하고, xv6가 페이지 테이블을 변경할 때 CPU에 해당 캐시된 TLB 항목을 무효화하라고 알려야 합니다. 그렇지 않으면 나중에 TLB가 이전에 캐시된 매핑을 사용하여 그동안 다른 프로세스에 할당된 실제 페이지를 가리키고 결과적으로 프로세스가 다른 프로세스의 메모리에 낙서할 수 있습니다. RISC-V에는 현재 CPU의 TLB를 플러시하는 명령어 sfence.vma가 있습니다 . Xv6는 satp 레지스터를 다시 로드한 후 kvminithart에서 sfence.vma를 실행하고, 트램폴린 코드에서

사용자 공간으로 반환되기 전 사용자 페이지 테이블 (kernel/trampoline.S:89).

satp를 변경하기 전에 sfence.vma를 발행하여 모든 미처리 로드 및 스토어가 완료될 때까지 기다려야 합니다. 이 대기는 페이지 테이블에 대한 이전 업데이트가 완료되었는지 확인하고 이전 로드 및 스토어가 새 페이지 테이블이 아닌 이전 페이지 테이블을 사용했는지 확인합니다.

하나.

TLB 전체를 플러시하지 않기 위해 RISC-V CPU는 주소 공간 식별자(ASID)[3]를 지원할 수 있습니다. 그런 다음 커널은 특정 주소 공간에 대한 TLB 항목만 플러시할 수 있습니다. Xv6는 이 기능을 사용하지 않습니다.

3.4 물리적 메모리 할당

커널은 페이지 테이블, 사용자 메모리, 커널 스택, 파이프 버퍼를 위해 런타임에 물리적 메모리를 할당하고 해제해야 합니다.

Xv6는 커널 끝과 PHYSTOP 사이의 물리적 메모리를 런타임 할당에 사용합니다. 한 번에 전체 4096바이트 페이지를 할당하고 해제합니다. 연결 목록을 페이지 자체에 스레드하여 어떤 페이지가 해제되었는지 추적합니다. 할당은 연결 목록에서 페이지를 제거하는 것으로 구성되고 해제는 해제된 페이지를 목록에 추가하는 것으로 구성됩니다.

3.5 코드: 물리적 메모리 할당기

할당자는 kalloc.c (kernel/kalloc.c:1)에 있습니다. 할당자의 데이터 구조는 할당에 사용할 수 있는 물리적 메모리 페이지의 자유 목록입니다. 각 자유 페이지의 목록 요소는 struct run (kernel/kalloc.c:17)입니다. 할당자는 그 데이터 구조를 보관할 메모리를 어디서 얻나요? 할당자는 각 자유 페이지의 실행 구조를 자유 페이지 자체에 저장합니다. 거기에는 다른 것이 저장되어 있지 않기 때문입니다. 자유 목록은 스핀 잠금 (kernel/kalloc.c:21-24) 으로 보호됩니다 . 리스트와 잠금은 구조체에 래핑되어 잠금이 구조체의 필드를 보호한다는 것을 명확히 합니다. 지금은 잠금과 acquire 및 release 호출을 무시합니다 . 6장에서 잠금을 자세히 살펴봅니다.

main 함수는 kinit을 호출하여 할당자 (kernel/kalloc.c:27)를 초기화합니다. kinit는 커널 끝과 PHYSTOP 사이의 모든 페이지를 보관하기 위해 자유 목록을 초기화합니다. Xv6는 하드웨어에서 제공하는 구성 정보를 구문 분석하여 얼마나 많은 실제 메모리를 사용할 수 있는지 확인해야 합니다. 대신 xv6는 머신에 128메가바이트의 RAM이 있다고 가정합니다. kinit는 kfree 에 대한 페이지별 호출을 통해 freerange를 호출하여 메모리를 자유 목록에 추가합니다. PTE는 4096바이트 경계에 정렬된 실제 주소만 참조할 수 있으므로(4096의 배수임) freerange는 PGROUNDUP을 사용하여 정렬된 실제 주소만 해제합니다. 할당자는 메모리 없이 시작합니다. kfree 에 대한 이러한 호출은 관리할 메모리를 제공합니다.

할당자는 때때로 주소를 정수로 처리하여 산술 연산을 수행하고(예: freerange의 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 메모리 유형을 변경하기 때문입니다.

kfree 함수 (kernel/kalloc.c:47) 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. 다행히도 이런 코드가 더 빨리 중단될 것입니다.

그런 다음 kfree는 해당 페이지를 자유 공간 목록에 추가합니다. pa를 struct run 에 대한 포인터로 캐스팅하고 , r->next 에 자유 공간 목록의 이전 시작을 기록하고 , 자유 공간 목록을 r과 동일하게 설정합니다. kalloc은 자유 공간 목록의 첫 번째 요소를 제거하고 반환합니다.

3.6 프로세스 주소 공간

각 프로세스는 별도의 페이지 테이블을 가지고 있으며, xv6가 프로세스 간에 전환할 때 페이지 테이블도 변경합니다. 그림 3.4는 그림 2.3보다 프로세스의 주소 공간을 더 자세히 보여줍니다. 프로세스의 사용자 메모리는 가상 주소 0에서 시작하여 MAXVA (kernel/riscv.h:360)까지 커질 수 있습니다. 원칙적으로 프로세스가 256기가바이트의 메모리를 처리할 수 있도록 합니다.

프로세스의 주소 공간은 프로그램 텍스트를 포함하는 페이지(xv6는 PTE_R, PTE_X, PTE_U 권한으로 매핑함), 프로그램의 사전 초기화된 데이터를 포함하는 페이지, 스택을 위한 페이지, 힙을 위한 페이지로 구성됩니다. Xv6는 데이터, 스택, 힙을 PTE_R, PTE_W, PTE_U 권한 으로 매핑합니다 .

사용자 주소 공간 내에서 권한을 사용하는 것은 사용자 프로세스를 강화하는 일반적인 기술입니다.
텍스트가 PTE_W 로 매핑된 경우 프로세스가 실수로 자체 프로그램을 수정할 수 있습니다. 예를 들어, 프로그래밍 오류로 인해 프로그램이 널 포인터에 쓰고 주소 0의 명령어를 수정한 다음 계속 실행되어 더 큰 혼란을 일으킬 수 있습니다. 이러한 오류를 즉시 감지하기 위해 xv6는 PTE_W 없이 텍스트를 매핑합니다. 프로그램이 실수로 주소 0에 저장하려고 하면 하드웨어가 저장을 실행하지 않고 페이지 폴트를 발생시킵니다(4.6절 참조).

그러면 커널은 해당 프로세스를 종료하고 개발자가 문제를 추적할 수 있도록 정보 메시지를 출력합니다.

마찬가지로 PTE_X 없이 데이터를 매핑하면 사용자 프로그램이 실수로 다음으로 점프할 수 없습니다.
프로그램 데이터의 주소를 입력하고 해당 주소에서 실행을 시작합니다.

실제 세계에서 권한을 신중하게 설정하여 프로세스를 강화하는 것은 보안 공격에 대한 방어에도 도움이 됩니다. 적대자는 버그를 악용하려는 의도로 프로그램(예: 웹 서버)에 신중하게 구성된 입력을 공급하여 프로그램의 버그를 트리거할 수 있습니다[14].
권한을 신중하게 설정하고 사용자 주소 공간의 레이아웃을 무작위로 지정하는 등의 다른 기술을 사용하면 이러한 공격이 더 어려워집니다.

스택은 단일 페이지이며 exec에서 생성된 초기 내용과 함께 표시됩니다. 명령줄 인수를 포함하는 문자열과 이에 대한 포인터 배열은 스택의 맨 위에 있습니다. 바로 아래에는 함수 main(argc, argv) 가 방금 호출된 것처럼 프로그램이 main 에서 시작할 수 있도록 하는 값이 있습니다.

할당된 스택 메모리를 오버플로하는 사용자 스택을 감지하기 위해 xv6는 PTE_U 플래그를 지워서 스택 바로 아래에 접근할 수 없는 가드 페이지를 배치합니다 . 사용자 스택이 오버플로되고 프로세스가 스택 아래의 주소를 사용하려고 하면 하드웨어는 가드 페이지에 사용자 모드에서 실행되는 프로그램에 접근할 수 없기 때문에 페이지 오류 예외를 생성합니다. 대신 실제 운영 체제는 오버플로될 때 사용자 스택에 자동으로 더 많은 메모리를 할당할 수 있습니다.

프로세스가 xv6에 더 많은 사용자 메모리를 요청하면 xv6는 프로세스의 힙을 늘립니다. Xv6는 먼저 다음을 사용합니다.

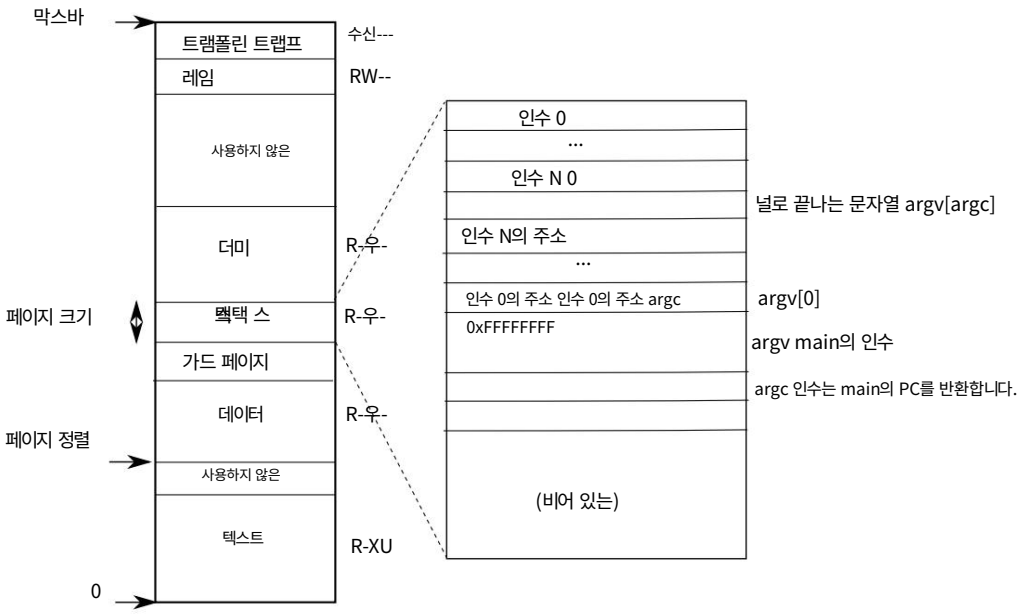

그림 3.4: 초기 스택을 포함한 프로세스의 사용자 주소 공간.

kalloc을 사용하여 물리적 페이지를 할당합니다. 그런 다음 프로세스의 페이지 테이블에 새 물리적 페이지를 가리키는 PTE를 추가합니다. Xv6는 이러한 PTE에서 PTE_W, PTE_R, PTE_U 및 PTE_V 플래그를 설정합니다. 대부분 프로세스는 전체 사용자 주소 공간을 사용하지 않습니다. xv6은 사용되지 않는 PTE에서 PTE_V를 비워 둡니다.

여기서는 페이지 테이블 사용의 몇 가지 좋은 예를 볼 수 있습니다. 첫째, 다른 프로세스의 페이지 테이블은 사용자 주소를 다른 물리 메모리 페이지로 변환하여 각 프로세스가 개인 사용자 메모리를 갖도록 합니다. 둘째, 각 프로세스는 메모리를 0에서 시작하는 연속된 가상 주소로 보는 반면, 프로세스의 물리 메모리는 비연속적일 수 있습니다. 셋째, 커널은 사용자 주소 공간의 맨 위에 트램폴린 코드(PTE_U 없음) 가 있는 페이지를 매핑 하므로 모든 주소 공간에 단일 물리 메모리 페이지가 표시되지만 커널에서만 사용할 수 있습니다.

3.7 코드: sbrk

sbrk는 프로세스가 메모리를 축소하거나 확장하기 위한 시스템 호출입니다. 이 시스템 호출은 growproc 함수 (kernel/proc.c:260)에 의해 구현됩니다. growproc는 n 이 양수인지 음수 인지에 따라 uvmalloc 또는 uvmdealloc를 호출합니다 . uvmalloc (kernel/vm.c:226) kalloc을 사용하여 물리적 메모리를 할당하고 mappages를 사용하여 PTE를 사용자 페이지 테이블에 추가합니다. uvmdealloc은 uvmunmap (kernel/vm.c:171)을 호출합니다. walk를 사용하여 PTE를 찾고 kfree를 사용하여 참조하는 물리적 메모리를 해제합니다.

Xv6는 하드웨어에 사용자 가상 주소를 매핑하는 방법을 알려주는 데만 프로세스의 페이지 테이블을 사용하는 것이 아니라 해당 프로세스에 할당된 실제 메모리 페이지의 유일한 레코드로도 사용합니다. 그래서 사용자 메모리를 해제하려면 (uvmunmap에서) 사용자 페이지 테이블을 검사해야 합니다.

3.8 코드: exec

exec는 프로세스의 사용자 주소 공간을 바이너리 또는 실행 파일이라고 하는 파일에서 읽은 데이터로 대체하는 시스템 호출입니다. 바이너리는 일반적으로 컴파일러와 링커의 출력이며 머신 명령어와 프로그램 데이터를 보관합니다. exec (kernel/exec.c:23) namei (kernel/exec.c:36)를 사용하여 명명된 바이너리 경로를 엽니다. 8장에서 설명합니다. 그런 다음 ELF 헤더를 읽습니다. Xv6 바이너리는 (kernel/elf.h) 에 정의된 널리 사용되는 ELF 형식으로 포맷됩니다 . ELF 바이너리는 ELF 헤더, struct elfhdr (kernel/elf.h:6)로 구성됩니다. 그 뒤에 일련의 프로그램 섹션 헤더, struct proghdr (kernel/elf.h:25)가 이어집니다. 각 progvhdr은 메모리에 로드되어야 하는 애플리케이션 섹션을 설명합니다. xv6 프로그램에는 두 개의 프로그램 섹션 헤더가 있습니다. 하나는 명령어용이고 다른 하나는 데이터용입니다.

첫 번째 단계는 파일에 ELF 바이너리가 들어 있는지 빠르게 확인하는 것입니다. ELF 바이너리는 4바이트 "매직 넘버" 0x7F, 'E', 'L', 'F' 또는 ELF_MAGIC (kernel/elf.h:3)로 시작합니다. ELF 헤더에 올바른 매직 번호가 있으면 exec는 바이너리가 올바르게 구성되었다고 가정합니다.

exec는 proc_pagetable (kernel/exec.c:49) 을 사용하여 사용자 매핑이 없는 새 페이지 테이블을 할당합니다 . uvmalloc (kernel/exec.c:65)를 사용하여 각 ELF 세그먼트에 대한 메모리를 할당합니다 . 그리고 loadseg (kernel/exec.c:10) 를 통해 각 세그먼트를 메모리에 로드합니다 . loadseg는 walkaddr를 사용하여 ELF 세그먼트의 각 페이지를 쓸 할당된 메모리의 물리적 주소를 찾고, readi를 사용하여 파일에서 읽습니다.

exec 로 생성된 첫 번째 사용자 프로그램 인 /init 의 프로그램 섹션 헤더는 다음과 같습니다.

```
# objdump -p 사용자/_init

사용자/_init:                 파일 형식 elf64-little

프로그램 헤더: 0x70000003 꺼짐
                              0x00000000000006bb0 가상 주소 0x0000000000000000
                                                  paddr 0x0000000000000000 align 2**0 filesz 0x000000000000004a memsz
                0x0000000000000000 플래그 r--
    로드 오프                 0x0000000000001000 가상 주소 0x0000000000000000
                                                  paddr 0x0000000000000000 align 2**12 filesz 0x0000000000001000 memsz
                0x0000000000001000 플래그 rx
    로드 오프                 0x0000000000002000 가상 주소 0x0000000000001000
                                                  paddr 0x0000000000001000 align 2**12 filesz 0x0000000000000010 memsz
                0x0000000000000030 flags rw-
    스택 오프                 0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**4
                                                  filesz 0x0000000000000000 memsz 0x0000000000000000 플래그 rw-
```

우리는 텍스트 세그먼트가 파일의 오프셋 0x1000에 있는 콘텐츠에서 메모리의 가상 주소 0x0에 로드되어야 한다는 것을 봅니다(쓰기 권한 없음). 또한 데이터가 페이지 경계에 있는 주소 0x1000에 로드되어야 하며 실행 권한 없음을 봅니다.

프로그램 섹션 헤더의 filesz는 memsz 보다 작을 수 있는데 , 이는 그 사이의 간격을 파일에서 읽는 대신 0으로 채워야 함을 나타냅니다(C 전역 변수의 경우). /init 의 경우 데이터 filesz 는 0x10바이트이고 memsz 는 0x30바이트이므로 uvmalloc은 0x30바이트를 보관할 만큼 충분한 물리적 메모리를 할당하지만 파일 /init에서 0x10바이트만 읽습니다.

이제 exec는 사용자 스택을 할당하고 초기화합니다. 스택 페이지 하나만 할당합니다. exec는 인수 문자열을 스택 맨 위에 한 번에 하나씩 복사하고, ustack에 해당 문자열에 대한 포인터를 기록합니다.
이것은 main 에 전달 될 argv 리스트 의 끝에 널 포인터를 배치합니다 . ustack 의 처음 세 항목 은 가짜 리턴 프로그램 카운터, argc, argv 포인터 입니다.

exec는 스택 페이지 바로 아래에 접근할 수 없는 페이지를 배치하여, 두 개 이상의 페이지를 사용하려는 프로그램이 오류를 발생시킵니다. 이 접근할 수 없는 페이지는 exec가 너무 큰 인수를 처리할 수 있도록 합니다. 이 상황에서 copyout (kernel/vm.c:352) exec가 인수를 스택에 복사하는 데 사용하는 함수는 대상 페이지에 접근할 수 없다는 것을 알아차리고 -1을 반환합니다.

새 메모리 이미지를 준비하는 동안 exec가 잘못된 프로그램 세그먼트와 같은 오류를 감지하면 레이블 bad 로 점프하여 새 이미지를 해제하고 -1을 반환합니다. exec는 시스템 호출이 성공할 것이라고 확신할 때까지 이전 이미지를 해제하기 위해 기다려야 합니다. 이전 이미지가 없어지면 시스템 호출은 -1을 반환할 수 없습니다. exec 에서 유일한 오류 사례는 이미지 생성 중에 발생합니다. 이미지가 완료되면 exec는 새 페이지 테이블 (kernel/exec.c:125) 에 커밋할 수 있습니다 . 그리고 이전 것을 해제합니다 (kernel/exec.c:129).

exec는 ELF 파일에서 바이트를 ELF 파일에 지정된 주소의 메모리로 로드합니다. 사용자 또는 프로세스는 원하는 주소를 ELF 파일에 넣을 수 있습니다. 따라서 exec는 ELF 파일의 주소가 실수로 또는 의도적으로 커널을 참조할 수 있으므로 위험합니다. 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. 예를 들어 if(ph.vaddr + ph.memsz < ph.vaddr) 는 합계가 64비트 정수를 오버플로하는지 확인합니다. 위험한 점은 사용자가 사용자가 선택한 주소를 가리키는 ph.vaddr 와 합계가 0x1000으로 오버플로될 만큼 큰 ph.memsz를 사용하여 ELF 바이너리를 구성할 수 있다는 것입니다. 이는 유효한 값처럼 보일 것입니다. 사용자 주소 공간에 커널도 포함된(하지만 사용자 모드에서는 읽기/쓰기가 불가능) 이전 버전의 xv6에서 사용자는 커널 메모리에 해당하는 주소를 선택하여 ELF 바이너리에서 커널로 데이터를 복사할 수 있었습니다. xv6의 RISC-V 버전에서는 커널에 별도의 페이지 테이블이 있기 때문에 이런 일이 일어날 수 없습니다. loadseg는 커널의 페이지 테이블이 아니라 프로세스의 페이지 테이블로 로드됩니다.

커널 개발자가 중요한 검사를 생략하는 것은 쉬운 일이며, 실제 커널은 사용자 프로그램이 커널 권한을 획득하기 위해 악용할 수 있는 검사가 누락된 오랜 역사를 가지고 있습니다. xv6가 커널에 제공된 사용자 수준 데이터의 유효성을 완벽하게 검사하지 못할 가능성이 높으며, 악의적인 사용자 프로그램이 xv6의 격리를 우회하기 위해 악용할 수 있습니다.

3.9 현실 세계

대부분의 운영체제와 마찬가지로 xv6는 메모리 보호 및 매핑을 위해 페이징 하드웨어를 사용합니다.
대부분의 운영 체제는 페이징과 페이지 오류 예외를 결합하여 xv6보다 훨씬 더 정교하게 페이징을 사용합니다. 이에 대해서는 4장에서 다루겠습니다.

Xv6는 커널이 가상 주소와 물리적 주소 사이의 직접 맵을 사용하고, 커널이 로드될 것으로 예상하는 주소 0x8000000에 물리적 RAM이 있다고 가정함으로써 간소화됩니다. 이는 QEMU에서는 작동하지만 실제 하드웨어에서는 나쁜 생각으로 판명됩니다. 실제 하드웨어는 RAM과 장치를 예측할 수 없는 물리적 주소에 배치하므로 (예를 들어) xv6가 커널을 저장할 수 있을 것으로 예상하는 0x8000000에 RAM이 없을 수 있습니다. 더 심각한 커널

디자인은 페이지 테이블을 이용하여 임의의 하드웨어 물리적 메모리 레이아웃을 예측 가능한 커널 가상 주소 레이아웃으로 전환합니다.

RISC-V는 물리적 주소 수준에서 보호를 지원하지만 xv6는 해당 기능을 사용하지 않습니다.

메모리가 많은 머신에서는 RISC-V의 "슈퍼 페이지" 지원을 사용하는 것이 합리적일 수 있습니다. 물리적 메모리가 작을 때 작은 페이지는 디스크에 대한 할당 및 페이지 아웃을 세밀하게 허용하기 위해 합리적입니다. 예를 들어, 프로그램이 8킬로바이트의 메모리만 사용하는 경우 물리적 메모리의 전체 4메가바이트 슈퍼 페이지를 제공하는 것은 낭비입니다. RAM이 많은 머신에서는 더 큰 페이지가 합리적이며 페이지 테이블 조작에 대한 오버헤드를 줄일 수 있습니다.

xv6 커널에는 작은 객체에 대한 메모리를 제공할 수 있는 malloc과 같은 할당자가 없기 때문에 커널은 동적 할당이 필요한 정교한 데이터 구조를 사용할 수 없습니다.
더 정교한 커널은 (xv6처럼) 4096바이트 블록만 할당하는 것이 아니라, 여러 가지 크기의 작은 블록을 할당할 가능성이 높습니다. 실제 커널 할당자는 큰 할당뿐만 아니라 작은 할당도 처리해야 합니다.

메모리 할당은 다년간 화제가 되어 온 주제이며, 기본적인 문제는 제한된 메모리의 효율적인 사용과 알려지지 않은 미래 요청에 대비하는 것입니다[9]. 오늘날 사람들은 공간 효율성보다 속도를 더 중요하게 여깁니다.

3.10 연습문제

1. RISC-V의 장치 트리를 분석하여 컴퓨터의 실제 메모리 양을 확인하세요.

2. sbrk(1)를 호출하여 주소 공간을 1바이트씩 늘리는 사용자 프로그램을 작성합니다 . 프로그램을 실행하고 sbrk를 호출하기 전과 sbrk 를 호출한 후에 프로그램의 페이지 테이블을 조사합니다 . 커널은 얼마나 많은 공간을 할당했습니까? 새 메모리의 PTE에는 무엇이 들어 있습니까?

3. 커널의 슈퍼 페이지를 사용하도록 xv6를 수정합니다.

4. exec 의 Unix 구현은 전통적으로 쉘 스크립트에 대한 특수 처리를 포함합니다. 실행할 파일이 텍스트 #! 로 시작하면 첫 번째 줄은 파일을 해석하기 위해 실행할 프로그램으로 간주됩니다. 예를 들어, exec 가 myprog arg1을 실행하기 위해 호출되고 myprog 의 첫 번째 줄이 #!/interp 이면 exec는 명령줄 /interp myprog arg1로 /interp를 실행합니다 .

   xv6에서 이 규칙에 대한 지원을 구현합니다.

5. 커널에 대한 주소 공간 레이아웃 무작위화를 구현합니다.

4장

트랩 및 시스템 호출

CPU가 명령어의 일반적인 실행을 제쳐두고 이벤트를 처리하는 특수 코드로 제어를 강제로 이전하게 하는 세 가지 종류의 이벤트가 있습니다. 한 가지 상황은 시스템 호출로, 사용자 프로그램이 ecall 명령어를 실행하여 커널에 무언가를 하도록 요청하는 경우입니다. 또 다른 상황은 예외입니다. 명령어(사용자 또는 커널)가 0으로 나누거나 잘못된 가상 주소를 사용하는 것과 같이 불법적인 일을 하는 경우입니다. 세 번째 상황은 장치 인터럽트로, 예를 들어 디스크 하드웨어가 읽기 또는 쓰기 요청을 완료할 때 장치가 주의가 필요하다는 신호를 보내는 경우입니다.

이 책에서는 이런 상황에 대한 일반적인 용어로 트랩을 사용합니다. 일반적으로 트랩 시점에 실행 중이던 코드는 나중에 재개해야 하며, 특별한 일이 발생했다는 것을 알 필요가 없습니다. 즉, 우리는 종종 트랩이 투명하기를 원합니다. 이는 특히 중단된 코드가 일반적으로 예상하지 못하는 장치 인터럽트의 경우 중요합니다. 일반적인 순서는 트랩이 커널로 제어를 강제로 전송하고, 커널이 레지스터와 기타 상태를 저장하여 실행을 재개할 수 있도록 하고, 커널이 적절한 핸들러 코드(예: 시스템 호출 구현 또는 장치 드라이버)를 실행하고, 커널이 저장된 상태를 복원하고 트랩에서 돌아오고, 원래 코드가 중단된 곳에서 재개된다는 것입니다.

Xv6는 커널에서 모든 트랩을 처리합니다. 트랩은 사용자 코드에 전달되지 않습니다. 커널에서 트랩을 처리하는 것은 시스템 호출에 자연스러운 일입니다. 격리는 커널만 장치를 사용할 수 있도록 요구하고, 커널은 여러 프로세스 간에 장치를 공유하는 편리한 메커니즘이기 때문에 인터럽트에 적합합니다. 또한 xv6는 사용자 공간의 모든 예외에 대응하여 문제가 있는 프로그램을 종료하기 때문에 예외에도 적합합니다.

Xv6 트랩 처리 과정은 4단계로 진행됩니다. RISC-V CPU가 수행하는 하드웨어 작업, 커널 C 코드를 위한 길을 준비하는 일부 어셈블리 명령어, 트랩을 어떻게 처리할지 결정하는 C 함수, 시스템 호출 또는 장치 드라이버 서비스 루틴입니다. 세 가지 트랩 유형 간의 공통점은 커널이 단일 코드 경로로 모든 트랩을 처리할 수 있음을 시사하지만, 사용자 공간의 트랩, 커널 공간의 트랩, 타이머 인터럽트의 세 가지 경우를 위해 별도의 코드를 갖는 것이 편리하다는 것이 밝혀졌습니다. 트랩을 처리하는 커널 코드(어셈블러 또는 C)는 종종 핸들러라고 합니다. 첫 번째 핸들러 명령어는 일반적으로 어셈블러(C가 아님)로 작성되며 벡터라고도 합니다.

4.1 RISC-V 트랩 장치

각 RISC-V CPU에는 커널이 CPU에 트랩을 처리하는 방법을 알려주기 위해 쓰는 제어 레지스터 세트가 있으며, 커널은 이를 읽어서 발생한 트랩을 알아낼 수 있습니다. RISC-V 문서에는 전체 스토리가 포함되어 있습니다[3]. riscv.h (kernel/riscv.h:1) xv6에서 사용하는 정의를 포함합니다.
가장 중요한 레지스터의 개요는 다음과 같습니다.

- stvec: 커널은 여기에 트랩 핸들러의 주소를 씁니다. RISC-V는 다음으로 점프합니다.
  트랩을 처리하기 위한 stvec의 주소입니다.
- sepc: 트랩이 발생하면 RISC-V는 프로그램 카운터를 여기에 저장합니다(그러면 pc가 stvec의 값으로 덮어쓰기 때문입니다).
  sret(트랩에서 복귀) 명령어는 sepc를 pc로 복사합니다. 커널은 sepc를 작성하여 sret이 어디로 가는지 제어할 수 있습니다.
- scause: RISC-V는 여기에 트랩의 이유를 설명하는 숫자를 입력합니다.
- sscratch: 트랩 핸들러 코드는 sscratch를 사용하여 사용자 레지스터 덮어쓰기를 방지합니다.
  저장하기 전에.
- sstatus: sstatus의 SIE 비트는 장치 인터럽트가 활성화되는지 여부를 제어합니다. 커널이 SIE를 지우면 RISC-V는 커널이 SIE를 설정할 때까지 장치 인터럽트를 연기합니다. SPP 비트는 트랩이 사용자 모드 또는 감독자 모드에서 왔는지 여부를 나타내고 sret이 어떤 모드로 반환되는지 제어합니다.

위의 레지스터는 슈퍼바이저 모드에서 처리되는 트랩과 관련이 있으며, 사용자 모드에서는 읽거나 쓸 수 없습니다. 머신 모드에서 처리되는 트랩에 대한 유사한 제어 레지스터 세트가 있습니다. xv6는 타이머 인터럽트의 특수한 경우에만 이를 사용합니다.

멀티코어 칩의 각 CPU는 자체적인 레지스터 세트를 갖고 있으며, 언제든지 두 개 이상의 CPU가 트랩을 처리할 수 있습니다.

트랩을 강제로 실행해야 하는 경우 RISC-V 하드웨어는 모든 트랩 유형(타이머 인터럽트 제외)에 대해 다음을 수행합니다.

1. 트랩이 장치 인터럽트이고 sstatus SIE 비트가 지워진 경우 다음 작업을 수행하지 마십시오.
   수행원.
2. sstatus의 SIE 비트를 지워 인터럽트를 비활성화합니다.
3. PC를 sepc로 복사합니다.
4. sstatus의 SPP 비트에 현재 모드(사용자 또는 감독자)를 저장합니다.
5. 함정의 원인을 반영하도록 scause를 설정합니다.
6. 모드를 감독자로 설정하세요.
7. stvec를 pc로 복사합니다.

8. 새로운 PC에서 실행을 시작합니다.

CPU가 커널 페이지 테이블로 전환하지 않고, 커널의 스택으로 전환하지 않으며, PC 이외의 레지스터를 저장하지 않는다는 점에 유의하세요. 커널 소프트웨어는 이러한 작업을 수행해야 합니다.
트랩 중에 CPU가 최소한의 작업을 수행하는 한 가지 이유는 소프트웨어에 유연성을 제공하기 위한 것입니다. 예를 들어, 일부 운영 체제는 트랩 성능을 높이기 위해 특정 상황에서 페이지 테이블 전환을 생략합니다.

위에 나열된 단계 중 어떤 것이 생략될 수 있는지 생각해 볼 가치가 있습니다. 아마도 더 빠른 트랩을 찾는 것이겠죠. 더 간단한 시퀀스가 작동할 수 있는 상황이 있기는 하지만, 일반적으로 생략하기에는 많은 단계가 위험할 것입니다. 예를 들어, CPU가 프로그램 카운터를 전환하지 않았다고 가정해 보겠습니다. 그러면 사용자 공간의 트랩이 사용자 명령어를 계속 실행하는 동안 감독자 모드로 전환될 수 있습니다. 이러한 사용자 명령어는 사용자/커널 격리를 깨뜨릴 수 있습니다. 예를 들어, satp 레지스터를 수정하여 모든 물리적 메모리에 액세스할 수 있는 페이지 테이블을 가리키도록 하면 됩니다.

그러므로 CPU를 커널이 지정한 명령어 주소, 즉 stvec로 전환하는 것이 중요합니다.

4.2 사용자 공간의 트랩

Xv6는 트랩이 커널에서 실행되는지 아니면 사용자 코드에서 실행되는지에 따라 트랩을 다르게 처리합니다. 사용자 코드의 트랩에 대한 스토리는 다음과 같습니다. 섹션 4.5에서는 커널 코드의 트랩을 설명합니다.

사용자 공간에서 실행하는 동안 사용자 프로그램이 시스템 호출(ecall 명령어)을 하거나 불법적인 작업을 하거나 장치가 인터럽트를 하는 경우 트랩이 발생할 수 있습니다. 사용자 공간에서 트랩의 상위 경로는 uservec (kernel/trampoline.S:21)입니다. 그 다음 usertrap (kernel/trap.c:37); 그리고 다시 돌아올때 usertrapret (kernel/trap.c:90) 그리고 userret (kernel/trampoline.S:101).
xv6의 트랩 처리 설계에 대한 주요 제약은 RISC-V 하드웨어가 트랩을 강제로 실행할 때 페이지 테이블을 전환하지 않는다는 사실입니다. 즉, stvec의 트랩 처리기 주소는 트랩 처리 코드가 실행을 시작할 때 적용되는 페이지 테이블이기 때문에 사용자 페이지 테이블에 유효한 매핑이 있어야 합니다. 또한 xv6의 트랩 처리 코드는 커널 페이지 테이블로 전환해야 합니다. 전환 후에도 실행을 계속하려면 커널 페이지 테이블에 stvec가 가리키는 처리기에 대한 매핑도 있어야 합니다.

Xv6는 트램폴린 페이지를 사용하여 이러한 요구 사항을 충족합니다. 트램폴린 페이지에는 uservec가 포함되어 있습니다.
stvec가 가리키는 xv6 트랩 처리 코드. 트램폴린 페이지는 모든 프로세스의 페이지 테이블에서 가상 주소 공간의 맨 위에 있는 주소 TRAMPOLINE 에 매핑되므로 프로그램이 스스로 사용하는 메모리 위에 있게 됩니다. 트램폴린 페이지는 커널 페이지 테이블의 주소 TRAMPOLINE에도 매핑됩니다. 그림 2.3과 그림 3.3을 참조하세요. 트램폴린 페이지는 PTE_U 플래그 없이 사용자 페이지 테이블에 매핑되므로 트랩은 관리자 모드에서 해당 페이지에서 실행을 시작할 수 있습니다. 트램폴린 페이지는 커널 주소 공간의 동일한 주소에 매핑되므로 트랩 처리기는 커널 페이지 테이블로 전환한 후에도 계속 실행할 수 있습니다.

uservec 트랩 핸들러의 코드는 trampoline.S (kernel/trampoline.S:21)에 있습니다. uservec가 시작되면 모든 32개 레지스터에는 중단된 사용자 코드가 소유한 값이 포함됩니다. 이 32개 값은 메모리의 어딘가에 저장되어야 트랩이 사용자 공간으로 돌아올 때 복원될 수 있습니다. 메모리에 저장하려면 주소를 보관하기 위해 레지스터를 사용해야 하지만 이 시점에서

범용 레지스터를 사용할 수 없습니다! 다행히도 RISC-V는 sscratch 레지스터 형태로 도움을 제공합니다. uservec의 시작 부분에 있는 csrw 명령어는 sscratch에 a0을 저장합니다.
이제 uservec에는 놀 수 있는 레지스터(a0)가 하나 있습니다.

uservec의 다음 작업은 32개의 사용자 레지스터를 저장하는 것입니다. 커널은 각 프로세스에 대해 trapframe 구조에 대한 메모리 페이지를 할당하는데, 이 구조는 (다른 것들 중에서) 32개의 사용자 레지스터를 저장할 공간이 있습니다 (kernel/proc.h:43). satp가 여전히 사용자 페이지 테이블을 참조하기 때문에 uservec는 트랩프레임이 사용자 주소 공간에 매핑되어야 합니다. Xv6는 각 프로세스의 트랩프레임을 해당 프로세스의 사용자 페이지 테이블의 가상 주소 TRAPFRAME에 매핑합니다. TRAPFRAME은 TRAMPOLINE 바로 아래에 있습니다.
프로세스의 p->trapframe도 trapframe을 가리키지만, trapframe의 물리적 주소를 가리키므로 커널은 커널 페이지 테이블을 통해 trapframe을 사용할 수 있습니다.

따라서 uservec는 주소 TRAPFRAME을 a0에 로드하고 사용자의 a0을 포함하여 모든 사용자 레지스터를 거기에 저장하고 sscratch부터 다시 읽습니다.

trapframe에는 현재 프로세스의 커널 스택 주소, 현재 CPU의 hartid, usertrap 함수의 주소, 커널 페이지 테이블의 주소가 포함됩니다. uservec는 이러한 값을 검색하고, satp를 커널 페이지 테이블로 전환하고, usertrap을 호출합니다.

usertrap의 역할은 트랩의 원인을 파악하고 이를 처리한 뒤 반환하는 것입니다 (kernel/- trap.c:37). 먼저 stvec를 변경하여 커널에 있는 동안 트랩이 uservec가 아닌 kernelvec에 의해 처리되도록 합니다. sepc 레지스터(저장된 사용자 프로그램 카운터)를 저장합니다. usertrap이 yield를 호출하여 다른 프로세스의 커널 스레드로 전환하고 해당 프로세스가 사용자 공간으로 돌아갈 수 있기 때문입니다. 이 과정에서 sepc를 수정합니다. 트랩이 시스템 호출인 경우 usertrap이 syscall을 호출하여 처리합니다. 장치가 인터럽트인 경우 devintr을 호출합니다. 그렇지 않으면 예외이며 커널은 오류가 발생한 프로세스를 종료합니다. 시스템 호출 경로는 저장된 사용자 프로그램 카운터에 4를 추가합니다. RISC-V는 시스템 호출의 경우 프로그램 포인터를 ecall 명령어를 가리키도록 두지만 사용자 코드는 후속 명령어에서 실행을 재개해야 하기 때문입니다.

나갈 때, usertrap은 프로세스가 종료되었는지 또는 CPU를 양보해야 하는지 확인합니다(이 트랩이 타이머 인터럽트인 경우).

사용자 공간으로 돌아가기 위한 첫 번째 단계는 usertrapret (kernel/trap.c:90)을 호출하는 것입니다. 이 함수는 사용자 공간에서 미래의 트랩을 준비하기 위해 RISC-V 제어 레지스터를 설정합니다. 여기에는 stvec를 uservec를 참조하도록 변경하고, uservec가 의존하는 trapframe 필드를 준비하고, sepc를 이전에 저장된 사용자 프로그램 카운터로 설정하는 것이 포함됩니다. 마지막으로 usertrapret은 사용자 및 커널 페이지 테이블 모두에 매핑된 트램폴린 페이지에서 userret을 호출합니다. 그 이유는 userret의 어셈블리 코드가 페이지 테이블을 전환하기 때문입니다.

usertrapret의 userret 호출은 a0(kernel/trampoline.S:101)에 있는 프로세스의 사용자 페이지 테이블에 대한 포인터를 전달합니다 . userret은 satp를 프로세스의 사용자 페이지 테이블로 전환합니다. 사용자 페이지 테이블은 트램폴린 페이지와 TRAPFRAME을 모두 매핑하지만 커널의 다른 것은 매핑하지 않는다는 점을 기억하세요.
사용자 및 커널 페이지 테이블에서 동일한 가상 주소에 대한 트램폴린 페이지 매핑은 userret이 satp를 변경한 후에도 계속 실행할 수 있도록 합니다. 이 지점부터 userret이 사용할 수 있는 유일한 데이터는 레지스터 내용과 트랩프레임의 내용입니다. userret은 TRAPFRAME 주소를 a0에 로드하고, 트랩프레임에서 a0을 통해 저장된 사용자 레지스터를 복원하고, 저장된 사용자 a0을 복원하고, sret을 실행하여 사용자 공간으로 돌아갑니다.

4.3 코드: 시스템 호출 호출

2장은 initcode.S 가 exec 시스템 호출 (user/initcode.S:11)을 호출하는 것으로 끝났습니다. 사용자 호출이 커널에서 exec 시스템 호출의 구현으로 어떻게 전달되는지 살펴보겠습니다.

initcode.S는 exec 에 대한 인수를 레지스터 a0과 a1에 넣고 시스템 호출 번호를 a7에 넣습니다. 시스템 호출 번호는 함수 포인터 표인 syscalls 배열의 항목과 일치합니다 (kernel/syscall.c:107). ecall 명령어 는 커널에 트랩을 걸어 uservec, usertrap, 그리고 syscall을 실행하게 하는데, 이는 위에서 본 바와 같습니다.

syscall (커널/syscall.c:132) 트랩프레임에 저장된 a7에서 시스템 호출 번호를 검색합니다.
그리고 이를 사용하여 syscall을 인덱싱합니다. 첫 번째 시스템 호출의 경우 a7에는 SYS_exec(kernel/syscall.h:8)가 포함됩니다. 시스템 호출 구현 함수 sys_exec를 호출하게 됩니다 .

sys_exec가 반환 되면 syscall은 반환 값을 p->trapframe->a0에 기록합니다. 이렇게 하면 RISC-V의 C 호출 규칙이 반환 값을 a0에 두기 때문에 exec()에 대한 원래 사용자 공간 호출이 해당 값을 반환하게 됩니다. 시스템 호출은 일반적으로 오류를 나타낼 때는 음수를 반환하고 성공을 나타낼 때는 0 또는 양수를 반환합니다. 시스템 호출 번호가 잘못되었으면 syscall은 오류를 인쇄하고 -1을 반환합니다.

4.4 코드: 시스템 호출 인수

커널의 시스템 호출 구현은 사용자 코드에서 전달된 인수를 찾아야 합니다. 사용자 코드는 시스템 호출 래퍼 함수를 호출하기 때문에 인수는 처음에 RISC-V C 호출 규칙이 배치하는 곳, 즉 레지스터에 있습니다. 커널 트랩 코드는 사용자 레지스터를 현재 프로세스의 트랩 프레임에 저장하고, 커널 코드는 여기에서 찾을 수 있습니다. 커널 함수 argint, argaddr 및 argfd는 트랩 프레임에서 n번째 시스템 호출 인수를 정수, 포인터 또는 파일 설명자로 검색합니다. 이들은 모두 argraw를 호출하여 적절한 저장된 사용자 레지스터 (kernel/syscall.c:34)를 검색합니다.

일부 시스템 호출은 포인터를 인수로 전달하고, 커널은 이러한 포인터를 사용하여 사용자 메모리를 읽거나 써야 합니다. 예를 들어, exec 시스템 호출은 사용자 공간의 문자열 인수를 참조하는 포인터 배열을 커널에 전달합니다. 이러한 포인터는 두 가지 과제를 제기합니다. 첫째, 사용자 프로그램은 버그가 있거나 악의적일 수 있으며, 커널에 잘못된 포인터나 커널이 사용자 메모리 대신 커널 메모리에 액세스하도록 속이는 포인터를 전달할 수 있습니다. 둘째, xv6 커널 페이지 테이블 매핑은 사용자 페이지 테이블 매핑과 동일하지 않으므로 커널은 일반 명령어를 사용하여 사용자가 제공한 주소에서 로드하거나 저장할 수 없습니다.

커널은 사용자가 제공한 주소와 데이터를 안전하게 주고받는 기능을 구현합니다. fetchstr은 그 예입니다 (kernel/syscall.c:25). exec와 같은 파일 시스템 호출은 fetchstr을 사용하여 사용자 공간에서 문자열 파일 이름 인수를 검색합니다. fetchstr은 copyinstr을 호출하여 어려운 작업을 수행합니다. copyinstr (kernel/vm.c:403) 사용자 페이지 테이블 pagetable의 가상 주소 srcva 에서 최대 바이트 까지 dst 로 복사합니다 . pagetable이 현재 페이지 테이블이

아니므 로 copyinstr은 walkaddr(walk를 호출)를 사용하여 pagetable 에서 srcva를 조회하여 물리적 주소 pa0을 생성합니다. 커널은 각 물리적 RAM 주소를 해당 커널 가상 주소에 매핑하므로 copyinstr은 문자열 바이트를 pa0에서 dst로 직접 복사할 수 있습니다. walkaddr (kernel/vm.c:109) 사용자가 제공한 가상 주소가 프로세스의 사용자 주소 공간의 일부인지 확인하여 프로그램을 실행합니다.

커널을 속여 다른 메모리를 읽게 할 수 없습니다. 비슷한 기능인 copyout은 커널에서 사용자가 제공한 주소로 데이터를 복사합니다.

4.5 커널 공간의 트랩

Xv6는 사용자 또는 커널 코드가 실행되는지에 따라 CPU 트랩 레지스터를 다소 다르게 구성합니다. 커널이 CPU에서 실행 중일 때, 커널은 stvec를 kernelvec (kernel/kernelvec.S:12)의 어셈블리 코드로 가리킵니다. xv6가 이미 커널에 있으므로 kernelvec는 satp가 커널 페이지 테이블로 설정되어 있고, 스택 포인터가 유효한 커널 스택을 참조한다는 점에 의존할 수 있습니다. kernelvec는 32개의 레지스터를 모두 스택에 푸시하고, 나중에 이를 복원하여 중단된 커널 코드가 방해 없이 다시 시작할 수 있도록 합니다.

kernelvec는 중단된 커널 스레드의 스택에 레지스터를 저장합니다. 레지스터 값이 해당 스레드에 속하기 때문에 의미가 있습니다. 트랩이 다른 스레드로 전환되는 경우 특히 중요합니다. 이 경우 트랩은 실제로 새 스레드의 스택에서 반환되어 중단된 스레드의 저장된 레지스터가 스택에 안전하게 남게 됩니다.

kernelvec는 kerneltrap으로 점프합니다 (kernel/trap.c:135) 레지스터를 저장한 후 kerneltrap은 장치 인터럽트와 예외라는 두 가지 유형의 트랩에 대비합니다. devintr (kernel/- trap.c:178) 을 호출합니다. 전자를 확인하고 처리합니다. 트랩이 장치 인터럽트가 아니라면 예외여야 하며, xv6 커널에서 발생하면 항상 치명적인 오류입니다. 커널은 패닉을 호출 하고 실행을 멈춥니다.

kerneltrap이 타이머 인터럽트로 인해 호출되고 프로세스의 커널 스레드가 실행 중이면(스케줄러 스레드와 대조적으로), kerneltrap은 다른 스레드에 실행할 기회를 주기 위해 yield를 호출합니다. 어느 시점에서 그 스레드 중 하나가 yield하고, 우리 스레드와 해당 kerneltrap이 다시 시작되도록 합니다. 7장에서는 yield에서 무슨 일이 일어나는지 설명합니다.

kerneltrap의 작업이 완료되면 trap에 의해 중단된 코드로 돌아가야 합니다. yield가 sepc와 sstatus의 이전 모드를 방해했을 수 있으므로 kerneltrap은 시작할 때 이를 저장합니다. 이제 해당 제어 레지스터를 복원하고 kernelvec (kernel/kernelvec.S:50)로 돌아갑니다. kernelvec는 스택에서 저장된 레지스터를 팝하고 sret을 실행하는데, 이는 sepc를 pc로 복사하고 중단된 커널 코드를 재개합니다.

타이머 인터럽트로 인해 kerneltrap이 yield를 호출하면 트랩이 어떻게 반환되는지 생각해 볼 가치가 있습니다.

Xv6는 CPU가 사용자 공간에서 커널에 들어갈 때 CPU의 stvec를 kernelvec로 설정합니다. 이는 usertrap (kernel/trap.c:29)에서 확인할 수 있습니다. 커널이 실행을 시작했지만 stvec가 여전히 uservec로 설정된 시간 창이 있으며, 그 창 동안 장치 인터럽트가 발생하지 않는 것이 중요합니다. 다행히도 RISC-V는 트랩을 받기 시작할 때 항상 인터럽트를 비활성화하고, xv6는 stvec를 설정한 후에야 인터럽트를 다시 활성화합니다.

4.6 페이지 오류 예외

Xv6의 예외에 대한 대응은 매우 지루합니다. 사용자 공간에서 예외가 발생하면 커널은 오류가 발생한 프로세스를 종료합니다. 커널에서 예외가 발생하면 커널이 패닉합니다. 실제 운영

시스템은 종종 훨씬 더 흥미로운 방식으로 반응합니다.

예를 들어, 많은 커널은 COW(copy-on-write) 포크를 구현하기 위해 페이지 폴트를 사용합니다. COW(copy-on-write) 포크를 설명하기 위해 3장에서 설명한 xv6의 포크를 생각해 보세요. fork는 포크 시점에 자식의 초기 메모리 내용이 부모의 내용과 동일하도록 합니다. Xv6는 uvmcopy (kernel/vm.c:306)로 fork를 구현합니다. 자식에 대한 물리적 메모리를 할당하고 부모의 메모리를 복사합니다. 자식과 부모가 부모의 물리적 메모리를 공유할 수 있다면 더 효율적일 것입니다. 그러나 이것을 간단하게 구현하면 부모와 자식이 공유 스택과 힙에 쓰기를 함으로써 서로의 실행을 방해하게 되므로 작동하지 않습니다.

부모와 자식은 페이지 테이블 권한과 페이지 폴트를 적절히 사용하여 물리적 메모리를 안전하게 공유할 수 있습니다. CPU는 페이지 테이블에 매핑이 없거나 PTE_V 플래그가 지워진 매핑이 있거나 권한 비트 (PTE_R, PTE_W, PTE_X, PTE_U) 가 시도 중인 작업을 금지하는 매핑이 있는 가상 주소가 사용될 때 페이지 폴트 예외를 발생시킵니다. RISC-V는 세 가지 종류의 페이지 폴트를 구분합니다. 로드 페이지 폴트(로드 명령어가 가상 주소를 변환할 수 없는 경우), 저장 페이지 폴트(저장 명령어가 가상 주소를 변환할 수 없는 경우), 명령어 페이지 폴트(프로그램 카운터의 주소가 변환되지 않는 경우)입니다. scause 레지스터는 페이지 폴트의 유형을 나타내고 stval 레지스터는 변환할 수 없는 주소를 포함합니다.

COW 포크의 기본 계획은 부모와 자식이 처음에는 모든 물리적 페이지를 공유하지만 각각은 읽기 전용으로 매핑하는 것입니다(PTE_W 플래그가 지워짐). 부모와 자식은 공유된 물리적 메모리에서 읽을 수 있습니다. 둘 중 하나가 주어진 페이지를 쓰면 RISC-V CPU가 페이지 오류 예외를 발생시킵니다. 커널의 트랩 핸들러는 물리적 메모리의 새 페이지를 할당하고 오류가 발생한 주소가 매핑되는 물리적 페이지를 복사하여 응답합니다. 커널은 오류가 발생한 프로세스의 페이지 테이블에서 관련 PTE를 변경하여 복사본을 가리키고 읽기뿐만 아니라 쓰기도 허용한 다음 오류가 발생한 명령어에서 오류가 발생한 프로세스를 재개합니다. PTE가 쓰기를 허용하기 때문에 다시 실행된 명령어는 이제 오류 없이 실행됩니다. 쓰기 시 복사는 각 페이지가 포크, 페이지 오류, exec 및 종료의 기록에 따라 다양한 수의 페이지 테이블에서 참조될 수 있으므로 물리적 페이지를 언제 해제할 수 있는지 결정하는 데 도움이 되는 기록 보관이 필요합니다.

이러한 회계 처리로 중요한 최적화가 가능해졌습니다. 프로세스에서 저장 페이지 오류가 발생하고 물리적 페이지가 해당 프로세스의 페이지 테이블에서만 참조되는 경우, 복사가 필요하지 않습니다.

복사-쓰기는 fork가 메모리를 복사할 필요가 없기 때문에 fork를 더 빠르게 만듭니다 . 일부 메모리는 나중에 쓰일 때 복사해야 하지만, 대부분의 메모리는 결코 복사할 필요가 없는 경우가 많습니다. 일반적인 예로 fork 다음에 exec가 있습니다. fork 후에 몇 페이지가 쓰여질 수 있지만 , 그러면 자식의 exec가 부모로부터 상속받은 대부분의 메모리를 해제합니다.

Copy-on-write 포크는 이 메모리를 복사할 필요성을 없애줍니다. 게다가 COW 포크는 투명합니다. 애플리케이션을 수정하지 않아도 이점을 얻을 수 있습니다.

페이지 테이블과 페이지 폴트의 조합은 COW 포크 외에도 다양한 흥미로운 가능성을 열어줍니다. 널리 사용되는 또 다른 기능은 지연 할당이라고 하며, 두 부분으로 구성됩니다. 첫째, 애플리케이션이 sbrk를 호출하여 더 많은 메모리를 요청하면 커널은 크기가 증가한 것을 알아차리지만 물리적 메모리를 할당하지 않고 새로운 가상 주소 범위에 대한 PTE를 생성하지 않습니다. 둘째, 이러한 새로운 주소 중 하나에서 페이지 폴트가 발생하면 커널은 물리적 메모리 페이지를 할당하여 페이지 테이블에 매핑합니다. COW 포크와 마찬가지로 커널은 다음을 구현할 수 있습니다.

애플리케이션에 투명하게 지연 할당을 적용합니다.

애플리케이션은 종종 필요한 것보다 더 많은 메모리를 요청하기 때문에, 지연 할당은 이깁니다. 커널은 애플리케이션이 전혀 사용하지 않는 페이지에 대해 전혀 작업할 필요가 없습니다. 게다가 애플리케이션이 주소 공간을 많이 늘리라고 요청하는 경우, 지연 할당이 없는 sbrk는 비쌉니다. 애플리케이션이 기가바이트의 메모리를 요청하는 경우, 커널은 262,144개의 4096바이트 페이지를 할당하고 0으로 만들어야 합니다. 지연 할당은 이 비용을 시간에 걸쳐 분산할 수 있습니다. 반면에 지연 할당은 커널/사용자 전환을 수반하는 페이지 오류의 추가 오버헤드를 초래합니다. 운영 체제는 한 페이지 대신 페이지 오류당 연속된 페이지를 할당하고 이러한 페이지 오류에 대한 커널 진입/종료 코드를 특수화하여 이 비용을 줄일 수 있습니다.

페이지 폴트를 악용하는 또 다른 널리 사용되는 기능은 요구 페이징입니다. exec 에서 xv6는 애플리케이션의 모든 텍스트와 데이터를 메모리에 열심히 로드합니다. 애플리케이션은 크고 디스크에서 읽는 것이 비쌀 수 있으므로 이러한 시작 비용은 사용자에게 눈에 띄게 나타날 수 있습니다. 사용자가 셸에서 큰 애플리케이션을 시작하면 사용자가 응답을 보는 데 오랜 시간이 걸릴 수 있습니다. 응답 시간을 개선하기 위해 최신 커널은 사용자 주소 공간에 대한 페이지 테이블을 만들지만 페이지의 PTE를 유효하지 않은 것으로 표시합니다. 페이지 폴트 시 커널은 디스크에서 페이지의 내용을 읽고 사용자 주소 공간에 매핑합니다. COW 포크 및 지연 할당과 마찬가지로 커널은 이 기능을 애플리케이션에 투명하게 구현할 수 있습니다.

컴퓨터에서 실행되는 프로그램에는 컴퓨터의 RAM보다 많은 메모리가 필요할 수 있습니다.
우아하게 대처하기 위해 운영 체제는 디스크로 페이징을 구현할 수 있습니다. 아이디어는 사용자 페이지의 일부만 RAM에 저장하고 나머지는 페이징 영역의 디스크에 저장하는 것입니다. 커널은 페이징 영역에 저장된 메모리(따라서 RAM에 저장되지 않음)에 해당하는 PTE를 무효로 표시합니다. 애플리케이션이 디스크로 페이지 아웃된 페이지 중 하나를 사용하려고 하면 애플리케이션에 페이지 폴트가 발생하고 해당 페이지를 페이지 인해야 합니다. 커널 트랩 핸들러는 물리적 RAM의 페이지를 할당하고, 디스크에서 RAM으로 페이지를 읽고, 관련 PTE를 수정하여 RAM을 가리키도록 합니다.

페이지를 페이지 인해야 하지만 사용 가능한 실제 RAM이 없는 경우 어떻게 되나요? 이 경우 커널은 먼저 실제 페이지를 페이지 아웃하거나 디스크의 페이징 영역으로 내보내서 사용 가능한 페이지를 해제하고 해당 실제 페이지를 참조하는 PTE를 무효로 표시해야 합니다. 제거는 비용이 많이 들기 때문에 페이징은 드물게 수행되는 경우, 즉 애플리케이션이 메모리 페이지의 하위 집합만 사용하고 하위 집합의 합집합이 RAM에 맞는 경우 가장 잘 수행됩니다. 이 속성은 종종 참조의 좋은 지역성을 갖는 것으로 언급됩니다. 많은 가상 메모리 기술과 마찬가지로 커널은 일반적으로 애플리케이션에 투명한 방식으로 디스크로 페이징을 구현합니다.

컴퓨터는 종종 하드웨어가 제공하는 RAM 양에 관계없이 거의 또는 전혀 없는 실제 메모리로 작동합니다. 예를 들어, 클라우드 제공자는 많은 고객을 단일 머신에 멀티플렉싱하여 하드웨어를 비용 효율적으로 사용합니다. 또 다른 예로, 사용자는 스마트폰에서 소량의 실제 메모리로 많은 애플리케이션을 실행합니다. 이러한 설정에서 페이지를 할당하려면 먼저 기존 페이지를 제거해야 할 수 있습니다. 따라서 실제 메모리가 부족하면 할당 비용이 많이 듭니다.

여유 메모리가 부족할 때 지연 할당과 요구 시 페이징이 특히 유용합니다.
sbrk 또는 exec 에서 메모리를 열렬히 할당하면 메모리를 사용 가능하게 하기 위해 퇴거 비용이 추가로 발생합니다. 게다가 애플리케이션이 페이지를 사용하기 전에 운영 체제가 페이지를 퇴거했을 수 있으므로 열렬히 작업한 것이 낭비될 위험이 있습니다.

페이지 및 페이지 오류 예외를 결합하는 기타 기능에는 다음이 포함됩니다.
스택과 메모리 맵 파일.

4.7 현실 세계

트램펄린과 트랩프레임은 지나치게 복잡해 보일 수 있습니다. 추진력은 RISC-V가 트랩을 강제할 때 의도적으로 가능한 한 적게 수행하여 매우 빠른 트랩 처리가 가능하다는 것입니다. 이는 중요한 것으로 밝혀졌습니다. 결과적으로 커널 트랩 핸들러의 처음 몇 가지 명령어는 사용자 환경에서 효과적으로 실행되어야 합니다. 즉, 사용자 페이지 테이블과 사용자 레지스터 내용입니다. 그리고 트랩 핸들러는 처음에는 실행 중인 프로세스의 ID나 커널 페이지 테이블의 주소와 같은 유용한 사실을 모릅니다. RISC-V가 커널이 사용자 공간에 들어가기 전에 정보를 숨길 수 있는 보호된 장소, 즉 sscratch 레지스터와 커널 메모리를 가리키지만 PTE_U가 없어서 보호되는 사용자 페이지 테이블 항목을 제공하기 때문에 해결책이 가능합니다. Xv6의 트램펄린과 트랩프레임은 이러한 RISC-V 기능을 활용합니다.

커널 메모리가 모든 프로세스의 사용자 페이지 테이블에 매핑되면(적절한 PTE 권한 플래그 사용) 특수 트램폴린 페이지에 대한 필요성이 없어질 수 있습니다. 그러면 사용자 공간에서 커널로 트래핑할 때 페이지 테이블 전환이 필요 없게 됩니다. 그러면 커널의 시스템 호출 구현이 매핑되는 현재 프로세스의 사용자 메모리를 활용하여 커널 코드가 사용자 포인터를 직접 역참조할 수 있습니다.

많은 운영 체제가 효율성을 높이기 위해 이러한 아이디어를 사용했습니다. Xv6는 사용자 포인터의 부주의한 사용으로 인해 커널에서 보안 버그가 발생할 가능성을 줄이고 사용자 및 커널 가상 주소가 겹치지 않도록 하는 데 필요한 복잡성을 줄이기 위해 이를 피합니다.

프로덕션 운영 체제는 copy-on-write 포크, 지연 할당, 수요 페이징, 디스크 페이징, 메모리 매핑 파일 등을 구현합니다. 나아가 프로덕션 운영 체제는 모든 물리적 메모리를 애플리케이션이나 캐시(예: 파일 시스템의 버퍼 캐시, 나중에 섹션 8.2에서 다룰 예정)에 사용하려고 합니다. Xv6는 이 점에서 순진합니다. 사용자는 운영 체제가 사용자가 지불한 물리적 메모리를 사용하기를 원하지만 xv6는 그렇지 않습니다. 나아가 xv6에서 메모리가 부족하면 실행 중인 애플리케이션에 오류를 반환하거나 다른 애플리케이션의 페이지를 제거하는 대신 해당 애플리케이션을 종료합니다.

4.8 연습문제

1. copyin 및 copyinstr 함수는 소프트웨어에서 사용자 페이지 테이블을 탐색합니다. 커널 페이지 테이블을 설정하여 커널이 사용자 프로그램을 매핑하고, copyin 및 copyinstr이 memcpy를 사용하여 시스템 호출 인수를 커널 공간으로 복사할 수 있도록 하며, 하드웨어가 페이지 테이블 탐색을 수행하도록 합니다.

2. 지연 메모리 할당을 구현합니다.

3. COW 포크를 구현합니다.

4. 모든 사용자 주소 공간에서 특수 TRAPFRAME 페이지 매핑을 제거할 방법이 있습니까? 예를 들어, uservec를 수정하여 32개의 사용자 레지스터를 커널 스택에 푸시하거나 proc 구조에 저장할 수 있습니까?

5. xv6를 수정하여 특수 TRAMPOLINE 페이지 매핑을 제거할 수 있습니까?

5장

인터럽트 및 장치 드라이버

드라이버는 특정 장치를 관리하는 운영 체제의 코드입니다. 장치 하드웨어를 구성하고, 장치에 작업을 수행하도록 지시하고, 결과 인터럽트를 처리하고, 장치의 I/O를 기다리고 있을 수 있는 프로세스와 상호 작용합니다. 드라이버 코드는 드라이버가 관리하는 장치와 동시에 실행되기 때문에 까다로울 수 있습니다. 또한 드라이버는 장치의 하드웨어 인터페이스를 이해해야 하는데, 이는 복잡하고 제대로 문서화되지 않을 수 있습니다.

운영 체제의 주의가 필요한 장치는 일반적으로 인터럽트를 생성하도록 구성할 수 있으며, 이는 트랩의 한 유형입니다. 커널 트랩 처리 코드는 장치가 인터럽트를 발생시키고 드라이버의 인터럽트 핸들러를 호출할 때를 인식합니다. xv6에서 이 디스패치는 devintr (kernel/trap.c:178)에서 발생합니다.

많은 장치 드라이버는 두 가지 맥락에서 코드를 실행합니다. 프로세스의 커널 스레드에서 실행되는 상단 절반과 인터럽트 시간에 실행되는 하단 절반입니다. 상단 절반은 장치가 I/O를 수행하도록 원하는 read 및 write와 같은 시스템 호출을 통해 호출됩니다. 이 코드는 하드웨어에 작업을 시작하도록 요청할 수 있습니다(예: 디스크에 블록을 읽도록 요청). 그런 다음 코드는 작업이 완료될 때까지 기다립니다. 결국 장치는 작업을 완료하고 인터럽트를 발생시킵니다. 하단 절반 역할을 하는 드라이버의 인터럽트 핸들러는 어떤 작업이 완료되었는지 파악하고, 적절한 경우 대기 중인 프로세스를 깨우고, 하드웨어에 대기 중인 다음 작업을 시작하도록 지시합니다.

5.1 코드: 콘솔 입력

콘솔 드라이버 (kernel/console.c) 드라이버 구조의 간단한 예시입니다. 콘솔 드라이버는 RISC-V에 연결된 UART 직렬 포트 하드웨어를 통해 사람이 입력한 문자를 허용합니다.
콘솔 드라이버는 백스페이스 및 컨트롤-u와 같은 특수 입력 문자를 처리하면서 한 번에 한 줄씩 입력을 축적합니다. 셸과 같은 사용자 프로세스는 read 시스템 호출을 사용하여 콘솔에서 입력 줄을 페치합니다. QEMU에서 xv6에 입력을 입력하면 키 입력은 QEMU의 시뮬레이션된 UART 하드웨어를 통해 xv6에 전달됩니다.

드라이버가 통신하는 UART 하드웨어는 QEMU에서 에뮬레이션한 16550 칩[13]입니다. 실제 컴퓨터에서 16550은 터미널이나 다른 컴퓨터에 연결하는 RS232 직렬 링크를 관리합니다.
QEMU를 실행하면 키보드와 디스플레이에 연결됩니다.
UART 하드웨어는 소프트웨어에 메모리 매핑 제어 레지스터 세트로 나타납니다.

즉, RISC-V 하드웨어가 UART 장치에 연결하는 몇 가지 물리적 주소가 있어 로드 및 저장이 RAM이 아닌 장치 하드웨어와 상호 작용합니다. UART의 메모리 매핑 주소는 0x10000000 또는 UART0 (kernel/memlayout.h:21)에서 시작합니다. UART 제어 레지스터는 몇 개 있으며, 각각은 바이트 너비입니다. UART0에서의 오프셋은 (kernel/uart.c:22)에 정의되어 있습니다. 예를 들어, LSR 레지스터에는 소프트웨어가 입력 문자를 읽을 때까지 기다리고 있는지 여부를 나타내는 비트가 들어 있습니다. 이러한 문자(있는 경우)는 RHR 레지스터에서 읽을 수 있습니다. 문자가 읽힐 때마다 UART 하드웨어는 대기 문자의 내부 FIFO에서 삭제하고 FIFO가 비어 있으면 LSR의 "준비" 비트를 지웁니다. UART 전송 하드웨어는 수신 하드웨어와 크게 독립적입니다. 소프트웨어가 THR에 바이트를 쓰면 UART가 해당 바이트를 전송합니다.

Xv6의 메인 호출 consoleinit (kernel/console.c:182) UART 하드웨어를 초기화합니다. 이 코드는 UART가 각 입력 바이트를 수신할 때 수신 인터럽트를 생성하고, UART가 출력 바이트를 보내는 것을 마칠 때마다 전송 완료 인터럽트를 생성하도록 UART를 구성합니다 (kernel/uart.c:53).

xv6 쉘은 init.c (user/init.c:19) 에 의해 열린 파일 기술자를 통해 콘솔에서 읽습니다 .
read 시스템 호출에 대한 호출은 커널을 거쳐 consoleread (kernel/ console.c:80) 로 전달됩니다 . consoleread는 입력이 도착할 때까지(인터럽트를 통해) 기다리고 cons.buf에 버퍼링하고, 입력을 사용자 공간에 복사하고(전체 줄이 도착한 후) 사용자 프로세스로 돌아갑니다. 사용자가 아직 전체 줄을 입력하지 않은 경우 모든 읽기 프로세스는 sleep 호출 (kernel/ console.c:96) 에서 기다립니다. (7장에서는 수면에 대한 자세한 내용을 설명합니다.)

사용자가 문자를 입력하면 UART 하드웨어는 RISC-V에 인터럽트를 발생시키라고 요청하고, 이는 xv6의 트랩 핸들러를 활성화합니다. 트랩 핸들러는 devintr (kernel/trap.c:178)을 호출합니다. RISC-V scause 레지스터를 살펴보고 인터럽트가 외부 장치에서 발생한 것임을 알아냅니다. 그런 다음 PLIC[3]라는 하드웨어 장치에 어떤 장치가 인터럽트를 발생시켰는지 알려달라고 요청합니다 (kernel/trap.c:187). UART인 경우 devintr는 uartintr를 호출합니다.

uartintr (커널/uart.c:176) UART 하드웨어에서 대기 중인 입력 문자를 읽고 이를 consoleintr에 전달합니다 (kernel/console.c:136); 문자를 기다리지 않습니다. 이후 입력이 새로운 인터럽트를 발생시키기 때문입니다. consoleintr의 역할은 전체 줄이 도착할 때까지 cons.buf에 입력 문자를 누적하는 것입니다. consoleintr는 백스페이스와 몇 가지 다른 문자를 특별히 처리합니다. 줄 바꿈이 도착하면 consoleintr는 대기 중인 consoleread(있는 경우)를 깨웁니다.

consoleread는 깨어나면 cons.buf에서 전체 줄을 관찰하고, 이를 사용자 공간에 복사하고, (시스템 호출 메커니즘을 통해) 사용자 공간으로 반환합니다.

5.2 코드: 콘솔 출력

콘솔에 연결된 파일 기술자에 대한 쓰기 시스템 호출은 결국 uartputc (kernel/uart.c:87)에 도착합니다. 장치 드라이버는 쓰기 프로세스가 UART가 전송을 마칠 때까지 기다릴 필요가 없도록 출력 버퍼(uart_tx_buf)를 유지합니다. 대신 uartputc는 각 문자를 버퍼에 추가하고, uartstart를 호출하여 장치 전송을 시작하고(아직 시작되지 않은 경우) 반환합니다. uartputc가 기다리는 유일한 상황은 버퍼가 이미 가득 찬 경우입니다.

UART가 바이트 전송을 완료할 때마다 인터럽트가 생성됩니다. uartintr은 uartstart를 호출합니다.

장치가 실제로 전송을 마쳤는지 확인하고 장치에 다음 버퍼링된 출력 문자를 전달합니다. 따라서 프로세스가 콘솔에 여러 바이트를 쓰는 경우 일반적으로 첫 번째 바이트는 uartputc의 uartstart 호출에 의해 전송되고 나머지 버퍼링된 바이트는 전송 완료 인터럽트가 도착하면 uartintr의 uartstart 호출에 의해 전송됩니다.

주목할 일반적인 패턴은 버퍼링과 인터럽트를 통해 장치 활동을 프로세스 활동에서 분리하는 것입니다. 콘솔 드라이버는 프로세스가 읽기를 기다리고 있지 않더라도 입력을 처리할 수 있습니다. 후속 읽기는 입력을 볼 것입니다. 마찬가지로 프로세스는 장치를 기다리지 않고도 출력을 보낼 수 있습니다. 이러한 분리는 프로세스가 장치 I/O와 동시에 실행되도록 하여 성능을 높일 수 있으며, 장치가 느리거나(UART의 경우) 즉각적인 주의가 필요한 경우(입력된 문자를 에코하는 경우) 특히 중요합니다. 이 아이디어는 때때로 I/O라고 합니다.

동시성.

5.3 드라이버의 동시성

consoleread와 consoleintr에서 acquire에 대한 호출을 알아차렸을 것입니다. 이러한 호출은 콘솔 드라이버의 데이터 구조를 동시 액세스로부터 보호하는 잠금을 획득합니다. 여기에는 세 가지 동시성 위험이 있습니다. 서로 다른 CPU의 두 프로세스가 동시에 consoleread를 호출할 수 있습니다. 하드웨어가 CPU에 콘솔(실제로는 UART) 인터럽트를 전달하도록 요청할 수 있는데, 이 CPU가 이미 consoleread 내부에서 실행 중일 수 있습니다. 하드웨어가 consoleread가 실행되는 동안 다른 CPU에 콘솔 인터럽트를 전달할 수 있습니다. 이러한 위험은 경쟁이나 교착 상태를 초래할 수 있습니다. 6장에서는 이러한 문제와 잠금이 이를 어떻게 해결할 수 있는지 살펴봅니다.

드라이버에서 동시성이 주의를 필요로 하는 또 다른 방법은 한 프로세스가 장치로부터 입력을 기다리고 있지만, 입력의 인터럽트 신호 도착이 다른 프로세스(또는 전혀 프로세스가 실행되지 않음)가 실행 중일 때 도착할 수 있다는 것입니다. 따라서 인터럽트 핸들러는 자신이 인터럽트한 프로세스나 코드에 대해 생각할 수 없습니다. 예를 들어, 인터럽트 핸들러는 현재 프로세스의 페이지 테이블로 안전하게 copyout을 호출할 수 없습니다. 인터럽트 핸들러는 일반적으로 비교적 적은 작업(예: 입력 데이터를 버퍼에 복사하기만 함)을 수행하고 나머지 작업을 수행하기 위해 상위 절반 코드를 깨웁니다.

5.4 타이머 인터럽트

Xv6는 타이머 인터럽트를 사용하여 클록을 유지하고 컴퓨트 바운드 프로세스 간에 전환할 수 있도록 합니다. usertrap 및 kerneltrap의 yield 호출이 이 전환을 발생시킵니다. 타이머 인터럽트는 각 RISC-V CPU에 연결된 클록 하드웨어에서 발생합니다. Xv6는 이 클록 하드웨어를 프로그래밍하여 각 CPU를 주기적으로 인터럽트합니다.

RISC-V는 타이머 인터럽트가 슈퍼바이저 모드가 아닌 머신 모드에서 수행되어야 합니다. RISC-V 머신 모드는 페이징 없이 실행되고 별도의 제어 레지스터 세트가 있으므로 머신 모드에서 일반 xv6 커널 코드를 실행하는 것은 실용적이지 않습니다. 결과적으로 xv6는 위에 제시된 트랩 메커니즘과 완전히 별도로 타이머 인터럽트를 처리합니다.

main 전에 start.c에서 머신 모드로 실행되는 코드는 타이머 인터럽트를 수신하도록 설정됩니다 (kernel/start.c:63). 작업의 일부는 특정 지연 후 인터럽트를 생성하도록 CLINT 하드웨어(코어 로컬 인터럽터)를 프로그래밍하는 것입니다. 또 다른 부분은 스크래치 영역을 설정하는 것입니다.

트랩프레임은 타이머 인터럽트 핸들러가 레지스터와 CLINT 레지스터의 주소를 저장하는 데 도움을 줍니다.
마지막으로, start는 mtvec를 timervec로 설정하고 타이머 인터럽트를 활성화합니다.

타이머 인터럽트는 사용자 또는 커널 코드가 실행되는 어느 시점에서나 발생할 수 있습니다. 커널이 중요한 작업 중에 타이머 인터럽트를 비활성화할 수 있는 방법은 없습니다. 따라서 타이머 인터럽트 핸들러는 인터럽트된 커널 코드를 방해하지 않는 방식으로 작업을 수행해야 합니다. 기본 전략은 핸들러가 RISC-V에 "소프트웨어 인터럽트"를 발생시키고 즉시 반환하도록 요청하는 것입니다. RISC-V는 일반적인 트랩 메커니즘으로 소프트웨어 인터럽트를 커널에 전달하고 커널이 이를 비활성화할 수 있도록 합니다. 타이머 인터럽트에 의해 생성된 소프트웨어 인터럽트를 처리하는 코드는 devintr (kernel/trap.c:205)에서 볼 수 있습니다.

머신 모드 타이머 인터럽트 핸들러는 timervec (kernel/kernelvec.S:95)입니다. 시작으로 준비된 스크래치 영역에 몇 개의 레지스터를 저장하고, CLINT에 다음 타이머 인터럽트를 생성할 시기를 알리고, RISC-V에 소프트웨어 인터럽트를 발생시키라고 요청하고, 레지스터를 복원하고 반환합니다. 타이머 인터럽트 핸들러에는 C 코드가 없습니다.

5.5 현실 세계

Xv6는 커널에서 실행하는 동안과 사용자 프로그램을 실행하는 동안 장치 및 타이머 인터럽트를 허용합니다. 타이머 인터럽트는 커널에서 실행하는 경우에도 타이머 인터럽트 핸들러에서 스레드 전환(yield 호출)을 강제합니다. 커널 스레드 간에 CPU를 공정하게 시간 분할하는 기능은 커널 스레드가 사용자 공간으로 돌아가지 않고 가끔 많은 시간을 컴퓨팅에 할애하는 경우에 유용합니다. 그러나 커널 코드가 타이머 인터럽트로 인해 일시 중단되었다가 나중에 다른 CPU에서 재개될 수 있다는 것을 염두에 두어야 하는 필요성은 xv6에서 약간의 복잡성의 원인입니다(6.6절 참조). 장치 및 타이머 인터럽트가 사용자 코드를 실행하는 동안만 발생하면 커널을 다소 더 단순하게 만들 수 있습니다.

일반적인 컴퓨터에서 모든 장치를 완벽하게 지원하는 것은 많은 작업입니다. 장치가 많고, 장치에는 많은 기능이 있으며, 장치와 드라이버 간의 프로토콜은 복잡하고 제대로 문서화되지 않을 수 있기 때문입니다. 많은 운영 체제에서 드라이버는 코어 커널보다 더 많은 코드를 차지합니다.

UART 드라이버는 UART 제어 레지스터를 읽어 한 번에 한 바이트씩 데이터를 검색합니다. 이 패턴을 프로그래밍된 I/O라고 하는데, 소프트웨어가 데이터 이동을 주도하기 때문입니다. 프로그래밍된 I/O는 간단하지만 너무 느려서 높은 데이터 속도에서 사용할 수 없습니다. 많은 양의 데이터를 고속으로 이동해야 하는 장치는 일반적으로 직접 메모리 액세스(DMA)를 사용합니다.
DMA 장치 하드웨어는 들어오는 데이터를 RAM에 직접 쓰고 나가는 데이터를 RAM에서 읽습니다. 최신 디스크 및 네트워크 장치는 DMA를 사용합니다.
DMA 장치용 드라이버는 RAM에 데이터를 준비한 다음 제어 레지스터에 한 번 쓰기를 사용하여 장치에 준비된 데이터를 처리하도록 지시합니다.

장치에 예측할 수 없는 시간에 주의가 필요하고, 너무 자주 필요하지 않을 때 인터럽트가 발생하는 것은 합리적입니다.
하지만 인터럽트는 CPU 오버헤드가 높습니다. 따라서 네트워크 및 디스크 컨트롤러와 같은 고속 장치는 인터럽트 필요성을 줄이는 트릭을 사용합니다. 한 가지 트릭은 들어오거나 나가는 요청 전체에 대해 단일 인터럽트를 발생시키는 것입니다. 또 다른 트릭은 드라이버가 인터럽트를 완전히 비활성화하고 주기적으로 장치를 확인하여 주의가 필요한지 확인하는 것입니다. 이 기술을 폴링이라고 합니다. 폴링은 장치가 매우 빠르게 작업을 수행하는 경우 의미가 있지만 장치가 대부분 유휴 상태인 경우 CPU 시간을 낭비합니다. 일부 드라이버는 폴링과 인터럽트 간에 동적으로 전환합니다.

현재 장치 부하에 따라 다릅니다.

UART 드라이버는 들어오는 데이터를 먼저 커널의 버퍼에 복사한 다음 사용자 공간에 복사합니다.
이는 낮은 데이터 전송 속도에서는 의미가 있지만, 이러한 이중 복사는 데이터를 매우 빠르게 생성하거나 소비하는 장치의 성능을 크게 저하시킬 수 있습니다. 일부 운영 체제는 종종 DMA를 사용하여 사용자 공간 버퍼와 장치 하드웨어 간에 데이터를 직접 이동할 수 있습니다.

1장에서 언급했듯이 콘솔은 애플리케이션에 일반 파일로 나타나고 애플리케이션은 read 및 write 시스템 호출을 사용하여 입력을 읽고 출력을 씁니다. 애플리케이션은 표준 파일 시스템 호출을 통해 표현할 수 없는 장치의 측면을 제어하고 싶을 수 있습니다(예: 콘솔 드라이버에서 라인 버퍼링 활성화/비활성화). Unix 운영 체제는 이러한 경우에 ioctl 시스템 호출을 지원합니다.

일부 컴퓨터 사용에는 시스템이 제한된 시간 내에 응답해야 합니다. 예를 들어, 안전이 중요한 시스템에서 마감일을 놓치면 재해가 발생할 수 있습니다. Xv6는 하드 실시간 설정에 적합하지 않습니다. 하드 실시간을 위한 운영 체제는 최악의 경우 응답 시간을 결정하기 위한 분석을 허용하는 방식으로 애플리케이션과 연결되는 라이브러리인 경향이 있습니다. Xv6는 또한 소프트 실시간 애플리케이션에도 적합하지 않습니다. 마감일을 가끔 놓치는 것이 허용되는 경우, xv6의 스케줄러가 너무 단순하고 인터럽트가 오랫동안 비활성화되는 커널 코드 경로가 있기 때문입니다.

5.6 연습문제

1. uart.c를 수정하여 인터럽트를 전혀 사용하지 않도록 합니다. console.c도 수정해야 할 수 있습니다.
2. 이더넷 카드용 드라이버를 추가합니다.

6장

잠금

xv6를 포함한 대부분의 커널은 여러 활동의 실행을 인터리빙합니다. 인터리빙의 한 가지 원인은 멀티프로세서 하드웨어입니다. 즉, xv6의 RISC-V와 같이 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 단계에 있거나 여러 CPU가 동시에 동일한 데이터를 업데이트할 가능성을 높입니다. 신중한 설계 없이 이러한 병렬 액세스는 잘못된 결과나 손상된 데이터 구조를 생성할 가능성이 높습니다. 단일 프로세서에서도 커널은 여러 스레드 간에 CPU를 전환하여 실행이 인터리빙되도록 할 수 있습니다. 마지막으로 인터럽트가 잘못된 시간에 발생하면 일부 인터럽트 가능 코드와 동일한 데이터를 수정하는 장치 인터럽트 핸들러가 데이터를 손상시킬 수 있습니다. 동시성이라는 단어는 멀티프로세서 병렬성, 스레드 전환 또는 인터럽트로 인해 여러 명령 스트림이 인터리빙되는 상황을 나타냅니다.

커널은 동시에 액세스되는 데이터로 가득 차 있습니다. 예를 들어, 두 개의 CPU가 동시에 kalloc을 호출하여 자유 목록의 머리에서 동시에 튀어나올 수 있습니다. 커널 설계자는 병렬성을 통해 성능을 높이고 응답성을 높일 수 있기 때문에 많은 동시성을 허용하고 싶어합니다. 그러나 결과적으로 커널 설계자는 이러한 동시성에도 불구하고 정확성을 확신해야 합니다. 올바른 코드에 도달하는 방법은 여러 가지가 있으며, 어떤 방법은 다른 방법보다 추론하기 쉽습니다. 동시성 하에서 정확성을 목표로 하는 전략과 이를 지원하는 추상화를 동시성 제어 기술이라고 합니다.

Xv6는 상황에 따라 여러 가지 동시성 제어 기술을 사용합니다. 더 많은 기술을 사용할 수 있습니다. 이 장에서는 널리 사용되는 기술인 잠금에 초점을 맞춥니다. 잠금은 상호 배제를 제공하여 한 번에 하나의 CPU만 잠금을 보유할 수 있도록 합니다. 프로그래머가 각 공유 데이터 항목에 잠금을 연결하고 코드가 항목을 사용할 때 항상 연결된 잠금을 유지하는 경우 해당 항목은 한 번에 하나의 CPU에서만 사용됩니다. 이 상황에서 잠금이 데이터 항목을 보호한다고 합니다. 잠금은 이해하기 쉬운 동시성 제어 메커니즘이지만 잠금의 단점은 동시 작업을 직렬화하기 때문에 성능을 제한할 수 있다는 것입니다.

이 장의 나머지 부분에서는 xv6에 잠금이 필요한 이유, xv6에서 잠금을 구현하는 방법, 그리고 잠금을 사용하는 방법을 설명합니다.

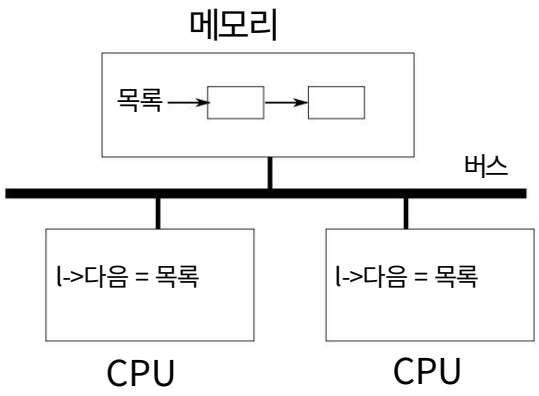

그림 6.1: 단순화된 SMP 아키텍처

6.1 종족

잠금이 필요한 이유의 예로 종료된 자식이 있는 두 프로세스가 wait을 호출하는 것을 생각해 보세요.
두 개의 다른 CPU. wait은 자식의 메모리를 해제합니다. 따라서 각 CPU에서 커널은 kfree를 호출합니다.
자식의 메모리 페이지를 해제합니다. 커널 할당자는 연결 리스트를 유지합니다: kalloc() (kernel/kalloc.c:69) 여유
페이지 목록에서 메모리 페이지를 팝하고 kfree()를 실행합니다 (kernel/kalloc.c:47)
페이지를 자유 목록에 푸시합니다. 최상의 성능을 위해 kfrees가
두 개의 부모 프로세스는 서로를 기다릴 필요 없이 병렬로 실행되지만
xv6의 kfree 구현을 고려하면 정확하지 않습니다.

그림 6.1은 설정을 더 자세히 보여줍니다. 즉, 사용 가능한 페이지의 연결 목록은 메모리에 있습니다.
로드 및 저장 명령을 사용하여 목록을 조작하는 두 CPU가 공유합니다. (실제로는
프로세서에는 캐시가 있지만 개념적으로 다중 프로세서 시스템은 마치 단일 프로세서가 있는 것처럼 작동합니다.
공유 메모리) 동시 요청이 없는 경우 목록 푸시 작업을 구현할 수 있습니다.
다음과 같습니다:

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

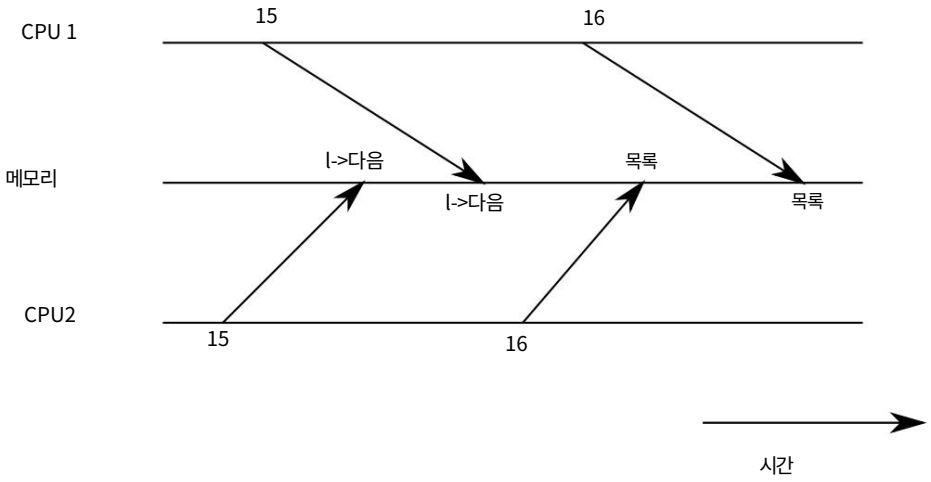

그림 6.2: 예시 레이스

```
17          }
```

이 구현은 격리된 상태에서 실행하면 정확합니다. 그러나 더 많은 코드가 실행되면 정확하지 않습니다.
두 개 이상의 사본이 동시에 실행됩니다. 두 개의 CPU가 동시에 push를 실행하면 둘 다
그림 6.1에 표시된 대로 15번째 줄을 실행한 다음 16번째 줄을 실행하면 잘못된 결과가 발생합니다.
그림 6.2에서 설명한 대로 결과가 나타납니다. 그런 다음 다음이 다음 으로 설정된 두 개의 목록 요소가 있습니다.
목록 의 이전 값입니다 . 목록 에 대한 두 가지 할당이 16번째 줄에서 발생하면 두 번째 할당은
첫 번째를 덮어쓰면 첫 번째 할당에 포함된 요소가 손실됩니다.

16번째 줄에서 손실된 업데이트는 레이스의 한 예입니다. 레이스는 메모리가
위치가 동시에 액세스되고 적어도 하나의 액세스가 쓰기입니다. 경쟁은 종종 버그의 신호입니다.
업데이트가 손실되었거나(쓰기 액세스인 경우) 불완전하게 업데이트된 데이터 구조를 읽은 것입니다.
레이스의 결과는 컴파일러가 생성한 머신 코드, 타이밍에 따라 달라집니다.
관련된 두 CPU와 메모리 시스템이 해당 메모리 작업을 어떻게 정렬하는지
이는 경쟁으로 인한 오류를 재현하고 디버깅하기 어렵게 만들 수 있습니다. 예를 들어, 인쇄 추가
디버깅하는 동안 명령문을 실행 하면 실행 타이밍이 충분히 변경될 수 있습니다.
인종이 사라지다.

경쟁을 피하는 일반적인 방법은 잠금을 사용하는 것입니다. 잠금은 상호 배제를 보장하므로 하나만
CPU가 한 번에 하나의 민감한 푸시 라인을 실행할 수 있기 때문에 위의 시나리오는 불가능합니다.
위 코드의 올바르게 잠긴 버전은 몇 줄만 추가합니다(노란색으로 강조 표시됨):

```
6           구조체 요소 *리스트 = 0;
7           구조체 잠금 목록 잠금;
8
9           무효의
10          푸시(int 데이터)
11          {
12              구조체 요소 *l;
13              l = malloc(크기 *l);
14              l->데이터 = 데이터;
```

```
15
16          획득(&listlock); l->다음 = 목록; 목록
17          = l; 해제(&listlock);
18
19
20      }
```

획득 과 해제 사이의 명령어 시퀀스를 종종 중요 섹션이라고 합니다.
잠금은 일반적으로 목록을 보호하는 것으로 간주됩니다 .

잠금이 데이터를 보호한다고 말할 때, 실제로는 잠금이 데이터에 적용되는 일부 불변식 컬렉션을 보호한다는 것을 의미합니다. 불변식은 작업 간에 유지되는 데이터 구조의 속성입니다. 일반적으로 작업의 올바른 동작은 작업이 시작될 때 불변식이 참이어야 합니다. 작업은 일시적으로 불변식을 위반할 수 있지만 완료되기 전에 다시 설정해야 합니다. 예를 들어, 연결 목록의 경우 불변식은 list 가 목록의 첫 번째 요소를 가리키고 각 요소의 다음 필드가 다음 요소를 가리킨다는 것입니다. push 의 구현은 이 불변식을 일시적으로 위반합니다. 17번째 줄에서 l은 다음 목록 요소를 가리키지만 list는 아직 l을 가리키지 않습니다 (18번째 줄에서 다시 설정). 위에서 살펴본 경쟁은 두 번째 CPU가 목록 불변식이 (일시적으로) 위반되는 동안 목록 불변식에 의존하는 코드를 실행했기 때문에 발생했습니다. 잠금을 적절히 사용하면 한 번에 하나의 CPU만 중요 섹션의 데이터 구조에서 작업할 수 있으므로 데이터 구조의 불변성이 유지되지 않을 때 어떤 CPU도 데이터 구조 작업을 실행하지 않습니다.

잠금은 동시에 발생하는 중요 섹션을 직렬화하여 한 번에 하나씩 실행되도록 하고, 따라서 불변성을 보존하는 것으로 생각할 수 있습니다(중요 섹션이 격리 상태에서 정확하다고 가정). 또한 동일한 잠금으로 보호되는 중요 섹션은 서로에 대해 원자적이라고 생각할 수 있으므로 각각은 이전 중요 섹션의 전체 변경 사항만 보고 부분적으로 완료된 업데이트는 절대 보지 못합니다.

정확성에 유용하지만 잠금은 본질적으로 성능을 제한합니다. 예를 들어 두 프로세스가 kfree를 동시에 호출하면 잠금이 두 개의 중요 섹션을 직렬화하므로 다른 CPU에서 실행하는 데 이점이 없습니다. 여러 프로세스가 동시에 동일한 잠금을 원하거나 잠금이 경합을 경험하는 경우 여러 프로세스가 충돌한다고 말합니다. 커널 설계의 주요 과제는 병렬성을 추구하면서 잠금 경합을 피하는 것입니다. Xv6는 이를 거의 수행하지 않지만 정교한 커널은 잠금 경합을 피하기 위해 특별히 데이터 구조와 알고리즘을 구성합니다.

목록 예에서 커널은 CPU마다 별도의 사용 가능 목록을 유지하고 현재 CPU의 목록이 비어 있고 다른 CPU에서 메모리를 훔쳐야 하는 경우에만 다른 CPU의 사용 가능 목록에 접근합니다.
다른 사용 사례에서는 더 복잡한 설계가 필요할 수도 있습니다.

잠금의 배치도 성능에 중요합니다. 예를 들어, push 에서 acquire를 13번째 줄 앞까지 옮기는 것이 옳을 것입니다. 하지만 이렇게 하면 malloc 에 대한 호출이 직렬화되기 때문에 성능이 저하될 가능성이 있습니다. 아래의 "잠금 사용" 섹션에서는 acquire 및 release 호출을 삽입할 위치에 대한 몇 가지 지침을 제공합니다.

6.2 코드: 잠금

Xv6에는 스핀락과 슬립락이라는 두 가지 유형의 잠금이 있습니다. 스핀락부터 시작하겠습니다. Xv6는
구조체 spinlock 으로서의 spinlock (kernel/spinlock.h:2). 구조의 중요한 필드는 다음과 같습니다.
잠긴, 잠금이 사용 가능할 때는 0이고, 잠금이 유지될 때는 0이 아닌 단어. 논리적으로 xv6
다음과 같은 코드를 실행하여 잠금을 획득해야 합니다.

```
21      무효의
22      acquire(struct spinlock *lk) // 작동하지 않습니다!
23      {
24          을 위한(;;) {
25              if(lk->잠금 == 0) {
26                  잠금->잠금 = 1;
27                  부서지다;
28              }
29          }
30      }
```

불행히도, 이 구현은 멀티프로세서에서 상호 배제를 보장하지 않습니다.
두 개의 CPU가 동시에 25번째 줄에 도달하여 lk->locked 가 0인 것을 확인한 다음
둘 다 26번째 줄을 실행하여 잠금을 잡습니다. 이 시점에서 두 개의 다른 CPU가 잠금을 잡고 있습니다.
상호 배타 속성을 위반합니다. 우리에게 필요한 것은 25행과 26행을 다음과 같이 실행하는 방법입니다.
원자적(즉, 나눌 수 없는) 단계.

잠금이 널리 사용되기 때문에 멀티코어 프로세서는 일반적으로 25행과 26행의 원자 버전을 구현하는 명령어를 제공합니다.
RISC-V에서 이 명령어는 amoswap r, a입니다.
amoswap은 메모리 주소 a의 값을 읽고, 레지스터 r의 내용을 해당 주소에 씁니다.
그리고 읽은 값을 r에 넣습니다. 즉, 레지스터의 내용과 메모리의 내용을 바꿉니다.
주소. 특수 하드웨어를 사용하여 다른 CPU가 작동하지 않도록 이 시퀀스를 원자적으로 수행합니다.
읽기와 쓰기 사이에 메모리 주소를 사용합니다.

Xv6의 획득 (kernel/spinlock.c:22) 이식 가능한 C 라이브러리 호출 __sync_lock_test_and_set을 사용합니다.
이는 amoswap 명령어 로 요약되며 반환 값은 이전(스왑된) 내용입니다.
lk->locked. acquire 함수 는 스왑을 루프로 감싸서 다시 시도(회전)하여 잠금 상태가 될 때까지 계속합니다.
잠금을 획득했습니다. 각 반복은 하나를 lk->locked 로 바꾸고 이전 값을 확인합니다.
이전 값이 0이면 잠금을 획득한 것이고 스왑은 lk->locked를 설정합니다.
1로. 이전 값이 1이면 다른 CPU가 잠금을 유지하고 우리가
원자적으로 하나를 lk->locked 로 바꿔도 값은 변경되지 않습니다.

잠금이 획득되면 디버깅을 위해 잠금을 획득한 CPU에 대한 레코드를 획득합니다 .
lk- >cpu 필드는 잠금으로 보호되므로 잠금을 유지하는 동안에만 변경해야 합니다.

함수 릴리스 (kernel/spinlock.c:47) acquire 의 반대입니다 : lk->cpu 를 지웁니다.
필드를 닫은 다음 잠금을 해제합니다. 개념적으로 해제하려면 lk->locked에 0을 할당하기만 하면 됩니다.
C 표준은 컴파일러가 여러 저장 명령어로 할당을 구현할 수 있도록 허용합니다.
C 할당은 동시 코드와 관련하여 비원자적일 수 있습니다. 대신 release는 C를 사용합니다.
원자 할당을 수행하는 라이브러리 함수 __sync_lock_release . 이 함수는 또한
결국 RISC-V amoswap 명령어로 귀결됩니다.

6.3 코드: 잠금 사용

Xv6는 경쟁을 피하기 위해 많은 곳에서 잠금을 사용합니다. 위에서 설명한 대로 kalloc (kernel/kalloc.c:69) 그리고 kfree (kernel/kalloc.c:47) 좋은 예를 형성합니다. 연습 문제 1과 2를 시도하여 해당 함수가 잠금을 생략하면 어떤 일이 일어나는지 확인합니다. 잘못된 동작을 트리거하기 어렵다는 것을 알게 될 가능성이 높으며, 이는 코드에 잠금 오류와 경쟁이 없는지 안정적으로 테스트하기 어렵다는 것을 암시합니다. Xv6에는 아직 발견되지 않은 경쟁이 있을 수 있습니다.

잠금을 사용하는 데 있어 어려운 부분은 사용할 잠금의 수와 각 잠금이 보호해야 할 데이터와 불변식을 결정하는 것입니다. 몇 가지 기본 원칙이 있습니다. 첫째, 한 CPU가 변수를 쓸 수 있고 다른 CPU가 변수를 읽거나 쓸 수 있는 경우, 잠금을 사용하여 두 작업이 겹치지 않도록 해야 합니다. 둘째, 잠금은 불변식을 보호한다는 것을 기억하세요. 불변식에 여러 메모리 위치가 포함되는 경우 일반적으로 모든 위치를 단일 잠금으로 보호하여 불변식을 유지해야 합니다.

위의 규칙은 잠금이 필요한 시점을 말하지만 잠금이 불필요한 시점에 대해서는 아무 말도 하지 않으며, 효율성을 위해 잠금을 너무 많이 하지 않는 것이 중요합니다. 잠금은 병렬성을 감소시키기 때문입니다. 병렬성이 중요하지 않다면 단일 스레드만 두고 잠금에 대해 걱정하지 않도록 할 수 있습니다. 간단한 커널은 커널에 들어갈 때 획득해야 하고 커널을 나갈 때 해제해야 하는 단일 잠금을 갖는 방식으로 멀티프로세서에서 이를 수행할 수 있습니다(파이프 읽기나 대기와 같은 차단 시스템 호출은 문제 가 될 수 있음). 많은 단일 프로세서 운영 체제가 이 접근 방식(때로는 "빅 커널 잠금"이라고 함)을 사용하여 멀티프로세서에서 실행되도록 변환되었지만, 이 접근 방식은 병렬성을 희생합니다. 한 번에 하나의 CPU만 커널에서 실행할 수 있습니다. 커널이 무거운 계산을 수행하는 경우 더 세분화된 잠금 세트를 더 많이 사용하여 커널이 여러 CPU에서 동시에 실행될 수 있도록 하는 것이 더 효율적입니다.

거친 그레인 잠금의 예로, xv6의 kalloc.c 할당자는 단일 잠금으로 보호되는 단일 자유 목록을 가지고 있습니다. 다른 CPU의 여러 프로세스가 동시에 페이지를 할당하려고 하면 각각은 획득을 위해 회전하여 차례를 기다려야 합니다. 회전은 유용한 작업이 아니기 때문에 CPU 시간을 낭비합니다. 잠금에 대한 경합이 CPU 시간의 상당 부분을 낭비했다면, 각각이 자체 잠금을 가진 여러 자유 목록을 갖도록 할당자 설계를 변경하여 진정한 병렬 할당을 허용함으로써 성능을 개선할 수 있습니다.

세분화된 잠금의 예로, xv6는 각 파일에 대해 별도의 잠금을 가지고 있으므로 다른 파일을 조작하는 프로세스가 종종 서로의 잠금을 기다리지 않고 진행할 수 있습니다. 프로세스가 동일한 파일의 다른 영역을 동시에 쓸 수 있도록 하려면 파일 잠금 체계를 훨씬 더 세분화할 수 있습니다. 궁극적으로 잠금 세분성 결정은 성능 측정과 복잡성 고려 사항에 따라 이루어져야 합니다.

이후 장에서는 xv6의 각 부분을 설명하면서 xv6의 사용 예를 언급할 것입니다.
동시성을 처리하기 위한 잠금. 미리보기로, 그림 6.3은 xv6의 모든 잠금을 나열합니다.

6.4 교착 상태 및 잠금 순서

커널을 통한 코드 경로가 동시에 여러 잠금을 보유해야 하는 경우 모든 코드 경로가 동일한 순서로 해당 잠금을 획득하는 것이 중요합니다. 그렇지 않으면 교착 상태의 위험이 있습니다. xv6의 두 코드 경로에 잠금 A와 B가 필요하지만 코드 경로 1이 잠금 A 순서로 잠금을 획득한다고 가정해 보겠습니다.

잠그다	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.
itable.lock	메모리 내 inode 항목 할당을 보호합니다.
vdisk_lock	디스크 하드웨어 및 DMA 기술자 대기열에 대한 액세스를 직렬화합니다.
kmem.lock	메모리 할당을 직렬화합니다
log.lock	트랜잭션 로그에 대한 작업을 직렬화합니다.
파이프의 pi->lock	은 각 파이프에 대한 작업을 직렬화합니다.
pid_lock	은 next_pid의 증가를 직렬화합니다.
proc의 p->lock	은 프로세스 상태의 변경 사항을 직렬화합니다.
wait_lock	은 웨이크업 손실을 방지하는 데 도움이 됩니다.
tickslock	tick 카운터에 대한 작업을 직렬화합니다.
inode의 ip->lock	은 각 inode와 해당 콘텐츠에 대한 작업을 직렬화합니다.
buf의 b->lock	각 블록 버퍼에 대한 작업을 직렬화합니다.

그림 6.3: xv6의 잠금 장치

B, 그리고 다른 경로는 B, 그 다음 A 순서로 이를 획득합니다. 스레드 T1이 코드 경로 1을 실행한다고 가정합니다.
그리고 잠금 A를 획득하고 스레드 T2는 코드 경로 2를 실행하고 잠금 B를 획득합니다. 그 다음 T1은 다음을 시도합니다.
잠금 B를 획득하고 T2는 잠금 A를 획득하려고 시도합니다. 두 획득 모두 무기한 차단됩니다.
두 경우 모두 다른 스레드가 필요한 잠금을 보유하고 획득이 반환될 때까지 잠금을 해제하지 않습니다.
이러한 교착 상태를 피하기 위해 모든 코드 경로는 동일한 순서로 잠금을 획득해야 합니다. 글로벌
잠금 획득 순서는 잠금이 효과적으로 각 기능 사양의 일부임을 의미합니다. 즉, 호출자입니다.
합의된 순서에 따라 잠금이 획득되도록 함수를 호출해야 합니다.

Xv6에는 프로세스별 잠금(각 프로세스의 잠금)을 포함하는 길이가 2인 많은 잠금 순서 체인이 있습니다.
struct proc) sleep이 작동하는 방식 때문에 (7장 참조). 예를 들어, consoleintr
(커널/콘솔.c:136) 입력된 문자를 처리하는 인터럽트 루틴입니다. 줄바꿈이 도착하면 콘솔 입력을 기다리는 모든 프로세스가 깨어나야 합니다. 이를 위해 consoleintr
wakeup을 호출하는 동안 cons.lock을 유지 하여 대기 중인 프로세스의 잠금을 획득합니다.
깨워라. 결과적으로 글로벌 교착 상태 회피 잠금 순서에는 cons.lock 규칙이 포함됩니다.
프로세스 잠금 전에 획득해야 합니다. 파일 시스템 코드에는 xv6의 가장 긴 잠금 체인이 포함되어 있습니다.
예를 들어, 파일을 생성하려면 디렉토리에 대한 잠금과
새 파일의 inode, 디스크 블록 버퍼의 잠금, 디스크 드라이버의 vdisk_lock, 호출 프로세스의 p->lock. 교착 상태를 피하기 위해 파일 시스템 코드는 항상 언급된 순서대로 잠금을 획득합니다.
이전 문장에서.

글로벌 교착 상태 회피 명령을 존중하는 것은 놀랍게도 어려울 수 있습니다. 때때로 잠금
순서는 논리적 프로그램 구조와 충돌합니다. 예를 들어 코드 모듈 M1이 모듈 M2를 호출하지만
잠금 순서는 M1의 잠금보다 M2의 잠금이 먼저 획득되어야 함을 요구합니다. 때때로 ID
잠금 장치의 수는 미리 알 수 없습니다. 아마도 잠금 장치를 하나 열어야 잠금 장치를 알아낼 수 있기 때문일 것입니다.
다음에 획득할 잠금의 ID입니다. 이런 종류의 상황은 파일 시스템에서 조회할 때 발생합니다.
경로 이름에 연속적인 구성 요소와 테이블을 검색할 때 대기하고 종료하는 코드

자식 프로세스를 찾는 프로세스의. 마지막으로, 교착 상태의 위험은 종종 잠금 체계를 얼마나 세밀하게 만들 수 있는지에 대한 제약이 되는데, 잠금이 많을수록 교착 상태의 기회가 더 많아지기 때문입니다. 교착 상태를 피해야 할 필요성은 종종 커널 구현에서 중요한 요소입니다.

6.5 재진입 잠금

재진입 잠금(re-entrant locks)을 사용하면 일부 교착 상태와 잠금 순서 지정 문제를 피할 수 있는 것처럼 보일 수 있습니다. 재귀 잠금이라고도 합니다. 아이디어는 잠금이 프로세스에 의해 유지되고 해당 프로세스가 잠금을 다시 획득하려고 시도하는 경우 커널은 xv6 커널이 하는 것처럼 panic을 호출하는 대신(프로세스가 이미 잠금을 가지고 있기 때문에) 이를 허용할 수 있다는 것입니다.

그러나 재진입 잠금은 동시성에 대한 추론을 더 어렵게 만듭니다. 재진입 잠금은 잠금이 임계 섹션을 다른 임계 섹션과 관련하여 원자적으로 만든다는 직관을 깨뜨립니다. 다음 두 함수 f 와 g를 고려하세요.

```
struct spinlock lock; int data = 0; // 잠금
으로 보호됨

f() { 획득
    (&lock); if(데이터 == 0)
    { call_once(); h(); 데이터 =
        1;




    } 잠금 해제(&lock);
}

g() { 획득
    (&lock); if(데이터 == 0)
    { call_once(); 데이터 = 1;




    } 잠금 해제(&lock);
}
```

이 코드 조각을 살펴보면 call_once가 한 번만 호출된다는 것을 알 수 있습니다.

f 또는 g에 의해서만 가능 하지만, 둘 다에 의해서는 불가능합니다.

하지만 재진입 잠금이 허용되고 h가 g를 호출하면 call_once가 두 번 호출됩니다.

재진입 잠금이 허용되지 않으면 h가 g를 호출 하면 교착 상태가 발생하는데, 이것도 좋지 않습니다. 하지만 call_once를 호출하는 것이 심각한 오류라고 가정하면 교착 상태가 더 바람직합니다. 커널 개발자는 교착 상태(커널 패닉)를 관찰하고 코드를 수정하여 피할 수 있지만 call_once를 두 번 호출하면 추적하기 어려운 오류가 자동으로 발생할 수 있습니다.

이러한 이유로 xv6는 이해하기 쉬운 비재진입 잠금을 사용합니다. 그러나 프로그래머가 잠금 규칙을 염두에 두는 한 두 가지 접근 방식을 모두 작동시킬 수 있습니다. xv6가

재진입 잠금을 사용하려면 호출 스레드가 현재 잠금을 보유하고 있음을 알리기 위해 acquire를 수정해야 합니다. 또한 push_off와 비슷한 스타일로 struct spinlock에 중첩된 acquire의 수를 추가해야 합니다. 다음에 설명합니다.

6.6 잠금 및 인터럽트 핸들러

일부 xv6 스핀록은 스레드와 인터럽트 핸들러에서 모두 사용하는 데이터를 보호합니다. 예를 들어, clockintr 타이머 인터럽트 핸들러는 틱을 증가시킬 수 있습니다 (kernel/trap.c:164) 커널 스레드가 sys_sleep (kernel/sysproc.c:59) 에서 tick을 읽는 것과 거의 같은 시간입니다 . 잠금 tickslock은 두 가지 접근을 직렬화합니다.

스핀락과 인터럽트의 상호작용은 잠재적 위험을 야기합니다. sys_sleep 이 tickslock을 보유하고 있고 CPU가 타이머 인터럽트로 인터럽트되었다고 가정해 보겠습니다. clockintr은 tickslock을 획득하려고 시도하고 , 그것이 보유되었음을 보고, 그것이 해제될 때까지 기다립니다. 이 상황에서 tickslock은 결코 해제되지 않습니다. 오직 sys_sleep 만이 그것을 해제할 수 있지만, sys_sleep은 clockintr 이 반환될 때까지 계속 실행되지 않습니다 . 그래서 CPU는 교착 상태가 되고, 두 잠금이 필요한 모든 코드도 동결됩니다.

이런 상황을 피하기 위해 인터럽트 핸들러에서 스핀락을 사용하는 경우 CPU는 인터럽트가 활성화된 상태에서는 절대로 해당 잠금을 유지해서는 안 됩니다. Xv6는 더 보수적입니다. CPU가 잠금을 획득하면 xv6는 항상 해당 CPU에서 인터럽트를 비활성화합니다. 다른 CPU에서도 인터럽트가 발생할 수 있으므로 인터럽트의 획득은 스레드가 스핀락을 해제할 때까지 기다릴 수 있습니다. 다만 동일한 CPU에서는 기다릴 수 없습니다.

CPU가 스핀록을 보유하지 않을 때 Xv6는 인터럽트를 다시 활성화합니다. 중첩된 중요 섹션을 처리하기 위해 약간의 회계 작업을 수행해야 합니다. acquire는 push_off (kernel/spinlock.c:89) 를 호출합니다. 그리고 릴리스는 pop_off (kernel/spinlock.c:100) 를 호출합니다 . 현재 CPU에서 잠금의 중첩 수준을 추적합니다. 해당 카운트가 0에 도달하면 pop_off는 가장 바깥쪽 중요 섹션의 시작 부분에 존재했던 인터럽트 활성화 상태를 복원합니다. intr_off 및 intr_on 함수는 RISC-V 명령어를 실행하여 각각 인터럽트를 비활성화하고 활성화합니다.

lk->locked(kernel/spin- lock.c:28)를 설정하기 전에 호출 push_off를 엄격히 호출하는 것이 중요합니다 . 두 가지가 역전되면 인터럽트가 활성화된 상태에서 잠금이 유지될 때 짧은 창이 생기고, 불행히도 타이밍이 정해진 인터럽트가 시스템을 교착 상태로 만들 것입니다. 마찬가지로 잠금을 해제한 후에만 release 가 pop_off를 호출하는 것이 중요합니다 (kernel/spinlock.c:66).

6.7 명령어 및 메모리 순서

소스 코드 문장이 나타나는 순서대로 프로그램이 실행된다고 생각하는 것은 자연스러운 일입니다.
그것은 단일 스레드 코드에 대한 합리적인 정신 모델이지만, 여러 스레드가 공유 메모리를 통해 상호 작용하는 경우에는 잘못된 것입니다[2, 4]. 한 가지 이유는 컴파일러가 소스 코드에서 암시하는 것과 다른 순서로 로드 및 저장 명령을 내보내고, 이를 완전히 생략할 수 있기 때문입니다(예: 레지스터에 데이터를 캐싱하여). 또 다른 이유는 CPU가 성능을 높이기 위해 명령을 순서 없이 실행할 수 있기 때문입니다. 예를 들어, CPU는 일련의 명령 A와 B가 서로 종속되지 않는다는 것을 알아차릴 수 있습니다. CPU는 명령 B를 먼저 시작할 수 있는데, 이는 명령 B의 입력이 A의 입력보다 먼저 준비되었거나 A와 B의 실행을 겹치기 위해서입니다.

이 푸시 코드에서 잘못될 수 있는 일의 예로 , 다음과 같은 경우 재앙이 발생합니다.
컴파일러나 CPU는 4번째 줄에 해당하는 저장소를 6번째 줄의 릴리스 이후 지점으로 옮겼습니다 .

```
1          l = malloc(sizeof *l); l->데이터 = 데이터; 획
2          득(&listlock); l->다음 = 목록;
3          목록 = l; 해제(&listlock);
4
5
6
```

이러한 재정렬이 발생하면 다른 CPU가 잠금을 획득하고 업데이트된 목록을 관찰할 수 있는 창이 생기지 만 초기화되지 않은 목록이 다음으로 표시됩니다.

좋은 소식은 컴파일러와 CPU가 메모리 모델이라는 일련의 규칙을 따르고 프로그래머가 재정렬을 제어하는 데 도움이 되는 몇 가지 기본 요소를 제공함으로써 동시성 프로그래머를 돕는다는 것입니다.

하드웨어와 컴파일러에 재정렬하지 말라고 알리기 위해 xv6은 획득 (kernel/spinlock.c:22) 모두에서 __sync_synchronize()를 사용합니다. 그리고 릴리스 (kernel/spinlock.c:47). __sync_synchronize() 는 메모리 장벽입니다. 이 장벽은 컴파일러와 CPU에 장벽을 가로질러 로드나 스토어를 재정렬하지 말라고 지시합니다. xv6의 획득 및 해제 장벽은 거의 모든 경우에 강제로 순서를 정합니다. xv6는 공유 데이터에 대한 액세스에 잠금을 사용하기 때문입니다. 9장에서는 몇 가지 예외에 대해 설명합니다.

6.8 Sleep 잠금

때때로 xv6는 장시간 잠금을 유지해야 합니다. 예를 들어, 파일 시스템(8장)은 디스크에서 내용을 읽고 쓰는 동안 파일을 잠근 상태로 유지하며, 이러한 디스크 작업은 수십 밀리초가 걸릴 수 있습니다. 그렇게 오랫동안 스핀락을 유지하면 다른 프로세스가 스핀락을 획득하려고 하면 낭비로 이어질 수 있습니다. 획득하는 프로세스가 회전하는 동안 CPU를 오랫동안 낭비하게 되기 때문입니다. 스핀락의 또 다른 단점은 프로세스가 스핀락을 유지하는 동안 CPU를 양보할 수 없다는 것입니다. 잠금이 있는 프로세스가 디스크를 기다리는 동안 다른 프로세스가 CPU를 사용할 수 있도록 이렇게 하려고 합니다.

스핀락을 잡고 있는 동안 양보하는 것은 불법입니다. 두 번째 스레드가 스핀락을 획득하려고 시도하면 교착 상태로 이어질 수 있기 때문입니다. 획득은 CPU를 양보하지 않으므로 두 번째 스레드의 회전으로 인해 첫 번째 스레드가 실행되고 잠금을 해제하지 못할 수 있습니다. 잠금을 잡고 있는 동안 양보하는 것은 스핀락이 유지되는 동안 인터럽트가 꺼져 있어야 한다는 요구 사항을 위반합니다. 따라서 획득을 기다리는 동안 CPU를 양보하고 잠금이 유지되는 동안 양보(및 인터럽트)를 허용하는 유형의 잠금이 필요합니다.

Xv6는 sleep-locks 형태로 이러한 잠금을 제공합니다. acquiresleep (kernel/sleeplock.c:22) 7장에서 설명할 기술을 사용하여 대기하는 동안 CPU를 제공합니다. 높은 수준에서 sleep-lock은 spinlock으로 보호되는 잠긴 필드를 가지고 있으며 acquiresleep 의 sleep 호출은 원자적으로 CPU를 제공하고 spinlock을 해제합니다. 결과적으로 acquiresleep이 대기하는 동안 다른 스레드가 실행될 수 있습니다.

sleep-locks는 인터럽트를 활성화 상태로 두기 때문에 인터럽트 핸들러에서 사용할 수 없습니다. acquiresleep이 CPU를 양보할 수 있기 때문에 sleep-locks는 spinlock 중요 섹션 내부에서 사용할 수 없습니다(spinlocks는 sleep-lock 중요 섹션 내부에서 사용할 수 있음).

스핀 잠금은 짧은 중요 섹션에 가장 적합합니다. 이를 기다리면 CPU 시간이 낭비되기 때문입니다.
sleep-locks는 장기 작업에 적합합니다.

6.9 현실 세계

동시성 기본 요소와 병렬성에 대한 수년간의 연구에도 불구하고 잠금을 사용한 프로그래밍은 여전히 어렵습니다. 동기화된 큐와 같은 상위 수준 구조 내에 잠금을 숨기는 것이 가장 좋지만 xv6에서는 그렇지 않습니다. 잠금을 사용하여 프로그래밍하는 경우 경쟁을 식별하려는 도구를 사용하는 것이 좋습니다. 잠금이 필요한 불변성을 놓치기 쉽기 때문입니다.

대부분의 운영 체제는 POSIX 스레드(Pthread)를 지원하는데, 이를 통해 사용자 프로세스는 여러 스레드를 다른 CPU에서 동시에 실행할 수 있습니다. Pthread는 사용자 수준 잠금, 배리어 등을 지원합니다. Pthread는 또한 프로그래머가 잠금을 다시 지정하도록 허용합니다.
참가자.

사용자 수준에서 Pthread를 지원하려면 운영 체제의 지원이 필요합니다. 예를 들어, 한 pthread가 시스템 호출에서 차단되면 같은 프로세스의 다른 pthread가 해당 CPU에서 실행될 수 있어야 합니다. 또 다른 예로, pthread가 프로세스의 주소 공간을 변경하면(예: 메모리 매핑 또는 언매핑) 커널은 같은 프로세스의 스레드를 실행하는 다른 CPU가 하드웨어 페이지 테이블을 업데이트하여 주소 공간의 변경 사항을 반영하도록 해야 합니다.

원자 명령어 없이 잠금을 구현하는 것은 가능하지만 비용이 많이 들고 대부분
운영체제는 원자적 명령어를 사용합니다.

많은 CPU가 동시에 동일한 잠금을 획득하려고 하면 잠금이 비쌀 수 있습니다. 한 CPU가 로컬 캐시에 잠금을 캐시하고 다른 CPU가 잠금을 획득해야 하는 경우 잠금을 보유한 캐시 라인을 업데이트하는 원자적 명령어는 라인을 한 CPU의 캐시에서 다른 CPU의 캐시로 이동해야 하며, 아마도 캐시 라인의 다른 사본을 무효화해야 합니다. 다른 CPU의 캐시에서 캐시 라인을 가져오는 것은 로컬 캐시에서 라인을 가져오는 것보다 훨씬 더 비쌀 수 있습니다.

잠금과 관련된 비용을 피하기 위해 많은 운영 체제는 잠금 없는 데이터 구조와 알고리즘을 사용합니다[6, 12]. 예를 들어, 목록 검색 중에 잠금이 필요 없고 목록에 항목을 삽입하는 원자적 명령어가 하나인 이 장의 시작 부분에 있는 것과 같은 연결 목록을 구현할 수 있습니다. 그러나 잠금 없는 프로그래밍은 잠금을 프로그래밍하는 것보다 더 복잡합니다. 예를 들어 명령어와 메모리 재정렬에 대해 걱정해야 합니다. 잠금을 사용한 프로그래밍은 이미 어렵기 때문에 xv6은 잠금 없는 프로그래밍의 추가적인 복잡성을 피합니다.

6.10 연습문제

1. kalloc 에서 획득 및 해제 호출을 주석으로 처리합니다 (kernel/kalloc.c:69). 이것은 kalloc을 호출하는 커널 코드에 문제를 일으킬 것 같습니다 . 어떤 증상이 나타날 것으로 예상하십니까? xv6을 실행할 때 이러한 증상이 나타나십니까? usertest를 실행할 때는 어떻습니까? 문제가 보이지 않는다면, 왜 그럴까요? kalloc의 중요 섹션에 더미 루프를 삽입하여 문제를 일으킬 수 있는지 살펴보십시오 .

2. 대신 kfree 에서 잠금을 주석 처리했다고 가정해 보겠습니다 (kalloc에서 잠금을 복원한 후). 이제 무엇이 잘못될 수 있을까요? kfree 에서 잠금이 없는 것이 kalloc 에서보다 덜 해롭 습니까 ?

3. 두 CPU가 동시에 kalloc을 호출하면 하나는 다른 하나를 기다려야 하므로 성능에 좋지 않습니다. kalloc.c를 수정하여 병렬성을 높여 다른 CPU에서 kalloc을 동시에 호출할 때 서로를 기다리지 않고 진행할 수 있도록 합니다.

4. 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.

5. xv6에서 Pthread의 하위 집합을 구현합니다. 즉, 사용자 프로세스가 2개 이상의 스레드를 가질 수 있도록 사용자 수준 스레드 라이브러리를 구현하고 이러한 스레드가 다른 CPU에서 병렬로 실행될 수 있도록 합니다. 블로킹 시스템 호출을 하고 공유 주소 공간을 변경하는 스레드를 올바르게 처리하는 디자인을 생각해 보세요.

7장

스케줄링

모든 운영 체제는 컴퓨터의 CPU보다 많은 프로세스로 실행될 가능성이 높으므로 프로세스 간에 CPU를 시간 공유할 계획이 필요합니다. 이상적으로는 공유가 사용자 프로세스에 투명해야 합니다. 일반적인 접근 방식은 각 프로세스에 하드웨어 CPU에 프로세스를 멀티플렉싱하여 자체 가상 CPU가 있다는 환상을 제공하는 것입니다. 이 장에서는 xv6가 이러한 멀티플렉싱을 어떻게 달성하는지 설명합니다.

7.1 멀티플렉싱

Xv6는 두 가지 상황에서 각 CPU를 한 프로세스에서 다른 프로세스로 전환하여 멀티플렉싱합니다. 첫째, xv6의 sleep 및 wakeup 메커니즘은 프로세스가 장치 또는 파이프 I/O가 완료되기를 기다리거나 자식이 종료되기를 기다리거나 sleep 시스템 호출에서 기다릴 때 전환합니다. 둘째, xv6는 주기적으로 스위치가 sleep 없이 오랜 시간 동안 계산하는 프로세스를 처리하도록 강제합니다. 이 멀티플렉싱은 각 프로세스가 자체 CPU를 가지고 있다는 환상을 만드는데, xv6가 메모리 할당자와 하드웨어 페이지 테이블을 사용하여 각 프로세스가 자체 메모리를 가지고 있다는 환상을 만드는 것과 비슷합니다.

멀티플렉싱을 구현하는 데는 몇 가지 과제가 있습니다. 첫째, 한 프로세스에서 다른 프로세스로 전환하는 방법은 무엇일까요? 컨텍스트 전환이라는 개념은 간단하지만 구현은 xv6에서 가장 불투명한 코드 중 하나입니다. 둘째, 사용자 프로세스에 투명한 방식으로 전환을 강제하는 방법은 무엇일까요? Xv6은 하드웨어 타이머의 인터럽트가 컨텍스트 전환을 구동하는 표준 기술을 사용합니다. 셋째, 모든 CPU가 동일한 공유 프로세스 집합 간에 전환하고, 경쟁을 피하기 위해 잠금 계획이 필요합니다. 넷째, 프로세스가 종료되면 프로세스의 메모리와 기타 리소스를 해제해야 하지만, (예를 들어) 여전히 사용 중인 자체 커널 스택을 해제할 수 없기 때문에 이 모든 것을 스스로 할 수는 없습니다. 다섯째, 멀티코어 머신의 각 코어는 시스템 호출이 올바른 프로세스의 커널 상태에 영향을 미치도록 실행 중인 프로세스를 기억해야 합니다. 마지막으로, 슬립 과 웨이크업을 통해 프로세스는 CPU를 포기하고 다른 프로세스나 인터럽트가 깨울 때까지 기다릴 수 있습니다. 웨이크업 알림이 손실되는 경쟁을 피하기 위해 주의해야 합니다. Xv6는 이런 문제를 가능한 한 간단하게 해결하려고 노력했지만, 그럼에도 불구하고 결과적으로 나오는 코드는 까다롭습니다.

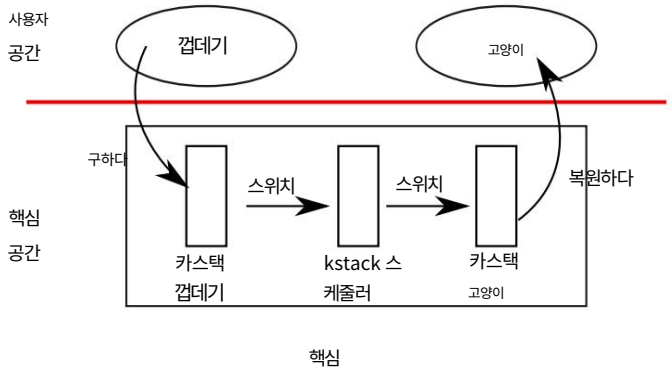

그림 7.1: 한 사용자 프로세스에서 다른 사용자 프로세스로 전환. 이 예에서 xv6는 하나의 CPU(따라서 하나의 스케줄러 스레드)로 실행됩니다.

7.2 코드: 컨텍스트 전환

그림 7.1은 한 사용자 프로세스에서 다른 사용자 프로세스로 전환하는 데 필요한 단계를 간략하게 설명합니다. 이전 프로세스의 커널 스레드로의 사용자-커널 전환(시스템 호출 또는 인터럽트), 현재 CPU의 스케줄러 스레드로의 컨텍스트 전환, 새 프로세스의 커널 스레드로의 컨텍스트 전환, 사용자 수준 프로세스로의 트랩 리턴입니다. xv6 스케줄러는 CPU당 전용 스레드(저장된 레지스터 및 스택)를 갖는데, 스케줄러가 이전 프로세스의 커널 스택에서 실행하는 것이 안전하지 않기 때문입니다. 다른 코어가 프로세스를 깨워서 실행할 수 있으며, 두 개의 다른 코어에서 동일한 스택을 사용하는 것은 재앙이 될 것입니다. 이 섹션에서는 커널 스레드와 스케줄러 스레드 간 전환의 메커니즘을 살펴보겠습니다.

한 스레드에서 다른 스레드로 전환하려면 이전 스레드의 CPU 레지스터를 저장하고 새 스레드의 이전에 저장된 레지스터를 복원해야 합니다. 스택 포인터와 프로그램 카운터가 저장되고 복원된다는 것은 CPU가 스택을 전환하고 실행하는 코드를 전환한다는 것을 의미합니다.

swtch 함수는 커널 스레드 전환에 대한 저장 및 복원을 수행합니다. swtch는 스레드에 대해 직접적으로 알지 못하고, 컨텍스트라고 하는 32개의 RISC-V 레지스터 집합만 저장하고 복원합니다.
프로세스가 CPU를 포기할 때가 되면 프로세스의 커널 스레드는 swtch를 호출하여 자체 컨텍스트를 저장하고 스케줄러 컨텍스트로 돌아갑니다. 각 컨텍스트는 구조체 컨텍스트 (kernel/proc.h:2)에 포함되어 있습니다. 프로세스의 struct proc 또는 CPU의 struct cpu에 포함된 자체. swtch는 struct context *old 및 struct context *new라는 두 개의 인수를 취합니다 . old 에 현재 레지스터를 저장하고 new 에서 레지스터를 로드한 다음 반환합니다.

swtch를 통해 스케줄러로 가는 프로세스를 따라가 봅시다 . 4장에서 인터럽트가 끝날 때 한 가지 가능성이 있다는 것을 보았습니다 . usertrap이 yield를 호출합니다 . yield는 차례로 sched를 호출하고, sched는 swtch를 호출하여 현재 컨텍스트를 p->context 에 저장 하고 이전에 cpu->context에 저장된 스케줄러 컨텍스트로 전환합니다 (kernel/proc.c:497).

스위치 (커널/swtch.S:3) 호출자가 저장한 레지스터만 저장합니다. C 컴파일러는 호출자가 저장한 레지스터를 스택에 저장하기 위해 호출자에서 코드를 생성합니다. swtch는 구조체 컨텍스트 에서 각 레지스터 필드의 오프셋을 알고 있습니다 . 프로그램 카운터는 저장하지 않습니다. 대신 swtch는 swtch가 호출된 반환 주소를 보관하는 ra 레지스터를 저장합니다 . 이제 swtch는 레지스터를 복원합니다 .

이전 swtch에서 저장된 레지스터 값을 보유하는 새 컨텍스트 . swtch가 반환 되면 복원된 ra 레지스터 가 가리키는 명령어 , 즉 새 스레드가 이전에 swtch를 호출한 명령어로 돌아갑니다. 또한 복원된 sp가 가리키는 곳이기 때문에 새 스레드의 스택에서 반환합니다 .

우리의 예에서 sched는 swtch를 호출하여 CPU->context, 즉 CPU당 스케줄러 컨텍스트 로 전환했습니다 . 해당 컨텍스트는 스케줄러가 swtch를 호출 한 과거 시점에 저장되었습니다 (kernel /proc.c:463) 지금 CPU를 포기하고 있는 프로세스로 전환합니다. 추적하고 있던 스위치가 돌아오면 sched가 아니라 scheduler로 돌아가고, 스택 포인터는 현재 CPU의 scheduler 스택에 있습니다.

7.3 코드: 스케줄링

마지막 섹션에서는 swtch 의 저수준 세부 사항을 살펴보았습니다 . 이제 swtch를 주어진 것으로 간주하고 스케줄러를 통해 한 프로세스의 커널 스레드에서 다른 프로세스로 전환하는 것을 살펴보겠습니다. 스케줄러는 CPU당 특수 스레드 형태로 존재하며, 각각 스케줄러 함수를 실행합니다. 이 함수는 다음에 실행할 프로세스를 선택하는 역할을 합니다. CPU를 포기하려는 프로세스는 자체 프로세스 잠금 p->lock을 획득하고, 보유하고 있는 다른 잠금을 해제하고, 자체 상태 (p->state) 를 업데이트한 다음 sched를 호출해야 합니다. 이 시퀀스는 yield(kernel/proc.c:503)에서 확인할 수 있습니다 . sleep하고 종료합니다. sched는 이러한 요구 사항 중 일부를 다시 확인합니다 (kernel/proc.c:487-492) 그런 다음 암시를 확인합니다. 잠금이 유지되었으므로 인터럽트를 비활성화해야 합니다. 마지막으로 sched는 swtch를 호출하여 현재 컨텍스트를 p->context 에 저장하고 cpu->context 에서 스케줄러 컨텍스트로 전환합니다 . swtch는 스케줄러의 swtch가 반환된 것처럼 스케줄러의 스택에서 반환합니다 (kernel/proc.c:463). 스케줄러는 for 루프를 계속 실행하고, 실행할 프로세스를 찾아서 해당 프로세스로 전환하고, 이 순환이 반복됩니다.

방금 xv6가 swtch 에 대한 호출에서 p->lock을 유지하는 것을 보았습니다 .swtch 의 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 p->lock이 swtch 에서 실행하는 동안 참이 아닌 프로세스의 상태 및 컨텍스트 필드에 대한 불변성을 보호하기 때문에 이 규칙을 어길 필요가 있습니다 .swtch 중에 p->lock을 유지하지 않으면 발생할 수 있는 문제의 한 예는 다른 CPU가 yield가 상태를 RUNNABLE 로 설정한 후 swtch가 자체 커널 스택 사용을 중지시키기 전에 프로세스를 실행하기로 결정할 수 있다는 것 입니다.결과적으로 두 개의 CPU가 동일한 스택에서 실행되어 혼란이 발생합니다.

커널 스레드가 CPU를 포기하는 유일한 장소는 sched이며, 항상 스케줄러 에서 같은 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. (kernel/proc.c:463), (커널/proc.c:497), (커널/proc.c:463), (커널/proc.c:497), 등등. 스레드 스위치를 통해 의도적으로 서로에게 제어권을 넘기는 프로시저는 때때로 코루틴이라고도 합니다. 이 예에서 sched 와 scheduler는 서로의 코루틴입니다.

스케줄러의 swtch 호출이 sched 로 끝나지 않는 경우가 하나 있습니다 . allocproc는 새 프로세스의 컨텍스트 ra 레지스터를 forkret으로 설정합니다 (kernel/proc.c:515). 그래서 첫 번째 스위치가

해당 함수의 시작으로 "돌아갑니다". forkret은 p->lock을 해제하기 위해 존재합니다 . 그렇지 않으면 새 프로세스가 fork 에서 돌아오는 것처럼 사용자 공간으로 돌아가야 하므로 대신 usertrapret에서 시작할 수 있습니다. scheduler (kernel/proc.c:445) 루프를 실행합니다. 실행할 프로세스를 찾고, 결과가 나올 때까지 실행하고, 반복합니

다.

스케줄러는 프로세스 테이블을 순환하여 실행 가능한 프로세스, 즉 p->state == RUNNABLE 인 프로세스를 찾습니다 . 프로세스를 찾으면 CPU당 현재 프로세스 변수 c->proc를 설정하고 프로세스를 RUNNING 으로 표시한 다음 swtch를 호출하여 실행을 시작합니다 (kernel/proc.c:458-463).

스케줄링 코드의 구조에 대해 생각하는 한 가지 방법은 각 프로세스에 대한 일련의 불변식을 적용하고 이러한 불변식이 참이 아닐 때마다 p->lock을 유지한다는 것입니다. 한 가지 불변식은 프로세스가 실행 중이 면 타이머 인터럽트의 yield가 프로세스에서 안전하게 전환될 수 있어야 한다는 것입니다. 즉, CPU 레지스터는 프로세스의 레지스터 값을 유지해야 하고(즉, swtch가 컨텍스트 로 이동하지 않음) c- >proc는 프로세스를 참조해야 합니다. 또 다른 불변식은 프로세스가 실행 가능 이면 유휴 CPU의 스케줄러가 이를 실행하는 데 안전해야 한다는 것입니다 . 즉, p->context는 프로세스의 레지스터를 유지해야 하고(즉, 실제 레지스터에 실제로 없음), 프로세스의 커널 스택에서 실행 중인 CPU가 없고, CPU의 c->proc가 프로세스를 참조하지 않아야 합니다. 이러한 속성은 p->lock이 유지되는 동안 종종 참이 아님에 유의하십시오 .

위의 불변식을 유지하는 것이 xv6가 종종 한 스레드에서 p->lock을 획득하고 다른 스레드에서 해제하는 이유입니다 . 예를 들어, yield 에서 획득 하고 scheduler에서 해제합니다. yield가 실행 중인 프로세스의 상태를 수정하여 RUNNABLE 로 만들기 시작 하면 불변식이 복원될 때까지 잠금을 유지해야 합니다. 가장 빠른 올바른 해제 지점은 scheduler (자체 스택에서 실행)가 c->proc를 지운 후입니다. 마찬가지로 scheduler가 RUNNABLE 프로세스를 RUNNING 으로 변환하기 시작 하면 커널 스레드가 완전히 실행될 때까지(예를 들어 , yield에서 switch 후) 잠금을 해제할 수 없습니다.

7.4 코드: mycpu 및 myproc

Xv6는 종종 현재 프로세스의 proc 구조에 대한 포인터가 필요합니다. 단일 프로세서에서는 현재 proc를 가리키는 전역 변수가 있을 수 있습니다 . 이는 각 코어가 다른 프로세스를 실행하기 때문에 멀티 코어 머신에서는 작동하지 않습니다. 이 문제를 해결하는 방법은 각 코어가 고유한 레지스터 세트를 가지고 있다는 사실을 이용하는 것입니다. 이러한 레지스터 중 하나를 사용하여 코어별 정보를 찾을 수 있습니다.

Xv6는 각 CPU에 대한 struct cpu (kernel/proc.h:22)를 유지합니다. 현재 해당 CPU에서 실행 중인 프로세스(있는 경우)를 기록하고, CPU 스케줄러 스레드에 대한 저장된 레지스터와 인터럽트 비활성화를 관리하는 데 필요한 중첩 스핀록의 수를 기록합니다. 함수 mycpu (kernel/proc.c:74) 현재 CPU의 struct cpu에 대한 포인터를 반환합니다. RISC-V는 CPU에 번호를 매기고 각각에 hartid를 부여합니다. Xv6는 각 CPU의 hartid가 커널에 있는 동안 해당 CPU의 tp 레지스터 에 저장되도록 합니다 .

이를 통해 mycpu는 tp를 사용하여 CPU 구조 배열을 인덱싱하여 올바른 구조를 찾을 수 있습니다.

CPU의 tp가 항상 CPU의 hartid를 유지하도록 하는 것은 약간 복잡합니다. start는 CPU 부팅 시퀀스의 초반에 tp 레지스터를 설정하며 , 이는 여전히 머신 모드에 있는 동안입니다 (kernel/start.c:51). usertrapret은 사용자 프로세스가 tp를 수정할 수 있기 때문에 트램폴린 페이지에 tp를 저장합니다 . 마지막으로 uservec은 사용자 공간에서 커널에 들어갈 때 저장된 tp를 복원합니다 (kernel/trampoline.S:77).

컴파일러는 tp 레지스터를 절대 사용하지 않을 것을 보장합니다. xv6에서 할 수 있다면 더 편리할 것입니다.

필요할 때마다 RISC-V 하드웨어에 현재 하드웨어를 요청하지만 RISC-V에서는 다음 경우에만 허용합니다.
감독자 모드가 아닌 머신 모드입니다.

cpuid 및 mycpu 의 반환 값은 취약합니다. 타이머가 중단되어 다음과 같은 문제가 발생하는 경우
스레드가 생성되고 다른 CPU로 이동하면 이전에 반환된 값이 더 이상 반환되지 않습니다.
맞습니다. 이 문제를 피하기 위해 xv6에서는 호출자가 인터럽트를 비활성화하고 인터럽트만 활성화해야 합니다.
반환된 struct cpu를 사용한 후 .

myproc 함수 (kernel/proc.c:83) 프로세스에 대한 struct proc 포인터를 반환합니다 .
현재 CPU에서 실행 중입니다. myproc는 인터럽트를 비활성화하고 mycpu를 호출하고 현재를 가져옵니다.
struct cpu 에서 프로세스 포인터 (c->proc) 를 꺼낸 다음 인터럽트를 활성화합니다. 반환 값
myproc 는 인터럽트가 활성화되어 있어도 안전하게 사용할 수 있습니다. 타이머 인터럽트가 호출 프로세스를 이동하는 경우
다른 CPU에 대해서는 struct proc 포인터가 동일하게 유지됩니다.

7.5 수면과 각성

스케줄링과 잠금은 한 스레드의 동작을 다른 스레드에서 숨기는 데 도움이 되지만 스레드가 의도적으로 상호 작용하는 데
도움이 되는 추상화도 필요합니다. 예를 들어, xv6의 파이프 리더는
쓰기 프로세스가 데이터를 생성할 때까지 기다려야 합니다. 부모가 기다리 라고 요청하면 기다려야 할 수도 있습니다.
자식이 종료되고, 디스크를 읽는 프로세스는 디스크 하드웨어가 읽기를 마칠 때까지 기다려야 합니다.
xv6 커널은 이러한 상황(및 기타 여러 상황)에서 Sleep 및 Wakeup이라는 메커니즘을 사용합니다.
Sleep은 커널 스레드가 특정 이벤트를 기다릴 수 있도록 허용합니다. 다른 스레드는 wakeup을 호출하여 이벤트를 기다
리는 스레드가 재개되어야 함을 나타낼 수 있습니다. Sleep과 wakeup은 종종 시퀀스라고 합니다.
조정 또는 조건부 동기화 메커니즘.

수면과 깨어남은 비교적 낮은 수준의 동기화 인터페이스를 제공합니다. 동기를 부여하려면
xv6에서 작동하는 방식을 사용하여 이를 사용하여 더 높은 수준의 동기화 메커니즘을 구축합니다.
생산자와 소비자를 조정하는 세마포어[5](xv6에서는 세마포어를 사용하지 않음).
세마포어는 카운트를 유지하고 두 가지 연산을 제공합니다. "V" 연산(생산자용)
카운트를 증가시킵니다. "P" 작업(소비자용)은 카운트가 0이 아닐 때까지 기다립니다.
그리고 그것을 감소시키고 반환합니다. 생산자 스레드가 하나뿐이고 소비자 스레드가 하나뿐인 경우
스레드가 있고 서로 다른 CPU에서 실행되었으며 컴파일러가 너무 적극적으로 최적화하지 않았습니다.
이 구현은 정확할 것입니다:

```
100 구조체 세마포어 {
101 구조체 스핀락 잠금;
102         정수 카운트;
103     };
104
105 무효
106 V(구조 세마포어 *s)
107 {
108         획득(&s->잠금);
109         s->카운트 += 1;
110         잠금 해제(&s->lock);
111     }
```

```
112
113 무효
114 P(구조체 세마포어 *s)
115 {
116             while(s->count == 0)
117                 ;
118             획득(&s->잠금);
119             s->카운트 -= 1;
120             잠금 해제(&s->lock);
121         }
```

위의 구현은 비용이 많이 듭니다. 생산자가 거의 행동하지 않으면 소비자는 비용을 지출하게 됩니다.
대부분의 시간은 while 루프에서 0이 아닌 카운트를 바라며 회전합니다. 소비자의 CPU
아마도 s->count를 반복적으로 폴링하여 바쁜 대기보다 더 생산적인 작업을 찾을 수 있을 것입니다 .
바쁜 대기를 피하려면 소비자가 CPU를 양보하고 V 이후에만 재개할 수 있는 방법이 필요합니다.
카운트를 증가시킵니다.

여기 그 방향으로 한 걸음 나아가지만, 우리가 보게 되듯이 그것만으로는 충분하지 않습니다. 한 쌍을 상상해 봅시다.
호출, sleep 및 wakeup 은 다음과 같이 작동합니다. sleep(chan)은 임의의 값에서 sleep합니다.
chan은 대기 채널이라고 합니다. sleep은 호출 프로세스를 절전 모드로 전환하여 다른 작업을 위해 CPU를 해제합니다.
작업. wakeup(chan)은 chan 에서 잠자고 있는 모든 프로세스를 깨우고 (있는 경우) 해당 프로세스의 sleep 호출을 실행합니다.
반환. chan 에서 기다리는 프로세스가 없으면 wakeup은 아무것도 하지 않습니다. 세마포어를 변경할 수 있습니다.
sleep 과 wakeup을 사용하기 위한 구현 (노란색으로 강조된 변경 사항):

```
200 무효
201 V(구조 세마포어 *s)
202 {
203             획득(&s->잠금);
204             s->카운트 += 1;
205             깨어나다;
206             잠금 해제(&s->lock);
207         }
208
209 무효
210 P(구조체 세마포어 *s)
211 {
212             while(s->count == 0)
213                 수면;
214             획득(&s->잠금);
215             s->카운트 -= 1;
216             잠금 해제(&s->lock);
217         }
```

P는 이제 회전하는 대신 CPU를 포기하는데, 이는 좋습니다. 그러나 그것은 그렇지 않은 것으로 밝혀졌습니다.
이 인터페이스를 사용하면 수면 과 깨우기를 쉽게 설계할 수 있으며 어떤 문제가 발생하더라도 걱정할 필요가 없습니다.
잃어버린 웨이크업 문제로 알려져 있습니다. P가 212번째 줄에서 s->count == 0 을 찾는 다고 가정합니다.
P는 212번째 줄과 213번째 줄 사이에 있으며 V는 다른 CPU에서 실행됩니다. 즉 s->count를 0이 아닌 값으로 변경합니다 .

wakeup을 호출하는데 , 이는 잠자고 있는 프로세스가 없다는 것을 발견하고 아무 일도 하지 않습니다. 이제 P는 실행을 계속합니다.
213번째 줄에서: sleep을 호출 하고 sleep 상태로 전환됩니다. 이로 인해 문제가 발생합니다. P는 V 호출을 기다리며 sleep 상태입니다.
그것은 이미 일어났습니다. 우리가 운이 좋고 생산자가 V를 다시 호출하지 않는 한 소비자는
카운트가 0이 아니더라도 영원히 기다립니다.

이 문제의 근원은 s->count == 0 일 때만 P가 절전 모드로 전환된다는 불변성이 위반된다는 것입니다.
V가 잘못된 순간에 실행 됨으로써 . 불변성을 보호하는 잘못된 방법은 이동하는 것입니다.
P 에서 잠금 취득(아래 노란색으로 강조 표시)을 수행하여 카운트 확인 및 호출을 수행합니다.
sleep 은 원자적이다:

```
300 무효
301 V(구조 세마포어 *s)
302 {
303             획득(&s->잠금);
304             s->카운트 += 1;
305             깨어나다;
306             잠금 해제(&s->lock);
307         }
308
309 무효
310 P(구조체 세마포어 *s)
311 {
312             획득(&s->잠금);
313             while(s->count == 0)
314                 수면;
315             s->카운트 -= 1;
316             잠금 해제(&s->lock);
317         }
```

P 의 이 버전은 잠금이 V를 방지하기 때문에 손실된 웨이크업을 피할 수 있기 를 바랄 수 있습니다.
313라인과 314라인 사이에서 실행하지 마십시오. 그렇게 하지만 교착상태에 빠지기도 합니다. P가 잠금을 유지합니다 .
V는 잠자는 동안 잠금을 기다리며 영원히 차단됩니다.

우리는 sleep의 인터페이스를 변경하여 이전 계획을 수정할 것입니다 : 호출자는 호출 프로세스가 sleep으로 표시된 후 잠금을 해제할 수 있도록 조건 잠금을 sleep 에 전달해야 합니다.
수면 채널에서 대기 중입니다. 잠금은 동시 V가 P가 완료될 때까지 대기하도록 강제합니다.
자신을 잠자게 하여, 웨이크업이 잠자는 소비자를 찾아서 깨울 수 있도록 합니다 . 소비자가 다시 깨어나면, 슬립은 돌아오기 전에 잠금을 다시 획득합니다. 우리의 새로운 올바른 슬립/웨이크업
이 계획은 다음과 같이 사용 가능합니다(변경 사항은 노란색으로 강조 표시됨):

```
400 무효
401 V(구조 세마포어 *s)
402 {
403             획득(&s->잠금);
404             s->카운트 += 1;
405             깨어나다;
406             잠금 해제(&s->lock);
407         }
```

```
408
409 무효
410 P(구조체 세마포어 *s) 411 {

412             획득(&s->잠금); s->카운트 == 0인 동
413             안
414                 sleep(s, &s->lock); s->count -= 1;
415             release(&s->lock);
416
417         }
```

P 가 s->lock을 보유하고 있다는 사실은 V가 P의 s->count 확인 과 sleep 호출 사이에 P를 깨우려고 시도하는 것을 방지합니다 . 그러나 sleep이 필요하여 s->lock을 원자적으로 해제 하고 소비 프로세스를 sleep 상태로 만들어서 깨어남을 잃어버리는 것을 방지합니다.

7.6 코드: 수면 및 깨우기

Xv6의 sleep (커널/proc.c:536) 그리고 웨이크업 (kernel/proc.c:567) 위의 마지막 예에서 보여준 인터페이스를 제공하고, 이를 구현하면(사용 방법에 대한 규칙 포함) 웨이크업 손실이 발생하지 않습니다. 기본 아이디어는 sleep이 현재 프로세스를 SLEEPING 으로 표시한 다음 sched를 호출하여 CPU를 해제하는 것입니다. wakeup은 주어진 대기 채널에서 휴면 중인 프로세스를 찾아 RUNNABLE로 표시합니다. sleep 및 wakeup 의 호출자는 상호 편리한 번호를 채널로 사용할 수 있습니다. Xv6는 종종 대기에 관련된 커널 데이터 구조의 주소를 사용합니다.

sleep은 p->lock을 획득합니다 (kernel/proc.c:547). 이제 절전 모드로 전환되는 프로세스는 p->lock을 모두 유지합니다.
그리고 lk. 호출자(예시에서 P)에서 lk를 유지하는 것이 필요했습니다 . 다른 프로세스(예시에서 V를 실행하는 프로세스)가 wakeup(chan)에 대한 호출을 시작할 수 없도록 했습니다 . 이제 sleep 이 p->lock을 유지하므로 lk를 해제하는 것이 안전합니다 . 다른 프로세스가 wakeup(chan) 에 대한 호출을 시작할 수 있지만 wakeup은 p->lock을 획득하기 위해 기다릴 것이므로 sleep 이 프로세스를 sleep으로 전환하는 것을 마칠 때까지 기다릴 것이므로 wakeup 이 sleep 을 놓치지 않도록 합니다 .

이제 sleep은 p->lock 만 보유 하고 다른 것은 보유하지 않으므로 sleep 채널을 기록하고, 프로세스 상태를 SLEEPING으로 변경하고, sched를 호출하여 프로세스를 sleep 상태로 전환할 수 있습니다 (kernel/proc.c:551-554). 잠시 후에, p->lock이 프로세스가 SLEEPING으로 표시될 때까지 (스케줄러 에 의해) 해제되지 않는 것이 왜 중요한지 분명해질 것입니다.

어느 시점에서 프로세스는 조건 잠금을 획득하고, sleeper가 기다리는 조건을 설정하고, wakeup(chan)을 호출합니다. 조건 잠금1을 유지하는 동안 wakeup을 호출하는 것이 중요합니다 . wakeup은 프로세스 테이블(kernel/proc.c:567)을 반복합니다 . 그것은 검사하는 각 프로세스의 p->lock을 획득하는데, 그 이유는 프로세스의 상태를 조작할 수 있고 p->lock이 sleep 과 wakeup이 서로를 놓치지 않도록 보장하기 때문입니다. wakeup이 일치하는 chan을 가진 SLEEPING 상태의 프로세스를 찾으면 그 프로세스의 상태를 RUNNABLE로 변경합니다. 스케줄러가 다음에 실행될 때 프로세스가 실행될 준비가 되었음을 알게 될 것입니다.

1엄밀히 말하면 웨이크업이 획득 에 바로 이어지면 충분합니다. (즉, 해제 후에 웨이크업을 호출할 수도 있습니다 .)

sleep 과 wakeup 에 대한 잠금 규칙 이 sleep 중인 프로세스가 wakeup을 놓치지 않도록 보장하는 이유는 무엇일까요? sleep 중인 프로세스는 조건을 확인하기 전의 지점부터 SLEEPING으로 표시된 후의 지점까지 조건 잠금이나 자체 p->lock 또는 둘 다를 보유합니다. wakeup을 호출하는 프로세스는 wakeup의 루프에서 두 잠금을 모두 보유합니다. 따라서 waker는 소비 스레드가 조건을 확인하기 전에 조건을 참으로 만들거나 waker의 wakeup이 sleep 중인 스레드를 SLEEPING으로 표시된 직후에 엄격하게 검사합니다. 그런 다음 wakeup은 sleep 중인 프로세스를 보고 깨웁니다(다른 것이 먼저 깨우지 않는 한).

때로는 여러 프로세스가 같은 채널에서 sleep 상태에 있는 경우가 있습니다. 예를 들어, 파이프에서 읽는 프로세스가 두 개 이상인 경우입니다. wakeup 에 대한 단일 호출로 모든 프로세스가 깨어납니다. 그중 하나가 먼저 실행되어 sleep이 호출된 잠금을 획득 하고(파이프의 경우) 파이프에서 대기 중인 모든 데이터를 읽습니다. 다른 프로세스는 깨어났지만 읽을 데이터가 없다는 것을 알게 됩니다. 그들의 관점에서 보면 wakeup은 "가짜"였으며 다시 sleep 상태에 들어가야 합니다. 이러한 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).

7.7 코드: 파이프

생산자와 소비자를 동기화하기 위해 sleep 과 wakeup을 사용하는 더 복잡한 예는 xv6의 파이프 구현입니다. 우리는 1장에서 파이프의 인터페이스를 보았습니다. 파이프의 한쪽 끝에 쓰여진 바이트는 커널 내부 버퍼에 복사된 다음 파이프의 다른 쪽 끝에서 읽을 수 있습니다.
이후 장에서는 파이프를 둘러싼 파일 기술자 지원에 대해 살펴보겠지만, 지금은 pipewrite 와 piperead 의 구현에 대해 살펴보겠습니다 .

각 파이프는 잠금 과 데이터 버퍼를 포함하는 구조체 파이프 로 표현됩니다 .
nread 및 nwrite 필드는 버퍼에서 읽은 총 바이트 수와 버퍼에 쓴 총 바이트 수를 계산합니다. 버퍼는 래핑됩니다. buf[PIPESIZE-1] 다음에 쓰여진 다음 바이트는 buf[0] 입니다 . 카운트는 래핑되지 않습니다. 이 규칙을 사용하면 구현에서 전체 버퍼 (nwrite == nread+PIPESIZE) 와 빈 버퍼 (nwrite == nread)를 구별할 수 있지만, 버퍼에 대한 인덱싱은 buf[nread] 대신 buf [nread % PIPESIZE]를 사용해야 함을 의미합니다 (nwrite의 경우도 마찬가지임).

piperead 와 pipewrite 에 대한 호출이 두 개의 다른 CPU에서 동시에 발생한다고 가정해 보겠습니다 . pipewrite (kernel/pipe.c:77) 파이프의 잠금을 획득하여 시작합니다. 이 잠금은 카운트, 데이터 및 관련 불변식을 보호합니다. piperead (kernel/pipe.c:106) 그런 다음 잠금을 획득하려고 시도하지만 획득할 수 없습니다. 획득 (kernel/spinlock.c:22) 에서 회전합니다. 잠금을 기다리는 중입니다. piperead가 기다리는 동안 pipewrite는 쓰여지는 바이트 (addr[0..n-1])를 반복하면서 각각을 차례로 파이프에 추가합니다 (kernel/pipe.c:95). 이 루프 중에 버퍼가 채워질 수 있습니다 (kernel/pipe.c:88). 이 경우, pipewrite는 wakeup을 호출하여 대기 중인 리더에게 버퍼에 대기 중인 데이터가 있다는 사실을 알리고, &pi->nwrite 에서 대기하여 리더가 버퍼에서 바이트를 꺼낼 때까지 기다립니다. sleep은 pipewrite의 프로세스를 대기 상태로 전환하는 일부로 pi->lock을 해제합니다 .

이제 pi->lock을 사용할 수 있으므로 piperead는 이를 획득하고 중요 섹션으로 들어갑니다. 여기서 pi->nread != pi->nwrite (kernel/pipe.c:113) 임을 확인합니다. (pi->nwrite == pi->nread+PIPESIZE (kernel/pipe.c:88) 때문에 pipewrite가 절전 모드로 전환됨) 그래서 for 루프 로 넘어가서 파이프 (kernel/pipe.c:120) 에서 데이터를 복사합니다 . 그리고 복사된 바이트 수만큼 nread를 증가시킵니다. 그 많은 바이트가 이제 쓰기에 사용 가능하므로, piperead는 wakeup(kernel /pipe.c:127) 을 호출합니다. 반환하기 전에 모든 잠자는 작성자를 깨웁니다. wakeup은 &pi->nwrite 에서 잠자는 프로세스를 찾습니다 . 이 프로세스는 pipewrite를 실행했지만 버퍼가 채워졌을 때 멈췄습니다. 해당 프로세스를 RUNNABLE로 표시합니다.

파이프 코드는 리더와 라이터에 대해 별도의 슬립 채널 (pi->nread 및 pi->nwrite)을 사용합니다.
이렇게 하면 많은 독자와 작성자가 같은 파이프를 기다리는 드문 경우에 시스템을 더 효율적으로 만들 수 있습니다. 파이프 코드는 슬립 조건을 확인하는 루프 내부에서 슬립합니다. 여러 독자나 작성자가 있는 경우 깨어나는 첫 번째 프로세스만 제외하고 모든 프로세스는 조건이 여전히 거짓임을 보고 다시 슬립합니다.

7.8 코드: 대기, 종료 및 종료

sleep 과 wakeup은 여러 종류의 대기에 사용될 수 있습니다. 1장에서 소개한 흥미로운 예는 자식의 exit 와 부모의 wait 간의 상호 작용입니다. 자식이 죽을 때 부모는 이미 wait에서 잠을 자고 있거나 다른 일을 하고 있을 수 있습니다. 후자의 경우 wait에 대한 후속 호출은 아마도 exit를 호출한 후 오랜 시간이 지난 후에 자식의 죽음을 관찰해야 합니다. wait이 관찰할 때까지 xv6가 자식의 죽음을 기록하는 방식은 exit가 호출자를 좀비 상태로 만드는 것입니다. 이 상태에서 부모의 wait이 이를 알아차리고 자식의 상태를 UNUSED로 변경하고 자식의 종료 상태를 복사하고 자식의 프로세스 ID를 부모에게 반환할 때까지 머무릅니다. 부모가 자식보다 먼저 종료되면 부모는 자식을 init 프로세스에 제공하고, init 프로세스는 끊임없이 wait을 호출합니다. 따라서 모든 자식은 정리할 부모를 갖게 됩니다. 과제는 부모와 자식이 동시에 대기 하고 종료하는 상황, 그리고 종료 와 종료가 동시에 발생하는 상황 간의 경쟁과 교착 상태를 피하는 것입니다.

대기는 wait_lock(kernel/proc.c:391)을 획득하는 것으로 시작됩니다 . 그 이유는 wait_lock 이 부모가 종료되는 자식의 웨이크업을 놓치지 않도록 보장하는 조건 잠금 역할을 하기 때문 입니다 .
그런 다음 wait은 프로세스 테이블을 스캔합니다. ZOMBIE 상태의 자식을 찾으면 해당 자식의 리소스와 proc 구조를 해제하고 자식의 종료 상태를 wait 에 제공된 주소 (0이 아닌 경우)에 복사하고 자식의 프로세스 ID를 반환합니다. wait이 자식을 찾았지만 종료된 자식이 없으면 sleep을 호출 하여 종료될 때까지 기다 립니다 (kernel/proc.c:433). 그런 다음 다시 스캔합니다. wait은 종종 wait_lock 과 일부 프로세스의 pp->lock이라는 두 개의 잠금을 보유합니다. 교착 상태를 피하기 위한 순서는 먼저 wait_lock을 실행 한 다음 pp->lock을 실행합니다. 종료 (kernel/proc.c:347) 종료 상태를 기록하고, 일부 리소스를 해제하고, reparent를 호출하여 자식을 init 프로세스에 제공하고, 대기 중인 경우 부모를 깨우고 , 호출자를 좀비로 표시하고, CPU를 영구적으로 양보합니다. exit는 이 시퀀스

동안 wait_lock 과 p->lock을 모두 보유합니다 . wait_lock 은 wakeup(p->parent) 에 대한 조건 잠금이기 때문에 wait_lock을 보유하여 대기 중인 부모가 wakeup을 잃지 않도록 합니다. exit는 또한 이 시퀀스에서 p->lock을 보유하여 대기 중인 부모가 자식 이 swtch를 마지막으로 호출하기 전에 자식 이 ZOMBIE 상태인 것을 보지 못하도록 해야 합니다 . exit는 교착 상태를 피하기 위해 wait 과 동일한 순서로 이러한 잠금을 획득합니다 .

부모 프로세스를 ZOMBIE 로 설정하기 전에 exit 가 부모 프로세스를 깨우는 것은 잘못된 것처럼 보일 수 있지만 , 이 방법은 안전합니다. wakeup 으로 부모가 실행될 수 있지만, wait 의 루프는 스케줄러가 자식의 p->lock을 해제할 때까지 자식을 검사할 수 없습니다 . 따라서 wait은 exit가 자식 프로세스를 ZOMBIE 로 상태를 설정한 후 훨씬 지나서야 종료되는 프로세스를 볼 수 있습니다 (kernel/proc.c:379).

exit를 사용 하면 프로세스가 스스로를 종료할 수 있지만 kill (kernel/proc.c:586)을 사용 하면 됩니다. 한 프로세스가 다른 프로세스를 종료하도록 요청하게 합니다. kill이 피해자 프로세스를 직접 파괴하기에는 너무 복잡할 것입니다 . 피해자가 다른 CPU에서 실행 중일 수 있고, 아마도 커널 데이터 구조에 대한 민감한 업데이트 시퀀스 중간에 있을 수 있기 때문입니다. 따라서 kill은 거의 아무것도 하지 않습니다. 피해자의 p->killed를 설정하고 , 절전 모드이면 깨웁니다. 결국 피해자는 커널에 들어가거나 나가고, 이때 usertrap 의 코드는 p->killed가 설정되어 있으면 exit를 호출합니다 (killed (kernel/proc.c:615)를 호출하여 확인합니다). 피해자가 사용자 공간에서 실행 중이라면 시스템 호출을 하거나 타이머(또는 다른 장치)가 인터럽트를 발생시켜 곧 커널에 진입하게 됩니다.

피해자 프로세스가 sleep 상태 인 경우 kill의 wakeup 호출 로 인해 피해자가 sleep에서 돌아옵니다. 이는 기다리고 있는 조건이 사실이 아닐 수 있으므로 잠재적으로 위험합니다.
그러나 xv6에서 sleep 에 대한 호출은 sleep이 반환된 후 조건을 다시 테스트하는 while 루프 에 항상 래핑됩니다 . sleep 에 대한 일부 호출은 루프에서 p->killed를 테스트 하고 현재 활동이 설정되어 있으면 중단합니다. 이는 이러한 중단이 올바를 때에만 수행됩니다. 예를 들어, 파이프 read 및 write 코드는 killed 플래그가 설정되어 있으면 반환됩니다. 결국 코드는 trap으로 돌아가고, 여기서 다시 p->killed를 확인 하고 종료합니다.

일부 xv6 sleep 루프는 p->killed를 확인하지 않습니다 . 왜냐하면 코드가 원자적이어야 하는 다단계 시스템 호출의 중간에 있기 때문입니다. virtio 드라이버 (kernel/virtio_disk.c:285) 예: 디스크 작업이 파일 시스템을 올바른 상태로 유지하기 위해 필요한 쓰기 세트 중 하나일 수 있기 때문에 p->killed를 확인하지 않습니다. 디스크 I/O를 기다리는 동안 종료된 프로세스는 현재 시스템 호출을 완료하고 usertrap이 종료된 플래그를 볼 때까지 종료되지 않습니다.

7.9 프로세스 잠금

각 프로세스와 연관된 잠금 (p->lock) 은 xv6에서 가장 복잡한 잠금입니다. p->lock 에 대해 생각하는 간단한 방법은 다음 struct proc 필드 중 하나를 읽거나 쓰는 동안 유지해야 한다는 것입니다 : p->state, p->chan, p->killed, p->xstate, 및 p->pid. 이러한 필드는 다른 프로세스 또는 다른 코어의 스케줄러 스레드에서 사용할 수 있으므로 잠금으로 보호해야 하는 것은 당연합니다.

그러나 p->lock 의 대부분 사용은 xv6 프로세스 데이터 구조의 상위 수준 측면을 보호하는 것입니다.
tures와 알고리즘. p->lock이 하는 모든 것의 전체 세트는 다음과 같습니다 .

- p->state 와 함께 새로운 프로세스에 대한 proc[] 슬롯 할당 시 경쟁을 방지합니다 .
- 생성되거나 파괴되는 동안 프로세스를 보이지 않게 숨깁니다.
- 부모의 대기가 상태를 ZOMBIE 로 설정한 프로세스를 수집하는 것을 방지합니다 .
  아직 CPU가 나오지 않았습니다.
- 다른 코어의 스케줄러가 해당 코어의 스케줄러가 설정한 후에 양보 프로세스를 실행하기로 결정하지 못하도록 방지합니다.
  switch가 완료되기 전에 상태를 RUNNABLE 로 전환합니다 .

- 단 하나의 코어 스케줄러만이 RUNNABLE 프로세스를 실행하기로 결정하도록 보장합니다.
- 스위치 중에 타이머 인터럽트가 발생하여 프로세스가 양보하는 것을 방지합니다 .
- 조건 잠금과 함께 웨이크업 이 프로세스를 간과하는 것을 방지하는 데 도움이 됩니다.
  sleep을 호출 했지만 CPU 할당을 완료하지 못했습니다.
- kill 의 희생자 프로세스가 종료되고 다시 할당되는 것을 방지합니다.
  kill은 p->pid 를 검사 하고 p->killed를 설정합니다.
- p->state 의 kill 검사와 쓰기를 원자적으로 만듭니다 .

p- >parent 필드는 p->lock 이 아닌 전역 잠금 wait_lock 으로 보호됩니다 .
프로세스의 부모만 p->parent를 수정하지만, 이 필드는 프로세스 자체와 자식을 검색하는 다른 프로세스에서 모두 읽습니다. wait_lock 의 목적은 wait 이 자식이 종료될 때까지 기다리는 동안 조건 잠금 역할을 하는 것입니다 . 종료하는 자식은 상태를 ZOMBIE 로 설정하고 부모를 깨우고 CPU를 넘겨줄 때까지 wait_lock 또는 p->lock을 보유합니다. wait_lock은 또한 부모와 자식의 동시 종료를 직렬화하므로 자식을 상속하는 init 프로세스가 wait에서 깨어나는 것이 보장됩니다 . wait_lock 은 각 부모에서 프로세스별 잠금이 아니라 전역 잠금입니다. 프로세스가 wait_lock을 획득하기 전까지는 부모가 누구인지 알 수 없기 때문입니다.

7.10 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. 실행 가능한 우선순위가 높은 프로세스가 스케줄러에 의해 실행 가능한 우선순위가 낮은 프로세스보다 선호된다는 아이디어입니다. 이러한 정책은 종종 경쟁적인 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 모두 특정 잠금을 사용할 때 발생할 수 있으며, 우선순위가 낮은 프로세스가 잠금을 획득하면 우선순위가 높은 프로세스가 진행되지 않을 수 있습니다. 대기 중인 프로세스의 긴 호송은 많은 우선순위가 높은 프로세스가 공유 잠금을 획득하는 우선순위가 낮은 프로세스를 기다릴 때 형성될 수 있습니다. 호송이 형성되면 오랫동안 지속될 수 있습니다. 이런 종류의 문제를 피하기 위해서는 정교한 스케줄러에 추가적인 메커니즘이 필요합니다.

sleep 과 wakeup은 간단하고 효과적인 동기화 방법이지만, 그 외에도 많은 방법이 있습니다. 이 모든 방법에서 첫 번째 과제는 이 장의 시작 부분에서 본 "wakeup 손실" 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 wakeup 손실을 방지하기 위해 sleep 조건에 대한 마지막 순간 확인 역할을 합니다. Linux 커널의

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

웨이크업 에서 프로세스 전체 세트를 스캔하는 것은 비효율적입니다. 더 나은 해결책은 sleep 과 wakeup 모두에서 chan을 해당 구조에서 sleep하는 프로세스 목록을 보관하는 데이터 구조로 바꾸는 것입니다(예: Linux의 대기 큐). Plan 9 의 sleep 과 wakeup은 랑데부 지점을 구조화합니다. 많은 스레드 라이브러리는 동일한 구조를 조건 변수라고 합니다. 이 맥락에서 sleep 과 wakeup 작업은 wait 과 signal 이라고 합니다 . 이러한 모든 메커니즘은 동일한 특징을 공유합니다. sleep 조건은 sleep 중에 원자적으로 삭제된 어떤 종류의 잠금으로 보호됩니다.

웨이크업 구현은 특정 채널에서 대기 중인 모든 프로세스를 깨우고, 많은 프로세스가 해당 특정 채널을 기다리고 있을 수 있습니다. 운영 체제는 이러한 모든 프로세스를 스케줄링하고, 이들은 슬립 조건을 확인하기 위해 경쟁합니다.

이런 식으로 행동하는 프로세스를 때때로 천둥치는 무리라고 부르기도 하는데, 이는 피하는 것이 가장 좋습니다.
대부분의 조건 변수에는 웨이크업을 위한 두 가지 기본 요소가 있습니다. 신호(signal) 는 한 프로세스를 깨우고, 브로드캐스트(broadcast)는 대기 중인 모든 프로세스를 깨웁니다.

세마포어는 종종 동기화에 사용됩니다. 카운트는 일반적으로 파이프 버퍼에서 사용 가능한 바이트 수 또는 프로세스가 가진 좀비 자식 수와 같은 것에 해당합니다. 추상화의 일부로 명시적 카운트를 사용하면 "잃어버린 웨이크업" 문제를 피할 수 있습니다. 발생한 웨이크업 수에 대한 명시적 카운트가 있습니다. 또한 카운트는 잘못된 웨이크업 및 천둥 무리 문제를 피합니다.

프로세스를 종료하고 정리하면 xv6에서 복잡성이 많이 발생합니다. 대부분의 운영 체제에서는 훨씬 더 복잡합니다. 예를 들어 피해자 프로세스가 커널 깊숙이 잠자고 있을 수 있고, 스택을 풀려면 각 함수가 정리를 해야 할 수 있으므로 주의가 필요합니다. 일부 언어는 예외 메커니즘을 제공하여 도움을 주지만 C는 그렇지 않습니다. 게다가, 아직 기다리던 이벤트가 발생하지 않았더라도 잠자고 있는 프로세스가 깨어날 수 있는 다른 이벤트가 있습니다. 예를 들어, Unix 프로세스가 잠자고 있을 때 다른 프로세스가 해당 프로세스에 신호를 보낼 수 있습니다 . 이 경우 프로세스는 중단된 시스템 호출에서 값 -1과 오류 코드가 EINTR로 설정된 상태로 반환됩니다. 애플리케이션은 이러한 값을 확인하고 무엇을 할지 결정할 수 있습니다. Xv6는 신호를 지원하지 않으므로 이러한 복잡성이 발생하지 않습니다.

Xv6의 kill 지원은 전적으로 만족스럽지 않습니다. p->killed를 확인해야 할 sleep 루프가 있습니다 . 관련된 문제는 p->killed를 확인하는 sleep 루프의 경우에도 sleep 과 kill 사이에 경쟁이 있다는 것 입니다 . kill은 p->killed를 설정 하고 피해자의 루프가 p->killed를 확인한 직후 sleep을 호출하기 전에 피해자를 깨우려고 할 수 있습니다 . 이 문제가 발생하면 피해자는 기다리는 조건이 발생할 때까지 p->killed를 알아차리지 못합니다 . 이는 꽤 늦을 수도 있고 전혀 없을 수도 있습니다(예: 피해자가 콘솔에서 입력을 기다리고 있지만 사용자가 아무 입력도 입력하지 않은 경우).

실제 운영 체제는 상수에 명시적인 자유 목록을 사용하여 자유 proc 구조를 찾습니다.
allocproc 의 선형 시간 검색 대신 시간을 사용합니다. xv6은 단순성을 위해 선형 스캔을 사용합니다.

7.11 연습문제

1. Sleep은 교착 상태를 피하기 위해 lk != &p->lock을 확인해야 합니다. 특수한 경우는 다음과 같습니다.
   교체하여 제거

   ```
   if(lk != &p->lock){
       (&p->lock을) 획득합니다. (lk를) 해제
       합니다.
   }
   ```

   ~와 함께

   ```
   해제(lk); 획득(&p->lock);
   ```

   이렇게 하면 잠이 깨질 거야. 어떻게?

2. sleep 과 wakeup을 사용하지 않고 xv6에서 세마포어를 구현합니다 (하지만 spin lock을 사용해도 됩니다). xv6에서 sleep과 wakeup의 사용을 세마포어로 대체합니다. 결과를 판단합니다.

3. 위에서 언급한 kill 과 sleep 간의 경쟁을 수정하여 피해자의 sleep 루프가 p->killed를 확인한 후 sleep을 호출하기 전에 kill이 발생하면 피해자 가 현재 시스템 호출을 중단하게 됩니다.

4. 모든 sleep 루프가 p->killed를 확인하도록 계획을 설계하여 예를 들어 virtio 드라이버에 있는 프로세스가 다른 프로세스에 의해 종료된 경우 while 루프에서 빠르게 반환될 수 있도록 합니다.
   프로세스.

5. 스케줄러 스레드를 통해 전환하는 대신, 한 프로세스의 커널 스레드에서 다른 프로세스의 커널 스레드로 전환할 때 하나의 컨텍스트 스위치만 사용하도록 xv6를 수정합니다. 양보하는 스레드는 다음 스레드를 직접 선택하고 swtch를 호출해야 합니다. 과제는 여러 코어가 실수로 같은 스레드를 실행하는 것을 방지하고, 잠금을 올바르게 하고, 교착 상태를 피하는 것입니다.

6. 실행 가능한 프로세스가 없을 때 RISC-V WFI (인터럽트 대기) 명령어를 사용하도록 xv6의 스케줄러를 수정합니다 . 실행 가능한 프로세스가 실행을 기다리고 있을 때마다 WFI에서 코어가 일시 중지되지 않도록 합니다.

8장

파일 시스템

파일 시스템의 목적은 데이터를 구성하고 저장하는 것입니다. 파일 시스템은 일반적으로 사용자와 애플리케이션 간의 데이터 공유를 지원하고, 재부팅 후에도 데이터를 계속 사용할 수 있도록 지속성을 지원합니다.

xv6 파일 시스템은 Unix와 유사한 파일, 디렉토리 및 경로 이름을 제공합니다(1장 참조).
지속성을 위해 virtio 디스크에 데이터를 저장합니다. 파일 시스템은 여러 가지 과제를 해결합니다.

- 파일 시스템은 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 크래시 복구를 지원해야 합니다. 즉, 크래시(예: 정전)가 발생하더라도 파일 시스템은 재시작 후에도 여전히 올바르게 작동해야 합니다. 크래시로 인해 업데이트 시퀀스가 중단되고 디스크 데이터 구조가 일관되지 않을 수 있습니다(예: 파일에서 사용되고 비어 있는 것으로 표시된 블록).

- 여러 프로세스가 동시에 파일 시스템에서 작동할 수 있으므로 파일 시스템 코드는 불변성을 유지하기 위해 조정해야 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.

8.1 개요

xv6 파일 시스템 구현은 그림 8.1에 표시된 7개 계층으로 구성됩니다. 디스크 계층은 virtio 하드 드라이브의 블록을 읽고 씁니다. 버퍼 캐시 계층은 디스크 블록을 캐시하고 이에 대한 액세스를 동기화하여 한 번에 하나의 커널 프로세스만 특정 블록에 저장된 데이터를 수정할 수 있도록 합니다. 로깅 계층은 상위 계층이 트랜잭션에서 여러 블록에 대한 업데이트를 래핑할 수 있도록 하며, 충돌 시 블록이 원자적으로 업데이트되도록 합니다(즉, 모든 블록이 업데이트되거나 전혀 업데이트되지 않음). inode 계층은 개별 파일을 제공하며 각각

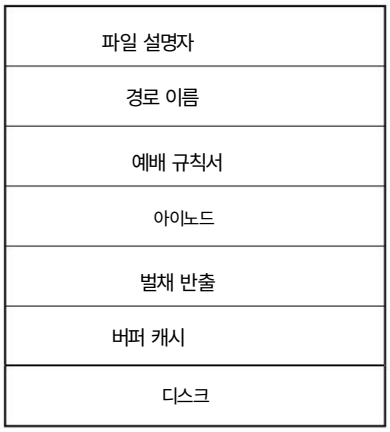

그림 8.1: xv6 파일 시스템의 계층.

고유한 i-번호와 파일 데이터를 보관하는 몇몇 블록이 있는 inode로 표현됩니다. 디렉토리 계층은 각 디렉토리를 특수한 종류의 inode로 구현하는데, 그 내용은 디렉토리 항목의 시퀀스이며, 각각에는 파일 이름과 i-번호가 포함됩니다. 경로명 계층은 /usr/rtm/xv6/fs.c 와 같은 계층적 경로명을 제공하고 재귀적 조회로 이를 해결합니다. 파일 설명자 계층은 파일 시스템 인터페이스를 사용하여 많은 Unix 리소스(예: 파이프, 장치, 파일 등)를 추상화하여 애플리케이션 프로그래머의 삶을 단순화합니다.

디스크 하드웨어는 전통적으로 디스크의 데이터를 512바이트 블록(섹터라고도 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. Xv6는 메모리에 읽은 블록의 사본을 struct buf (kernel/buf.h:1) 유형의 객체에 보관합니다. 이 구조에 저장된 데이터는 때때로 디스크와 동기화되지 않을 수 있습니다. 즉, 디스크에서 아직 읽혀지지 않았을 수도 있고(디스크가 작업하고 있지만 섹터의 내용을 아직 반환하지 않았을 수도 있음), 소프트웨어에 의해 업데이트되었지만 아직 디스크에 쓰여지지 않았을 수도 있습니다.

파일 시스템은 디스크에서 inode와 콘텐츠 블록을 저장하는 위치에 대한 계획이 있어야 합니다. 이를 위해 xv6는 그림 8.2와 같이 디스크를 여러 섹션으로 나눕니다. 파일 시스템은 블록 0을 사용하지 않습니다(부트 섹터를 보유). 블록 1은 슈퍼블록이라고 하며, 파일 시스템에 대한 메타데이터(블록 단위의 파일 시스템 크기, 데이터 블록 수, inode 수, 로그의 블록 수)를 포함합니다. 2에서 시작하는 블록은 로그를 보유합니다. 로그 다음에는 블록당 여러 inode가 있는 inode가 있습니다. 그 다음에는 사용 중인 데이터 블록을 추적하는 비트맵 블록이 옵니다. 나머지 블록은 데이터 블록입니다. 각각은 비트맵 블록에서 사용 가능으로 표시되거나 파일이나 디렉토리의 콘텐츠를 보유합니다. 슈퍼블록은 초기 파일 시스템을 빌드하는 mkfs 라는 별도의 프로그램으로 채워집니다.

이 장의 나머지 부분에서는 버퍼 캐시부터 시작하여 각 계층에 대해 설명합니다. 상황에 주의하세요.
하위 계층에서 잘 선택된 추상화가 상위 계층의 설계를 용이하게 하는 방식입니다.

그림 8.2: xv6 파일 시스템의 구조.

8.2 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.c 에 있습니다.

버퍼 캐시에서 내보내는 주요 인터페이스는 bread 와 bwrite로 구성됩니다. 전자는 메모리에서 읽거나 수정할 수 있는 블록의 사본이 포함된 buf를 가져오고, 후자는 수정된 버퍼를 디스크의 해당 블록에 씁니다. 커널 스레드는 버퍼를 다 사용하면 brelse를 호출하여 버퍼를 해제해야 합니다. 버퍼 캐시는 버퍼당 sleep-lock을 사용하여 한 번에 한 스레드만 각 버퍼(따라서 각 디스크 블록)를 사용하도록 합니다. bread는 잠긴 버퍼를 반환하고 brelse는 잠금을 해제합니다.

버퍼 캐시로 돌아가 봅시다. 버퍼 캐시는 디스크를 보관하기 위해 고정된 수의 버퍼를 가지고 있습니다.
블록, 즉 파일 시스템이 캐시에 없는 블록을 요청하는 경우 버퍼 캐시는 현재 다른 블록을 보유하고 있는 버퍼를 재활용해야 합니다. 버퍼 캐시는 새 블록에 대해 가장 최근에 사용되지 않은 버퍼를 재활용합니다. 가장 최근에 사용되지 않은 버퍼는 곧 다시 사용될 가능성이 가장 낮다는 가정이 있습니다.

8.3 코드: 버퍼 캐시

버퍼 캐시는 버퍼의 이중 연결 리스트입니다. main (kernel/- main.c:27) 에서 호출하는 함수 binit 은 정적 배열 buf에 있는 NBUF 버퍼 로 목록을 초기화합니다 (kernel/bio.c:43-52). 버퍼 캐시에 대한 다른 모든 액세스는 buf 배열이 아닌 bcache.head를 통한 연결 목록을 참조합니다 .

버퍼에는 두 개의 상태 필드가 연관되어 있습니다. 필드 valid는 버퍼에 블록 사본이 들어 있음을 나타냅니다. 필드 disk는 버퍼 내용이 디스크에 전달되어 버퍼가 변경될 수 있음을 나타냅니다(예: 디스크에서 데이터로 데이터 쓰기).

빵 (커널/bio.c:93) 주어진 섹터 (kernel/bio.c:97) 에 대한 버퍼를 가져오기 위해 bget을 호출합니다 . 버퍼를 디스크에서 읽어야 하는 경우 bread는 버퍼를 반환하기 전에 virtio_disk_rw를 호출하여 해당 작업을 수행합니다. bget (kernel/bio.c:59) 주어진 장치와 섹터 번호 (kernel/bio.c:65-73)를 가진 버퍼를 버퍼 목

록에서 스캔합니다. 해당 버퍼가 있으면 bget은 버퍼에 대한 sleep-lock을 획득합니다. 그런 다음 bget은 잠긴 버퍼를 반환합니다.

주어진 섹터에 대한 캐시된 버퍼가 없는 경우, bget은 버퍼를 만들어야 하며, 다른 섹터를 보관했던 버퍼를 재사용할 수 있습니다. 버퍼 목록을 두 번째로 스캔하여 사용되지 않는 버퍼를 찾습니다 (b->refcnt = 0). 이러한 버퍼는 모두 사용할 수 있습니다. bget은 버퍼 메타데이터를 편집하여 새 장치와 섹터 번호를 기록하고 해당 sleep-lock을 획득 합니다 . 할당 b->valid = 0 은 bread가 버퍼의

이전 내용.

디스크 섹터당 최대 하나의 캐시된 버퍼가 있어야 독자가 쓰기를 볼 수 있고 파일 시스템이 동기화를 위해 버퍼에 잠금을 사용하기 때문에 중요합니다. bget은 블록이 캐시되었는지에 대한 첫 번째 루프의 확인에서 블록이 이제 캐시되었다는 두 번째 루프의 선언(dev, blockno 및 refcnt 설정) 까지 bache.lock 을 지속적으로 유지하여 이 불변성을 보장합니다. 이렇게 하면 블록의 존재 여부와(존재하지 않는 경우) 블록을 원자적으로 유지하는 버퍼 지정을 확인합니다.

bget이 bcache.lock 중요 섹션 외부에서 버퍼의 sleep-lock을 획득하는 것은 안전합니다. 0이 아닌 b->refcnt가 버퍼가 다른 디스크 블록에 재사용되는 것을 방지하기 때문입니다. sleep-lock은 블록의 버퍼링된 콘텐츠에 대한 읽기 및 쓰기를 보호하는 반면, bcache.lock 은 캐시된 블록에 대한 정보를 보호합니다.

모든 버퍼가 바쁘다면 너무 많은 프로세스가 동시에 파일 시스템 호출을 실행하고 있다는 뜻입니다 . bget 패닉이 발생합니다. 버퍼가 비워질 때까지 sleep하는 것이 더 우아한 대응책이 될 수 있지만, 그러면 교착 상태가 발생할 가능성이 있습니다.

bread가 디스크를 읽고(필요한 경우) 버퍼를 호출자에게 반환하면 호출자는 버퍼를 독점적으로 사용할 수 있으며 데이터 바이트를 읽거나 쓸 수 있습니다. 호출자가 버퍼를 수정하는 경우 버퍼를 해제하기 전에 bwrite를 호출하여 변경된 데이터를 디스크에 써야 합니다 . bwrite (kernel/bio.c:107) virtio_disk_rw를 호출하여 디스크 하드웨어와 통신합니다.

호출자가 버퍼를 다 사용하면 brelse를 호출 하여 해제해야 합니다. (brelse 라는 이름은 b-release의 줄임말로, 난해하지만 배울 가치가 있습니다. Unix에서 유래했으며 BSD, Linux, Solaris에서도 사용됩니다.) brelse (kernel/bio.c:117) 슬립 잠금을 해제하고 버퍼를 연결 목록의 앞으로 옮깁니다 (kernel/bio.c:128-133). 버퍼를 이동하면 목록이 버퍼가 얼마나 최근에 사용되었는지(즉, 해제되었는지)에 따라 정렬됩니다. 목록의 첫 번째 버퍼가 가장 최근에 사용된 버퍼이고, 마지막 버퍼가 가장 최근에 사용되지 않은 버퍼입니다. bget 의 두 루프는 이를 활용합니다. 기존 버퍼에 대한 스캔은 최악의 경우 전체 목록을 처리해야 하지만 가장 최근에 사용된 버퍼를 먼저 확인(bcache.head 에서 시작하여 다음 포인터를 따라감)하면 참조의 지역성이 양호할 때 스캔 시간이 줄어듭니다. 재사용할 버퍼를 선택하는 스캔은 뒤로 스캔(이전 포인터 를 따라감)하여 가장 최근에 사용되지 않은 버퍼를 선택합니다 .

8.4 로깅 계층

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. xv6가 여러 사용자를 지원했다면 이 상황은 보안 문제가 될 수 있습니다. 이전 파일의 소유자가 다음을 읽을 수 있기 때문입니다.

다른 사용자가 소유한 새 파일에 블록을 작성합니다.

Xv6는 간단한 로깅 형식을 통해 파일 시스템 작업 중에 발생하는 충돌 문제를 해결합니다.
xv6 시스템 호출은 디스크상 파일 시스템 데이터 구조를 직접 쓰지 않습니다. 대신 디스크의 로그에 수행하려는 모든 디스크 쓰기에 대한 설명을 넣습니다. 시스템 호출이 모든 쓰기를 기록한 후 디스크에 특별한 커밋 레코드를 기록하여 로그에 완전한 작업이 포함되어 있음을 나타냅니다. 그 시점에서 시스템 호출은 쓰기를 디스크상 파일 시스템 데이터 구조에 복사합니다. 쓰기가 완료된 후 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.

xv6의 로그가 파일 시스템 작업 중 충돌 문제를 해결하는 이유는 무엇입니까? 작업이 커밋되기 전에 충돌이 발생하면 디스크의 로그가 완료로 표시되지 않고 복구 코드는 이를 무시하며 디스크 상태는 작업이 시작되지 않은 것처럼 됩니다. 작업이 커밋된 후에 충돌이 발생하면 복구는 작업의 모든 쓰기를 재생하고 작업이 디스크 데이터 구조에 쓰기를 시작한 경우 반복할 수 있습니다. 어느 경우든 로그는 충돌과 관련하여 작업을 원자화합니다. 복구 후에 작업의 모든 쓰기가 디스크에 나타나거나 아무것도 나타나지 않습니다.

8.5 로그 디자인

로그는 슈퍼블록에 지정된 알려진 고정 위치에 있습니다. 헤더 블록과 업데이트된 블록 사본("로깅된 블록") 시퀀스로 구성됩니다. 헤더 블록에는 로깅된 블록마다 하나씩 있는 섹터 번호 배열과 로그 블록 수가 포함됩니다. 디스크의 헤더 블록에 있는 수는 0으로, 로그에 트랜잭션이 없음을 나타내거나 0이 아닌 값으로, 로그에 지정된 수의 로깅된 블록이 있는 완전한 커밋된 트랜잭션이 포함되어 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 프로세스에 의한 파일 시스템 작업의 동시 실행을 허용하기 위해 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 트랜잭션을 함께 커밋하는 아이디어는 그룹 커밋이라고 합니다. 그룹 커밋은 커밋의 고정 비용을 여러 작업에 걸쳐 상각하기 때문에 디스크 작업 수를 줄입니다. 또한 그룹 커밋은 디스크 시스템에 동시에 더 많은 쓰기를 제공하여 디스크가 단일 디스크 회전 중에 모든 쓰기를 할 수 있도록 합니다. Xv6의 virtio 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 로그 공간보다 많은 개별 블록을 쓸 수 없습니다.
이는 대부분 시스템 호출에 문제가 되지 않지만, 그 중 두 개는 잠재적으로 많은 블록을 쓸 수 있습니다: write 와 unlink. 큰 파일 쓰기는 많은 데이터 블록과 많은 비트맵 블록, 그리고 inode 블록을 쓸 수 있습니다. 큰 파일의 unlink는 많은 비트맵 블록과 inode를 쓸 수 있습니다. Xv6의 write 시스템 호출은 큰 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.

8.6 코드: 로깅

시스템 호출에서 로그를 일반적으로 사용하는 예는 다음과 같습니다.

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

 ...
종료_연산();
```

begin_op (커널/log.c:127) 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 모두 증가시키면 공간이 예약되고 이 시스템 호출 중에 커밋이 발생하지 않습니다. 이 코드는 각 시스템 호출이 최대 MAXOPBLOCKS 개의 개별 블록을 쓸 수 있다고 보수적으로 가정합니다.

log_write (커널/log.c:215) bwrite 의 프록시 역할을 합니다 . 메모리에 블록의 섹터 번호를 기록하여 디스크의 로그에 슬롯을 예약하고, 블록 캐시에 버퍼를 고정하여 블록 캐시가 블록을 제거하지 못하도록 합니다. 블록은 커밋될 때까지 캐시에 남아 있어야 합니다. 그때까지 캐시된 사본은 수정 사항의 유일한 기록이며, 커밋 후에야 디스크의 해당 위치에 쓸 수 있으며, 동일한 트랜잭션의 다른 읽기는 수정 사항을 확인해야 합니다. log_write는 단일 트랜잭션 중에 블록이 여러 번 쓰여지는 경우 이를 감지하고 로그의 동일한 슬롯에 해당 블록을 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 (커널/log.c:147) 먼저 미처리 시스템 호출의 카운트를 감소시킵니다. 카운트가 이제 0이면 commit()을 호출하여 현재 트랜잭션을 커밋합니다. 이 프로세스에는 4단계가 있습니다. write_log() (kernel/log.c:179) 트랜잭션에서 수정된 각 블록을 버퍼 캐시에서 디스크의 로그에 있는 슬롯으로 복사합니다. write_head() (kernel/log.c:103) 헤더 블록을 디스크에 씁니다. 이것은 커밋 지점이며 쓰기 후 충돌이 발생하면 복구가 재생됩니다.

로그에서 트랜잭션 쓰기. install_trans (kernel/log.c:69) 로그에서 각 블록을 읽고 파일 시스템의 적절한 위치에 씁니다. 마지막으로 end_op는 카운트 0으로 로그 헤더를 씁니다. 이는 다음 트랜잭션이 로깅된 블록을 쓰기 시작하기 전에 수행되어야 하므로 충돌로 인해 후속 트랜잭션의 로깅된 블록을 사용하여 한 트랜잭션의 헤더를 사용하여 복구되지 않습니다. recover_from_log (kernel/log.c:117) initlog (kernel/log.c:55) 에서 호출됩니다 . fsinit(kernel/fs.c:42) 에서 호출됩니다 . 부팅 중 첫 번째 사용자 프로세스가 실행되기 전 (kernel/proc.c:527). 로그 헤더를 읽고 헤더가 로그에 커밋된 트랜잭션이 포함되어 있음을 나타내는 경우 end_op 의

동작을 모방합니다.

로그의 사용 예는 filewrite (kernel/file.c:135)에서 나타납니다. 거래 내용은 다음과 같습니다.

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

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.

8.7 코드: 블록 할당기

파일 및 디렉토리 내용은 디스크 블록에 저장되며, 이는 빈 풀에서 할당해야 합니다. Xv6의 블록 할당자는 블록당 1비트씩 디스크에 빈 비트맵을 유지합니다. 0비트는 해당 블록이 비어 있음을 나타내고, 1비트는 사용 중임을 나타냅니다. 프로그램 mkfs는 부트 섹터, 슈퍼 블록, 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 의 루프는 (kernel/fs.c:72) 에 있습니다 . 블록 0에서 시작하여 sb.size 까지 모든 블록을 고려합니다 . 이는 파일 시스템의 블록 수입니다. 비트맵 비트가 0인 블록을 찾습니다. 이는 해당 블록이 비어 있음을 나타냅니다. balloc이 이러한 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(Bits-Per-Block) 비트를 확인합니다. 두 프로세스가 동시에 블록을 할당하려고 하면 발생할 수 있는 경쟁은 버퍼 캐시가 한 번에 하나의 프로세스만 하나의 비트맵 블록을 사용할 수 있도록 허용한다는 사실로 방지됩니다. bfree (kernel/fs.c:92) 올바른 비트맵 블록을 찾아 올바른 비트를 지웁니다. 다시 독점

bread 와 brelse가 암시적으로 사용되기 때문에 명시적 잠금이 필요 없습니다.

이 장의 나머지 부분에서 설명하는 대부분의 코드와 마찬가지로 balloc 및 bfree 는 트랜잭션 내부에서 호출됩니다.

8.8 아이노드 층

inode라는 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.

디스크상의 inode는 struct dinode (kernel/fs.h:32)에 의해 정의됩니다. type 필드 는 파일, 디렉토리, 특수 파일(장치)을 구분합니다. type이 0이면 디스크상의 inode가 비어 있음을 나타냅니다. nlink 필드는 이 inode를 참조하는 디렉토리 항목의 수를 계산하여 디스크상의 inode와 해당 데이터 블록을 언제 비워야 하는지 인식합니다. size 필드는 파일의 내용 바이트 수를 기록합니다. addrs 배열은 파일의 내용을 보관하는 디스크 블록의 블록 번호를 기록합니다.

커널은 itable이라는 테이블의 메모리에 있는 활성 inode 세트를 보관합니다 . struct inode (kernel/file.h:17) 디스크에 있는 struct dinode 의 메모리 내 복사본입니다 . 커널은 해당 inode를 참조하는 C 포인터가 있는 경우에만 inode를 메모리에 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. itable.lock은 inode가 inode 테이블에 최대 한 번만 존재한다는 불변성과 메모리 내 inode의 ref 필드가 inode에 대한 메모리 내 포인터의 수를 세는 불변성을 보호합니다. 각 메모리 내 inode에는 sleep-lock을 포함하는 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 inode를 테이블에 유지하고 다른 inode에 대해 테이블 항목을 재사용하지 않습니다. 마지막으로 각 inode에는 파일을 참조하는 디렉토리 항목의 수를 세는 nlink 필드(디스크에 있고 메모리에 있는 경우 메모리에 복사됨)가 있습니다. xv6은 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 부분은 iget() 의 이러한 동작에 의존 하여 inode에 대한 장기 참조(열린 파일 및 현재 디렉토리)를 보관하고 여러 inode를 조작하는 코드(예: 경로명 조회)에서 교착 상태를 피하면서 경쟁을 방지합니다.

iget 이 반환하는 struct inode에는 유용한 내용이 없을 수 있습니다. 디스크에 있는 inode의 사본을 보관하기 위해 코드는 ilock을 호출해야 합니다. 이렇게 하면 inode가 잠기고(다른 프로세스가 ilock 할 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 다음을 보관할 수 있습니다.

iget 이 반환한 inode에 대한 C 포인터 이지만 , 한 번에 하나의 프로세스만 inode를 잠글 수 있습니다.

inode 테이블은 커널 코드나 데이터 구조가 C 포인터를 보관하는 inode만 저장합니다. 주요 작업은 여러 프로세스의 액세스를 동기화하는 것입니다. inode 테이블은 또한 자주 사용되는 inode를 캐시하지만 캐싱은 부차적입니다. inode가 자주 사용되는 경우 버퍼 캐시는 아마도 메모리에 보관할 것입니다. 메모리 내 inode를 수정하는 코드는 iupdate를 사용하여 디스크에 씁니다.

8.9 코드: Inodes

새로운 inode를 할당하기 위해(예를 들어, 파일을 생성할 때) xv6는 ialloc (kernel/fs.c:199)을 호출합니다. ialloc은 balloc 과 유사합니다 . 디스크의 inode 구조를 한 번에 한 블록씩 반복하면서 free로 표시된 것을 찾습니다. 하나를 찾으면 새 유형을 디스크에 쓰고 iget에 대한 tail 호출과 함께 inode 테이블에서 항목을 반환합니다 (kernel/fs.c:213). ialloc 의 올바른 작동은 한 번에 하나의 프로세스만 bp 에 대한 참조를 보유할 수 있다는 사실에 달려 있습니다 . ialloc은 다른 프로세스가 동시에 inode가 사용 가능하다는 것을 보고 이를 청구하려고 하지 않는다는 것을 확신할 수 있습니다. iget (kernel/fs.c:247) 원하는 장치와 inode 번호가 있는 활성 항목 (ip->ref > 0)을 inode 테이블에서 찾습니다. 하나를 찾으면 해당 inode에 대한 새 참조를 반환합니다 (kernel/fs.c:256-260). iget 이 스캔 할 때 첫 번째 빈 슬롯의 위치 (kernel/

fs.c:261- 262)를 기록합니다. 테이블 항목을 할당해야 하는 경우 사용합니다.

코드는 메타데이터나 콘텐츠를 읽거나 쓰기 전에 ilock을 사용하여 inode를 잠가야 합니다 . ilock (kernel/fs.c:293) 이 목적을 위해 sleep-lock을 사용합니다. ilock이 inode에 대한 독점적 액세스 권한을 얻으면 필요한 경우 디스크(더 가능성이 높은 버퍼 캐시)에서 inode를 읽습니다. iunlock 함수 (kernel/fs.c:321) 슬립 잠금을 해제하여 슬립 상태에 있는 모든 프로세스를 깨웁니다.

iput (커널/fs.c:337) 참조 카운트를 감소시켜 inode에 대한 C 포인터를 해제합니다 (kernel/fs.c:360). 이것이 마지막 참조인 경우, inode 테이블에서 해당 inode의 슬롯은 이제 비어 있으며 다른 inode에 다시 사용될 수 있습니다.

iput이 inode에 대한 C 포인터 참조가 없고, inode가 해당 inode에 대한 링크도 없음(디렉토리에 없음)을 확인한 경우 , inode와 해당 데이터 블록을 해제해야 합니다. iput은 itrunc를 호출하여 파일을 0바이트로 자르고, 데이터 블록을 해제합니다. inode 유형을 0(할당되지 않음)으로 설정하고, inode를 디스크에 씁니다 (kernel/fs.c:342).

iput 에서 inode를 해제하는 경우의 잠금 프로토콜은 자세히 살펴볼 가치가 있습니다. 한 가지 위험은 동시 스레드가 ilock 에서 이 inode를 사용하기 위해 기다리고 있을 수 있다는 것입니다(예: 파일을 읽거나 디렉토리를 나열하기 위해). 그리고 inode가 더 이상 할당되지 않았다는 것을 알게 될 준비가 되어 있지 않을 수 있습니다. 이는 시스템 호출이 해당 inode에 대한 링크가 없고 ip->ref 가 하나일 경우 메모리 내 inode에 대한 포인터를 가져올 방법이 없기 때문에 발생할 수 없습니다. 해당 참조는 iput을 호출하는 스레드가 소유한 참조입니다 . iput이 itable.lock 중요 섹션 외부에서 참조 카운트가 1인지 확인하는 것은 사실이지만 , 그 시점에서 링크 카운트는 0으로 알려져 있으므로 어떤 스레드도 새 참조를 획득하려고 하지 않습니다. 또 다른 주요 위험은 ialloc 에 대한 동시 호출이 iput 이 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 획득할 때까지 정중하게 기다린 후, 이때 iput 이 해당 작업을 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, 읽기 전용 시스템 호출조차도 파일 시스템을 사용하는 경우 트랜잭션으로 래핑해야 합니다.

iput() 과 충돌 사이에는 까다로운 상호 작용이 있습니다 . iput()은 파일의 링크 카운트가 0으로 떨어지면 파일을 즉시 자르지 않습니다. 어떤 프로세스가 메모리에서 inode에 대한 참조를 여전히 보관할 수 있기 때문입니다. 프로세스가 성공적으로 파일을 열었기 때문에 여전히 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 제거하면 해당 inode를 목록에서 제거하여 디스크 목록을 업데이트합니다. 복구 시 파일 시스템은 목록에 있는 모든 파일을 해제합니다.

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

8.10 코드: Inode 내용

디스크상의 inode 구조인 struct dinode는 크기와 블록 번호 배열을 포함합니다(그림 8.3 참조). inode 데이터는 dinode 의 addrs 배열에 나열된 블록에서 찾을 수 있습니다. 첫 번째 NDIRECT 데이터 블록은 배열의 첫 번째 NDIRECT 항목에 나열되어 있습니다. 이러한 블록을 직접 블록이라고 합니다. 다음 NINDIRECT 데이터 블록은 inode가 아니라 간접 블록이라고 하는 데이터 블록에 나열되어 있습니다. addrs 배열의 마지막 항목은 간접 블록의 주소를 제공합니다.

따라서 파일의 처음 12kB(NDIRECT x BSIZE) 바이트는 inode에 나열된 블록에서 로드할 수 있는 반면, 다음 256kB (NINDIRECT x BSIZE) 바이트는 간접 블록을 참조한 후에만 로드할 수 있습니다. 이는 디스크상에서 좋은 표현이지만 클라이언트에게는 복잡한 표현입니다. bmap 함수는 곧 보게 될 readi 및 writei 와 같은 상위 수준 루틴이 이 복잡성을 관리할 필요가 없도록 표현을 관리합니다 . bmap은 inode ip에 대한 bn번째 데이터 블록 의 디스크 블록 번호를 반환합니다 . ip에 아직 그러한 블록이 없으면 bmap 은 하나를 할당합니다.

bmap 함수 (kernel/fs.c:383) 쉬운 경우부터 시작합니다. 첫 번째 NDIRECT 블록은 inode 자체에 나열됩니다 (kernel/fs.c:388-396). 다음 NINDIRECT 블록은 ip->addrs[NDIRECT] 의 간접 블록에 나열되어 있습니다 . bmap은 간접 블록 (kernel/fs.c:407) 을 읽습니다. 그런 다음 블록 내의 오른쪽 위치에서 블록 번호를 읽습니다 (kernel/fs.c:408). 블록 번호가 NDIRECT+NINDIRECT를 초과하면 bmap이 패닉을 일으킵니다. writei에는 이런 일이 발생하지 않도록 하는 검사 기능이 포함되어 있습니다 (kernel/fs.c:513). bmap은 필요에 따라 블록을 할당합니다. ip->addrs[] 또는 간접 항목 0은 블록이 할당되지 않았음을 나타냅니다. bmap이 0을 만나면 새로

운 블록의 번호로 대체합니다.

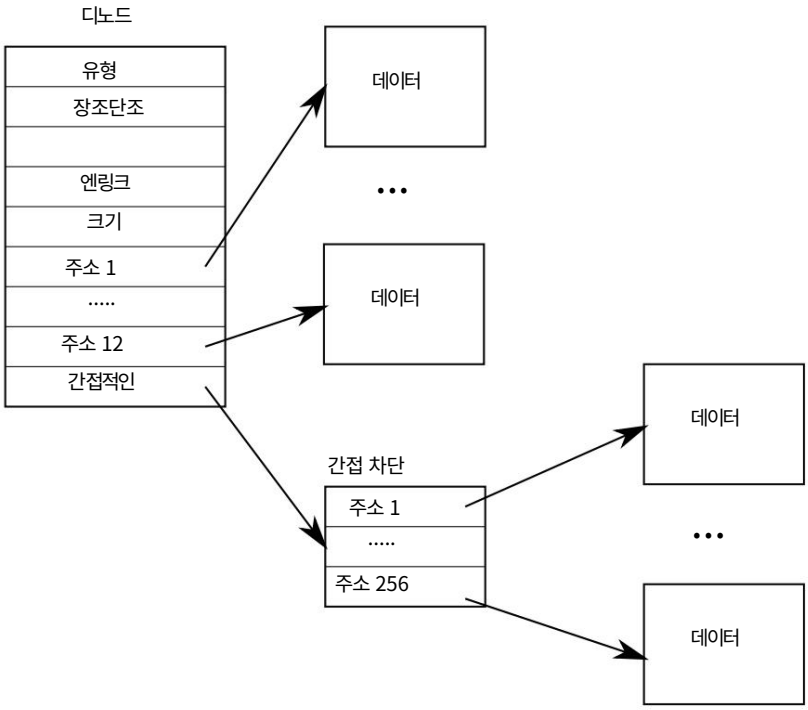

그림 8.3: 디스크에 있는 파일의 표현.

요구에 따라 할당됨 (kernel/fs.c:389-390) (kernel/fs.c:401-402).

itrunc는 파일의 블록을 해제하고 inode 크기를 0으로 재설정합니다. itrunc (kernel/fs.c:426) 직접 블록 (kernel/fs.c:432-437)을 해제하는 것으로 시작합니다. 그런 다음 간접 블록 (kernel/fs.c:442-445)에 나열된 항목은 다음과 같습니다. 마지막으로 간접 블록 자체 (kernel/fs.c:447-448).

bmap을 사용 하면 readi 와 writei가 inode의 데이터를 쉽게 얻을 수 있습니다. readi (kernel/fs.c:472) 오프셋과 카운트가 파일 끝을 넘지 않도록 하는 것으로 시작합니다. 파일 끝을 넘은 읽기는 오류를 반환합니다 (kernel/fs.c:477-478) 파일 끝에서 시작하거나 끝을 넘는 읽기는 요청한 것보다 적은 바이트를 반환합니다 (kernel/fs.c:479-480). 메인 루프는 파일의 각 블록을 처리하여 버퍼에서 dst로 데이터를 복사합니다 (kernel/fs.c:482-494). writei (커널/fs.c:506) readi 와 동일 하지만 세 가지 예외가 있습니다. 파일 끝에서 시작하거나 끝을 넘는 쓰기는 최대 파일 크기 (kernel/fs.c:513-514) 까지 파일을 늘립니다 . 루프는 외부가 아닌 버퍼에 데이터를 복사합니다 (kernel/fs.c:36); 그리고 쓰기가 파일을 확장한 경우 writei는 해당 파일의 크기를 업데이트해야 합니다.

함수 stati (kernel/fs.c:458) inode 메타데이터를 stat 구조 에 복사합니다 . 이 구조는 stat 시스템 호출을 통해 사용자 프로그램에 노출됩니다 .

8.11 코드: 디렉토리 계층

디렉토리는 파일과 매우 비슷하게 내부적으로 구현됩니다. inode는 T_DIR 유형 이고 데이터는 디렉토리 항목 시퀀스입니다. 각 항목은 struct dirent (kernel/fs.h:56)입니다. 이름과 inode 번호를 포함합니다. 이름은 최대 DIRSIZ (14)자입니다. 이보다 짧으면 NULL(0) 바이트로 끝납니다. inode 번호가 0인 디렉토리 항목은 무료입니다.

dirlookup 함수 (kernel/fs.c:552) 주어진 이름을 가진 항목을 디렉토리에서 검색합니다.
하나를 찾으면 해당 inode에 대한 포인터를 잠금 해제된 상태로 반환하고, 호출자가 편집하고자 할 경우 디렉토리 내 항목의 바이트 오프셋으로 *poff를 설정합니다. dirlookup이 올바른 이름의 항목을 찾으면 *poff를 업데이트 하고 iget을 통해 얻은 잠금 해제된 inode를 반환합니다 . dirlookup 은 iget이 잠금 해제된 inode를 반환하는 이유입니다 . 호출자는 dp를 잠갔 으므로 조회가 현재 디렉토리의 별칭을 위한 것이라면 반환하기 전에 inode를 잠그려고 하면 dp를 다시 잠그고 교착 상태가 됩니다. (여러 프로세스가 관련된 더 복잡한,교착 상태 시나리오가 있으며 유일한 문제는 아닙니다.) 호출자는 dp를 잠금 해제한 다음 ip를 잠가 한 번에 하나의 잠금만 유지하도록 할 수 있습니다.

‥, 부모 디렉토리의 별칭; .

dirlink 함수 (kernel/fs.c:580) 주어진 이름과 in-ode 번호로 디렉토리 dp에 새 디렉토리 항목을 씁니다. 이름이 이미 있으면 dirlink는 오류를 반환합니다 (kernel/fs.c:586-590). 메인 루프는 할당되지 않은 항목을 찾는 디렉토리 항목을 읽습니다. 항목을 찾으면 루프를 일찍 중지합니다 (kernel/fs.c:563-564). 사용 가능한 항목의 오프셋 으로 설정된 off . 그렇지 않으면 루프는 dp->size 로 설정된 off 로 끝납니다. 어느 쪽이든 dirlink는 offset off 에 쓰기를 통해 디렉토리에 새 항목을 추가합니다 .

8.12 코드: 경로 이름

경로 이름 조회에는 각 경로 구성 요소마다 하나씩 dirlookup 에 대한 일련의 호출이 포함됩니다 . namei (kernel/fs.c:687) path 를 평가 하고 해당 inode를 반환합니다. nameiparent 함수는 변형입니다. 마지막 요소 앞에서 멈추고 부모 디렉터리의 inode를 반환하고 마지막 요소를 name에 복사합니다. 둘 다 일반화된 함수 namex를 호출하여 실제 작업을 수행합니다.

namex (커널/fs.c:652) 경로 평가가 시작되는 위치를 결정하는 것으로 시작합니다. 경로가 슬래시로 시작하면 평가는 루트에서 시작합니다. 그렇지 않으면 현재 디렉토리 (kernel/fs.c:656-659)에서 시작합니다.
그런 다음 skipelem을 사용하여 경로의 각 요소를 차례로 고려합니다 (kernel/fs.c:661). 루프의 각 반복은 현재 inode ip 에서 name을 찾아야 합니다 . 반복은 ip를 잠그고 디렉토리인지 확인하는 것으로 시작합니다. 그렇지 않으면 검색이 실패합니다 (kernel/fs.c:662-666). (ip 잠금 이 필요한 이유는 ip->type이 변경될 수 있기 때문이 아니라(변경될 수 없음) ilock이 실행되기 전까지 ip->type이 디스크에서 로드되었는지 보장할 수 없기 때문입니다.) 호출이 nameiparent 이고 이것이 마지막 경로 요소인 경우 nameiparent 의 정의에 따라 루프가 일찍 중지됩니다. 마지막 경로 요소는 이미 name 에 복사되었으므로 namex는 잠금 해제된 ip만 반환하면 됩니다 (kernel/fs.c:667-671) .

마지막으로 루프는 dirlookup을 사용하여 경로 요소를 찾고 ip = next (kernel/fs.c:672-677)를 설정하여 다음 반복을 준비합니다 . 루프가 경로 요소를 모두 소모하면 ip를 반환합니다.

namex 프로시저를 완료하는 데 오랜 시간이 걸릴 수 있습니다. 경로 이름에서 탐색한 디렉토리에 대한 inode와 디렉토리 블록을 읽기 위해 여러 디스크 작업이 필요할 수 있습니다(버퍼 캐시에 없는 경우). Xv6는 한 커널이 namex 를 호출하면

디스크 I/O에서 스레드가 차단되면 다른 경로명을 찾는 다른 커널 스레드가 동시에 진행될 수 있습니다. namex는 경로에 있는 각 디렉토리를 별도로 잠그므로 다른 디렉토리에서의 조회가 병렬로 진행될 수 있습니다.

이 동시성은 몇 가지 과제를 야기합니다. 예를 들어, 한 커널 스레드가 경로명을 찾는 동안 다른 커널 스레드는 디렉토리를 연결 해제하여 디렉토리 트리를 변경할 수 있습니다.
잠재적인 위험은 다른 커널 스레드에 의해 삭제된 디렉토리를 조회하고 해당 블록이 다른 디렉토리나 파일에 재사용될 수 있다는 것입니다.

Xv6는 이러한 경쟁을 피합니다. 예를 들어, namex 에서 dirlookup을 실행할 때, lookup 스레드는 디렉토리에 대한 잠금을 유지하고 dirlookup은 iget을 사용하여 얻은 inode를 반환합니다 . iget은 inode의 참조 카운트를 증가시킵니다. dirlookup 에서 inode를 수신한 후에야 namex는 디렉토리에 대한 잠금을 해제 합니다 . 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 ip 와 동일한 inode를 가리킵니다 . ip 에 대한 잠금을 해제하기 전에 next를 잠그면 교착 상태가 발생합니다. 이 교착 상태를 피하기 위해 namex는 next 에 대한 잠금을 얻기 전에 디렉토리를 잠금 해제합니다 . 여기서도 iget 과 ilock을 분리하는 것이 중요한 이유를 알 수 있습니다.

8.13 파일 기술자 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.

1장에서 살펴본 것처럼 Xv6는 각 프로세스에 열려 있는 파일이나 파일 설명자의 테이블을 제공합니다.
각 열려 있는 파일은 구조 파일 (kernel/file.h:1) 로 표현됩니다 . 이는 inode 또는 파이프를 감싸는 래퍼이며 I/O 오프셋이 더해집니다. open 에 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. 반면에 단일 열린 파일(동일한 구조체 파일) 은 한 프로세스의 파일 테이블과 여러 프로세스의 파일 테이블에 여러 번 나타날 수 있습니다. 이는 한 프로세스가 open 을 사용하여 파일을 연 다음 dup를 사용하여 별칭을 만들 거나 fork를 사용하여 자식과 공유하는 경우 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.

첫 번째 세 가지는 이제 익숙한 형식을 따릅니다. filealloc (kernel/file.c:30) 참조되지 않은 파일 (f->ref == 0)에 대한 파일 테이블을 스캔하고 새 참조를 반환합니다. filedup (kernel/file.c:48) 참조 카운트를 증가시키고 fileclose (kernel/file.c:60) 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 (kernel/file.c:88) inode에서만 허용되며 stati.fileread를 호출 합니다 .

그리고 filewrite는 작업이 open 모드에서 허용되는지 확인한 다음 파이프나 inode 구현으로 호출을 전달합니다. 파일이 inode를 나타내는 경우 fileread와 filewrite는 I/O 오프셋을 작업의 오프셋으로 사용한 다음 이를 진행합니다 (kernel/file.c:122-123) (kernel/file.c:153-154). 파이프에는 오프셋 개념이 없습니다. inode 함수는 호출자가 잠금을 처리해야 한다는 점을 기억하세요 (kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166). 인-오드 잠금은 읽기 및 쓰기 오프셋이 원자적으로 업데이트된다는 편리한 부작용이 있습니다. 즉, 같은 파일에 여러 번 쓰는 작업이 동시에 진행되어도 서로의 데이터를 덮어쓸 수 없지만 쓰기가 엇갈리게 될 수는 있습니다.

8.14 코드: 시스템 호출

하위 계층이 제공하는 기능을 사용하면 대부분 시스템 호출의 구현이 간단합니다((kernel/sysfile.c) 참조). 좀 더 자세히 살펴볼 만한 호출이 몇 가지 있습니다.

sys_link 및 sys_unlink 함수는 디렉토리를 편집하여 inode에 대한 참조를 만들거나 제거합니다. 이는 트랜잭션 사용의 힘을 보여주는 또 다른 좋은 예입니다. sys_link(kernel/sys- file.c:124) 두 개의 문자열(old 및 new) 인 인수를 가져오는 것으로 시작합니다 (kernel/sysfile.c:129). old가 존재하고 디렉토리가 아닌 경우 (kernel/sysfile.c:133-136) sys_link는 ip->nlink 카운트를 증가시킵니다 . 그런 다음 sys_link는 nameiparent를 호출하여 new (kernel/sysfile.c:149) 의 부모 디렉토리와 최종 경로 요소를 찾습니다. 그리고 old 의 inode (kernel/sys- file.c:152) 를 가리키는 새로운 디렉토리 항목을 생성합니다 . 새로운 부모 디렉토리는 존재해야 하며 기존 inode와 동일한 장치에 있어야 합니다. inode 번호는 단일 디스크에서만 고유한 의미를 갖습니다. 이와 같은 오류가 발생하면 sys_link는 뒤로 돌아가서 ip->nlink를 감소시켜야 합니다.

트랜잭션은 여러 디스크 블록을 업데이트해야 하기 때문에 구현을 간소화하지만, 우리는 그것을 하는 순서에 대해 걱정할 필요가 없습니다. 모두 성공하거나 아무것도 성공하지 못할 것입니다. 예를 들어, 트랜잭션이 없다면 링크를 만들기 전에 ip->nlink를 업데이트하면 파일 시스템이 일시적으로 안전하지 않은 상태가 되고, 그 사이에 충돌이 발생하면 대혼란이 발생할 수 있습니다. 트랜잭션이 있으면 이에 대해 걱정할 필요가 없습니다.

sys_link는 기존 inode에 대한 새 이름을 만듭니다. create 함수 (kernel/sysfile.c:246) 새 inode에 대한 새 이름을 만듭니다. 세 가지 파일 생성 시스템 호출을 일반화한 것입니다. O_CREATE 플래그 로 open하면 새 일반 파일이 만들어지고, mkdir은 새 디렉토리가 만들어지고, mkdev는 새 장치 파일이 만들어집니다. sys_link와 마찬가지로 create는 nameiparent를 호출하여 부모 디렉토리의 inode를 가져옵니다. 그런 다음 dirlookup을 호출하여 이름이 이미 있는지 확인합니다 (kernel/sysfile.c:256). 이름이 실제로 존재하는 경우 create의 동작은 어떤 시스템 호출에 사용되는지에 따라 달라집니다. open은 mkdir 및 mkdev 와 다른 의미를 갖습니다 . create가 open (type == T_FILE)을 대신하여 사용되고 존재하는 이름이 그 자체로 일반 파일인 경우 open은 이를 성공으로 처리하므로 create 도 마찬가지입니다 (kernel/sysfile.c:260). 그렇지 않으면 오류입니다 (kernel/sysfile.c:261-262). 이름이 아직 존재하지 않으면 create now는 ialloc (kernel/sysfile.c:265) 으로 새로운 inode를 할당합니다 . 새로운 inode가 디렉토리인 경우, create는 . 및 항목 으로 초기화합니다 . 마지막으로, 이제 데이터가 제대로 초기화되었으므로 create는 부모 디렉토리 (kernel/sysfile.c:278) 에 연결할 수 있습니다 . create는 sys_link 처럼 두 개의 inode 잠금을 동시에 보유합니다: ip 와 dp. inode ip가 새로 할당되었기 때문에 교착 상태가 발생할 가능성이 없습니다 : 시스템의 다른 프로세스는 ip 의 잠금을 보유한 다음 dp를 잠그려고 하지 않습니다 .

..

create를 사용하면 sys_open, sys_mkdir 및 sys_mknod를 쉽게 구현할 수 있습니다 . sys_open (kernel/sysfile.c:305) 가장 복잡한데, 새 파일을 만드는 것은 할 수 있는 일의 일부에 불과하기 때문입니다. open에 O_CREATE 플래그가 전달되면 create (kernel/sysfile.c:320) 를 호출합니다 . 그렇지 않으면 namei (kernel/sysfile.c:326)를 호출합니다 . create는 잠긴 inode를 반환하지만 namei는 반환하지 않으므로 sys_open은 inode 자체를 잠가야 합니다. 이것은 디렉토리가 쓰기가 아닌 읽기 전용으로만 열려 있는지 확인할 수 있는 편리한 장소를 제공합니다. inode가 어떤 식으로든 획득되었다고 가정하면 sys_open은 파일과 파일 설명자 (kernel/sysfile.c:344) 를 할당합니다. 그런 다음 파일 (kernel/sysfile.c:356- 361)을 채웁니다. 부분적으로 초기화된 파일은 현재 프로세스의 테이블에만 있기 때문에 다른 프로세스가 액세스할 수 없습니다.

7장에서는 파일 시스템이 생기기도 전에 파이프 구현을 살펴보았습니다. sys_pipe 함수는 파이프 쌍을 만드는 방법을 제공하여 해당 구현을 파일 시스템에 연결합니다.
인수는 두 개의 정수에 대한 공간을 포인터로, 여기에 두 개의 새로운 파일 기술자를 기록합니다.
그런 다음 파이프를 할당하고 파일 기술자를 설치합니다.

8.15 현실 세계

실제 운영 체제의 버퍼 캐시는 xv6의 버퍼 캐시보다 훨씬 더 복잡하지만, 캐싱과 디스크 액세스 동기화라는 두 가지 목적을 제공합니다. V6와 마찬가지로 Xv6의 버퍼 캐시는 간단한 LRU(최근에 가장 적게 사용된) 제거 정책을 사용합니다. 구현할 수 있는 훨씬 더 복잡한 정책이 많이 있으며, 각각은 일부 워크로드에는 좋지만 다른 워크로드에는 좋지 않습니다. 더 효율적인 LRU 캐시는 연결 목록을 제거하고 대신 조회에는 해시 테이블을 사용하고 LRU 제거에는 힙을 사용합니다. 최신 버퍼 캐시는 일반적으로 메모리 매핑 파일을 지원하기 위해 가상 메모리 시스템과 통합됩니다.

Xv6의 로깅 시스템은 비효율적입니다. 커밋은 파일 시스템 시스템 호출과 동시에 발생할 수 없습니다. 시스템은 블록의 몇 바이트만 변경되더라도 전체 블록을 로깅합니다. 한 번에 한 블록씩 동기 로그 쓰기를 수행하는데, 각각 전체 디스크 회전 시간이 필요할 가능성이 높습니다. 실제 로깅 시스템은 이러한 모든 문제를 해결합니다.

로깅은 크래시 복구를 제공하는 유일한 방법은 아닙니다. 초기 파일 시스템은 재부팅 중에 스캐빈저(예: UNIX fsck 프로그램)를 사용하여 모든 파일과 디렉토리, 블록 및 inode 사용 가능 목록을 검사하여 불일치를 찾고 해결했습니다. 스캐빈징은 대용량 파일 시스템의 경우 몇 시간이 걸릴 수 있으며, 원래 시스템 호출이 원자적이 되도록 불일치를 해결할 수 없는 상황이 있습니다. 로그에서 복구하는 것이 훨씬 빠르고 시스템 호출이 크래시 발생 시 원자적이 되도록 합니다.

Xv6는 초기 UNIX와 동일한 기본적인 디스크상 inode 및 디렉토리 레이아웃을 사용합니다. 이 방식은 수년에 걸쳐 놀라울 정도로 지속되었습니다. BSD의 UFS/FFS와 Linux의 ext2/ext3는 본질적으로 동일한 데이터 구조를 사용합니다. 파일 시스템 레이아웃에서 가장 비효율적인 부분은 디렉토리로, 각 조회 중에 모든 디스크 블록에 대한 선형 스캔이 필요합니다. 디렉토리가 몇 개의 디스크 블록에 불과할 때는 합리적이지만 많은 파일을 보관하는 디렉토리의 경우 비용이 많이 듭니다. Microsoft Windows의 NTFS, macOS의 HFS, Solaris의 ZFS 등을 예로 들면 디렉토리를 디스크상 균형 블록 트리로 구현합니다. 이는 복잡하지만 대수 시간 디렉토리 조회를 보장합니다.

Xv6는 디스크 오류에 대해 순진합니다. 디스크 작업이 실패하면 xv6가 패닉합니다. 이것이 합리적인지 여부

하드웨어에 따라 다릅니다. 운영 체제가 중복성을 사용하여 디스크 오류를 가리는 특수 하드웨어 위에 있는 경우 운영 체제가 오류를 매우 드물게 보기 때문에 당황하는 것이 괜찮을 수 있습니다. 반면 일반 디스크를 사용하는 운영 체제는 오류를 예상하고 더 우아하게 처리해야 하므로 한 파일의 블록이 손실되어도 나머지 파일 시스템의 사용에 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 소프트웨어로 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 inode 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 많은 다른 기능이 부족합니다. 예를 들어, 지원이 부족합니다.
스냅샷과 증분 백업을 위한 포트.

최신 Unix 시스템은 디스크 저장소와 동일한 시스템 호출을 사용하여 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 filewrite 의 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.

8.16 연습문제

1. 왜 balloc 에서 패닉을 하는가 ? xv6가 회복할 수 있을까?

2. ialloc 에서 패닉이 발생하는 이유는 ? xv6가 복구할 수 있나요?

3. filealloc 이 파일이 부족할 때 패닉을 일으키지 않는 이유는 무엇입니까 ? 이것이 더 흔하고 따라서 처리할 가치가 있는 이유는 무엇입니까?

4. sys_link 의 iunlock(ip) 와 dirlink 에 대한 호출 사이에 다른 프로세스가 ip 에 해당하는 파일을 연결 해제한다고 가정합니다 . 링크가 올바르게 생성될까요? 왜 그럴까요?

5. create는 성공하는 데 필요한 네 개의 함수 호출(하나는 ialloc 에 , 세 개는 dirlink에) 을 합니다 . 성공하지 못하면 create는 패닉을 호출합니다 . 왜 이게 용납될 수 있을까요? 왜 그 네 개의 호출 중 어느 것도 실패할 수 없을까요?

6. sys_chdir은 iput(cp->cwd) 전에 iunlock(ip)를 호출하는데 , 이는 cp->cwd를 잠그려고 시도할 수 있지만 , iput 이후까지 iunlock(ip)를 연기해도 교착 상태가 발생하지 않습니다. 왜 그럴까요?

7. lseek 시스템 호출을 구현합니다. lseek를 지원하려면 lseek가 f->ip->size 를 넘어선 경우 파일의 구멍을 0으로 채우도록 filewrite를 수정 해야 합니다 .
8. O_TRUNC 와 O_APPEND를 추가 하여 > 와 >> 연산자가 셸에서 작동하도록 합니다 .
9. 심볼릭 링크를 지원하도록 파일 시스템을 수정합니다.
10. 명명된 파이프를 지원하도록 파일 시스템을 수정합니다.
11. 메모리 맵 파일을 지원하도록 파일과 VM 시스템을 수정합니다.

9장

동시성 재검토

동시에 좋은 병렬 성능, 동시성에 대한 정확성, 이해하기 쉬운 코드를 얻는 것은 커널 설계에서 큰 과제입니다. 잠금을 직접 사용하는 것이 정확성을 향한 최선의 경로이지만 항상 가능한 것은 아닙니다. 이 장에서는 xv6가 복잡한 방식으로 잠금을 사용하도록 강요받는 예와 xv6가 잠금과 유사한 기술을 사용하지만 잠금은 사용하지 않는 예를 강조합니다.

9.1 잠금 패턴

캐시된 항목은 종종 잠그기 어려운 문제입니다. 예를 들어, 파일 시스템의 블록 캐시 (kernel /bio.c:26) 최대 NBUF 디스크 블록의 사본을 저장합니다. 주어진 디스크 블록이 캐시에 최대 하나의 사본을 갖는 것이 중요합니다. 그렇지 않으면 다른 프로세스가 동일해야 하는 블록의 다른 사본에 충돌하는 변경을 할 수 있습니다. 캐시된 각 블록은 struct buf (kernel/buf.h:1)에 저장됩니다. struct buf에는 한 번에 하나의 프로세스만 주어진 디스크 블록을 사용하도록 하는 데 도움이 되는 lock 필드가 있습니다. 그러나 그 잠금만으로는 충분하지 않습니다. 블록이 캐시에 전혀 없고 두 프로세스가 동시에 사용하려고 하는 경우는 어떨까요? struct buf가 없으므로(블록이 아직 캐시되지 않았기 때문에) 잠글 것이 없습니다. Xv6는 캐시된 블록의 ID 집합과 추가 잠금(bcache.lock)을 연결하여 이 상황을 처리합니다.

블록이 캐시되었는지 확인해야 하는 코드(예: bget (kernel/bio.c:59)) 또는 캐시된 블록 세트를 변경하려면 bcache.lock을 보유해야 합니다. 그런 다음 코드가 필요한 블록과 구조체 buf를 찾은 후 bcache.lock을 해제하고 특정 블록만 잠글 수 있습니다. 이는 일반적인 패턴입니다. 항목 세트에 대한 잠금 하나와 항목당 잠금 하나.

일반적으로 잠금을 획득하는 동일한 함수가 잠금을 해제합니다. 그러나 사물을 보는 더 정확한 방법은 잠금이 원자적으로 보여야 하는 시퀀스의 시작 부분에서 획득되고 해당 시퀀스가 끝날 때 해제된다는 것입니다. 시퀀스가 다른 함수나 다른 스레드 또는 다른 CPU에서 시작하고 끝나면 잠금 획득 및 해제는 동일한 작업을 수행해야 합니다. 잠금의 기능은 특정 에이전트에 데이터 조각을 고정하는 것이 아니라 다른 사용자를 기다리게 하는 것입니다. 한 가지 예는 yield (kernel/proc.c:503)의 획득입니다. 획득 프로세스가 아닌 스케줄러 스레드에서 릴리스됩니다. 또 다른 예는 ilock (kernel/fs.c:293)의 acquiresleep입니다. 이 코드는 디스크를 읽는 동안 종종 잠자코 있습니다. 다른 CPU에서 깨어날 수도 있는데, 이는 잠금이 다른 CPU에서 획득되고 해제될 수 있음을 의미합니다.

객체에 내장된 잠금으로 보호되는 객체를 해제하는 것은 섬세한 작업입니다. 잠금을 소유하는 것만으로는 해제가 정확하다는 것을 보장할 수 없기 때문입니다. 문제는 다른 스레드가 객체를 사용하기 위해 획득 대기 중일 때 발생합니다. 객체를 해제하면 내장된 잠금이 암묵적으로 해제되어 대기 스레드가 오작동하게 됩니다. 한 가지 해결책은 객체에 대한 참조가 몇 개 있는지 추적하여 마지막 참조가 사라질 때만 해제되도록 하는 것입니다. pipeclose (kernel/pipe.c:59)를 참조하세요. 예를 들어 pi->readopen과 pi->writeopen은 파이프에 해당 파이프를 참조하는 파일 설명자가 있는지 추적합니다.

일반적으로 관련 항목 집합에 대한 읽기 및 쓰기 시퀀스 주변에 잠금이 적용됩니다. 이 잠금은 다른 스레드가 완료된 업데이트 시퀀스만 볼 수 있도록 보장합니다(해당 스레드도 잠금을 사용하는 경우).
업데이트가 단일 공유 변수에 대한 간단한 쓰기인 상황은 어떨까요? 예를 들어, setkilled 및 killed (kernel/proc.c:607) p->killed 의 간단한 용도를 둘러싼 잠금 . 잠금이 없다면 한 스레드가 p->killed를 쓰는 동시에 다른 스레드가 읽을 수 있습니다. 이것은 경쟁이며 C 언어 사양에 따르면 경쟁은 정의되지 않은 동작을 생성하므로 프로그램이 충돌하거나 잘못된 결과를 생성할 수 있습니다.1 잠금은 경쟁을 방지하고 정의되지 않은 동작을 피합니다.

.

경쟁이 프로그램을 망가뜨릴 수 있는 한 가지 이유는 잠금이나 동등한 구조가 없다면 컴파일러가 원래 C 코드와는 상당히 다른 방식으로 메모리를 읽고 쓰는 기계 코드를 생성할 수 있기 때문입니다. 예를 들어, killed를 호출하는 스레드의 기계 코드는 p->killed를 레지스터에 복사하고 캐시된 값만 읽을 수 있습니다. 즉, 스레드는 p->killed에 대한 쓰기를 전혀 볼 수 없습니다 . 잠금은 이러한 캐싱을 방지합니다.

9.2 잠금 장치와 같은 패턴

많은 곳에서 xv6는 객체가 할당되었으며 해제되거나 재사용되어서는 안 된다는 것을 나타내기 위해 잠금과 같은 방식으로 참조 카운트 또는 플래그를 사용합니다. 프로세스의 p->state는 file, inode 및 buf 구조의 참조 카운트와 마찬가지로 이런 방식으로 작동합니다. 각 경우 잠금은 플래그 또는 참조 카운트를 보호하지만 객체가 조기에 해제되는 것을 방지하는 것은 후자입니다.

파일 시스템은 여러 프로세스가 가질 수 있는 일종의 공유 잠금으로 struct inode 참조 카운트를 사용하는데, 이는 코드가 일반 잠금을 사용할 경우 발생할 수 있는 교착 상태를 방지하기 위한 것입니다.
예를 들어, namex의 루프 (kernel/fs.c:652) 각 경로명 구성 요소로 명명된 디렉토리를 차례로 잠급니다. 그러나 namex는 루프가 끝날 때 각 잠금을 해제해야 합니다. 경로명에 점이 포함된 경우(예: a/./b) 여러 잠금을 유지한 경우 자체적으로 교착 상태가 될 수 있기 때문입니다. 디렉토리와 관련된 동시 조회와 교착 상태가 될 수도 있습니다. 8장에서 설명한 대로 솔루션은 루프가 참조 카운트를 증가시키지만 잠그지 않고 디렉토리 inode를 다음 반복으로 넘기는 것입니다. …

일부 데이터 항목은 다른 시간에 다른 메커니즘으로 보호되며, 때로는 명시적 잠금이 아닌 xv6 코드의 구조에 의해 암묵적으로 동시 액세스로부터 보호될 수 있습니다. 예를 들어, 물리적 페이지가 비어 있는 경우 kmem.lock (kernel/kalloc.c:24)으로 보호됩니다.

그런 다음 페이지가 파이프 (kernel/pipe.c:23)로 할당되면, 다른 잠금(임베디드 pi->lock)으로 보호됩니다. 페이지가 새 프로세스의 사용자 메모리에 재할당되면 보호되지 않습니다.

[1] https://en.cppreference.com/w/c/language/memory_model 의 "스레드 및 데이터 경쟁"

잠금에 의해서는 전혀 불가능합니다. 대신 할당자가 해당 페이지를 다른 프로세스에 제공하지 않는다는 사실(해제될 때까지)이 동시 액세스로부터 보호합니다. 새 프로세스의 메모리 소유권은 복잡합니다. 먼저 부모가 fork에서 메모리를 할당하고 조작한 다음 자식이 사용하고(자식이 종료된 후) 부모가 다시 메모리를 소유하고 kfree에 전달합니다. 여기에는 두 가지 교훈이 있습니다. 데이터 객체는 수명의 다른 시점에서 다른 방식으로 동시성으로부터 보호될 수 있으며 보호는 명시적 잠금이 아닌 암묵적 구조의 형태를 취할 수 있습니다.

마지막 잠금과 유사한 예는 mycpu()에 대한 호출 주위에서 인터럽트를 비활성화해야 하는 필요성입니다 (kernel /proc.c:83). 인터럽트를 비활성화하면 호출 코드가 타이머 인터럽트와 관련하여 원자적이 되어 컨텍스트 전환이 강제되고 결과적으로 프로세스가 다른 CPU로 이동하게 됩니다.

9.3 잠금 장치가 전혀 없습니다

xv6가 잠금이 전혀 없는 가변 데이터를 공유하는 곳이 몇 군데 있습니다. 하나는 스핀락 구현에 있지만, RISC-V 원자 명령어가 하드웨어에 구현된 잠금에 의존한다고 볼 수도 있습니다. 또 다른 하나는 main.c (kernel/main.c:7)의 시작 변수입니다. CPU 0이 xv6 초기화를 완료할 때까지 다른 CPU가 실행되지 않도록 방지하는 데 사용됩니다. volatile은 컴파일러가 실제로 로드 및 저장 명령어를 생성하도록 보장합니다.

Xv6에는 한 CPU 또는 스레드가 데이터를 쓰고 다른 CPU 또는 스레드가 데이터를 읽는 경우가 있지만, 해당 데이터를 보호하기 위한 특정 잠금은 없습니다. 예를 들어, fork에서 부모는 자식의 사용자 메모리 페이지를 쓰고 자식(다른 스레드, 아마도 다른 CPU에 있음)은 해당 페이지를 읽습니다. 잠금은 해당 페이지를 명시적으로 보호하지 않습니다. 이는 엄밀히 말해 잠금 문제는 아닙니다. 자식은 부모가 쓰기를 마친 후에야 실행을 시작하기 때문입니다. 이는 잠재적인 메모리 순서 지정 문제입니다(6장 참조). 메모리 장벽이 없으면 한 CPU가 다른 CPU의 쓰기를 볼 이유가 없기 때문입니다. 그러나 부모가 잠금을 해제하고 자식은 시작하면서 잠금을 획득하기 때문에 acquire 및 release의 메모리 장벽은 자식의 CPU가 부모의 쓰기를 볼 수 있도록 보장합니다.

9.4 병렬성

잠금은 주로 정확성을 위해 병렬성을 억제하는 것입니다. 성능도 중요하기 때문에 커널 설계자는 종종 정확성을 달성하고 병렬성을 허용하는 방식으로 잠금을 사용하는 방법에 대해 생각해야 합니다. xv6는 체계적으로 고성능을 위해 설계되지 않았지만, 어떤 xv6 작업이 병렬로 실행될 수 있고 어떤 작업이 잠금에서 충돌할 수 있는지 고려하는 것은 여전히 가치가 있습니다.

xv6의 파이프는 상당히 좋은 병렬성의 예입니다. 각 파이프에는 자체 잠금이 있으므로 다른 프로세스가 다른 CPU에서 다른 파이프를 병렬로 읽고 쓸 수 있습니다. 그러나 주어진 파이프의 경우 작성자와 판독자는 서로가 잠금을 해제할 때까지 기다려야 합니다. 동시에 같은 파이프를 읽고 쓸 수 없습니다. 또한 빈 파이프에서 읽기(또는 가득 찬 파이프에 쓰기)는 차단되어야 하지만 이는 잠금 체계 때문이 아닙니다.

컨텍스트 스위칭은 더 복잡한 예입니다. 각각 자체 CPU에서 실행되는 두 개의 커널 스레드는 동시에 yield, sched, swtch를 호출할 수 있으며 호출은 병렬로 실행됩니다.

각 스레드는 잠금을 하나 가지고 있지만, 서로 다른 잠금이기 때문에 서로를 기다릴 필요가 없습니다.
그러나 스케줄러에 들어가면 두 CPU가 RUNNABLE인 프로세스 테이블을 검색하는 동안 잠금에서 충돌할 수 있습니다. 즉, xv6는 컨텍스트 전환 중에 여러 CPU에서 성능 이점을 얻을 가능성이 있지만 가능한 만큼은 아닐 수 있습니다.

또 다른 예는 서로 다른 CPU의 서로 다른 프로세스에서 동시에 fork를 호출하는 것입니다. 호출은 pid_lock과 kmem.lock을 위해 서로를 기다려야 할 수 있으며, 프로세스 테이블에서 UNUSED 프로세스를 검색하는 데 필요한 프로세스별 잠금을 위해 기다려야 할 수 있습니다. 반면에 두 개의 fork 프로세스는 사용자 메모리 페이지를 복사하고 페이지 테이블 페이지를 완전히 병렬로 포맷할 수 있습니다.

위의 각 예에서 잠금 방식은 특정 경우에 병렬 성능을 희생합니다. 각 경우에 더 정교한 설계를 사용하여 더 많은 병렬성을 얻을 수 있습니다. 가치가 있는지 여부는 세부 사항에 따라 달라집니다. 관련 작업이 얼마나 자주 호출되는지, 코드가 경합 잠금을 유지하는 데 얼마나 많은 시간이 걸리는지, 동시에 충돌하는 작업을 실행할 수 있는 CPU 수, 코드의 다른 부분이 더 제한적인 병목 현상인지 여부입니다. 주어진 잠금 방식이 성능 문제를 일으킬 수 있는지 또는 새로운 설계가 상당히 더 나은지 추측하기 어려울 수 있으므로 현실적인 작업 부하에 대한 측정이 종종 필요합니다.

9.5 연습문제

1. xv6의 파이프 구현을 수정하여 다른 코어에서 동일한 파이프에 대한 읽기와 쓰기가 병렬로 진행될 수 있도록 합니다.

2. 여러 코어가 동시에 실행 가능한 프로세스를 찾을 때 잠금 경합을 줄이기 위해 xv6의 스케줄러()를 수정합니다.

3. xv6의 fork()에서 일부 직렬화를 제거합니다.

제10장

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/k-ernel 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.

서지

[1] Linux 일반 취약점 및 노출(CVE). https://cve.mitre.org/
cgi-bin/cvekey.cgi?keyword=linux.

[2] RISC-V 명령어 세트 매뉴얼 제1권: 권한이 없는 ISA. https://github.com/riscv/riscv-isa-manual/releases/download/
Ratified-IMAFDQC/riscv-spec-20191213.pdf , 2019 .

[3] RISC-V 명령어 세트 매뉴얼 제2권: 특권 아키텍처. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-
v1.12/riscv-privileged-20211203.pdf , 2021 .

[4] Hans-J Boehm. 스레드는 라이브러리로 구현될 수 없습니다. ACM PLDI 컨퍼런스, 2005.

[5] Edsger Dijkstra. 협력 순차 프로세스. https://www.cs.utexas.edu/
사용자/EWD/전사/EWD01xx/EWD123.html, 1965.

[6] Maurice Herlihy 및 Nir Shavit. 멀티프로세서 프로그래밍의 기술, 개정판 재인쇄.
2012년.

[7] Brian W. Kernighan. C 프로그래밍 언어. Prentice Hall 전문 기술
참고문헌, 2판, 1988.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe,
Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch 및 Simon Winwood. Sel4: OS 커널
의 공식 검증. ACM SIGOPS 22nd Symposium on Operating Systems Principles의 회의록, 207-220페이지, 2009년.

[9] Donald Knuth. 기본 알고리즘. 컴퓨터 프로그래밍의 예술. (2판),
제1권. 1997년.

[10] L Lamport. Dijkstra의 동시 프로그래밍 문제에 대한 새로운 솔루션. ACM 커뮤니케이션, 1974.

[11] John Lions. UNIX 6판에 대한 해설. Peer to Peer Communications, 2000.

[12] Paul E. Mckenney, Silas Boyd-wickizer 및 Jonathan Walpole. 리눅스에서의 RCU 사용
커널: 10년 후, 2013년.

[13] Martin Michael 및 Daniel Durich. NS16550A: UART 설계 및 응용 프로그램 고려 사항. http://bitsavers.trailing-edge.com/components/national/_appNotes/AN-0491.pdf , 1987.

[14] Aleph One. 재미와 이익을 위해 스택을 부수기. http://phrack.org/issues/49/14.html#기사.

[15] David Patterson 및 Andrew Waterman. RISC-V Reader: 개방형 아키텍처 Atlas. 스트로베리 캐년, 2017년.

[16] Dave Presotto, Rob Pike, Ken Thompson 및 Howard Trickey. 분산 시스템인 Plan 9. 1991년 봄 EurOpen 컨퍼런스 회의록, 43-50쪽, 1991년.

[17] Dennis M. Ritchie 및 Ken Thompson. UNIX 타임 공유 시스템. Commun. ACM, 17(7):365–375, 1974년 7월.

색인