# 3 Project 2: User Programs

이제 Pintos와 협력하여 인프라 및 스레드 패키지에 익숙해 졌으므로 이제는 사용자 프로그램을 실행할 수 있는 시스템 부분에 대한 작업을 시작할 시간입니다. 기본 코드는 이미 사용자 프로그램로드 및 실행을 지원하지만 I / O 또는 상호 작용이 가능하지 않습니다. 이 프로젝트에서는 프로그램이 시스템 호출을 통해 OS와 상호 작용할 수 있습니다.

이 할당을 위해 'userprog'디렉토리에서 작업하게 될 것이지만, Pintos의 거의 모든 다른 부분과도 상호 작용할 것입니다. 아래에서 관련 부분을 설명 할 것입니다.

프로젝트 1 제출물 위에 프로젝트 2를 만들거나 새로 시작할 수 있습니다. 이 과제에는 프로젝트 1의 코드가 필요하지 않습니다. "알람 시계"기능은 프로젝트 3과 4에서 유용 할 수 있지만 반드시 필요한 것은 아닙니다.

돌아가서 테스트 실행 방법을 다시 읽는 것이 유용 할 수 있습니다 (섹션 1.2.1 [테스트], 5 페이지 참조).

3.1 Background

지금까지 Pintos에서 실행 한 모든 코드는 운영 체제 커널의 일부였습니다. 예를 들어, 마지막 할당의 모든 테스트 코드가 시스템의 권한 있는 부분에 완전히 액세스 할 서있는 커널의 일부로 실행되었음을 의미합니다. 운영 체제 위에서 사용자 프로그램을 실행하기 시작하면 더 이상 사실이 아닙니다. 이 프로젝트는 그 결과를 다룬다.

우리는 한 번에 하나 이상의 프로세스가 실행되도록 허용합니다. 각 프로세스에는 하나의 스레드가 있습니다 (다중 스레드 프로세스는 지원되지 않음). 사용자 프로그램은 컴퓨터 전체를 가지고 있다고 착각하여 작성됩니다. 즉, 한 번에 여러 프로세스를 로드하고 실행할 때 이러한 환상을 유지하려면 메모리, 일정 및 기타 상태를 올바르게 관리해 야합니다.

이전 프로젝트에서는 테스트 코드를 직접 커널에 컴파일 했으므로 커널 내에 특정 기능 인터페이스를 요구해야 했습니다. 이제부터는 사용자 프로그램을 실행하여 운영 체제를 테스트 할 것입니다. 이것은 당신에게 훨씬 더 큰 자유를 줍니다. 사용자 프로그램 인터페이스가 여기에 설명 된 스펙을 충족하는지 확인해야 하지만, 제한 조건 하에서 원하는 대로 커널 코드를 재구성하거나 다시 작성할 수 있습니다.

3.1.1 Source Files

프로그래밍에 대한 개요를 얻는 가장 쉬운 방법은 작업 할 각 부분을 간단히 살펴 보는 것입니다. 'userprog'에서 적은 수의 파일을 찾을 수 있지만 여기서 일의 대부분은 다음과 같습니다.

**‘process.c’ ‘process.h’**

Loads ELF binaries and starts processes.

**‘pagedir.c’ ‘pagedir.h’**

80x86 하드웨어 페이지 테이블을 위한 간단한 관리자. 이 프로젝트에 대해이 코드를 수정하고 싶지는 않지만 일부 함수를 호출 할 수 있습니다. 자세한 내용은 섹션 4.1.2.3 [페이지 테이블], 페이지 40을 참조하십시오.

**‘syscall.c’ ‘syscall.h’**

사용자 프로세스가 커널 기능에 액세스하려고 할 때마다 시스템 호출을 호출합니다. 스켈레톤 시스템 콜 핸들러입니다. 현재는 메시지를 인쇄하고 사용자 프로세스를 종료합니다. 이 프로젝트의 2 부에서는 시스템 호출에 필요한 모든 작업을 수행하는 코드를 추가합니다.

**‘exception.c’ ‘exception.h’**

사용자 프로세스가 권한 또는 금지 된 작업을 수행하면 "예외"또는 "오류"로 커널에 트랩됩니다.이 파일은 예외를 처리합니다. 현재 모든 예외는 단순히 메시지를 인쇄하고 프로세스를 종료합니다. 일부는 아니지만 프로젝트 2의 솔루션은이 파일에서 page\_fault ()를 수정해야합니다.

**‘gdt.c’ ‘gdt.h’**

80x86은 세그먼트 화 된 아키텍처입니다. GDT (Global Descriptor Table)는 사용중인 세그먼트를 설명하는 테이블입니다. 이 파일들은 GDT를 설정합니다. 프로젝트의 파일을 수정할 필요가 없습니다. GDT 작동 방식에 관심이 있다면 코드를 읽을 수 있습니다.

**‘tss.c’ ‘tss.h’**

TSS (Task-State Segment)는 80x86 아키텍처 작업 전환에 사용됩니다. Pintos는 Linux처럼 사용자 프로세스가 인터럽트 처리기를 시작할 때 스택을 전환하기 위해서만 TSS를 사용합니다. 프로젝트의 파일을 수정할 필요가 없습니다. TSS의 작동 방식에 관심이 있다면 코드를 읽을 수 있습니다.

3.1.2 Using the File System

사용자 프로그램이 파일 시스템에서 로드되고 구현해야 하는 많은 시스템 호출이 파일 시스템을 처리하기 때문에 이 프로젝트의 파일 시스템 코드와 인터페이스해야 합니다. 그러나이 프로젝트의 초점은 파일 시스템이 아니므로 간단하지만 완전한 파일 시스템을 'filesys'디렉토리에 제공했습니다. 'filesys.h'와 'file.h'인터페이스를 살펴보고 파일 시스템을 사용하는 방법, 특히 많은 제한 사항을 이해해야 합니다.

이 프로젝트의 파일 시스템 코드를 수정할 필요가 없으므로 그렇게 하지 않는 것이 좋습니다. 파일 시스템에서 작업하면 이 프로젝트의 초점에서 벗어날 가능성이 있습니다.

파일 시스템 루틴을 적절히 사용하면 파일 시스템 구현을 향상시킬 때 프로젝트 4의 작업이 훨씬 쉬워집니다. 그때까지 다음과 같은 한계를 용인해야 합니다.

• 내부 동기화가 없습니다. 동시 액세스는 서로를 방해합니다. 한 번에 하나의 프로세스 만 파일 시스템 코드를 실행하도록 하려면 동기화를 사용해야합니다.

• 파일 크기는 생성시 고정됩니다. 루트 디렉토리는 파일로 표시되므로 생성 될 수 있는 파일의 수가 제한됩니다.

• 파일 데이터는 단일 범위로 할당됩니다. 즉, 단일 파일의 데이터는 디스크의 연속 섹터 범위를 차지해야 합니다. 따라서 외부 분열은 파일 시스템이 시간이 지남에 따라 심각한 문제가 될 수 있습니다.

( File data is allocated as a single extent, that is, data in a single file must occupy a contiguous range of sectors on disk. External fragmentation can therefore become a serious problem as a file system is used over time. )

• 하위 디렉토리가 없습니다.

• 파일 이름은 14 자로 제한됩니다.

• 작동 중 시스템 충돌로 인해 자동으로 복구 할 수 없는 방식으로 디스크가 손상 될 수 있습니다. 어쨌든 파일 시스템 복구 도구는 없습니다.

• 하나의 중요한 기능이 포함되어 있습니다: filesys\_remove ()에 대한 유닉스와 비슷한 의미가 구현됩니다. 즉, 파일이 제거 될 때 열려 있으면 해당 블록의 할당이 해제되지 않고 마지막으로 닫힐 때까지 열려 있는 모든 스레드에서 액세스 할 수 있습니다. 자세한 내용은 35 페이지의 [열린 파일 제거]를 참조하십시오.

파일 시스템 파티션이 있는 시뮬레이트 된 디스크를 생성 할 수 있어야합니다. *pintos-mkdisk* 프로그램은이 기능을 제공합니다. *'userprog / build'*디렉토리에서 ***pintos-mkdisk filesys.dsk - filesys-size = 2***를 실행하십시오. 이 명령은 2MB Pintos 파일 시스템 파티션이 들어있는 *'filesys.dsk'*라는 가상 디스크를 생성합니다. 그런 다음 커널의 명령 행에서 *'-f -q'*를 전달하여 파일 시스템 파티션을 포맷하십시오: ***pintos -f -q***. *'-f'*옵션은 파일 시스템을 포맷하고, *'-q'*는 포맷이 완료되자 마자 Pintos를 종료 하게합니다.

시뮬레이트 된 파일 시스템 안팎으로 파일을 복사하는 방법이 필요합니다. 핀토 *'-p'* ("put") 및 *-g* ("get") 옵션은이 작업을 수행합니다. *'file'*을 Pintos 파일 시스템에 복사하려면 ***'pintos -p file -- -q'***명령을 사용하십시오. *'-p'*는 시뮬레이트 된 커널이 아닌 'pintos' 스크립트 용이므로 '-'가 필요합니다. *'newname'*이라는 이름으로 Pintos 파일 시스템에 복사하려면 *'-a newname'*을 추가하십시오. ***'pintos -p file -a newname -- -q '***. VM에서 파일을 복사하는 명령은 비슷하지만 *'-p'*대신 *'-g'*를 사용하십시오.

덧붙여 말하자면, 이 명령은 커널의 명령 행에 특수 명령 추출 및 추가를 전달하고 특수 시뮬레이션 "스크래치" 파티션 간에 복사하여 작동합니다. 매우 궁금하다면 *'filesys / fsutil.c’뿐만* 아니라 pintos 스크립트를 보고 구현 세부 사항을 학습 할 수 있습니다.

다음은 파일 시스템 파티션으로 디스크를 생성하고, 파일 시스템을 포맷하고, 에코 프로그램을 새 디스크에 복사 한 다음, 인수 x를 전달하여 echo를 실행하는 방법을 요약 한 것입니다. (인수를 전달할 때까지 인수 전달이 작동하지 않습니다.) 이미 *'examples'*에 예제를 작성했으며 현재 디렉토리가 *'userprog / build'*라고 가정합니다.

3.1.3 How User Programs Work

Pintos는 메모리에 저장되고 구현 한 시스템 호출 만 사용하는 한 일반 C 프로그램을 실행할 수 있습니다. 특히 이 프로젝트에 필요한 시스템 호출이 없기 때문에 malloc ()을 구현할 수 없습니다. 커널은 스레드를 전환 할 때 프로세서의 부동 소수점 단위를 저장 및 복원하지 않기 때문에 Pintos는 부동 소수점 연산을 사용하는 프로그램도 실행할 수 없습니다.

'src / examples'디렉토리에는 몇 가지 샘플 사용자 프로그램이 들어 있습니다. 이 디렉토리의 'Makefile'은 제공된 예제를 컴파일하고, 편집하여 자신의 프로그램을 컴파일 할 수도 있습니다. 예제 프로그램 중 일부는 프로젝트 3 또는 4가 구현 된 후에 만 ​​작동합니다.

Pintos는 'userprog / process.c'에 제공된 로더를 사용하여 ELF 실행 파일을 로드 할 수 있습니다. ELF는 Linux, Solaris 및 기타 여러 운영 체제에서 오브젝트 파일, 공유 라이브러리 및 실행 파일에 사용되는 파일 형식입니다. 실제로 80x86 ELF 실행 파일을 출력하는 컴파일러와 링커를 사용하여 Pintos 용 프로그램을 생성 할 수 있습니다. (우리는 컴파일러와 링커를 제공했다.)

시뮬레이트 된 파일 시스템에 테스트 프로그램을 복사 할 때까지 Pintos는 유용한 작업을 수행 할 수 없습니다. 다양한 프로그램을 파일 시스템에 복사 할 때까지는 재미있는 일을 할 수 없습니다. 클린 레퍼런스 파일 시스템 디스크를 생성하고 디버깅하는 동안 때때로 일어날 수 있는 유용한 상태를 넘어서는 'filesys.dsk'를 휴지통에 복사 할 때마다 이를 복사 할 수 있습니다.

3.1.4 Virtual Memory Layout

Pintos의 가상 메모리는 사용자 가상 메모리와 커널 가상 메모리의 두 영역으로 구분됩니다. 사용자 가상 메모리의 범위는 가상 주소 0에서 PHYS\_BASE까지이며, 'threads / vaddr.h'에 정의되어 있으며 기본값은 0xc0000000 (3GB)입니다. 커널 가상 메모리는 PHYS\_BASE에서 4GB까지 가상 주소 공간의 나머지 부분을 차지합니다.

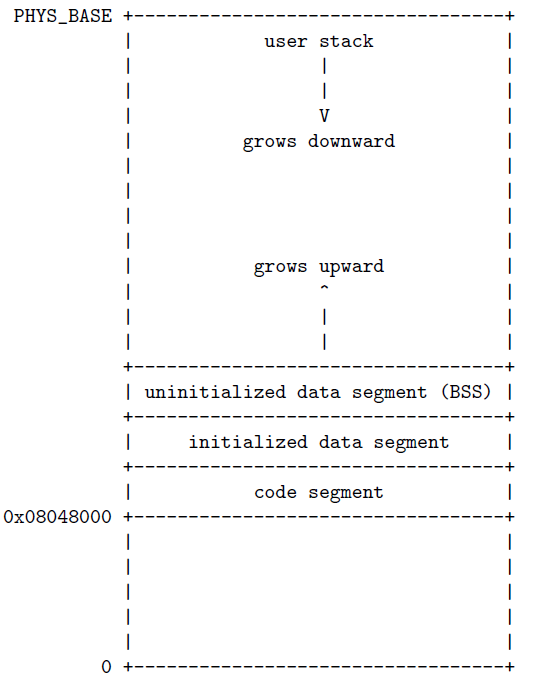
사용자 가상 메모리는 프로세스 별입니다. 커널이 한 프로세스에서 다른 프로세스로 전환 할 때 프로세서의 페이지 디렉토리 기본 레지스터 ('userprog / pagedir.c'의 pagedir\_activate () 참조)를 변경하여 사용자 가상 주소 공간을 전환합니다. struct thread는 프로세스의 페이지 테이블에 대한 포인터를 포함합니다.

커널 가상 메모리는 전역 적입니다. 실행중인 사용자 프로세스 또는 커널 스레드에 관계없이 항상 동일한 방식으로 매핑됩니다. Pintos에서 커널 가상 메모리는 PHYS\_BASE에서 시작하여 실제 메모리에 매핑됩니다. 즉 가상 주소 PHYS\_BASE는 실제 주소 0에 액세스하고 가상 주소 PHYS\_BASE + 0x1234는 실제 주소 0x1234에 액세스하며 컴퓨터의 실제 메모리 크기까지 증가합니다.

사용자 프로그램은 자체 사용자 가상 메모리에만 액세스 할 수 있습니다. 커널 가상 메모리에 액세스하려고 하면 'userprog / exception.c'에서 page\_fault ()가 처리하는 페이지 오류가 발생하고 프로세스가 종료됩니다. 커널 스레드는 커널 가상 메모리와 사용자 프로세스가 실행중인 경우 모두 실행중인 프로세스의 가상 메모리에 액세스 할 수 있습니다. 그러나 커널에서도 매핑 되지 않은 사용자 가상 주소에서 메모리에 액세스하려고 하면 페이지 오류가 발생합니다.

3.1.4.1 Typical Memory Layout

개념적으로, 각 프로세스는 자신의 사용자 가상 메모리를 자유롭게 레이아웃 할 수 있지만 자유롭게 선택할 수 있습니다. 실제로, 사용자 가상 메모리는 다음과 같이 배치됩니다.



이 프로젝트에서 사용자 스택의 크기는 고정되어 있지만 프로젝트 3에서는 확장이 허용됩니다. 전통적으로 초기화되지 않은 데이터 세그먼트의 크기는 시스템 호출로 조정할 수 있지만 이를 구현할 필요는 없습니다.

Pintos의 코드 세그먼트는 사용자 가상 주소 0x08084000에서 시작합니다. 주소 공간의 맨 아래에서 약 128MB입니다. 이 값은 [SysV-i386]에 명시되어 있으며 큰 의미는 없습니다.

링커는 다양한 프로그램 세그먼트의 이름과 위치를 알려주는 "링커 스크립트"의 지시에 따라 메모리에 사용자 프로그램의 레이아웃을 설정합니다. "infold"를 통해 액세스 할 수 있는 링커 설명서의 "스크립트"장을 참조하여 링커 스크립트에 대해 자세히 배울 수 있습니다.

특정 실행 파일의 레이아웃을 보려면 *'-p'*옵션과 함께 *objdump* (80x86) 또는 *i386-elf-objdump* (SPARC)를 실행하십시오.

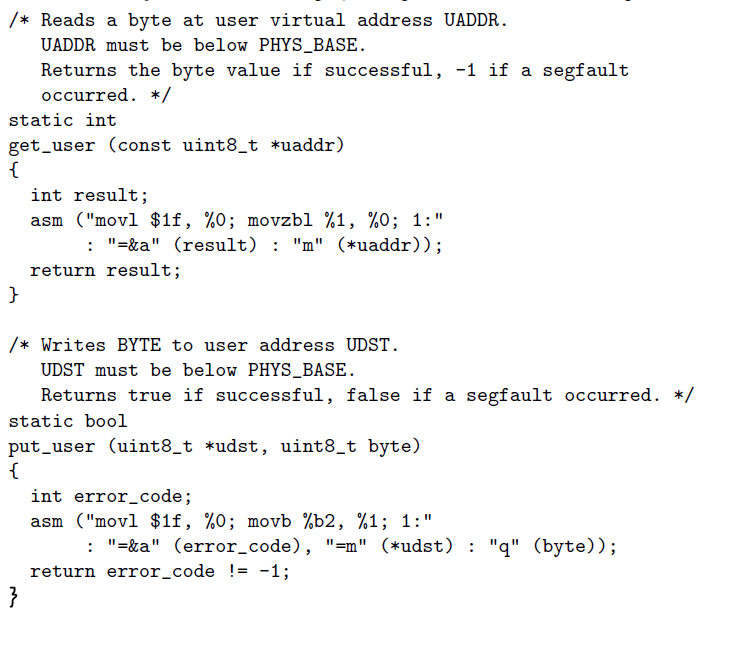
3.1.5 Accessing User Memory

시스템 호출의 일부로서 커널은 종종 사용자 프로그램이 제공하는 포인터를 통해 메모리에 액세스해야 합니다. 사용자가 널 포인터, 맵핑되지 않은 가상 메모리에 대한 포인터 또는 커널 가상 주소 공간 (PHYS\_BASE 이상)에 대한 포인터를 전달할 수 있기 때문에 커널은 매우 주의해야합니다. 이러한 유형의 모든 무효 포인터는 문제가 되는 프로세스를 종료하고 해당 리소스를 해제하여 커널이나 다른 실행중인 프로세스에 해를 끼치 지 않고 거부되어야 합니다.

이 작업을 올바르게 수행하는 데는 최소한 두 가지 합리적인 방법이 있습니다. 첫 번째 방법은 사용자 제공 포인터의 유효성을 확인한 다음 역 참조하는 것입니다. 이 경로를 선택하면 'userprog / pagedir.c'와 'threads / vaddr.h'에 있는 함수를 살펴볼 수 있습니다. 이것은 사용자 메모리 액세스를 처리하는 가장 간단한 방법입니다.

두 번째 방법은 사용자 포인터가 PHYS\_BASE 이하를 가리키는 지 확인한 다음 역 참조합니다. 잘못된 사용자 포인터는 'userprog / exception.c'에서 page\_fault ()에 대한 코드를 수정하여 처리 할 수 있는 "페이지 오류"를 발생시킵니다. 이 기술은 프로세서의 MMU를 이용하기 때문에 일반적으로 더 빠르기 때문에 실제 커널 (Linux 포함)에서 사용되는 경향이 있습니다.

두 경우 모두 리소스가 "누출되지"않도록 해야 합니다. 예를 들어, 시스템 호출이 malloc ()으로 잠금 또는 할당 된 메모리를 획득했다고 가정합니다. 이후에 유효하지 않은 사용자 포인터가 발생하면 잠금을 해제하거나 메모리 페이지를 확보해야 합니다. 참조를 취소하기 전에 사용자 포인터를 확인하도록 선택하면 이는 간단해야 합니다. 잘못된 포인터가 페이지 폴트를 일으키는 경우 처리하기가 더 어렵습니다. 메모리 액세스에서 오류 코드를 반환 할 방법이 없기 때문입니다. 그러므로, 후자의 기술을 시도하려는 사람들을 위해, 우리는 약간의 유용한 코드를 제공 할 것이다:



이 기능들 각각은 사용자 주소가 PHYS\_BASE보다 낮은 것으로 이미 확인되었다고 가정합니다. 또한 커널의 페이지 오류가 eax를 0xffffffff로 설정하고 이전 값을 eip로 복사하도록 page\_fault ()를 수정했다고 가정합니다.

3.2 Suggested Order of Implementation

• Argument passing (3.3.3 절 [인수 전달], 29 페이지 참조). 모든 사용자 프로그램은 인수 전달이 구현 될 때까지 즉시 오류 페이지를 표시합니다.

현재로서는, 당신은 단순히

\* esp = PHYS\_BASE;

를

\* esp = PHYS\_BASE – 12

로 바꿉니다; setup\_stack ()에 있습니다. 이름이 (null)로 인쇄 되더라도 인수를 검사하지 않는 테스트 프로그램에서는 작동합니다.

인수 전달을 구현할 때까지는 명령 줄 인수를 전달하지 않고 프로그램 만 실행해야 합니다. 인수를 프로그램에 전달하려고 시도하면 프로그램 이름에 인수가 포함되며 실패 할 수 있습니다.

• User memory access (3.1.5 [Accessing User Memory], 27 페이지 섹션 참조). 모든 시스템 호출은 사용자 메모리를 읽어야합니다. 사용자 메모리에 쓸 시스템 콜은 거의 없다.

• System call infrastructure (3.3.4 [시스템 호출], 29 페이지 섹션 참조). 사용자 스택에서 시스템 호출 번호를 읽고 이를 기반으로 처리기로 전달하기위한 충분한 코드를 구현하십시오.

• The exit system call. 정상적인 방법으로 끝내는 모든 사용자 프로그램은 종료됩니다. main () 호출에서 돌아 오는 프로그램조차도 간접적으로 종료됩니다 (*'lib / user / entry.c'*의 \_start () 참조).

• 시스템 콘솔 인 fd 1에 기록하기위한 쓰기 시스템 호출. 우리의 모든 테스트 프로그램은 콘솔에 쓰고 (printf ()의 사용자 프로세스 버전은 이런 식으로 구현된다), 쓰기가 가능할 때까지는 모두 오작동 할 것이다.

• 이제 process\_wait ()를 무한 루프 (영원히 기다리는 루프)로 변경하십시오. 제공된 구현은 즉시 반환되므로 Pintos는 프로세스가 실제로 실행되기 전에 꺼집니다. 결국에는 올바른 구현을 제공해야합니다.

(

* Argument passing (see Section 3.3.3 [Argument Passing], page 29). Every user program will page fault immediately until argument passing is implemented.

For now, you may simply wish to change

\*esp = PHYS\_BASE;

to

\*esp = PHYS\_BASE - 12; in setup\_stack(). That will work for any test program that doesn’t examine its arguments, although its name will be printed as (null).

Until you implement argument passing, you should only run programs without passing command-line arguments. Attempting to pass arguments to a program will include those arguments in the name of the program, which will probably fail.

* User memory access (see Section 3.1.5 [Accessing User Memory], page 27). All system calls need to read user memory. Few system calls need to write to user memory.
* System call infrastructure (see Section 3.3.4 [System Calls], page 29). Implement enough code to read the system call number from the user stack and dispatch to a handler based on it.
* The exit system call. Every user program that finishes in the normal way calls exit. Even a program that returns from main() calls exit indirectly (see \_start() in ‘lib/user/entry.c’).
* The write system call for writing to fd 1, the system console. All of our test programs write to the console (the user process version of printf() is implemented this way), so they will all malfunction until write is available.
* For now, change process\_wait() to an infinite loop (one that waits forever). The provided implementation returns immediately, so Pintos will power off before any processes actually get to run. You will eventually need to provide a correct implementation.

)

위를 구현 한 후에는 사용자 프로세스가 최소한으로 작동해야 합니다. 최소한 콘솔에 쓰고 올바르게 종료 할 수 있습니다. 그런 다음 구현을 구체화하여 일부 테스트가 통과하기 시작할 수 있습니다.

3.3 Requirements

3.3.1 Design Document

프로젝트를 시작하기 전에 프로젝트 2 디자인 문서 템플릿을 'pintos / src / userprog / DESIGNDOC'이름으로 소스 트리에 복사하고 입력해야 합니다. 작업을 시작하기 전에 디자인 문서 템플릿을 읽는 것이 좋습니다 프로젝트에. 가상 프로젝트와 함께 사용되는 샘플 디자인 문서는 부록 D [프로젝트 문서], 99 페이지를 참조하십시오.

3.3.2 Process Termination Messages

*exit* 또는 다른 이유로 인해 사용자 프로세스가 종료 될 때마다 *printf ( "% s : exit (% d) \ n", ...);*에 의해 인쇄 된 것처럼 포맷 된 프로세스의 이름과 종료 코드를 인쇄하십시오. 인쇄 된 이름은 process\_execute ()에 전달 된 전체 이름이어야 하며 명령 줄 인수는 생략해야 합니다. 사용자 프로세스가 아닌 커널 스레드가 종료되거나 시스템 호출 중지가 호출 될 때이 메시지를 인쇄하지 마십시오. 프로세스가 로드되지 않을 때 메시지는 선택적입니다.

이 외에도 Pintos가 이미 인쇄하지 않은 메시지는 인쇄하지 마십시오. 디버깅 중에 유용한 추가 메시지를 찾을 수는 있지만 점수 스크립트를 혼동하여 점수를 낮춥니다.

3.3.3 Argument Passing

현재 process\_execute ()는 인수를 새 프로세스로 전달하는 것을 지원하지 않습니다. process\_execute ()를 확장하여 이 기능을 구현하면 단순히 프로그램 파일 이름을 인수로 사용하는 대신 공백의 단어로 나눕니다. 첫 번째 단어는 프로그램 이름이고, 두 번째 단어는 첫 번째 인수입니다. 즉, process\_ execute ( "grep foo bar")는 grep이 두 개의 인수 foo와 bar를 전달하도록 실행해야 합니다.

명령 행에서 여러 개의 공백은 하나의 공백과 동일하므로 process\_execute ( "grep foo bar")는 원래 예제와 동일합니다. 명령 행 인수의 길이에 적당한 제한을 부과 할 수 있습니다. 예를 들어, 인수를 단일 페이지 (4KB)에 들어갈 수 있는 인수로 제한 할 수 있습니다. (pintos 유틸리티가 커널에 전달할 수 있는 명령 행 인수에는 128 바이트의 관련이 없습니다. ( There is an unrelated limit of 128 bytes on command-line arguments that the pintos utility can pass to the kernel.))

원하는 방식으로 인수 문자열을 구문 분석 할 수 있습니다. 길 잃은 경우 strtok\_r()을보고 *'lib / string.h'*에서 프로토 타입을 작성하고 *'lib / string.c'*에 철저한 주석으로 구현하십시오. 설명서 페이지를 보시면 (프롬프트에서 *man strtok\_r*을 실행하십시오) 자세한 내용을 볼 수 있습니다.

스택 설정 방법에 대한 자세한 내용은 섹션 3.5.1 [프로그램 시작 상세 정보], 36 페이지를 참조하십시오.

3.3.4 System Calls

*'userprog / syscall.c'*에 시스템 호출 핸들러를 구현하십시오. 우리가 제공하는 스켈레톤 구현은 프로세스를 종료하여 시스템 호출을 "처리"합니다. 시스템 호출 번호를 검색 한 다음 시스템 호출 인수를 검색하고 적절한 작업을 수행해야합니다.

다음 시스템 호출을 구현하십시오. 나열된 프로토 타입은 *'lib / user / syscall.h'*를 포함하는 사용자 프로그램에서 볼 수 있는 프로토 타입입니다. (이 헤더와 *'lib / user'*의 다른 모든 것들은 사용자 프로그램에서만 사용됩니다.) 각 시스템 호출에 대한 시스템 호출 번호는 *'lib / syscall-nr.h'*에 정의되어 있습니다:

**void halt (void) [시스템 콜]**

shutdown\_power\_off () ( 'devices / shutdown.h'에 선언되어 있음)를 호출하여 핀토스를 종료합니다. 가능한 교착 상태 상황 등에 대한 정보를 잃어 버리기 때문에 거의 사용하지 않아야 합니다.

**void exit (int status) [시스템 콜]**

현재 사용자 프로그램을 종료하고 상태를 커널에 반환합니다. 프로세스의 부모가 대기하는 경우 (아래 참조), 이것이 반환되는 상태입니다. 일반적으로 상태 0은 성공을 나타내고 0이 아닌 값은 오류를 나타냅니다.

**pid\_t exec (const char \*cmd\_line) [시스템 콜]**

cmd 행에 주어진 이름의 실행 파일을 실행하고 주어진 인수를 전달하고 새 프로세스의 프로그램 ID (pid)를 반환합니다. 어떤 이유로 든 프로그램을 로드하거나 실행할 수 없는 경우 pid -1을 반환해야 합니다. 그렇지 않으면 유효한 PID가 아니어야 합니다. 따라서 자식 프로세스가 실행 파일을 성공적으로 로드했는지 여부를 알 때까지 부모 프로세스는 exec에서 반환 할 수 없습니다. 이를 보장하려면 적절한 동기화를 사용해야합니다.

**int wait (pid\_t pid) [시스템 콜]**

자식 프로세스 pid를 기다리고 자식의 종료 상태를 가져옵니다.

pid가 아직 살아 있다면 종료 할 때까지 대기합니다. 그런 다음 pid가 종료 할 때 전달한 상태를 반환합니다. pid가 exit ()를 호출하지 않았지만 커널에 의해 종료 된 경우 (예 : 예외로 인해 종료 된 경우), wait (pid)는 -1을 반환해야 합니다. 부모 프로세스가 부모 호출 대기 시간까지 이미 종료 된 자식 프로세스를 기다리는 것은 매우 합법적입니다. 그러나 커널은 부모 프로세스가 자식 프로세스의 종료 상태를 검색하도록 하거나 커널 프로세스가 자식 프로세스를 종료했는지 확인해야합니다.

다음 조건 중 하나라도 해당되면 wait는 실패하고 -1을 즉시 반환해야 합니다:

1. pid는 호출 프로세스의 직접적인 하위를 참조하지 않습니다. pid는 호출 프로세스가 exec에 대한 성공적인 호출에서 반환 값으로 pid를 수신 한 경우에만 호출 프로세스의 직접적인 하위 프로세스입니다.

자식은 상속되지 않습니다. A가 자식 B를 생성하고 B가 자식 프로세스 C를 생성하면 A가 B가 죽더라도 C를 기다릴 수 없습니다. 프로세스 A에 의한 대기 (C) 호출은 실패해야 합니다. 마찬가지로 상위 프로세스가 종료되기 전에 고아 프로세스가 새 상위 프로세스에 할당되지 않습니다.

2. wait를 호출하는 프로세스는 이미 pid 대기를 호출했습니다. 즉, 프로세스는 주어진 자식을 최대 한 번 대기 할 수 있습니다.

프로세스는 임의의 수의 자식을 생성하고 순서에 관계없이 대기하며 일부 또는 모든 자식을 기다리지 않고 종료 할 수도 있습니다. 디자인은 대기가 발생할 수 있는 모든 방법을 고려해야합니다. struct thread를 포함하여 모든 프로세스의 자원은 부모가 이전에 기다렸는지 여부와 상관없이 그리고 그 부모가 부모 앞이나 뒤에 있는지에 관계없이 해제되어야 합니다.

초기 프로세스가 끝날 때까지 Pintos가 종료되지 않도록 해야합니다. 제공된 Pintos 코드는 main () ( *'threads / init.c'*)에서 process\_wait () ( *'userprog / process.c'*에 있음)를 호출하여 이 작업을 시도합니다. 함수의 맨 위에 있는 주석에 따라 process\_wait ()을 구현 한 다음 process\_wait ()에 대기 시스템 호출을 구현하는 것이 좋습니다.

이 시스템 호출을 구현하려면 나머지 모든 작업보다 훨씬 많은 작업이 필요합니다.

**bool create (const char \* file, unsigned initial\_size) [시스템 콜]**

초기 사이즈가 *initial\_size* bytes인 *file* 이라는 새 파일을 만듭니다. 성공하면 true를 반환하고, 그렇지 않으면 false를 반환합니다. 새 파일을 만들면 열리지 않습니다. 새 파일을 여는 것은 별도의 작업으로 개방 시스템 호출이 필요합니다.

**bool remove (const char \* file) [시스템 콜]**

*file*이라는 파일을 삭제합니다. 성공하면 true를 반환하고, 그렇지 않으면 false를 반환합니다. 파일이 열려 있는지 또는 닫혀 있는지에 관계없이 파일이 제거 될 수 있으며 열린 파일을 제거해도 파일이 닫히지 않습니다. 자세한 내용은 35 페이지, [Removing an Open File]를 참조하십시오.

**int open (const char \* file) [시스템 콜]**

*file*이라는 파일을 엽니다. "파일 기술자(file descriptor)"(fd)라고하는 음수가 아닌 정수 핸들을 반환합니다. 파일을 열 수 없으면 -1을 반환합니다.

fd 0 (STDIN\_FILENO)은 표준 입력이고, fd 1 (STDOUT\_FILENO)은 표준 출력입니다. The *open* system call will never return either of these file descriptors, which are valid as system call arguments only as explicitly described below.

각 프로세스에는 독립적인 file descriptors가 있습니다. file descriptors는 자식 프로세스에 상속되지 않습니다.

단일 프로세스 또는 다른 프로세스에 관계없이 단일 파일이 두 번 이상 열리면 각 열기는 새 file descriptors를 반환합니다. 단일 파일에 대한 서로 다른 file descriptors 는 닫을 별도의 호출에서 독립적으로 닫히고 파일 위치를 공유하지 않습니다.

**int filesize (int fd) [시스템 콜]**

fd로 여는 파일의 크기 (바이트)를 반환합니다.

**int read (int fd, void \* buffer, unsigned size) [시스템 콜]**

fd로 여는 파일로부터 size 바이트를 버퍼에 읽어들입니다. 실제로 읽힌 바이트 수를 반환합니다 (파일 끝에서 0). 파일을 읽을 수 없는 경우 -1 (파일 끝 이외의 조건으로 인해). Fd 0은 input\_getc ()를 사용하여 키보드에서 읽습니다.

**int write (int fd, const void \* buffer, 부호없는 크기) [시스템 콜]**

버퍼에서 열린 파일 fd로 size 바이트를 씁니다. 실제로 기입해진 바이트 수를 돌려줍니다. 몇 바이트를 기입 할 수 없는 경우는 사이즈보다 작을 수가 있습니다.

파일 끝을 지나면 정상적으로 파일이 확장되지만 파일 확장은 기본 파일 시스템에 의해 구현되지 않습니다. 예상되는 동작은 파일 끝까지 가능한 한 많은 바이트를 쓰고 기록 된 실제 수를 반환하거나 전혀 쓸 수 없는 경우 0을 반환하는 것입니다.

Fd 1은 콘솔에 기록합니다. 적어도 크기가 수백 바이트보다 크지 않은 한, 콘솔에 쓰는 코드는 putbuf ()에 대한 한 번의 호출로 모든 버퍼를 작성해야 합니다. (더 큰 버퍼를 분해하는 것이 합리적입니다.) 그렇지 않으면, 다른 프로세스에 의해 출력 된 텍스트 행이 콘솔에 인터리브되어 사람 판독기와 등급 매기기 스크립트를 혼동시킬 수 있습니다.

**void seek (int fd, unsigned position) [시스템 콜]**

열린 파일 fd에서 읽히거나 쓰여질 다음 바이트를 파일의 시작 부분에서 바이트 단위로 변경합니다. (따라서 0의 위치는 파일의 시작입니다.) 파일의 현재 끝을 지나친 탐색은 오류가 아닙니다. 나중의 읽기는 파일의 끝을 나타내는 0 바이트를 얻습니다. 이후의 쓰기는 파일을 확장하여 작성되지 않은 간격을 0으로 채 웁니다. 그러나 프로젝트 4가 완료 될 때까지 Pintos 파일의 길이는 고정되어 있으므로 파일 끝 부분을 기록하면 오류가 반환됩니다. 이러한 의미는 파일 시스템에서 구현되며 시스템 호출 구현 시 특별한 노력을 필요로 하지 않습니다.

**unsigned tell (int fd) [시스템 콜]**

열린 파일 fd에서 읽거나 쓰게 될 다음 바이트의 위치를 ​​파일의 시작 부분에서 바이트 단위로 반환합니다.

**void close (int fd) [시스템 콜]**

file descriptors fd를 닫습니다. 프로세스를 종료하거나 종료하면 각 프로세스에 대해이 함수를 호출 한 것처럼 모든 열린 파일 설명자가 암시 적으로 닫힙니다.

이 파일은 다른 syscall을 정의합니다. 지금은 무시하십시오. 프로젝트 3에서는 일부를 구현하고 프로젝트 4에서는 나머지를 구현하므로 확장 성을 염두에 두고 시스템을 설계해야 합니다.

syscall을 구현하려면 사용자 가상 주소 공간에서 데이터를 읽고 쓸 수 있는 방법을 제공해야합니다. 시스템 호출 번호는 사용자의 가상 주소 공간에 있는 사용자 스택에 있기 때문에 시스템 호출 번호를 얻기 전에 이 기능이 필요합니다. 이것은 약간 까다로울 수 있습니다 : 사용자가 유효하지 않은 포인터, 커널 메모리로의 포인터 또는 부분적으로 그 영역 중 하나에 있는 블록을 제공하면 어떻게 될까요? 사용자 프로세스를 종료하여 이러한 경우를 처리해야합니다. 다른 시스템 호출 기능을 구현하기 전에 이 코드를 작성하고 테스트 할 것을 권장합니다. 자세한 내용은 섹션 3.1.5 [사용자 메모리 액세스], 27 페이지를 참조하십시오.

원하는 수의 사용자 프로세스가 시스템 호출을 동시에 수행 할 수 있도록 시스템 호출을 동기화해야 합니다. 특히 'filesys'디렉토리에 제공된 파일 시스템 코드를 여러 스레드에서 동시에 호출하는 것은 안전하지 않습니다. 시스템 호출 구현은 파일 시스템 코드를 중요한 섹션으로 처리해야합니다. process\_execute ()도 파일에 액세스한다는 사실을 잊지 마십시오. 지금은 'filesys'디렉토리의 코드를 수정하지 않는 것이 좋습니다.

'lib / user / syscall.c'에서 각 시스템 호출에 대한 사용자 수준 함수를 제공했습니다. 이것은 사용자 프로세스가 C 프로그램에서 각 시스템 호출을 호출하는 f}을 제공합니다. 각각은 작은 인라인 어셈블리 코드를 사용하여 시스템 호출을 호출하고 (적절한 경우) 시스템 호출의 반환 값을 반환합니다.

이 부분을 끝내고 영원히 끝나면 핀토스는 종료해야 합니다. 사용자 프로그램이 할 수 있는 일은 OS가 충돌하거나 공포에 빠지거나 어설션에 실패하거나 기타 오작동을 일으키지 않아야 합니다. 이 점을 강조하는 것이 중요합니다. 우리의 테스트는 시스템 호출을 여러 가지 방법으로 해독하려고 시도 할 것입니다. 모든 사건을 생각하고 처리해야합니다. 사용자 프로그램이 OS를 중지시킬 수 있는 유일한 방법은 시스템 호출 중지를 호출하는 것입니다.

시스템 호출에 잘못된 인수가 전달되면 허용 가능한 옵션에는 오류 값 반환 (값을 반환하는 호출의 경우), 정의되지 않은 값 반환 또는 프로세스 종료가 포함됩니다.

시스템 호출의 작동 방법에 대한 자세한 내용은 섹션 3.5.2 [시스템 호출 세부 사항], 37 페이지를 참조하십시오.

3.3.5 Denying Writes to Executables

실행 파일로 사용중인 파일에 대한 쓰기를 거부하는 코드를 추가하십시오. 프로세스가 디스크에서 변경되는 코드를 실행하려고 하면 예기치 않은 결과로 인해 많은 OS가 이를 수행합니다. 일단 가상 메모리가 프로젝트 3에서 구현되면 이것은 특히 중요하지만, 지금도 해가 될 수는 없습니다.

file\_deny\_write ()를 사용하여 열린 파일에 쓰지 않도록 할 수 있습니다. 파일에 file\_allow\_write ()를 호출하면 파일이 다시 활성화됩니다 (다른 opener 가 파일 쓰기를 거부하지 않은 경우).

파일을 닫으면 쓰기도 다시 활성화됩니다. 따라서 프로세스의 실행 파일에 대한 쓰기를 거부하려면 프로세스가 실행 중일 때 열려 있어야합니다.

3.4 FAQ

3.4.1 Argument Passing FAQ

3.4.2 System Calls FAQ

3.5 80x86 Calling Convention

이 절에서는 Unix의 32 비트 80x86 구현에서 일반 함수 호출에 사용되는 규칙의 중요한 사항을 요약합니다. 간결성을 위해 일부 세부 사항은 생략되었습니다. 모든 세부 사항을 원하면 [SysV-i386]을 참조하십시오.

호출 규칙은 다음과 같이 작동합니다.

1. 호출자는 일반적으로 PUSH 어셈블리 언어 명령어를 사용하여 스택에서 함수의 각 인수를 하나씩 PUSH합니다. 인수는 오른쪽에서 왼쪽 순서로 푸시됩니다.

스택은 아래쪽으로 커집니다. 각 푸시는 스택 포인터를 감소시키고 C 식 표현 *'\* - sp = value'*와 같이 현재 가리키는 위치에 저장합니다.

2. 호출자가 스택에서 다음 명령어 (복귀 주소)의 주소를 밀어 넣고 호출 수신자의 첫 번째 명령어로 점프합니다. 단일 80x86 명령 인 CALL은 두 가지를 모두 수행합니다.

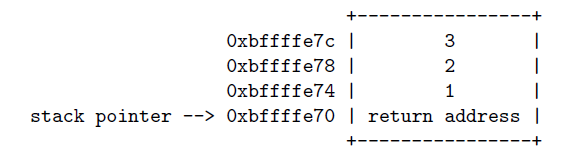
3. 피 호출자가 실행됩니다. 제어권을 얻으면 스택 포인터는 return address를 가리키고 첫 번째 인수는 그 바로 위에 있고 두 번째 인수는 첫 번째 인수 바로 위에 있는 식으로 계속됩니다.

4. 호출 수신자가 리턴 값을 갖는 경우, 레지스터 EAX에 이를 저장합니다.

5. 호출 수신자는 스택에서 반환 주소를 POPING하고 80x86 RET 명령어를 사용하여 지정된 주소로 점프하여 반환합니다.

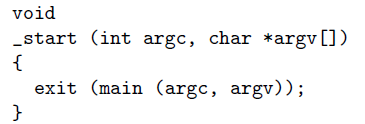
6. 호출자가 인수를 스택에서 POP합니다.

3 개의 int 인수를 취하는 f () 함수를 생각해보십시오. 이 다이어그램은 f ()가 f (1,2,3)로 호출 된 경우 위의 3 단계 시작 부분에서 호출 수신자가 보는 샘플 스택 프레임을 보여줍니다. 초기 스택 주소는 임의적입니다.



3.5.1 Program Startup Details

사용자 프로그램 용 Pintos C 라이브러리는 사용자 프로그램의 진입 점으로 *'lib / user / entry.c'*에 \_start ()를 지정합니다. 이 함수는 main ()이 다음을 반환하면 exit ()를 호출하는 main () 주위에 있습니다.



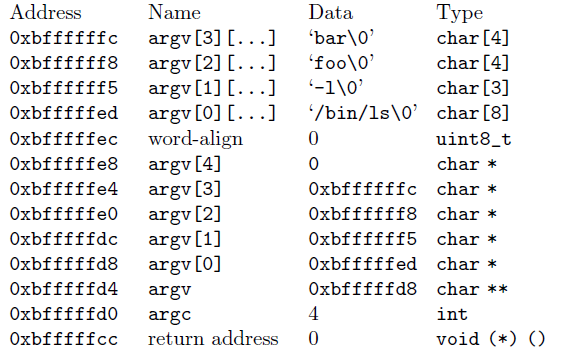
커널은 사용자 프로그램이 실행을 시작하기 전에 초기 함수에 대한 인수를 스택에 두어야합니다. 인수가 정상 호출 규칙과 같은 방식으로 전달됩니다 (섹션 3.5, 35 페이지 [80X86 호출 규칙] 참조).

***'/ bin / ls-l foobar'***명령 예제에 대한 인수를 처리하는 방법을 고려하십시오. 먼저 명령을 '/ bin / ls', '-l', 'foo', 'bar'로 분리하십시오. 단어를 스택 맨 위에 놓습니다. 그들은 포인터를 통해 참조되기 때문에 순서는 중요하지 않습니다.

그런 다음 스택에서 각 문자열의 주소와 널 포인터 센티널을 오른쪽에서 왼쪽 순서로 밀어 넣으십시오. 이것들은 argv의 요소입니다. null pointer sentinel은 C 표준에서 요구하는 대로 argv [argc]가 널 포인터입니다. 명령은 argv [0]이 가장 낮은 가상 주소에 있습니다. 워드 정렬 액세스는 정렬되지 않은 액세스보다 빠르기 때문에 최상의 성능을 위해 스택 포인터를 첫 번째 푸시 이전에 4의 배수로 반올림합니다.

그런 다음 argv (argv [0]의 주소)와 argc를 이 순서대로 PUSH 합니다. 마지막으로 가짜 "return address"를 PUSH합니다. 엔트리 함수가 돌아 오지는 않지만 스택 프레임은 다른 구조와 동일한 구조를 가져야합니다.

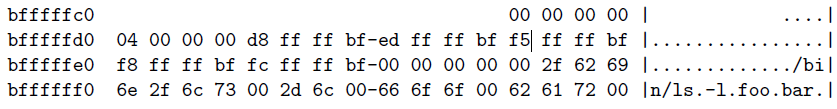
아래 표는 PHYS\_BASE가 0xc0000000이라고 가정 할 때 스택의 상태와 사용자 프로그램 시작 직전의 관련 레지스터를 보여줍니다.



이 예제에서 스택 포인터는 0xbfffffcc로 초기화됩니다.

위에 표시된 것처럼 코드는 가상 주소 PHYS\_BASE (*'threads / vaddr.h'*에 정의 됨) 바로 아래의 페이지에서 사용자 가상 주소 공간 맨 위에서 스택을 시작해야합니다.

'<stdio.h>'에 선언 된 비표준 hex\_dump() 함수는 인수 전달 코드를 디버깅하는 데 유용합니다. 위의 예에서 보여주는 내용은 다음과 같습니다.



3.5.2 System Call Details

첫 번째 프로젝트는 이미 운영 체제가 사용자 프로그램에서 제어권을 다시 얻을 수 있는 한 가지 방법, 즉 타이머 및 I / O 장치의 인터럽트를 처리했습니다. 이것은 "외부"인터럽트인데, 이는 CPU 외부의 엔티티로 인해 발생하기 때문입니다 (A.4.3 [외부 인터럽트 처리], 74 페이지 참조).

운영 체제는 프로그램 코드에서 발생하는 이벤트 인 소프트웨어 예외도 처리합니다 (섹션 A.4.2 [내부 인터럽트 처리], 73 페이지 참조). 페이지 오류 또는 0으로 나누기와 같은 오류가 있을 수 있습니다. 예외는 사용자 프로그램이 운영 체제에서 서비스 ( "시스템 호출")를 요청할 수 있는 수단이기도 합니다.

80x86 아키텍처에서 'int'명령어는 시스템 호출을 호출하는 데 가장 일반적으로 사용되는 방법입니다. 이 명령어는 다른 소프트웨어 예외와 동일한 방식으로 처리됩니다. Pintos에서 사용자 프로그램은 'int $ 0x30'을 호출하여 시스템 호출을 합니다. 시스템 호출 번호와 추가 인수는 인터럽트를 호출하기 전에 일반적인 방식으로 스택에 푸시 될 것으로 예상됩니다 (3.5 [80x86 호출 규칙], 35 페이지 참조).

따라서 시스템 호출 핸들러 syscall\_handler ()가 제어권을 얻으면 시스템 호출 번호는 호출자의 스택 포인터에서 32 비트 워드에 있고, 첫 번째 인수는 다음 상위 주소에서 32 비트 워드에 있는 식으로 계속됩니다. 호출자의 스택 포인터는 전달 된 structintr\_frame의 'esp'멤버로 syscall\_handler ()에 액세스 할 수 있습니다. (struct intr\_frame은 커널 스택에 있습니다.)

함수 반환 값에 대한 80x86 규칙은 EAX 레지스터에 값을 배치하는 것입니다. 값을 반환하는 시스템 호출은 struct int r\_ frame의 'eax'멤버를 수정하여 수행 할 수 있습니다.

시스템 호출을 구현하기 위해 많은 양의 반복적 인 코드를 작성하지 않도록 노력해야합니다. 각 시스템 호출 인수는 정수인지 포인터인지에 관계없이 스택에서 4 바이트를 차지합니다. 스택에서 각 시스템 호출의 인수를 검색하는 데 거의 동일한 코드를 작성하지 않으려면 이 기능을 활용할 수 있어야합니다.