**Pintos Project 1: User Program (1)**

담당 교수: 박성용 교수님

학번/이름: 20191286 김나현

개발 기간: ~2021.10.31

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

본 프로젝트는 System call, system call handler, argument passing, user stack 등이 구현되어 있지 않은 pintos에 앞서 언급한 것들을 포함한 OS의 많은 기능들을 구현하여 command를 하였을 때, 원하는 결과가 나오게 하도록 pintos 코드들을 수정하는 것입니다. 이때, command들은 커널 안에서만 부를 수 있는 I/O 등을 필요로 하므로 argument들을 user stack에 쌓고, 커널로 user stack의 주소를 넘겨줌으로써 해결할 수 있습니다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

‘echo x’와 같은 Command line을 입력 받으면 자동적으로 echo와 x로 구분해주는 것이 아니므로 파싱(parsing)하여 ‘echo’와 같은 파일 시스템을 열고 커널에서 해당 프로그램을 위한 기능을 수행하기 위해 command line을 통해 입력 받은 argument들을 stack에 쌓습니다.

1. User Memory Access

Kernel은 user program이 제공한 포인터를 통해서만 메모리에 접근이 가능하고, 유저는 null 포인터나 가상 메모리와 mapping되지 않는 포인터나, 커널의 가상 주소 공간(PHY\_BASE 위)을 가리키는 포인터를 넘겨줄 수 있기 때문에 커널은 유저가 제공한 포인터를 바로 사용하면 안 되고 유효한지 확인하는 과정을 반드시 거쳐야 합니다. 이때, 수행하는 과정을 통해 process를 끝내거나 리소스를 freeing함으로써 커널이나 수행 중인 다른 process의 손상을 막을 수 있습니다.

1. System Calls

User mode에서 돌고 있는 user program의 경우, 메모리나 디스크에 접근할 수 없고 I/O 등 또한 사용할 수 없기 때문에 userprog/syscall.c 안에 있는 system call handler를 구현하여 user program이 halt, exit, exec, wait, read(stdin), write(stdout)의 system call을 호출할 수 있도록 합니다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

커널은 유저 프로그램이 실행되기를 허락하기 전에 argument들을 쌓아야 하므로 command line을 띄어쓰기 단위로 나눠서 stack에 쌓아야 합니다. Argument를 쌓을 때는 PHY\_BASE에서 char[4], uint32\_t, char\*, char\*\*, int, void(\*)의 공간만큼을 뺀 후, 주소를 할당하므로 PHY\_BASE를 0xc0000000으로 가정하면 코드는 PHY\_BASE보다 아래에 있는 유저 가상 주소 공간은 존재하고, argument들은 PHY\_BASE에서 시작해서 주소 공간을 빼가며 저장하게 됩니다. 이때, 스택은 주소가 클수록 아래 있는 것을 의미하고 주소가 작을수록 위에 있는, 즉 top에 가깝다는 것을 의미합니다. 또한 stack에 argument들을 넣기 전에는 setup\_stack 함수를 호출하여 최소한의 stack page, 4KB를 할당해야 합니다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명

앞서 설명하였듯, 다른 process나 커널에 접근하는 포인터나 mapping 되지 않은 가상 메모리 영역을 가리키는 포인터, 또는 커널의 가상 주소 공간을 가리키는 포인터는 invalid한 pointer로 여겨집니다. 따라서, kernel이 잘못된 메모리에 access를 하지 않도록 방지하는 두 가지 방법이 있는데 하나는 유저가 제공한 포인터의 유효성을 검증하는 것으로, user memory access를 다루는 가장 쉬운 방법입니다. 다른 하나는 유저 포인터가 PHYS\_BASE 아래를 가리키는 것인지 판단하는 것으로, 유효하지 않은 유저 포인터는 page fault를 발생하므로 유효하지 않은 포인터라는 것을 알 수 있습니다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

Halt 시스템 콜은 shutdown\_power\_off() 함수를 호출합니다. Shutdown\_power\_off 함수는 Bochs나 QEMU에서 프로그램을 돌리고 있을 때 power을 끄는 역할을 합니다.

Exit 시스템 콜은 user program을 종료하고 커널에 프로그램의 상태를 반환합니다. 이때, thread\_name 함수를 이용하여 종료되는 프로그램의 이름과 상태를 print하고 thread\_exit 함수를 통해 thread를 종료합니다.

Exec 시스템 콜은 process\_execute 함수를 호출하여 child process를 생성하고, child process의 주소를 반환합니다. 이때, 에러가 발생해 child process가 만들어지지 않으면 page를 freeing해줍니다.

Wait 시스템 콜은 child process가 끝날 때까지 기다리고, child process가 끝나면 종료되는 상태를 반환합니다. 이때, child process가 없거나 넘겨주는 child process의 tid\_t가 유효하지 않거나 이미 성공적으로 종료된 경우, -1을 반환합니다.

Write 시스템 콜과 read 시스템 콜은 각각 stdout과 stdin에 대해서만 가능하게 하고, read 시스템 콜을 구현하기 위해 input\_getc 함수를 사용하고 write 시스템 콜을 구현하기 위해 putbuf를 사용합니다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

Argument passing 부분 구현: userprog 디렉토리 안에 있는 process.h와 process.c를 수정하여 load 함수 내부에서나 load 함수를 호출하는 start\_process 함수 내부에서 command line을 parsing하고 stack에 parsing한 argument들을 순서대로 쌓습니다.

User memory access 부분 구현 & System call handler 부분과 system call 부분 구현: userprog/syscall.c을 수정하여 system call 함수를 각각 구현하여 system call handler에서 system call을 호출할 수 있도록 작성해줍니다. 이때, system call 함수에 전달할 메모리 주소가 유효한지 확인하기 위해 is\_user\_vaddr 함수를 이용하여 memory access가 가능한 주소인지를 검증합니다.

Additional code 부분 구현: test case로 나와있는 함수 이외에도 추가적으로 수행할 수 있는 fibonacci 함수와 max\_of\_four\_int 함수를 구현하여 총 4개의 정수 argument에 대해서 additional이라는 명령어를 호출할 때, 두 가지 함수의 결과가 한 번에 출력할 수 있도록 구현합니다.

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

Userprog/process.c에 start\_process에서 load 함수를 호출하기 전에 parsing하여 load 함수를 호출할 때, file\_name에 argv[0]을 대입하여 load 함수가 argv[0]의 명령어 프로그램을 loading 할 수 있도록 하고 start\_process에서 load가 끝난 후에 stack이 setup 되면 stack에 argument들을 쌓을 수 있도록 구현해야 합니다.

System call handler에서 system call number에 따라 적절한 system call 함수를 호출하기 위해 src/lib/syscall-nr.h에 정의된 system call number을 이용하여 \*(uint32\_t\*)(f->esp)가 어떤 system call number냐에 따라 switch문으로 알맞은 system call 함수를 호출할 수 있게 합니다. 이때, userprog/syscall.c에서는 system call 함수도 구현해야 하고 system call handler에서 system call 함수를 호출할 때 정확한 data type으로 매개 변수를 넘겨줘야 합니다.

System call을 호출할 때, caller는 intr\_frame이라는 구조체에 있는 esp라는 포인터를 이용하여 stack에 system call number와 argument를 쌓고 system call handler에 전달하고 system call이 끝나면 그 결과를 intr\_frame의 구조체에 있는 eax라는 변수에 저장하여 반환합니다.

Thread 구조체에는 thread의 id를 저장하는 tid라는 변수가 있고, thread의 상태를 저장하는 status라는 변수, thread 이름을 저장하는, 크기가 16인 name이라는 문자열 배열 등이 있습니다. System call 중 exit 함수가 호출되면 thread가 종료되므로 thread\_name()이라는 함수를 사용하여 thread 구조체에 저장된 name 변수를 읽도록 구현하고, thread\_current() 함수로 구한 현재 thread 포인터를 이용해 exit\_status를 설정하고 마지막으로 thread\_exit() 함수를 이용해 thread를 종료합니다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 텍스트이(가) 표시된 사진

   자동 생성된 설명User Memory Access
2. 텍스트, 명함, 스크린샷이(가) 표시된 사진

   자동 생성된 설명System Calls

유저 프로그램에서 API를 호출하면 src/lib/user/syscall.c에서 system call number와 argument들을 caller의 stack에 쌓은 후, int $0x30이라는 instruction을 이용하여 system call을 이용하기 위해 interrupt가 발생시킵니다. Interrupt handler에서는 system call handler를 호출하고 system call handler로 가면 system call number에 따라 적절한 system call 함수를 호출하고 system call 함수가 반환하는 값을 eax에 저장하여 caller에게 전달합니다.

* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

우선, command line을 모두 입력 받아 file\_name이라는 배열 안에 넣은 후, args라는 char 자료형 이중 포인터를 이용해 첫 번째 argument는 args[0]에, 두 번째 argument는 args[1]에 순서대로 넣어줍니다. 이때, argument를 구분하는 공백이 여러 번 들어올 수 있기 때문에 이를 고려하여 코드를 작성해야 합니다. stack에 argument를 하나하나 저장할 것이기 때문에 각 argument를 넣은 stack의 주소를 가리키는 char\*\* 자료형 포인터 argv\_addr를 만들고 argv[0]이 저장된 주소를 가리키는 char\*\* 자료형 포인터 dargv\_addr를 생성합니다. 각 argument의 크기에 맞게 PHY\_BASE에서 시작하여 stack pointer를 감소해가며 argument의 문자열을 stack에 모두 저장한 후, stack 포인터를 4바이트를 맞춰주기 위해 word-align를 해줍니다. 이후, char\* 포인터가 가리키는 공간에 0을 저장하고 char\* 포인터가 가리키는 공간에 argv\_addr[n](n은 argument 개수에서 1을 뺀 정수를 의미함)~argv\_addr[0]을 순서대로 저장합니다. 이때, argv\_addr[0]가 저장되는 주소를 dargv\_addr에 저장하고 char\*\* 포인터가 가리키는 공간에 dargv\_addr을 저장합니다. 그 다음은, int 포인터가 가리키는 공간에 argument의 개수(n+1)를 저장합니다. Stack에 argument를 저장할 때, 저장하는 data의 type을 고려하여 스택 포인터의 자료형을 변환하며 저장해야 한다는 점을 유의해야 합니다.

1. User Memory Access

System call을 부르기 위해 user memory 주소를 전달할 때, 유효하지 않은 주소를 전달할 수 있기 때문에 system call handler에서는 is\_user\_vaddr 함수를 이용하여 system call 함수에 전달하는 주소가 유효한지를 판단해야 합니다. 이때, is\_user\_vaddr 함수가 0을 반환하면 유효하지 않다는 것이기 때문에 system call 함수 중 sys\_exit 함수를 이용하여 sys\_exit(-1)을 호출합니다. is\_user\_vaddr 함수를 사용하지 않으면 unvaild한 pointer에 대해서도 system call을 수행하기 때문에 에러가 발생할 수 있으므로 반드시 system call을 하기 전에 validity를 검증하는 과정을 거쳐야 합니다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

System call wait에서 process\_wait 함수를 부를 때, 자식 프로세스가 있다면 자식 프로세스가 종료되었는지를 확인해야 됩니다. 따라서, 자식 프로세스를 저장할 수 있는 list를 만들고 자식 프로세스가 죽으면 sema\_up을 하고 부모는 sema\_down 상태로 자식 프로세스의 종료를 기다리도록 구현하였습니다. 이때, 한 부모에 대해 여러 자식이 존재할 때 하나의 자식이 종료되면서 sema\_up을 실행하면 linked list로 구현된 부모의 자식 프로세스 list의 연결이 끊어질 수 있으므로 세마포어를 추가적으로 만들어 부모 프로세스가 list에서 종료되는 자식 프로세스의 정보를 제거한 후 자식 프로세스의 데이터를 free시켜 다른 자식 프로세스를 담은 linked list를 온전하게 유지할 수 있었습니다.

System call 중 exec 함수는 프로그램 이름을 전달 받아 해당 process를 실행하는 역할을 하므로 process\_execute 함수를 호출하도록 구현하였고, halt 함수에서는 실행 중인 process를 멈추기 위해 단순히 shutdown\_power\_off 함수를 호출하도록 구현하였습니다. 또한 read 함수와 write 함수는 각각 stdin과 stdout에 대해서만 수행이 되어야 하므로 fd을 확인하여 각각 0과 1일 때만 수행되고 그렇지 않을 때는 -1을 반환하도록 하였습니다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

src/examples 디렉토리에 additional.c를 만들고, Makefile을 수정하여 컴파일할 때, additional.c를 포함하여 컴파일하도록 하였습니다. 또한, system call 번호를 정의하기 위해 src/lib 디렉토리에 있는 syscall-nr.h에 SYS\_FIBO와 SYS\_MAX\_OF\_FOUR를 추가하였고, src/userprog 디렉토리에 있는 syscall.c와 syscall.h를 수정하여 system call handler에 sys\_fibonacci 함수와 sys\_max\_of\_four 함수를 구현하였고 max\_of\_four를 수행하기 위해 src/lib/user 디렉토리에 있는 syscall.c와 syscall.h를 수정하여 argument의 개수가 4개일 때 호출할 수 있는 syscall4를 추가적으로 정의하고 fibonacci 함수와 max\_of\_four 함수를 정의하였습니다.

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

텍스트이(가) 표시된 사진

자동 생성된 설명