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

담당 교수 : 김영재

조 / 조원 : 20191576 김윤성

개발 기간 : 2023.09.28 – 2023.10.03

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

Pintos에서의 명령어 처리를 구현한다. 이를 위해 Argument Parsing과 User Memory Access를 구현하고 이를 바탕으로 Argument Passing과 System Call을 구현해 명령어 처리를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

User stack에 유저가 입력한 명령어가 띄어쓰기 단위로 parsing 되어 순서대로 stack에 저장된다.

1. User Memory Access

User process에서의 유효하지 않은 메모리 접근을 차단하고 이러한 접근이 일어났을 때 해당 process를 종료한다.

1. System Calls

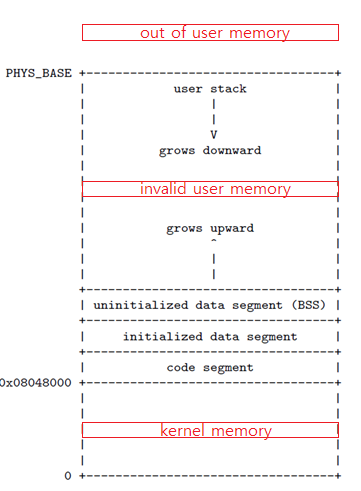
User process에서의 syscall에 의해 kernel side의 적절한 함수를 호출할 수 있다.

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

Pintos에서의 명령어를 처리하기 위해서는 virtual memory의 user stack에 명령어를 차례대로 쌓아야 한다. Stack의 top을 가리키는 esp 레지스터는 메모리 주소가 높은 곳에서 낮은 곳으로 나아가며 stack을 키움을 유의해야 한다. 이때 x86 calling convention을 지켜가며 명령어를 띄어쓰기 단위로 parsing 해 user stack에 쌓는다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명

User side에서의 memory access는 user side와 kernel side 중 user side로만 엄격히 제한된다. 따라서 user process는 kernel memory에 접근할 수 없다. 이와 달리 user memory 영역 안에 있지만 NULL 값을 갖거나 user memory 영역 밖으로 넘어가는 경우 또한 제한된다. 즉, kernel memory 영역으로의 접근, user memory 영역 안의 잘못된 접근, 그리고 user memory 영역 밖으로의 접근 세 가지 경우는 invalid memory access이다.



* + Invalid memory access를 어떻게 막을 것인지 설명

User process가 실행되는 동안 어떠한 memory으로의 접근이 이루어질 때는 항상 해당 memory가 유효한 접근인지 검사해야 한다. 먼저 kernel memory인지 판별한다. 만약 kernel memory가 아니면 이는 user memory이다. 다음으로 해당 user memory가 유효한memory인지 판별한다. 만약 어떤 memory access가 유효하지 않다고 판단했다면 해당 process를 종료해야 한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

User process는 kernel memory 영역에 일반적으로 접근할 수 없다. 하지만 어떤 경우에는 user process에서 kernel memory 영역에 반드시 접근해야 할 필요성이 생긴다. 이 경우 user process는 system call을 이용해 kernel memory 영역에 접근할 수 있다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

void halt (void)

- ctrl+c와 같은 방식으로 user process만이 아닌 pintos 자체를 종료하는 system call이다.

void exit (int status)

- user process 자기 자신을 종료하는 system call이다.

pid\_t exec (const char \*file)

- file에 해당하는 process를 실행하는 system call이다. 이때 user process의 child process가 하나 생성되고 user process는 parent process가 되어 child process가 모두 종료될 때까지 wait한다.

int wait (pid\_t)

- 해당 parent process의 모든 child process가 종료될 때까지 기다리는 system call이다.

int read (int fd, void \*buffer, unsigned length)

- file descriptor에 있는 내용을 length만큼 buffer에 읽는 system call이다.

int write (int fd, const void \*buffer, unsigned length)

- buffer에 있는 내용을 length만큼 file descriptor에 쓰는 system call이다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

User process에서 system call API를 호출하는 시점에 user side에서 kernel side로의 전환이 이루어진다. 이때 user code에서의 다음 instruction의 위치와 user code에서 사용했던 모든 변수들을 stack에 저장하고 kernel side로 전환된다. System call API에 의해 kernel side로 전환이 되면 user process는 호출한 시점에서 대기하고 kernel process가 실행된다. 그리고 kernel process가 모두 종료되면 stack에 저장되어 있던 user process의 변수들과 다음 instruction의 주소를 꺼내고 user process로 돌아와 user process의 나머지 코드를 실행한다.

텍스트, 라인, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

먼저 명령어를 parsing하여 user stack에 쌓아 유저의 명령어를 process에 전달한다. 그 이후 system call handler를 수정한다. 이때 잘못된 주소로의 접근을 막기 위해 user memory access를 검사하는 함수를 작성한다. 그리고 wait()과 exec()를 제외한 나머지 system call들을 처리한다. 해당 system call들을 모두 구현했으면 wait()과 exec()를 구현하기 위해 thread.h를 수정해 parent process가 자신의 child process들을 처리할 수 있도록 한다. Parent process가 child process를 저장할 수 있게 되면 wait()과 exec()를 모두 구현한다.

* 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)**
* Argument Passing

process.c의 load() 함수에서 setup\_stack() 함수에 의해 virtual memory가 생성이 된다. setup\_stack() 함수의 호출 이후 parsing된 argument를 user stack의 시작 지점인 PHYS\_BASE부터 4byte word-align에 맞추어 쌓는다.

* User Memory Access

Invalid memory access를 막기 위해 syscall.c에 접근하려는 주소값을 받아 해당 주소값이 유효한지 검사하는 check\_address() 함수를 새로 추가한다. 만약 유효하지 않은 주소값에 접근하려 하면 해당 process를 종료한다.

* System Calls

먼저 syscall.c의 syscall\_handler() 함수에서 상황에 따른 적절한 system call 함수를 적용하기 위해 <syscall-nr.h>에 있는 enum을 이용한다. 이때 switch 구문을 사용해 상황에 맞는 system call 함수를 호출한다. 특히 parent process는 child process가 끝날 때까지 wait 상태에 머무는데, 자신의 child precess들을 관리할 수 있도록 thread.h에 해당 변수들을 추가해야 한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

텍스트, 도표, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

도표, 텍스트, 라인이(가) 표시된 사진

자동 생성된 설명

1. System Calls

도표, 텍스트, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명

* 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

process.c의 load() 함수의 인자로 들어온 file\_name에 유저의 명령어가 들어 있다. 이file\_name을 parsing하여 새로운 임시 배열(char\* argv[128])에 저장한다. 이후 filesys\_open() 함수로 file\_name를 인자로 주어 실행하는데, parsing한 argv[0]으로 교체해야 유저가 원하는 명령어의 실행이 정상적으로 수행된다.

setup\_stack() 함수의 호출 이후 user stack의 top을 가리키는 esp 포인터를 적절히 줄여가며 argument를 쌓기 시작한다. 이때 4 byte씩 word-align하며 저장하는데, x86 calling convention에 맞게 PHYS\_BASE에서 4 떨어진 주소부터 argv[][ . . . ], padding bit, argv[], argv, argc, return address 순서대로 쌓아야 한다. 이때 목적지 주소값을 원하는 크기만큼 0으로 초기화하는 memset() 함수와 목적지 주소값을 소스값에서 원하는 크기만큼 복사하는 memcpy() 함수를 사용하면 편리하다. user stack에 argument를 모두 쌓고 해당 부분을 hex\_dump()로 출력하면 다음과 같이 parsing 된 argument가 출력되는 것을 확인할 수 있다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

Pintos에서 유효한 주소로의 접근인지 판단하는 함수들을 제공해 간단하게 구현할 수 있다. 먼저 kernel 혹은 user memory인지 판별하는 함수로 threads/vaddr.h의 is\_kernel\_vaddr() 함수와 is\_user\_vaddr() 함수가 있다. 해당 함수들은 인자로 들어온 주소값이 user stack의 최상단과 비교해 kernel memory인지 user memory인지 알려준다. 다음으로 어떤 주소값이 user memory에 위치한 주소이지만 mapping이 되어 있는지 판별하는 함수로 userprog/pagedir.c의 pagedir\_get\_page() 함수가 있다. 해당 함수는 현재 process가 사용하고 있는 page를 불러와 인자로 들어온 주소값이 어디에 위치하는지 알려준다. 만약 인자로 들어온 주소값이 존재하지 않는다면 NULL을 리턴한다.

syscall.c에 check\_address() 함수를 만들어 유효한 주소로의 접근인지 검사한다. 이 함수에서 위의 검사를 통해 잘못된 주소로의 접근이 시도되었다고 판단하면 해당 process를 exit(-1)로 종료한다.

1. System Calls

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

먼저 start\_process() 함수에서 어셈블리 코드를 이용해 다음 instruction이 가리키는 곳으로 jump한다. 그럼 system\_handler() 함수가 intr\_frame의 esp가 가리키는 syscall number를 분석해 그에 따른 syscall 함수를 호출해준다.

void halt (void)

- pintos에서 제공하는 shutdown\_power\_off() 함수를 사용하면 쉽게 구현할 수 있다.

void exit (int status)

- process가 종료될 때 해당 process를 실행한 thread의 이름과 status를 출력한다. 이후 해당 process를 실행한 thread의 status를 업데이트한 후 thread\_exit() 함수를 호출해 thread의 종료를 유도한다.

pid\_t exec (const char \*file)

- process\_execute() 함수를 호출해 새로운 child process의 생성을 유도한다. 이때 process\_execute() 함수가 끝날 때까지 exec() 함수가 리턴을 해서는 안 되며 child process의 process\_execute() 함수의 리턴값을 parent process에게 넘겨주어야 한다.

int wait (pid\_t)

- process\_wait() 함수를 호출해 parent process의 wait을 유도한다. 이때 parent process는 자신의 child process들이 모두 종료될 때까지 process\_wait() 함수에서 기다린다. process\_wait() 함수 내부에서는 parent process를 실행한 thread가 child process들을 저장하고 있는데, 여기에서 semaphore를 이용해 각각의 child process가 종료될 때까지 기다린다. 만약 child process들이 모두 종료되었다면 process\_wait() 함수를 종료하고 parent process는 wait 상태에서 빠져나오게 된다.

int read (int fd, void \*buffer, unsigned length)

- file descriptor가 0이면 STDIN, 1이면 STDOUT, 2이면 STDERR이다. 이번 프로젝트의 read() 함수는 file descriptor가 0일 경우에만 구현한다. 만약 file descriptor가 0이면 인자로 들어온 buffer에 STDIN에서 하나씩 읽어 size만큼의 문자를 저장한다.

int write (int fd, const void \*buffer, unsigned length)

- read() 함수와 마찬가지로 file descriptor가 0이면 STDIN, 1이면 STDOUT, 2이면 STDERR이다. 이번 프로젝트의 write() 함수는 file descriptor가 1일 경우에만 구현한다. 만약 file descriptor가 1이면 인자로 들어온 buffer에서 size만큼 문자를 읽어 STDOUT에 출력한다.

1. Additional System calls

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

새로운 system call을 처리하기 위해 syscall-nr.h에 SYS\_FIBONACCI와 SYS\_MAX\_OF\_FOUR를 추가한다. 이후 syscall\_handler() 함수에서 해당 system call number를 처리할 수 있게 switch 구문을 수정한다. 또한 syscall.h에 새로운 system call을 처리하기 위해 int fibonnacci() 함수와 int max\_of\_four\_int() 함수의 선언을 추가한다.

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

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명