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

**20191613**

**컴퓨터 공학과**

**윤상현**

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

시스템콜 핸들러를 작성하여, 정상적으로 exit이 되도록 구현

스택 및 파싱과정을 구현하여 argument passing   
유저가 커널 혹은 널포인터를 참조하지 못하게 하기

시스템 콜 핸들러를 작성하여 exec, exit, write등이 올바르게 작동하도록 구현

Fibonacci, max\_of\_four\_int 시스템 콜 구현

Additional 시스템 api 구현

을 할 예정입니다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

유저가 핀토스에 쓰고 싶은 파일명, 들어가는 인자 등을 파싱하여 스택에 쌓아놓고, 그것들을 통해 디스크에서 복사해온 파일들을 실행시킬 수 있게 됩니다.

1. User Memory Access

직접적으로 메모리를 사용할 때는, 프로세스들이나 커널과 충돌이 발생할 수 있는데, 가상 메모리를 사용함으로써, 이러한 문제를 해결할 수 있습니다.

또한, 미리 포인터의 정보를 확인하여 null pointer 나 unmapped 메모리를 사용하는 것을 방지할 수도 있습니다.

1. System Calls

위의 가상메모리 개념에 의하면, 각자의 유저프로그램은 커널의 핵심함수나 기능에도 접근할 수 없습니다. 그러한 부분을 해결하기위해서 커널에 접근가능한 커널 모드로 들어가게 해 주는 중간 단계가 system call입니다. 이를 통하여 프로세스 스케쥴, 파일 시스템 접근, 커널 메모리 접근 등을 가능케 합니다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명  
    핀토스에서 유저가 파일이름, 인수 등의 인자를 입력하면, agrv배열이 run\_action, Run\_task, process\_execute로 차례로 전달됩니다. 그 후, start\_process가 호출되고, load함수가 호출되는데, 여기서 스택에 argument가 쌓이게 됩니다. 스택의 top에 해당하는 esp를 이용하여, 해당 함수에서 따로 파싱 처리한 argument들을 스택에 쌓게 됩니다.
* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명

핀토스에서는 커널 메모리와 프로세스별 유저 메모리로 구분하여 사용합니다.

이때, 메모리를 가상화 하지 않고, 직접적으로 사용할 경우 서로의 영역을 침범하여, 메모리가 오염되거나 심하면 os가 손상될 수 있습니다.

또는 단순히 널포인터 참조또한 문제를 일으킬 수 있습니다.

* + Invalid memory access를 어떻게 막을 것인지 설명

가상 메모리를 사용하여, 각 프로세스가 전체 메모리를 사용하는 것과 같은 효과를 주게 합니다. 또한, is\_user\_vaddr(), is\_kernel\_vaddr()등을 사용하여, 미리 이 메모리가 unmapped인지 , 커널인지 null 인지등을 미리 파악하는 방법 또한 사용할 수 있습니다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

위의 가상 메모리를 사용하고, 커널 메모리와 유저 메모리를 분리하는 등의 이유로, 유저가 커널의 api, 메모리 등에 접근할 수 있도록 따로 중간단계로써 시스템 콜이 필요합니다.

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

Halt:

핀토스를 종료합니다 프로세스를 kill하게 되므로 많이 쓰이진 않는다고 합니다.

Exit:

현재 프로세스를 종료합니다. 정상적이지 못한 경우에는 보통 -1을 리턴하게 됩니다. 자식 프로세스의 exit후에는 그 exit\_status를 부모에게 전달하게 됩니다.

Exec:

새로운 프로세스를 실행하게됩니다. 현재 돌아가는 중인 프로세스의 자식 프로세스를 새로 만들게 되고, 그 pid를 부모 프로세스에게 줍니다.

Wait:

자식 프로세스중에 주어진 pid와 같은 pid를 가진 프로세스가 있다면, sema를 이용하여 그 프로세스가 종료될때까지 기다리다가, 그 exit\_status를 받아서 부모 프로세스에게 주는 역할을 합니다. 주로 자식 프로세스에게 컨트롤을 넘겨주는 기능을 합니다.

Read:

정보를 읽어들이는 기능으로 input\_getc로 작성했습니다. 지금은 stdin(0)만 읽어들일 수 있습니다.

Write:

정보를 putbuf로 작성합니다. 지금은 stdin(1)에만 쓸 수 있습니다.

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

유저레벨에서 스택에 정보를 쌓은 후에, 시스템 콜 api를 호출하게 되면, syscall1,2,3.. 이 호출되고, intr\_handler를 거쳐서, infr\_frame\* f가 핸들러로 가게 됩니다. 거기서 유효한 메모리인지, 등을 검사하고, esp의 가장 위에 있는 시스템콜 번호로 어떤 시스템을 콜했는지 알게 됩니다. 그 후, 스택의 쌓인 정보들을 기반으로 kernel api 가 실행되고, 그것의 결과값이 위의 intr\_frame의 멤버인 eax에 저장되어서 다시 유저에게로 결과값을 알려주게 됩니다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

먼저 argument passing을 위해 파싱과 스택을 먼저 구현한 이후,

비교적으로 간단한 halt, exit 을 구현할 것입니다.

그후에, 유저메모리 검사하는 부분을 작성하고 Exec, wait, read, write 시스템콜을 작성할 것입니다.

* 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 load함수 내부에서 file\_name을 파싱하고, esp스택 포인터를 이용하여 파싱한 결과를 쌓을 예정입니다.

User\_memory: exception.c의 page\_fault함수에서, 1차적으로 user의 커널 참조를 감지하여서 막고, syscall.c 내부에 따로 적절한 포인터인지 검사하는 cad(\*ptr)함수를 작성하여 검사할 것입니다.

Systemcall: syscall.c에서 스위치 구문을 사용하여 시스템콜 핸들러를 작성하고, exit, halt, wait, exec등의 함수를 작성할 것입니다. Wait와 exec구현 과정에서 thread.h의 thread자료구조에 semaphore 변수와 exit\_status 변수를 추가하였습니다. 그리고, process\_wait, process\_execute함수에 semaphore를 사용한 synchronize를 구현하였습니다.

1. **연구 결과**
   1. **Flow Chart**

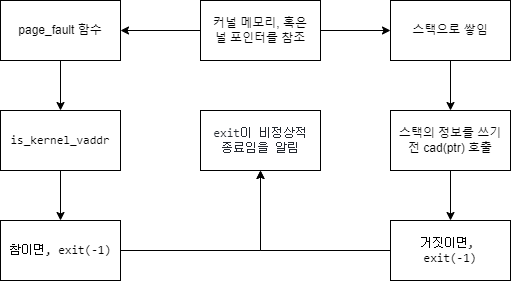
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

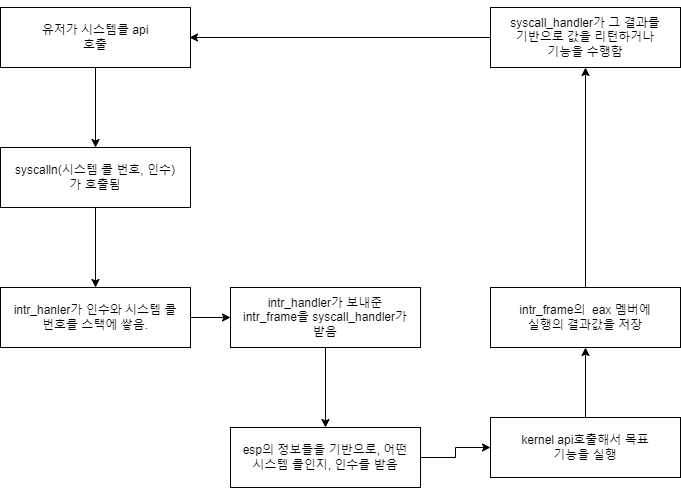
텍스트이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access



1. System Calls



* 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

먼저, file\_name을 받은 후에, 문자열을 수정하기 위하여, char\* fn\_copy에 palloc을 사용하여 페이지를 할당해준후, file\_name의 내용을 복사해 주었습니다. 그 후, for문을 돌려서, 스페이스로 구분된 각 문자열이 몇 개인지 세준 이후에 strtok\_k를 이용해서 잘라주었습니다. 그 후, 주어진 \*\*esp를 이용해서, 인수, 워드, 인수의 주소, 인수의 주소의 주소(esp상), return address를 넣어주는 것으로 구현을 하였습니다.

1. User Memory Access

먼저, exception.c의 page\_fault함수 내부에서 user의 접근인지 감지하는 부분이 있어서 user일 경우 그냥 exit(-1)하도록 하였습니다.

그 후에는 syscall.c내부에 cad(\*ptr) 함수를 만들어서, 이 포인터가 커널이거나 null인지 검사하도록 하였습니다.

1. System Calls

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

Halt:

shutdown\_power\_off를 호출하였습니다.

Exit:

스레드 이름과 종료상태를 출력하고, thread자료구조에 추가한 exit\_status를 업데이트 해줍니다. 그러면 나중에 부모가 방금 추가한 exit\_status를 가져가서 정상적인 종료였는지 알수 있게 해줍니다.

Exec:

process\_execute함수를 호출하도록 하였습니다.

Filename을 다시 파싱하여, thread\_create의 스레드이름으로 명령어 부분만 넣어줍니다. 그러면, thread\_create에 의해서 새로 생긴 스레드의 정보로 start\_create가 실행됩니다. 그 후, sema\_down(wait\_child)를 호출하여, 자식 프로세스의 생성을 기다립니다. (다시 sema\_up해줄대까지) 그 후, start\_process에서 sema\_up이 호출되면,

Thread\_create 의 리턴값으로 나온 pid값을 통해서 현재 돌아가는 중인 스레드의 자식 프로세스로써 새로운 프로세스를 넣어줍니다. (thread의 child\_list에 추가)

Wait:

process\_wait함수를 호출하도록 하였습니다.

자식 노드가 모든 수행을 마치면, process\_exit에서 sema\_up(wait\_child)와 sema\_down(exit\_child)를 호출해서, 다시 부모 프로세스로 넘어옵니다

thread자료구조에 child\_list를 이용하여, 지금 wait함수를 통해 들어온 pid를 가진 자식이 있는지 for문을 이용하여 돌린 후에, 있다면 sema\_down(wait\_child)를 호출하여, 자식노드의 exit가 끝날때까지 기다리게합니다..

그러면 자식의 exit\_status를 저장하고, child\_list 에서 삭제한 후에, sema\_up(exit\_child)를 호출해주어서 자식노드가 진짜로 종료 될 수 있게 해줍니다.

Read:

Fd = 0 (stdin) 이면 Input\_getc()를 이용하여 null이 나올때까지 받다가, 마지막으로 받은 것을 바로 리턴했습니다.

Stdio만 고려해서, 아니면 exit(-1)호출했습니다.

Write:

Fd == 1(stdout) 이면

버퍼에 사이즈만큼 putbuf()하였습니다.

Stdout만 고려해서 아니면 exit(-1) 호출했습니다.

1. Additional System calls

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

Syscall.c에서 시스템 콜 함수 Fibonacci, max\_of\_fout\_int 를 작성하였습니다.

그리고 syscall-nr.h의 시스템콜 enum의 마지막에 sys\_fibonacci, sys\_max\_of\_four\_int 를 추가하였습니다. (각 19, 20)

그리고 example 디렉토리에 있는 .c파일을 참고하여 additional.c 를 작성하엿습니다.

여기서는 위에서 작성한 두 시스템콜함수를 이용하여 출력을 해주는 역할입니다.

그리고 lib/user의 syscall.h에 프로토타입을 추가하고 syscall.c에 새로운 syscall4를 작성하였습니다. 인수가 4개인 타입의 함수입니다. 이 함수를 이용하여 max\_of\_four\_int에서 인수를 userprog/syscall.c 로 전달할수 있게 해주엇습니다. (fibonacci는 syscall1)

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**
* 텍스트이(가) 표시된 사진

  자동 생성된 설명