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

담당 교수 : 김영재

조 / 조원 : 우석민

개발 기간 : 2023-09-27~2023-10-06

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

Pintos에서 유저 프로그램을 돌릴 수 있는 것이 목표이다. 프로그램을 로드 한 뒤 스택을 초기화 된 뒤, argument를 스택에 쌓아 올려서 프로스램을 실행할 준비를 진행한다. 유저 프로그램에서 사용할 수 있는 halt, exit, exec, wait, read, write와 같은 system call interface를 구현한다. 또한 이런 시스템 콜에서 유저가 제공한 포인터의 validity를 확인해서 접근해서는 안되는 부분에 접근하면 강제 종료를 시킨다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

유저 프로그램을 돌릴 때 argument를 전달할 수 있다. 유저 프로그램이 로드가 되고 나서 스택이 만들어지면, argument들을 80x86 calling convention에 따라 스택에 push를 해준다. 이렇게 되면 프로그램이 실행될 때 argument들을 스택을 통해 접근할 수 있게 된다.

1. User Memory Access

유저 프로그램이 돌아가고 있을 때 접근하면 안 되는 메모리에 접근할 수 있다. 구체적으로는 kernel virtual address에 접근하거나, page table에 mapping이 이루어지지 않은 부분에 접근할 수 있다. 그렇기 때문에 유저가 syscall에게 보내는 모든 포인터는 validity를 확인해야 한다. 이를 구현하게 되면 유저가 특정 프로세스에서 invalid한 포인터를 보내도 dereferencing 하는 것을 방지한다. 따라서 커널이나 다른 프로세스에 영향이 가지 않는다.

1. System Calls

유저 프로그램은 보안 문제 때문에 일반적으로는 커널 메모리에 접근할 수 없다. 하지만 파일을 읽고 쓰는 등의 동작들은 커널 코드로 인해서 이루어진다. 커널에서는 유저에게 이러한 기능을 제공하기 위해 system call api를 제공한다. System call을 구현하게 되면 이를 통해 유저는 커널 코드를 실행하고, 커널의 기능을 안전하게 사용할 수 있다. 이번 프로젝트에서는 halt, exit, exec, wait, 그리고 read, write의 일부 기능을 구현하게 되고, 유저 프로그램은 이 system call 들을 사용할 수 있게 된다. Read와 write는 stdin, stdout에 한해서만 구현하게 된다.

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

커널에서 프로그램을 load하고, 스택을 initialization을 한 다음 유저가 보낸 argument들을 스택에 쌓게 된다. 먼저, 유저가 보낸 argument를 parsing을 한다. 그 다음에 뒤에 있는 argument부터 스택에 하나씩 데이터를 넣게 된다. 스택은 PHYS\_BASE인 0xC0000000부터 주소가 아래로 내려가면서 쌓인다. 이렇게 넣으면 word-align하지 않게 끝날 수 있는데, 이를 word-align하게 남는 부분을 0으로 채워 준다. 그 다음은, NULL pointer을 넣고 뒤쪽에 있는 argument부터 argument가 저장된 위치의 포인터를 차례대로 넣어준다. 이때 NULL을 넣는 이유는 argument list가 언제 끝나는지 알기 위해서이다. 그 다음 argument list의 시작점을 나타내는 포인터를 넣어주고, argument의 개수를 넣어준다. 마지막으로 return address로 0을 넣어주면 스택을 쌓는 것이 끝난다.

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

핀토스에서 invalid memory access는 NULL pointer로의 접근, mapping이 안된 가상 메모리로의 접근, 그리고 커널 메모리 영역에 접근하는 것이다.

* + Invalid memory access를 어떻게 막을 것인지 설명

이를 막기 위해서는 포인터를 dereferencing을 하는 부분이 있을 때마다 포인터의 validity를 확인하고 나서 dereferencing을 해야한다. Validity를 확인하기 위해서 먼저 NULL은 NULL인지 확인을 한다. 그리고 virtual memory에 mapping이 되어있는지 확인하기 위해서 pagedir\_get\_page 함수를 사용한다. 이 함수가 NULL을 return 한다면 mapping이 되어있지 않다는 뜻이다. 그리고 kernel memory에 접근하는지 확인하기 위해 is\_user\_vaddr() 함수 또는 is\_kernel\_vaddr() 함수를 써서 어디를 가리키는 포인터인지 확인을 한다. 이러한 invalid memory access가 있다면 exit(-1)을 통해 프로그램을 강제로 종료하게 된다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

시스템 콜은 user program이 커널이 제공하는 기능을 사용하기 위한 인터페이스이다. 시스템 콜을 통해 유저 프로그램은 하드웨어을 사용하거나, 커널이 관리하는 프로세스 관련 기능 등을 사용할 수 있다. 이를 위해서 평소에는 User mode로 코드를 돌리다가, system call이 오면, kernel mode로 변경하여 커널의 코드를 실행한다.

시스템 콜이 필요한 이유는 이런 커널이 제공하는 기능들을 안전하고 편리하게 사용하기 위해서이다. 만약에 유저 프로그램이 직접 커널에 있는 코드를 실행하여 커널의 기능을 사용하거나, 하드웨어에 직접적으로 접근한다면, 접근하면 안 되는 부분의 접근을 막는 등의 보안을 지키기 어렵다. 또한, 도중 에러가 나도 복구하기가 힘들다. 하지만 커널이 시스템 콜 인터페이스라는 중간 다리를 통해 유저 프로그램이 보내는 명령을 대신 실행하기 때문에, 잘못된 부분이 있으면 해당되는 유저 프로그램을 강제 종료 시키거나, 에러가 났다고 리턴 하는 등, 커널 자체나 다른 프로세스에 영향을 끼치지 않게 처리할 수 있다.

또한 커널은 시스템 리소스를 공정하게 프로세스에 나눠서 사용해야 하는데, 유저 프로그램이 직접 하드웨어를 사용하는 코드를 돌리게 된다면 이러한 리소스 관리가 어려워질 것이다. 시스템 콜 인터페이스를 통해 하드웨어에 접근하면 커널이 어떤 프로세스가 하드웨어를 사용할지 관리를 할 수 있게 된다.

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

Halt 시스템 콜은 shutdown\_power\_off() 함수를 호출하여 pintos를 종료시킨다.

Exit 시스템 콜은 현재 유저 프로세스를 종료시키고, 커널에 return status를 전달한다. 모든 프로세스는 이 시스템 콜을 통해서 종료가 이루어진다. Exec, Wait와 관련해서 동기화 문제를 처리해야 한다.

Exec는 인자로 주어진 유저 프로그램을 실행시키는 시스템 콜이다. 제대로 실행이 되면 적절한 pid를 리턴하고 아니면 -1을 리턴한다.

Wait는 인자로 주어진 pid에 해당하는 child 프로세스가 종료되기까지 기다리는 시스템 콜이다. 부모에 의해서 불리며, 같은 child에 대해 두 번 불릴 수 없다.

Read는 파일을 읽는 시스템 콜이다. 이번 프로젝트에서는 인자로 0이 들어온 경우, stdin만 input\_getc 함수를 활용하여 처리한다.

Write는 파일에 쓰는 시스템 콜이다. 이번 프로젝트에서는 인자로 1이 들어온 경우, stdout만 putbuf 함수를 통해 처리한다.

Fibonacci, max\_of\_four\_int 시스템 콜은 피보나치 수열을 계산하는 시스템 콜과, 4개의 정수 중 가장 큰 값을 리턴하는 시스템 콜이다.

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

핀토스가 실행될 때 syscall\_init에서 interrupt의 0x30을 interrupt vector table에 syscall\_handler로 등록하게 된다.

유저 레벨에서 시스템 콜 API를 호출하면, 먼저 스택에 유저가 전달한 인자들을 push하게 된다. 또한 정해진 시스템 콜 번호를 스택에 넣는다. 그 다음 0x30으로 interrupt를 걸어서 syscall\_handler을 호출하게 된다. syscall\_handler에서는 스택에 저장된 system call number이랑 저장된 인자들에 접근해서 해당하는 system call에 맞는 코드를 실행한다. 이때 스택에 접근할 때 유저 프로그램이 보낸 스택 포인터의 validity를 확인하고, 만약에 system call의 인자에 포인터가 있으면, 이도 포인터의 validity를 확인한다. 코드의 호출이 종료되었으면, 다시 유저 프로그램으로 돌아가게 된다. 이때 system call number이랑 인자들을 스택에 push를 했기 때문에 스택 포인터를 다시 원래대로 되돌리고, eax에 저장된 값을 system call api를 부른 곳에 리턴하게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

Argument Passing (09/27 ~ 09/28)

* + - 들어온 인자를 파싱하고, 스택에 쌓는 코드를 작성

User Memory Access (09/29 ~ 09/29)

* + - Invalid한 유저 메모리 접근을 검증하는 함수 작성

System Call (09/30~10/5)

* + - Halt, exit, exec, wait, read, write의 기능을 구현하는 함수를 작성. Exit, exec, wait 등 동기화가 필요한 부분에 대한 검증. Additional System call인 fibonacci랑 max\_of\_four\_int 작성.

보고서 작성 및 검토(10/6~10/7)

* 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

현재 thread에 프로그램을 로드 시켜주는 load 함수에서 argument passing에 관한 내용을 구현한다. 이 함수에는 file\_name이라는 이름으로 문자열이 넘어오게 되는데, 이 문자열에는 프로그램 파일 이름과, 유저가 보낸 인자들이 들어있다. 프로그램을 로드 하기 위해 먼저 filesys\_open 함수를 불러 프로그램 파일을 열게 되는데, 이 전에 argument를 parsing 해서 프로그램 파일 이름만 이 함수에 넣어주어야 한다. 그리고 나중에 스택이 setup되고 나서 스택에 인자를 push해줘야 하는데, 이를 위해 parsing한 내용을 저장해두고 있는다. 스택이 setup이 되면 80x86 convention에 따라 스택에 argument를 push해준다. 먼저 argument의 반대 순서대로 넣어주고, 이때 넣어준 address를 미리 저장해 둔다. 그 다음 word align에 맞춰서 esp를 조정하고, argv의 마지막을 나타내는 NULL을 push한다. 그 다음 argument가 어디 저장되어 있는지에 대한 포인터를 미리 저장해둔 것을 이용해 push해준다. Argv의 시작점을 나타내는 포인터를 push하고, return address인 0를 push해주면 끝나게 된다.

* + User Memory Access

유저가 syscall을 통해 kernel 모드로 들어가고, 전달해주는 모든 포인터의 validity를 확인한다. 이를 쉽게 하기 위해 check\_valid\_pointer 함수를 작성한다. 이 함수는 NULL인지, is\_user\_vaddr()를 써서 유저 가상 메모리 영역인지, 그리고 pagedir\_get\_page 함수를 써서 mapping이 된 메모리인지 확인을 한다. 확인하는 포인터는 유저가 보내주는 esp, 그리고 시스템 콜 인자로 포인터가 있는 exec, write, read 등의 시스템 콜에 전달되는 포인터를 확인한다.

* + System Call

기본적으로 syscalll\_handler에서 먼저 esp를 읽고, syscall 번호를 확인한다. 확인한 번호를 바탕으로 각 system call에 해당하는 함수를 불러줄 것이다. 실수를 줄이기 위해 esp에서 값을 읽어올 때는 타입별로 미리 작성한 함수를 써서 읽는다.

Exit, exec, wait의 시스템 콜에서의 동기화 문제를 해결하기 위해 semaphore을 추가적으로 저장을 해야 한다. 또한 부모 프로세스에 따른 자식 프로세스가 누군지 알기 위해 list에 저장을 해야 한다. List는 핀토스에서 제공하는 struct list를 사용한다. Struct thread를 list\_item 처럼 사용해도 될 것이지만, 추가한 요소들을 분리해서 확인하기 위해 struct list\_item\_thread를 따로 만들어서 추가적인 정보를 이 struct에 저장을 하고, struct thread에 struct list\_item\_thread를 넣는다.

Struct list\_item\_thread에는 동기화를 위한 semaphore called\_exit, can\_free\_resources, load\_done을 넣을 것이다. Called\_exit은 exit이 불려졌는지 확인하기 위해서, can\_free\_resources는 wait가 끝나고, 부모에서 child를 리스트에서 제거를 했는지 확인하기 위해, load\_done은 exec에서 load가 끝나고 성공, 실패 여부를 확인하는 동기화를 진행하기 위해 넣는다. 추가적으로 로드가 성공적으로 되었는지 저장하는 load\_success, exit number을 저장하는 exit\_status, 그리고 중복으로 wait하지 않게 방지를 해주는 waiting을 저장한다.

이렇게 struct thread에 struct list와 struct list\_item\_thread 등 여러가지를 추가했기 때문에, 이를 init을 시켜줘야 한다. Init\_thread 함수에서 추가한 것들을 init을 해준다. 또한 thread\_create에 부모 thread가 child thread를 list에 추가해주는 부분을 추가할 것이다.

* + - Halt

Halt는 shutdown\_power\_off()를 불러주는 것으로 구현을 한다.

* + - Exit

Exit은 wait와의 동기화 작업을 위해 called\_exit semaphore을 up 시켜주고, can\_free\_resources semaphore를 down 시켜준다. 이렇게 하면 wait에서 can\_free\_resources semaphore를 up 시켜주면 thread\_exit을 불러서 thread를 종료시킨다. 추가적으로 status 인자를 esp에서 읽어 프린트 해준다.

* + - Exec

Esp에 저장된 포인터를 읽고, 이 포인터에 저장된 파일명을 process\_execute 함수에 전달하여 새로운 프로세스를 생성한다. 만약에 로드에 실패하면 -1을 리턴해야 하기 때문에 load\_done semaphore을 down해두고, 프로그램이 load가 된 시점에서 load\_done semaphore을 up 해주어야 한다. Start\_process 함수가 child thread에서 돌아가기 때문에 이 함수에서 이런 동기화 작업을 해주면 된다. 그리고 로드가 잘 되었는지에 따라서 load\_success를 설정해주고, 이를 이용해서 tid를 리턴하거나 -1을 리턴한다.

* + - Wait

Wait 할 pid를 esp에서 읽고, process\_wait를 불러서 wait를 진행한다. Process\_wait 함수는 현재 프로세스의 child process를 저장한 child\_process\_list를 순회하면서 들어온 pid에 해당하는 child의 struct thread를 얻는다. 이를 통해 child의 called\_exit semaphore이 up이 된다면, child\_process\_list에서 child를 제거하고, can\_free\_resources를 up 해서 child 프로세스가 thread\_exit을 부를 수 있게 해준다.

* + - Write

Esp를 읽어서 fd, buffer, size를 얻어준다. 이번 프로젝트에서는 fd가 1인 경우만 처리하면 되기 때문에 putbuf 함수를 이용해 buffer에 size만큼을 읽어서 콘솔에 write 한다.

* + - Read

Esp를 읽어서 fd, buffer, size를 얻어준다. 이번 프로젝트는 fd가 0이 들어온 경우만 처리하면 되기 때문에 input\_getc()함수를 size번을 불러 input을 받고 buffer에 넣어준다.

* + Additional System call

lib/에 있는 syscall-nr에 새롭게 구현할 system call의 number을 넣어준다. lib/user에 유저가 부를 수 있는 api 함수를 정의해준다. Max\_of\_four\_int는 인자를 4개를 넘겨야 하기 때문에 정의가 되지 않은 syscall4를 정의해서 인자 4개를 push하고 int 30을 해주는 코드를 작성한다. 리턴이 되면 스택에 20을 더해서 원상복구 시켜준다.

* + - Fibonacci

Esp를 읽어서 인자인 n을 받아준다. Syscall.c 함수에서는 피보나치 수열을 리턴하는 함수를 넣어준다.

* + - Max\_of\_four\_int

Esp를 읽어서 인자인 a, b, c, d를 받아준다. Syscall.c에는 인자로 들어온 4개의 정수 중 maximum을 리턴하는 함수를 작성한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

텍스트, 도표, 평면도, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Argument passing 를 구현하기 위해 load 함수에서 위에 설명한 내용의 순서를 순서도로 나타낸 것이다.

1. User Memory Access

도표, 스케치, 그림, 패턴이(가) 표시된 사진

자동 생성된 설명

User memory access를 관리하기 위한 추가적인 함수에 대한 순서도를 나타낸 것이다. check\_valid\_pointer로 user가 제공한 주소가 valid한지 확인한다. 아래에 있는 read\_stack\_int32, read\_stack\_pointer 등은 check\_valid\_pointer 함수를 활용해 필요할 때 valid한 포인터인지 확인한다.

1. System Calls

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

System Call Handler의 전체적인 구현에 대한 순서도이다. 개발 방법에서 작성한 내용을 순서도로 나타내었다.

* 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

Argument passing은 load 함수에서 구현하였다. 먼저 load에 들어온 file\_name을 parsing 해야한다. 인자로 들어온 file\_name을 strlcpy를 이용해서 임의로 정의한 dup\_fn에 복사하였다. 그리고 strtok\_r 함수를 이용해 dup\_fn을 변형하면서 arguments\_list에 각 argument가 시작되는 포인터를 저장하였다. 마지막으로 나중에 exit에서 프로그램의 이름을 프린트 해야하기 때문에 struct thread에 있는 name을 parsing 한 것으로 업데이트 해주었다.

Setup\_stack이 실행되고 나서 argument를 스택에 push하는 작업을 진행하였다. Push를 할 때 esp를 조정하고 메모리를 복사해야하는 과정이 있는데, 이를 한번에 처리하기 위해 push\_to\_user\_stack이라는 함수를 만들었다. 0으로 초기화 해야하는 경우도 있기 때문에 push를 진행할 data가 NULL이면 memset을 하는 예외 사항을 포함한 함수이다. 그 이후는 argument\_list에 들어있는 argument를 뒤에서부터 넣어야 하기 때문에 arg\_idx-1부터 0까지 반복문을 돌면서 push\_to\_user\_stack을 호출하였다. 각 반복마다 argument가 저장된 위치의 포인터를 나중에 써야하기 때문에 arg\_pointers\_in\_stack에 미리 저장해 두었다.

그 다음 word align을 해야 하는데, word size가 4이기 때문에 esp%4 만큼 0을 push\_to\_user\_stack 함수에 NULL을 인자로 주어서 push를 진행하였다. Argv의 마지막을 나타내기 위한 NULL도 똑같이 진행하였다. 마지막으로 위에서 저장해둔 arg\_pointers\_in\_stack에 저장된 포인터를 arg\_idx-1부터 0까지 push를 해 주었다. 그 다음 현재 esp를 push\_to\_user\_stack해주었고, arg\_idx만큼을 push해주었다. 마지막으로 4바이트 만큼 0을 push해서 argument passing을 끝내었다.

1. User Memory Access

User memory access를 validate하기 위해서 check\_valid\_pointer이라는 함수를 새롭게 만들었다. 이 함수에서 pagedir\_get\_page를 사용하기 때문에, validity를 확인할 주소인 uaddr과 함께, uint32\_t로 struct thread에 정의되어 있는 pagedir도 인자로 넘겨 주는 형식으로 함수를 제작하였다. 만약에 valid하지 않은 메모리이면 actual\_exit(-1)을 불러서 -1을 exit status로 하여 프로그램이 종료되게 하였다.

후에 syscall에서 유저가 제공한 포인터를 dereference 할 때 check\_valid\_pointer을 선행해준다. 두 경우가 있는데, 하나는 esp에 접근하는 경우이다. Esp에 접근하기 전에 validation을 하지 않는 것을 방지하기 위해 read\_stack\_int32, read\_stack\_uint32, read\_stack\_pointer 등의 함수를 작성하였다. 이 함수들에서는 dereference를 하기 전에 check\_valid\_pointer을 먼저 호출하게 된다. 더불어서 read\_stack\_pointer은 포인터가 저장되어 있는 부분을 dereference 하고, 그 다음 저장된 pointer을 다시 check\_valid\_pointer 함수를 이용해 validation 한다. 그렇기 때문에 이 함수로 얻은 포인터는 validity에 대해서 걱정 하지 않아도 된다.

1. System Calls

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

Syscall\_handler이 호출이 되면 가장 먼저 read\_stack\_int32를 활용하여 syscall number을 얻어내었다. 그 다음 이 syscall number을 switch문을 이용해서 각 syscall에 해당하는 함수를 불러 내었다. 인자로는 struct thread\* 와, struct intr\_frame을 주었는데, 각 함수에서 인자를 뽑아내고, 리턴 값인 eax을 설정할 수 있기 위해서 두 구조체 포인터를 넘겨주었다.

Syscall\_handler에서 쓰기 위해 struct thread 구조체에 많은 것을 추가하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

먼저 child\_process\_list를 추가하여 부모가 exec로 만든 child process를 트래킹 할 수 있게 하였다. 그 다음 struct list\_item\_thread 타입의 thread\_item을 추가하여 작성자가 추가한 변수들을 따로 모아서 볼 수 있게 하였다. 위에 struct list\_item\_thread가 정의되어 있는데, 여기에는 동기화에 필요한 semaphore, 그리고 추가적으로 저장할 필요가 있는 정보들을 저장하는 부분이다. 이런 개별적인 내용은 개별 syscall을 설명하는 부분에 후술하였다.

Struct thread에 추가를 하였기 때문에 추가한 부분에 대한 init을 init\_thread 함수에 추가하였다. Init\_thread는 thread\_init 함수에서 부르기 때문에 가장 먼저 만들어지는 thread에서도 제대로 init을 진행할 수 있게 된다. 처음에는 thread\_create에 init을 진행했었는데, 최초의 thread에서 init이 진행되지 않아 문제가 된 적이 있었다. Thread\_create 함수에는 부모 thread가 child thread를 만들고 그 struct thread \*를 받아올 수 있는 유일한 부분이기 때문에 여기에서 list\_push\_back 함수를 통해 child\_process\_list에 child\_thread 포인터를 넣어 주었다.

개별적인 설명은 아래에 되어 있다.

* + Halt

Halt는 sys\_halt()에 구현되어 있고 shutdown\_power\_off()를 부른다.

* + Exit

Exit은 sys\_exit에 구현되어 있다. 스택에서 인자로 넘어온 exit status를 읽고 actual\_exit() 함수를 호출한다. Actual exit은 인자로 exit status를 받는데, 다른 함수에서도 간편하게 사용하기 위해 exit만 따로 함수를 두 단계로 나눠서 구현하게 되었다. 예를 들어 check\_valid\_pointer 함수에서 프로그램을 종료시키기 위해 이 함수를 부르게 된다. 그리고 현재 page fault 가 났을 때 thread\_exit을 부르게 되어있는데, 이러면 semaphore 때문에 제대로 종료가 되지 않아서 actual\_exit을 부르게 하였다. 이 함수는 파일명과 exit status를 출력하고, 동기화 관련 문제를 해결한다. 먼저 struct thread에 thread\_item에 저장되어 있는 called\_exit을 sema\_up을 시켜준다. 부모에서 process\_wait에서 sema\_down(called\_exit)을 하고 있는데, 이를 풀어주어야 하기 때문에 그렇다. 이 때 actual\_exit에서는 sema\_down(can\_free\_resources)를 하고 있어야 하는데, thread\_exit이 호출되어서 child thread가 먼저 사라져 버리면, process\_wait에서 부모에 있는 child\_process\_list에 들어있는 현재 child를 삭제시켜줘야 하는데, 이미 삭제된 포인터에 접근하게 되는 race condition에 빠지게 된다. Process\_wait에서 list\_remove가 끝난 뒤에 thread\_exit을 호출할 수 있도록 list\_remove 이후 sema\_up(can\_free\_resources)를 넣어서 동기화를 진행한다.

* + Exec

Exec는 sys\_exec를 호출하여 구현한다. Read\_stack\_pointer로 인자로 들어온 파일명을 인자로 넣어 process\_execute를 호출한다. Exec 시스템 콜의 리턴 값은 tid이거나, 실패한 경우 -1을 리턴해야 한다. 이를 위해서 새롭게 생성된 thread가 load 함수가 끝날 때 까지 sys\_exec는 대기를 하고 있어야 한다. 그렇기 때문에 sema\_down(load\_done)을 해놓고 있는다. Process\_execute에서 thread\_create를 하고, 그 thread가 load를 하고 난 뒤에 sema\_up(load\_done)이 되어 성공적으로 load가 되었는지 알게 되면, sema\_up(load\_done)을 하기 전에 load\_success 변수를 확인해 성공 했는지, 실패 했는지 여부를 알 수 있다. 이를 이용해서 eax의 값을 변경해 값을 리턴한다.

* + Wait

Wait 시스템 콜은 sys\_wait에서 구현하였다. Process\_wait 함수를 부르는데, 이 함수에서 wait를 할 수 있는지 확인하고, 동기화 작업을 진행한다. 먼저 can\_wait\_tid라는 함수를 추가적으로 작성하였다. 이 함수는 child\_tid가 주어졌을 때 현재 struct thread\*에 있는 child\_process\_list를 traversal하면서 그에 맞는 child\_thread struct thread의 list\_item\_thread를 반환해준다. 이 함수에서 또한 직계 child가 맞는지, waiting이 두 번 되었는지 확인한다. 먼저 traversal을 하면서 child\_tid가 없으면 직계 child가 아닌 것이기 때문에 NULL을 반환하고, 이렇게 되면 syscall은 -1을 반환하게 된다. 그 다음 child의 list\_item\_thread에 waiting이라는 변수가 false일 때 waiting을 true로 바꿔서 wait가 두 번 이루어지는 것을 방지한다. Waiting이 true라면 NULL을 반환을 해서 -1이 리턴이 된다. 그 다음은 동기화 작업이 진행되는데 위에 exit에서 설명이 되어있다. Exit이 불리기 전 까지 대기하고 있어야 하기 때문에 called\_exit에 sema\_down을 하고 있다가, exit이 되었다면 list\_remove를 해주고, 그 다음 child process에서 thread\_exit을 부를 수 있도록 sema\_up(can\_free\_resources)를 해주게 된다. 마지막으로 exit\_status를 리턴하면서 sys\_wait로 가게 되고, eax에 이 반환값을 넣어주어 유저 프로세스 까지 리턴되게 하였다.

* + Write

인자로 3가지가 들어오는데, 이를 이용해서 putbuf 함수를 이용해 write system call을 구현하게 된다. eax에는 size를 넣어주어 얼마나 write했는지 반환한다. 일단은 다른 fd가 들어오면 -1을 반환값으로 넣어주었다.

* + Read

마찬가지로 3가지의 인자가 들어오는데, fd가 0인 경우에 input\_getc()를 size 만큼 반복문을 돌아서 buffer에 저장하게 된다. Write와 마찬가지로 size를 eax에 넣어주고, 다른 fd가 들어오면 -1을 넣어주었다.

1. Additional System calls

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

추가적인 syscall을 작성해야하기 때문에 syscall-nr.h 파일에 enum으로 정의 되어 있는 syscall number에 SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INT를 추가하였다. 그 다음은 lib/user에 있는 syscall.c에 사용자가 사용할 수 있는 Fibonacci(int n)과 max\_of\_four\_int(int a, int b, int c, int d)를 추가하였다. Fibonacci는 인자가 하나이기 때문에 syscall1로 define되어 있는 어셈블리를 실행하였고, max\_of\_four\_int는 인자가 4개이기 때문에 syscall4라는 새로운 매크로를 정의하게 되었다. 다른 매크로와 비슷하게 argument를 push하고, int 0x30을 걸어준다. 그 다음 syscall\_handler의 호출이 끝나면 스택을 복구하기 위해 20을 더해주었다.

Userprog/syscall.c에 있는 syscall\_handler에서는 위에서 새롭게 정의한 SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INT를 확인해서 sys\_fibonacci, sys\_max\_of\_four\_int 함수를 호출하였다. Sys\_fibonacci 함수는 피보나치 수열에서 인자에 맞는 숫자를 계산하여 eax에 넣어주었고, sys\_max\_of\_four\_int는 eax에 인자로 들어온 int의 max를 계산해서 넣어주었다.

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

  자동 생성된 설명**