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

담당 교수 : 박성용 교수님

조 / 조원 : 남현준

개발 기간 :

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

Skeleton code 상태로 주어진 pintos는 필요한 부분들의 구현이 빠져있기 때문에 OS로서의 기능을 충실히 수행하지 못하는 상태이다.

Pinrtos project 1에서는 system call, system call handler, argument passing, user stack등의 작동방식을 이해한 다음, 이해한 내용을 통해 해당 기능들을 구현하여 user program의 실행이 가능해지도록 하는 것을 목표로 프로젝트를 진행하게 될 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

Argument Passing은 사용자에게 받은 입력을 토대로 user program을 실행시킬 때, 입력받은 명령어 뭉치를 공백을 기준으로 잘라 실행하려는 파일의 이름, 실행 옵션, 파일을 실행할 때 사용할 argument들로 나눠서 명시된 순서대로 구현을 하게 될 user stack에 push한다. User stack에 올바르게 push가 되었다면 해당 정보를 이용하게 프로그램에 실행할 파일, argument 등을 system call로 pass하여 해당 input에 맞는 결과가 pintos의 수행결과로 출력되게 될 것이다.

1. User Memory Access

User program이 실행될 때, 참조하는 주소가 user program의 영역이 아닌 kernel program의 영역이라면 exit status -1로 종료되어야 한다. 또한, 참조하려는 주소를 가져왔을 때 null pointer인 경우나 모종의 이유로 valid한 상태의 주소가 아닐 경우, 마찬가지로 프로그램의 실행이 종료될 수 있도록 구현되어야 한다.

1. System Calls

User program이 실행될 때 해당 program의 argument의 개수에 맞는 system call이 호출된다. 이렇게 호출되는 system call은 system call handler를 경유하여 source code로 주어진 system call api들을 호출하여 명령어에 맞는 실행이 이루어질 것이다. Project 1에서는 이 내용을 구현하여 halt(), exit(), wait(), exec(), read(), write() 함수가 실행될 때 해당 명령어에 맞는 결과가 나올 수 있을 것이다.

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

User program이 실행되며 명령어가 들어오면 공백을 기준으로 argument를 나누게 된다. 이 때, user stack에는 argument를 역순으로 스택에 저장하기 시작한다. 공백을 기준으로 나눈 argument의 저장이 완료되었다면 word alignment를 통해 참조되는 메모리의 단위가 4 byte로 나누어 떨어질 수 있도록 부족한 부분만큼 메꿔준다.

가령, 주어진 ppt에서의 예제인 “echo x”의 경우 “echo”를 저장하기 위해 5 byte, “x”를 저장하기 위해 2 byte를 사용하여 총 7 byte의 크기를 가진다. (이 떄, 문자열의 길이보다 1 큰 크기를 사용하는 것은 마지막에 추가되는 ‘\0’를 의식한 것이다.) 따라서, 메모리의 총 사용량이 4의 배수가 될 수 있도록 1 byte만큼을 0으로 채워주는 형식으로 명령어를 저장하게 된다. 이렇게 word alignment를 진행하는 이유로는 메모리 주소를 탐색할 때 주소 + 1 혹은 주소 – 1을 통해 다음/이전 주소로 접근할 경우 4 byte 만큼의 차이가 생기게 되는데 위의 경우와 같이 7 byte를 저장한 다음 메모리 주소를 통해 접근하는 경우 의도하지 않은 부분까지 읽어들일 가능성이 있기 때문에 alignment를 통해 규격을 맞춰주는 것이라고 할 수 있다.

이렇게 word alignment를 통한 명령어의 저장을 끝냈다면 다음 4 byte만큼은 0을 저장해 선을 그어 명령어와 주소의 저장 부분을 나눠주도록 한다. 그 다음 부분부터 개별 argument의 주소, argument들의 주소를 저장하는 배열의 시작주소 (&argv), argument의 개수를 나타내는 argc, return address 0를 차례대로 push 하는 것으로 커널 내 stack에 argument를 push하는 것이 완료된다.

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

User memory의 경우 user program이 프로그램에서 사용할 데이터가 저장된 영역을 참조하는 것으로 사용된다. 해당 부분을 참조하기 위해 포인터를 통해 주소를 가리키는 방식으로 사용하게 되는데 이 때, user program에서 포인터가 가리키는 주소가 kernel address의 영역이었거나 주소를 얻어올 것을 요청했는데 결과가 null pointer, 즉 일치하는 주소가 없는 경우 invalid한 memory access가 일어나게 된다.

Invalid memory access를 막기 위해 project1을 진행하기 위해 주어진 pintos의 skeleton code에는 valid한 user address인지를 판별하는 is\_user\_vaddr() 함수와 is\_kernel\_vaddr() 함수가 제공된다. 해당 함수들을 적절히 활용하여 user 영역에서 valid한 주소인지, kernel 영역에서 valid한 주소인지를 판별하는 것으로 프로그램의 실행을 종료할지, 아니면 속행할지를 판단할 수 있을 것이다.

* 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

위에서 정의한 것과 같이 user program은 기본적으로 kernel address 영역에 접근하지 못하도록 되어있다. 프로그램이 실행중인 경우, memory나 disk영역을 access하게 된다면 저장된 데이터에 변화가 생겨 core functionality들에 프로그램적 에러를 야기할 수 있기 때문이다. 따라서, 메모리/디스크에 접근하는 작동은 kernel mode에서 실행되게 되는데 이 때 OS에서 제공하는 방식이 system call을 통해 kernel mode로의 진입이다.

Project 1에서 구현한 system call의 목록은 다음과 같다.

* + Halt()
    - Shutdown\_power\_off() 함수를 호출하여 종료하게 되는 system call.
  + Exit()
    - 어떤 명령어가 실행되었는지와 그 때의 exit status의 출력하고 thread\_exit()을 통해 thread의 종료를 담당하는 system call.
  + Exec()
    - Process\_execute() 함수를 통해 process의 execution을 하도록 하는 system call
  + Wait()
    - Parent와 child process가 있을 때,, child process의 종료까지 parent process의 실행을 잠깐동안 중지시키도록 하는 system call.
  + Read()
    - STDIN의 내용을 읽는다. Fd가 0인 경우 length 길이만큼 반복문을 진행하다 ‘\0’(= null 문자)가 나왔을 때 해당 index의 값을 return하는 함수를 실행하는 system call.
  + Write()
    - STDOUT에 출력한다. Putbuf를 통해 buffer에 있는 내용을 length 길이만큼 가져오고 length를 return하는 함수를 실행하는 system call.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

9/18 ~ 9/25: Argument passing 구현

9/25 ~ 10/5: System call, system call handler, user memory access, additional 구현

9/18 ~ 10-8: 보고서 작성

* 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)**

1. Argument passing

Process.c의 load 함수에서 user stack을 구성하기에 앞서 실행하고자 하는 파일의 이름이 valid한지 확인을 해야 할 필요가 있다. User stack의 구성에 사용할 argv, argc를 구하는 과정에서 strtok\_r을 통해 얻은 string token이 executable한 file의 이름인지 filesys\_open() 함수를 통해 확인한다. Strtok() 대신 strtok\_r() 함수를 사용한 이유는 strtok\_r()이 thread-safe한 함수이고 이는 thread가 동시에 해당 함수를 사용하더라도 언제나 의도한대로의 실행결과가 나오기 때문이다.

Executable file 형식이라면 setup\_stack()의 수행값이 true일 때 user stack의 구성을 시작하게 된다. 우선 stack에 user로부터 받은 명령어를 역순으로 stack에 push하도록 한다. 이후, alignment를 통해 사용한 메모리의 총량이 word size (4 byte)가 될 수 있도록 조정해준 뒤 1 word size만큼 null 값으로 대체하여 실질적인 명령어와 해당 명령어를 저장하고 있는 배열의 주소를 확실한 선을 통해 나눠준다. 이후, 명령어의 주소, 해당 명령어를 담고 있는 argv의 시작주소, argc 등을 차례로 stack에 push하고 return address 0을 push 하는 것으로 argument passing을 완료함과 동시에 user stack의 구성도 마치게 된다.

1. User memory access

User가 invalid한 메모리 공간을 참조하는 경우 user program이 종료되어야 한다. threads폴더의 vaddr.h 헤더 파일에 정의된 함수들을 활용하여 해당 부분을 구현하도록 한다.

Syscall.c의 system call handler 구현 부분에서 interrupt stack frame을 통해 저장된 stack pointer에 접근할 수 있는데 해당 stack pointer에서 참조하는 주소가 user address space에 위치하는지 확인해야 할 필요가 있다. 따라서, vaddr.h 내부에 정의된 is\_user\_vaddr() 함수를 이용해 user address space를 참조한다면 system call에 맞는 함수를 실행시키고 그렇지 않다면 exit(-1)을 통해 프로그램이 종료될 수 있도록 구현하는 방향으로 수정했다.

Exception.c 파일에서 user가 null이거나 참조하는 address가 kernel 영역일 경우 마찬가지로 program이 종료되어야 한다. 따라서, is\_kernel\_vaddr() 함수를 통해 주어진 address가 kernel 영역을 참조하는 중이거나 user가 null인 경우 exit(-1)을 호출하여 프로그램이 종료될 수 있도록 수정했다.

1. System call

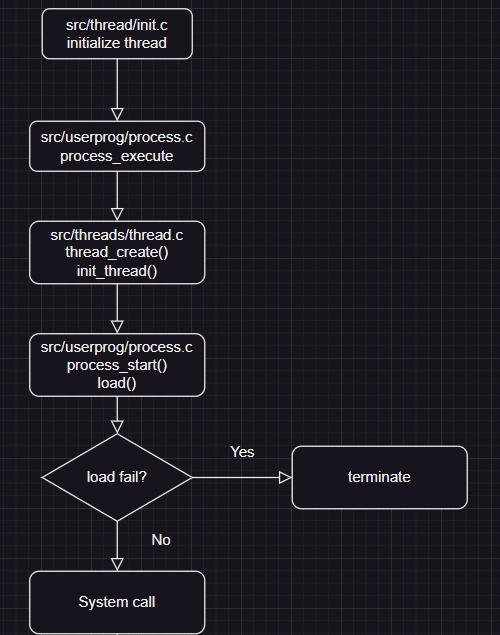
System call handler를 통해 주어지는 system call에 대해 switch문을 통해 해당 system call에 맞는 함수가 호출될 수 있도록 수정을 가한다.

호출되어 반환되는 결과는 interrupt stack frame의 eax에 저장될 수 있도록 했고 만약 stack frame의 esp가 user address space가 아닌 곳을 참조하고 있다면 exit(-1)을 통해 프로그램이 종료될 수 있도록 수정했다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

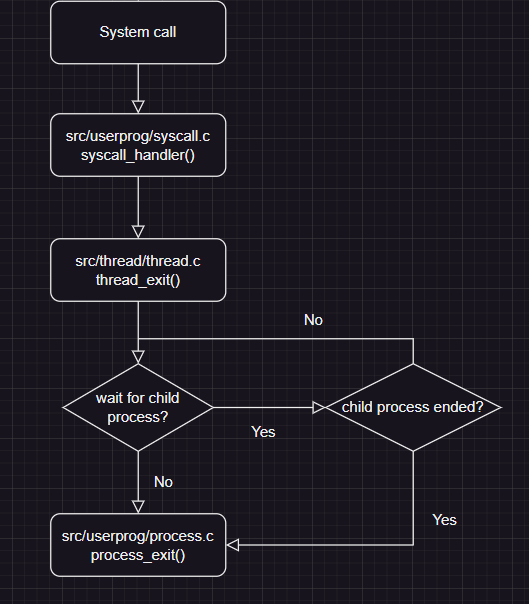


1. User Memory Access

텍스트, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명

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

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Executable한 파일을 load하여 프로그램을 실행하는 것인지 판단하기 위해 후에 user stack을 구성할 때 사용할 argv배열을 구성하며 발생하는 첫 번째 string token을 fileString에 저장한다. 공백을 기준으로 parsing하기 때문에 가장 처음에 token으로 잘리는 string이 file의 이름이기 때문이다.

Argv의 구성이 끝났다면 filesys\_open() 함수를 통해 유효한 executable file인지 확인한다. 만약 filesys\_open()이 return하는 값이 NULL이라면 유효하지 않은 executable file이므로 goto done을 통해 stack의 구성을 하지 않고 false를 return한다.

Filesys\_open()이 return하는 값이 NULL이 아니었다면 user stack의 구성을 시작한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

앞서 구성한 argv를 user stack에 push한다. 이 때, stack에 push되는 순서는 주어진 명령어의 오른쪽 -> 왼쪽 방향으로 push되기 때문에 반복문을 argc-1부터 0의 순서로 진행하며 stack에 push하도록 한다. 명령어를 나눈 argument를 저장한 다음은 해당 argument들의 주소 또한 저장해야 하기 때문에 stack에 push하며 주소를 argvAddress에 저장한다.

Argument의 push가 끝난 후 work alignment를 통해 사용한 메모리의 크기를 4의 배수로 맞춰준 다음 null pointer, argument address, argc, return address 순으로 user stack에 push를 해준다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Process\_execute() 함수에서 file\_name 또한 strtok\_r() 함수를 통해 적절하게 argument를 공백 문자를 기준으로 잘라주는 것으로 filesys\_open() 함수가 작동할 수 있도록 했다.

Thread를 구성할 때 전달된 명령어에서 프로그램에 해당하는 string 부분이 유효하지 않은 이름이라면 return -1을 통해 thread가 생성되지 않고 바로 종료될 수 있도록 하였다.

1. User Memory Access

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Userprog/sycall.c 파일에서 올바르지 못한 메모리 영역을 참조하게 되는 경우 exit(-1)을 할 수 있도록 if-else 문을 구성했다. 해당 작동을 위해 thread/vaddr.h 파일을 include하여 valid한 user address인지 확인하는 함수인 is\_user\_vaddr()을 사용할 수 있도록 수정했다.

Is\_user\_vaddr() 함수를 통해 얻은 값이 true라면 해당 system call에 맞는 함수를 실행하여 의도한 결과를 return받을 수 있도록 했고 그게 아니라면 exit(-1)을 호출해 종료될 수 있도록 system call handler를 구성했다.

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

Userprog/exception.c 에서 kernel address를 참조하는 경우 프로그램이 종료될 수 있도록 is\_kernel\_vaddr()을 추가했다.

1. System Calls

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

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* Halt()
  + Shutdown\_power\_off() 함수를 호출하고 break를 통해 switch문을 탈출하게 됨.
* Wait()
  + Is\_user\_vaddr() 함수를 통해 user address space를 참조하는지 확인한 후 결과에 맞는 작동을 하게 됨.

Valid했을 경우 wait() 함수를 실행한다.

폰트, 텍스트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

Process\_wait()를 실행하여 반환 받은 결과를 결과로 반환한다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Process\_wait() 함수는 현재 thread의 child list를 탐색해서 child\_tid를 만족하는 tid를 가진 thread가 child list 중에 있다면 synchronization을 통해 child가 terminate 된 다음 parent process가 성공적으로 reaping할 수 있도록 되어야 할 것이다.

처음에는 synchronization의 사용 없이 적절히 parent의 wait 상태, child의 exit 상태 등만을 thread에 저장하는 식으로 접근할 수 있지 않을까 생각했지만 결과적으로 wait를 구현했을 때 main : exit(-1)의 상태로 프로그램이 죽어버리는 것을 확인할 수 있었다.

해당 문제는 parent가 child를 wait하는 것까지는 성공했지만 child가 thread\_exit() 통해 종료되며 child process가 가지고 있던 memory 정보 또한 같이 사라지게 되는 것에서 생긴다. Parent process에서는 입력에 맞게 작동을 한 다음, list\_remove()를 통해 child의 정보를 list에서 삭제하는 code를 수행하게 되는데 이 때, child process가 종료됨과 동시에 관련된 정보가 모두 사라지게 되어 remove를 통해 제거할 수 있는 데이터가 남아있지 않을 가능성이 있다. parent에서 해당 작동을 올바르게 수행하기 전에, child process가 실행되는 thread가 exit()을 통해 종료되며 관련된 정보가 시스템 상에서 사라지게 되고, 뒤늦게 parent에서 list\_remove()를 통해 child의 정보를 list에서 삭제하기 위해 포인터를 참조하게 되며 해당 문제가 발생하게 되는 것이다. 해당 작동에 대해, parent는 이미 없어진 정보를 찾으려고 하기 때문에 의도했던대로 작동하지 못하고, 이로 인해 main : exit(-1)이라는 문구가 pintos 실행 중에 출력되게 되는 것이었다.

해당 문제를 해결하기 위해 synchronization의 설정을 위한 semaphore를 2가지 선언했다. 하나는 childSemaphore로 child의 종료를 기다릴 수 있게 하는 semaphore다. Process\_wait() 함수에서 현재 실행중인 thread의 child process의 list에서 일치하는 thread id를 가진 child가 있다면 parent는 해당 child의 작동이 완료될 때까지 기다려야 한다. 따라서, sema\_down(&tmp->childSempahore)를 통해 해당 값이 1 이상일 때 sema\_down() 이후에 적인 로직을 수행할 수 있도록 하였다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Process가 종료될 때, 현재 실행중인 thread의 childSemaphore가 sema\_up()을 통해 값이 변경되어 위의 process\_wait() 함수에서 sema\_down() 이후의 부분을 수행할 수 있게 된다. MemSemaphore 또한 명시되어 있는데 해당 semaphore의 역할은 child process의 메모리를 특정 시점까지 보관될 수 있도록 하는 것이다.

앞서 semaphore를 사용한 구현을 하기 전에 main : exit(-1)이라는 문구를 OS가 출력했다고 작성했는데 이는 parent process와 child process 사이에서 어떤 순서로 실행될지 모르는 thread의 특성 때문에 생기는 concurrency issue라고 하였다. childSemaphore가 process\_exit()에서 sema\_up이 되는 순간 process가 종료되게 되고 child proceses와 관련된 정보 또한 할당해제가 될 것이다. 따라서, parent 쪽에서 child의 정보를 list에서 삭제하려고 하지만 이미 할당해제가 된 상태라 찾을 수 없게 되고 이로 인해 main : exit(-1) error가 출력되는 것이었다.

따라서, semaphore를 하나 더 추가해야 할 필요를 느꼈고 해당 semaphore가 바로 MemSemaphore이다. Child process가 종료되었을 때 바로 메모리의 해제가 일어나는 것이 아닌 parent에서 child process에 대한 정보를 list\_remove()를 통해 지운 다음에 메모리의 할당이 해제되어야 올바르게 작동되는 것이다. 따라서, process\_exit() 단계에서 child process일 경우 sema\_down()을 통해 MemSemaphore의 값을 내리도록 하였다. 이 때, sema\_up()을 통해 MemSemaphore의 값이 변경되는 시점은 list\_remove()를 통해 child의 정보를 지운 다음이므로 이렇게 한다면 올바르게 reaping됨과 동시에 parent에서 list\_remove() 함수를 통해 child wait를 끝냈다는 사실을 반영한 후, 다음에 정의된 코드대로 작동하게 될 것이다.

* Exit()
  + Is\_user\_vaddr() 함수를 통해 user address space를 참조하는지 확인한 후 결과에 맞는 작동을 하게 됨.

Valid했을 경우 exit() 함수를 실행한다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

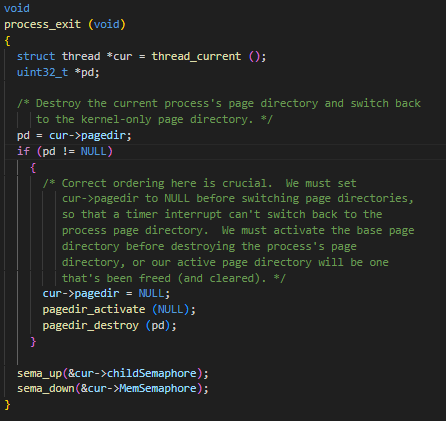
자동 생성된 설명

Exit() 함수에서는 현재 실행되고 있는 thread의 이름과 exit status를 출력한 다음 thread\_exit()를 호출하여 thread를 종료시킨다.

Thread/thread.c 의 thread\_exit() 함수에서 process\_exit()가 호출된다.

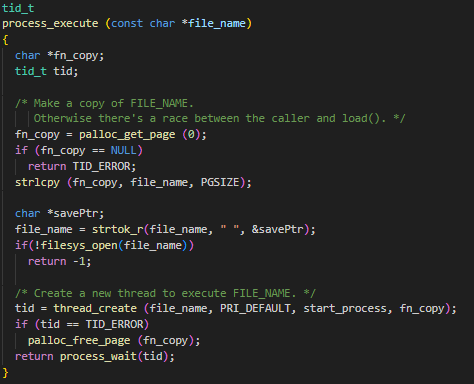
텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명



앞서 wait()에서 언급했던 것과 같이 thread id가 child process의 정보를 담고 있는 thread라면 해당 thread의 parent process가 child list에서 정보를 list\_remove()를 통해 지우기 전까지 정보가 유지되어야 할 필요가 있다. 따라서, 해당 thread가 child process 였을 경우 parent process에서 list에서 remove를 통해 연결을 해제한 다음 thread가 종료될 수 있도록 해야할 것이다.

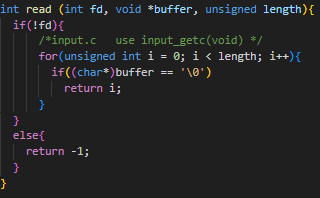
* Exec()
  + Process\_execute()를 통해 인자로 넘겨지는 이름을 가진 파일을 실행하도록 한다.



File\_name을 strtok\_r()을 통해 파싱하여 저장되어 있는 file 중 같은 이름을 가진 file을 메모리에 load할 수 있도록 한다. 이 때, file\_name이 존재하지 않는 file의 이름이라면 -1을 return하여 exit(0함수가 실행될 수 있도록 한다.

실행할 수 있는 파일의 이름이었다면 thread\_create()를 통해 thread를 생성하고 thread id를 할당한다.

Thread의 생성이 성공적으로 완료되었다면 process\_wait(tid)를 통해

* Read()
  + 

인자로 넘겨진 fd를 check한다. 만약 fd의 번호가 0이 아닐 경우 STDIN에 해당하는 file descriptor 아니므로 read 함수가 해당 fd로 호출되었을 경우 -1을 return하여 종료될 수 있도록 한다.

0이었을 경우 ‘\0’ (null 문자)를 만날 때까지 파일을 읽으며 null 문자 전까지의 파일 길이를 결과로 return하도록 한다.

* Write()
  + 텍스트, 스크린샷, 폰트이(가) 표시된 사진

    자동 생성된 설명

인자로 넘겨진 fd를 check한다. 만약 fd의 번호가 0일 경우 STDOUT에 해당하는 file descriptor 아니므로 write 함수가 해당 fd로 호출되었을 경우 -1을 return하여 종료될 수 있도록 한다.

0이 아니었을 경우 write 함수를 통해 console 상에 내용이 출력될 수 있도록 putbuf() 함수를 통해 인자로 넘겨진 buffer에 저장된 정보를 출력한 다음 length를 결과로 return한다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
  + Fibonacci와 max\_of\_four\_int 함수의 경우, 주어진 skeleton code에 system call number가 정의되어 있지 않다. 따라서, 해당 함수들을 system call handler를 통해 특정 system call이 들어왔을 때 호출하도록 하려면 system call number를 정의해주어야 한다.

Src/lib 폴더에 있는 syscall-nr.h 파일에서 해당 system call을 받았을 때 작동할 수 있게 system call을 추가해줘야 한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

다음과 같이 SYS\_FIBO와 SYS\_MAX\_FOUR system call을 추가하였다.

스크린샷, 텍스트, 폰트이(가) 표시된 사진

자동 생성된 설명

다른 system call들과 마찬가지로 system call이 발생했을 때 system call handler를 통해 kernel에서 해당 system call의 요청을 처리할 수 있도록 해주었다. 다른 system call과 마찬가지로 invalid한 user address space를 참조할 경우 exit(-1)을 통해 종료될 수 있도록 하였다.

다른 system call들과 마찬가지로 system call number에 맞는 함수가 호출 될 수 있어야 하기 때문에 해당 내용이 사전에 헤더파일에 정의되어야 한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Src/lib/user 폴더에서 syscall.h 파일에 다음과 같이 fibonacci에 해당하는 int fibo(int n)과 max of four integers에 해당하는 int maxOfFour(int a, int b, int c, int d)를 추가해주었다.

다른 함수들과 마찬가지로 src/userprog의 syscall.c 파일에서 해당 함수들의 작동을 정의했다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Fibonacci의 경우 fibo 함수에서 사용하는 argument의 수가 1개이기 때문에 사전에 정의되어 있는 sycall1을 사용하면 interrupt가 일어나지만 maxOfFour의 경우 argument의 수가 4개이고 skeleton code 상태의 pintos에서는 3개의 argument에 대응 가능한 syscall3까지만 정의되어 있기 때문에 새로운 sycall4를 정의할 필요가 있다.

Src/lib/user 폴더의 syscall.c 파일에 syscall에 대한 정의가 있다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

해당 파일에 다음과 같이 maxOfFour에서 사용 가능한 syscall4를 정의하였다. Syscall4 함수는 어셈블리어로 정의되어 있고 인자로 들어오는 number, arg0, arg1, arg2, arg3를 역순으로 esp 스택 push하는 형태로 선언되어 있다. 이 때, esp에 모든 데이터를 push한 다음 addl $20을 통해 return address를 복구하는 작업을 했다.

Project의 명세서를 보면 additional이라는 프로그램을 통해 fibonacci와 max\_of\_four\_integers를 실행하도록 되어있다. 하지만, skeleton code 상태의 pintos에는 additional이라는 프로그램이 정의되어 있지 않으므로 직접 추가해주어야 한다.

Src/examples 폴더 additional.c 파일을 생성하여 다음과 같이 작성해주었다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Fibonacci(arg 0)을 구하게 되는데 arg 0번째 Fibonacci number를 구했다면 maxFour에 maxOfFour()함수를 통해 argument로 들어오는 값들 중에서 가장 큰 값을 저장한다. 해당 값들을 출력한 다음 EXIT\_SUCCES를 return하여 성공적으로 호출되었다는 flag를 반환한다.

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

수행 결과는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Arg 0에 해당하는 10번째 Fibonacci number이 55와 해당 arguments 중 가장 큰 값인 62가 출력되는 것을 확인할 수 있다.