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

담당 교수 : 박성용 교수

조 / 조원 : 20181589 강민석

개발 기간 : 2022.09.18 ~ 2022.10.01

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

현재 주어진 pintos에서는 system call, system call handler, argument passing, 그리고 user stack이 구현되지 않았기 때문에 user program이 제대로 실행되지 않는다. 기본적으로, 현재 pintos는 위 기능들을 포함하여 많은 OS기능들을 구현하지 않는다. 이번 프로젝트에서는 이런 기능들을 구현해서 user program이 실행될 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

src/userprog 폴더 안에 있는 process.c에서 구현한다. load()에서 argument를 띄어쓰기 단위로 parsing을 진행하고, user stack의 stack pointer에 해당하는 esp 레지스터 값을 이용하여 parsing된 argument의 내용과 그 주소값을 스택에 저장한다. 예를 들어 “/bin/ls -l foo bar”라는 argument를 입력 받았을 때, 스택 저장 결과는 다음과 같이 표현할 수 있다.

테이블이(가) 표시된 사진

자동 생성된 설명

hex\_dump() 명령어를 통한 출력은 다음과 같아야 한다.

예를 들어, “echo x”를 입력 받았을 때,

텍스트, 전자기기, 키보드이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

User Memory 영역과 Kernel Memory 영역을 구분하여 서로 다른 구역을 침범하지 않도록 관리해 준다. src/userprog의 syscall.c에서 check\_user\_vaddr()라는 함수를 만든다. check\_user\_vaddr에서 내장되어 있는 함수인 is\_user\_vaddr()을 통해 user가 kernel memory 영역을 침범하지 않았는지 확인한다. 만약 침범했을 경우에는 user program을 죽이는데, 이는 침범한 thread를 종료하고 -1을 parent thread에게 return해준다.

1. System Calls

이번 프로젝트에서는 halt, exit, exec, wait, read(stdin), write(stdout), Fibonacci, max\_of\_four\_int의 system call을 구현한다. src/userprog의 syscall.c에서 구현하며, 스택 포인터 f->esp를 통해서 접근한다. 그리고 system call 함수의 인자로 넘겨준다. 수행 결과(Return값)는 f->eax에 저장한다.

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

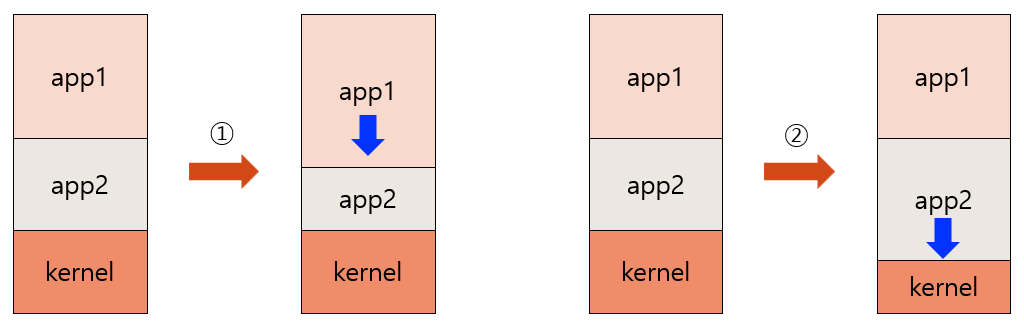
먼저, parsing을 수행해 주어야 한다. argument는 load function의 parameter인 file\_name으로 들어온다. file\_name을 통해서 들어온 argument의 개수를 세어서 argc라는 변수에 저장한다. 그 다음, parsing 완료된 argument들을 저장할 argv array를 생성한다. strtok\_r() 함수를 이용해 띄어쓰기 단위로 parsing을 하고 argv에 넣어준다. 예를 들어 ‘echo x’가 들어오면 argv[0]=”echo”, argv[1]=”x”와 같이 저장된다.

Parsing을 완료했다면, 이제 user stack을 만들어 passing을 진행해야 한다. stack에 쌓을 때, argv에 저장돼 있는 argument들을 뒤에서부터 쌓아야 한다. 스택의 top에 해당하는 esp를 argv[i]+1 크기만큼 뺀 다음, memcpy()를 통해 argv[i]의 값을 스택에 넣어준다. 그리고 해당되는 스택 element의 주소 값을 argv[i]에 저장해 둔다.

argv의 argument들을 스택에 다 쌓았다면, 이제 word size(4 byte)만큼 alignment를 수행해 준다. 이후에는 argv[i]에 저장해 뒀던 주소 값을 쌓고, argv의 주소값과 argc, 그리고 return address를 스택에 push한다.

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

Pintos는 메모리를 user memory와 kernel memory의 두 영역으로 나눈다. 이런 메모리 영역을 직접 사용하면 메모리를 관리하기가 어렵다. 예를 들어, 각 process가 서로에게 피해를 주거나, 운영 체제를 실행하는 데 중요한 kernel code를 손상시킬 수 있다. 이를 Invalid memory access라고 한다.



이러한 문제를 방지하기 위해서 운영체제는 가상 메모리(Virtual Memory) 시스템을 채택한다. 가상 메모리 덕분에 각 process는 고유한 메모리 영역을 가질 수 있으며, process가 마치 전체 메모리를 차지하는 것처럼 사용할 수 있다. Pintos 또한 가상 메모리로 메모리 영역을 관리하며, 가상 메모리는 user virtual memory와 kernel virtual memory로 나뉜다.

Pintos는 kernel에 PHYS\_BASE(3GB~4GB)의 1GB 영역을 할당한다. 또한, pintos에서의 메모리 단위는 페이지이고, 4KB의 크기를 갖는다.

* + Invalid memory access를 어떻게 막을 것인지 설명

Invaild memory access는 esp값과 kernel memory의 시작점인 PHYS\_BASE간의 비교를 통해 제어할 수 있다. src/userprog의 syscall.c에서 각 system call 함수를 호출하기 전에 is\_user\_vaddr을 이용한 wrapper함수를 만들어서 함수의 값이 false라면 커널 영역을 침범했으므로 exit(-1)을 통해서 종료시켜준다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

User mode에서는 kernel 영역을 수정 및 사용하면 안되는데, 그러면 kernel 영역에 있는 유용한 함수 및 기능을 이용하지 못한다. 이를 가능하게 하기 위해 만들어진 것이 system call이다.

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

1. SYS\_HALT

현재 돌아가고 있는 Pintos를 종료 시키는 system call이다. halt 함수를 구현하였으며 shutdown\_power\_off()라는 내장 함수를 사용했다.

1. SYS\_EXIT

현재 thread를 종료 시키는 system call이다. exit 함수를 구현하였으며 thread\_exit()라는 내장 함수를 사용하였다.

1. SYS\_EXEC

실행을 위해 새로운 thread를 생성해주는 system call이다. exec 함수를 구현하였으며 process\_execute() 함수를 호출하고 이 안에서 thread를 생성한다.

1. SYS\_WAIT

Parent thread가 child thread보다 미리 종료되지 않도록 해주는 system call이다. wait 함수를 구현하였으며 process\_wait() 함수를 사용했다.

1. SYS\_READ

file descriptor(fd)가 0인 경우(STDIN), input\_getc() 함수를 이용해 한 글자 씩 읽는다.

1. SYS\_WRITE

file descriptor(fd)가 1인 경우(STDOUT), putbuf() 함수를 이용해 출력한다.

1. SYS\_FIB

추가로 구현한 system call이고, n번째 Fibonacci 수를 구한다.

1. SYS\_MAX

추가로 구현한 system call이고, 네 수 중 가장 큰 수를 구한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명
    1. main()에서 시작되어 run\_actions(argv)를 호출한다.
    2. run\_actions(argv)안의 a->function(argv)가 run\_task를 호출한다.
    3. run\_task가 process\_wait에 들어갈 process\_execute를 호출한다.
    4. process\_execute안의 thread\_create가 user program 이름을 등록한다. 그리고 start\_process를 통해 user program을 실행시킨다.
    5. process scheduling이 호출되면 child process(user program)은 start\_process()에 의하여 실행된다. load()로 메모리를 할당 받고 사용자 프로그램을 적재한다. 만약 user program이 종료되면 exit이 호출되고 이는 process\_wait로 돌아간다.
    6. process\_execute에서 실행된 부모 process는 process\_wait에서 child process가 끝날 때까지 기다린다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**
* 9/18~9/21: Parsing 및 Argument passing 구현
* 9/22~9/26: System call 구현
* 9/27~10/1: Fibonacci 및 max\_of\_four\_int 구현 후 make check 결과 체크
  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

* userprog/process.c의 load()에 argument parsing 코드를 추가한다.
* userprog/process.c의 load()에 setup\_stack() 이후에 스택을 쌓는다.
* userprog/process.c의 process\_execute()에서 filesys\_open(name\_command)의 값이 NULL인 경우에(즉 아무런 파일도 open되지 않았을 때), return -1을 해준다.

1. User Memory Access

* userprog/expection.c의 page\_fault()에서 user가 kernel memory에 접근한 경우에는 exit(-1)를 호출한다.
* userprog/syscall.c에서 is\_user\_vaddr()을 이용한 wrapper 함수인 check\_user\_vaddr() 함수를 통해 user memory 영역인지 지속적으로 check해준다.

1. System Calls

* userprog/syscall.c에서 syscall\_handler()에 switch문을 만들어서 src/lib/syscall-nr.h에 있는 system call number에 따라 해당되는 system call function을 호출한다.
* userprog/syscall.c에서 system call function을 구현한다. 구현한 함수들은 다음과 같다. (추가구현 포함)

**void halt(void);**

**void exit(int status);**

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

**pid\_t exec (const char \*file);**

**int wait (pid\_t);**

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

**int fibonacci(int n);**

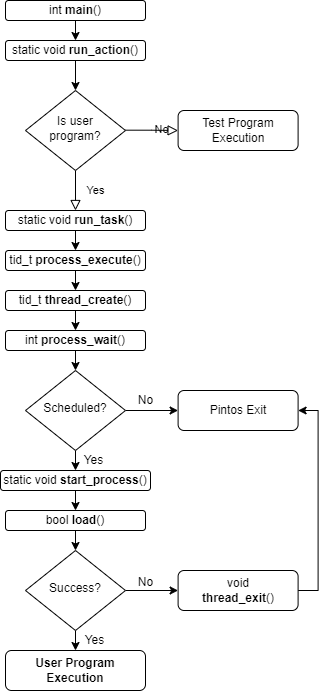
**int max\_of\_four\_int(int a, int b, int c, int d);**

* userprog/syscall.h에 위 함수들을 prototyping해준다.
* 추가 구현을 위해 lib/syscall-nr.h에 SYS\_FIB, SYS\_MAX의 system call number를 추가하고 lib/user/syscall.h에 system call API의 prototyping을 해준다. 또한, lib/user/syscall.c에 syscall4를 새로 define하고 system call API도 정의한다.
* exit(), wait()를 수행하기 위해 threads/thread.h에서 struct thread의 element들을 수정해 주는데, 이는 thread concurrency를 위해 semaphore를 설정하고 exit\_status를 추가해준다.
* threads/thread.c에서 sema\_init을 수행한다.

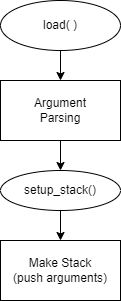
1. **연구 결과**
   * **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

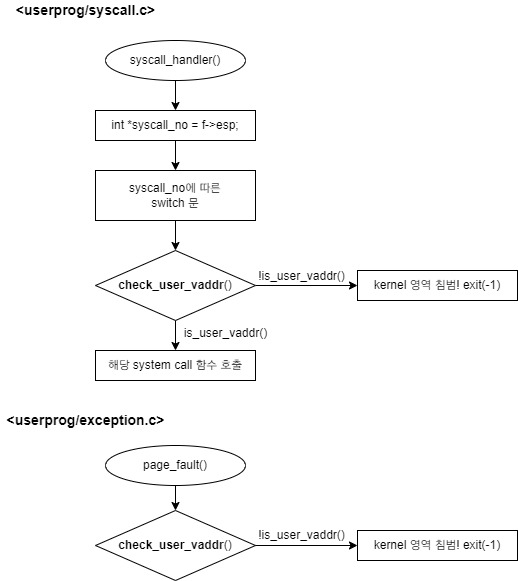
1. 전체적인 Flow



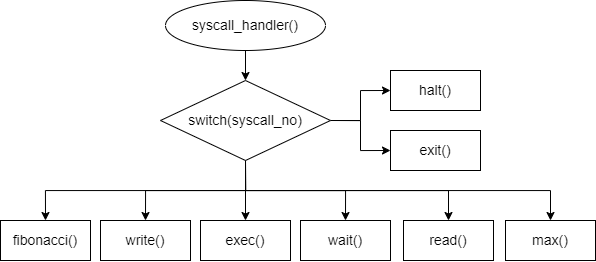
1. Argument Passing



1. User Memory Access



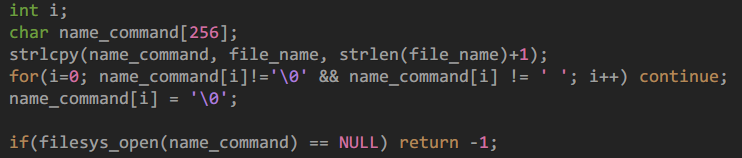
1. System Calls



* + **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

* userprog/process.c의 process\_execute()에서 파일 이름을 받고, filesys\_open했을 때 NULL이면 return -1 처리한다.



* userprog/process.c의 load()에서 parsing을 진행한다. 먼저, 다음 코드로 argc의 값을 구한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* 이후, argv 배열을 malloc해준 후, strtok\_r을 이용하여 공백을 기준으로 parsing을 수행한다.
* parsing이 끝나면, t->name에 argv[0], 즉, process의 이름을 넘겨주고, filesys\_open()을 수행한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

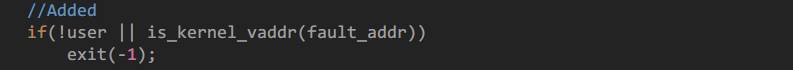
* userprog/process.c의 load()에서 setup\_stack()이 끝난 후 이전에 설명한 대로 stack을 쌓아준다. 모두 쌓은 후에는 hex\_dump()를 통해 스택이 잘 쌓여있는지를 확인한다. (주석처리 부분)

텍스트이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

* serprog/expection.c의 page\_fault()에서 user가 kernel memory에 접근한 경우에는 exit(-1)를 호출한다.



* serprog/syscall.c에서 is\_user\_vaddr()을 이용한 wrapper 함수인 check\_user\_vaddr() 함수를 통해 user memory 영역인지 지속적으로 check해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
* userprog/syscall.c에서 syscall\_handler()에 switch문을 만들어서 src/lib/syscall-nr.h에 있는 system call number에 따라 해당되는 system call function을 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* halt(): 현재 돌아가고 있는 Pintos를 종료 시키는 system call 함수이다. shutdown\_power\_off()라는 내장 함수를 사용했다.

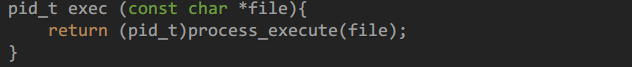


* exit(): 현재 thread를 종료 시키는 system call 함수이다. thread\_current()-> exit\_status의 값에 parameter로 들어온 status를 넣어준다. thread\_exit()라는 내장 함수를 사용하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* exec(): 실행을 위해 새로운 thread를 생성해주는 system call이다. process\_execute() 함수를 호출하고 이 안에서 thread를 생성한다.



* wait(): Parent thread가 child thread보다 미리 종료되지 않도록 해주는 system call이다. 여기선 pid를 받아 process\_wait()으로 넘겨주는데, 이는 종료된 child thread의 식별자이다. process\_wait()에서는 이를 입력받아 해당 pid를 갖는 child thread를 찾고, 만약에 찾을 수 없다면 -1을 return한다. 이후에 child가 process\_exit에서 sema\_up(&cur->child\_lock)으로 lock을 풀면 process\_wait의 sema\_down(&(found->lock\_child))에서 다시 코드가 진행되고 list\_remove를 진행한다.
* 하지만 계속해서 프로그램이 죽는 현상이 발생하였다. 그 이유를 곰곰히 생각해보니, child process가 process\_exit에서 sema\_up을 하는 순간, parent process는 list\_remove를 수행하는데, 이미 child process가 죽었으므로 list\_remove에 필요한 t가 메모리에 남아있지 않아서 그런 것 같았다.
* 그래서 lock\_memory라는 새로운 semaphore를 하나 더 설정해 주었다. process\_exit에서 child process가 lock\_child를 sema\_up하는 동시에 lock\_memory를 sema\_down함으로써 list\_remove에 필요한 메모리가 사라지지 않도록 해준다. 그런 다음 process\_wait에서 list\_remove()한 후에 lock\_memory를 sema\_up해주고 모든 과정이 끝난다.
* threads/thread.h에서 struct semaphore lock\_child, lock\_memory를 해주고, threads/thread.c의 init\_thread()에서 sema\_init으로 초기화해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명



텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

* read(): file descriptor(fd)가 0인 경우(STDIN), input\_getc() 함수를 이용해 한 글자 씩 읽는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* write(): file descriptor(fd)가 1인 경우(STDOUT), putbuf() 함수를 이용해 출력한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
* userprog/syscall.c에서 Fibonacci 및 max\_of\_four\_int 함수 구현

텍스트이(가) 표시된 사진

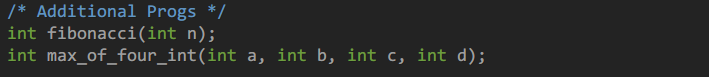
자동 생성된 설명

* lib/syscall-nr.h에 system call number 추가

텍스트이(가) 표시된 사진

자동 생성된 설명

* lib/user/syscall.h에 system call API prototype 추가



* lib/user/syscall.c에 syscall4()를 새로 정의하고 system call API도 새로 정의

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

* src/examples에 makefile 수정 및 additional.c를 생성



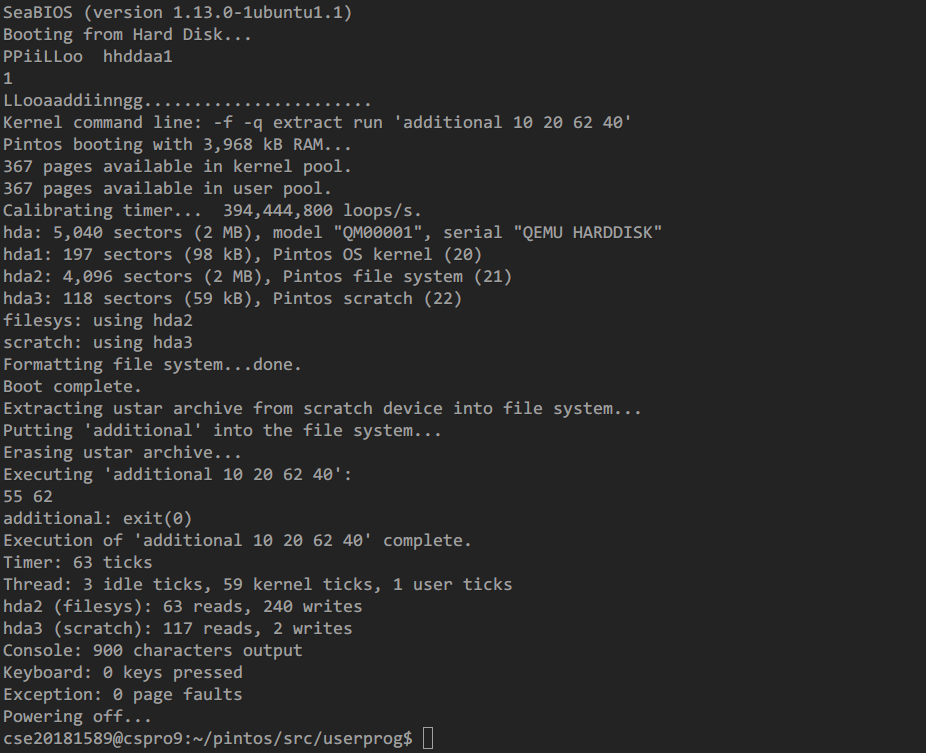
텍스트이(가) 표시된 사진

자동 생성된 설명

* 이제 아래 명령어로 pintos에서 ‘additional’이 실행 가능하다.

**pintos/src/userprog$** pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run 'additional 10 20 62 40'

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

****