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

담당 교수 : 김영재

조 / 조원 : 20171659 유경연

개발 기간 : 9/12~9/26

1. **개발 목표**

Pintos의 가장 기본적인 기능들을 구현한다. halt, exit, exec, wait, read(stdin), write(stdin) 등의 System Call이 작동할 수 있도록 코드를 작성한다. 추가적으로 새로운 System Call(fibonacci, max\_of\_four\_int)도 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* Argument Passing
  + 입력된 Argument를 Stack에 쌓아서 프로그램이 작동할 수 있도록 한다.
* User Memory Access
  + User Memory와 Kernel Memory를 구분하여 Invalid Memory Access가 발생하는 것을 막는다.
* System Calls
  + 이번 프로젝트에서 구현해야 하는 System Call들이 작동할 수 있도록 한다.
  1. **개발 내용**
* Argument Passing
  + 입력된 Argument를 공백을 단위로 하여 Parsing하고 그 값들을 Stack에 쌓는다. Word-align을 통해 Stack의 크기를 4의 배수로 크기를 맞추고 Parsing된 Argument들의 주소 값을 Stack에 쌓는다. Argument의 개수와 Return address도 Stack에 쌓아준다.
* User Memory Access
  + Pintos의 Memory는 크게 나누면 User Memory와 Kernel Memory 둘로 나뉜다. User mode일때는 Kernel Memory에 접근하려 한다면 Invalid memory access가 발생한다.
  + 참조할 Address가 User Memory인지 미리 검사하고 만약 아니라면 Invaild memory access이므로 해당 메모리를 참조하지 않는다.
* System Calls
  + User는 Kernel Memory에 접근할 수 없으므로 User는 Kernel이 제공하는 기능들을 직접 사용할 수 없다. 이 기능들을 사용할 수 있도록 하기 위해 System Call이 필요하다.
  + 이번 프로젝트에서 개발할 System Calls
    - **void halt (void)**

Pintos를 종료한다.

* + - **void exit (int status)**

현재 실행중인 User program을 종료한다.

* + - **pid\_t exec (const char \*cmd\_line)**

입력으로 주어진 실행파일을 실행하고 새로운 process의 program id를 return한다.

* + - **int wait (pid\_t pid)**

자식 process가 종료될 때까지 기다리고 자식 process의 exit status를 가져온다.

* + - **int read (int fd, void \*buffer, unsigned size)**

fd로 열린 file을 size byte만큼 읽어서 buffer에 저장한다.

* + - **int write (int fd, void \*buffer, unsigned size)**

buffer를 fd로 열린 file에 size byte만큼 출력한다.

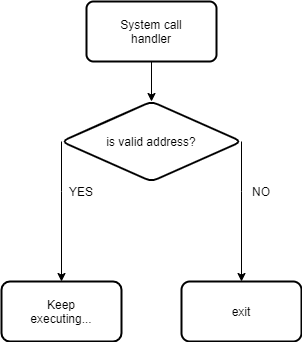
* + User program에서 System Call을 호출하면 src/lib/user/syscall.c에 있는 System Call 함수가 호출된다. 그 함수가 호출되면 int $0x30을 수행하고 Interrupt Vector table의 0x30번지는 System Call handler의 주소이므로 src/userprog/syscall.c의 syscall\_handler()가 호출된다. syscall\_handler()는 System Call number에 맞는 System Call 함수가 호출하고 작업이 수행된다. 작업이 완료되면 eax에 return값을 저장하고 다시 User level로 돌아온다.

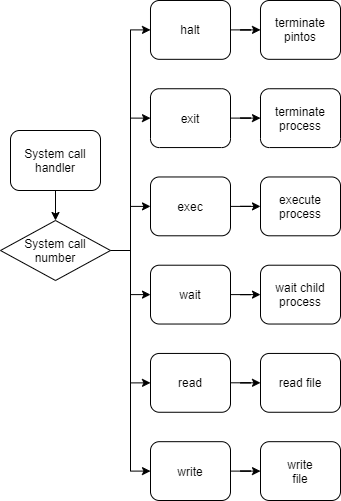
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 9/12
  + Visual Studio Code를 이용한 원격 개발 환경을 구축하였다.
  + 에러가 발생하여 cspro 서버에 Pintos를 재설치했다.
* 9/14
  + Argument를 Parsing하고 stack에 쌓아 Passing까지 하는데 성공하였다.
* 9/16

Exit와 write를 구현하였다. Ppt에 예시로 주어진 echo x가 정상적으로 실행되었다.

* 9/19
  + 남은 System Call을 구현하려 하였으나 잘 되지 않았다.
* 9/20
  + User Memory Access 구현에 성공하고 남은 System Call도 모두 구현 성공하였다.
  + Make check시 80개 test 중에 37개만 실패했다.
* 9/21
  + 추가적으로 Additional System Call 구현하는데 성공했다.
* 9/22
  + 개발과정에서 작성한 주석들을 삭제하고 코드를 다듬었다.
* 9/26
  + 보고서를 작성하고 제출용 파일을 만들었다.
  1. **개발 방법**
* Argument Passing
  + 수정할 소스코드: src/userprog/process.c
  + Load()함수의 인자로 들어온 file\_name을 Parsing하고 esp Stack에 쌓는다.
* User Memory Access
  + 수정할 소스코드: src/userprog/syscall.h, syscall.c
  + address가 valid한지 검사해주는 check\_vaddr()함수를 추가해서 Memory에 접근할 때마다 확인해준다.
* System Calls
  + 수정할 소스코드: src/userprog/syscall.h, syscall.c, process.c
  + Syscall\_handler() 함수를 수정해서 입력에 따라 함수를 호출할 수 있도록 수정하고 각 System Call에 맞는 함수들을 추가한다. 또한 그 기능들을 구현하기 위해서 process.c에 추가 수정을 해준다.

1. **연구 결과**
   1. **Flow Chart**
2. **텍스트이(가) 표시된 사진

   자동 생성된 설명**Argument Passing 2. User Memory Access
3. System Calls
   1. **제작 내용.**
4. Argument Passing
   * 우선 Argument를 Program name과 Argument들로 Parsing하기 위해 load() 함수의 filename이라는 함수인자를 공백을 기준으로 Parsing한다. 이때 pintos에서 제공하는 strtok\_r()을 사용한다. 또한 file\_name를 수정하지 않기 위해 strlcpy()를 이용하여 복사한 문자열을 Parsing하는데 사용한다. Parsing한 문자열은 argv를 동적 할당하여 저장한다. Parsing하면서 Argument의 개수(argc)도 알아낸다.
   * 문자열 처리가 제대로 됐는지 확인해보려 했는데 아직 process\_wait()함수가 구현되지 않아서 출력이 나오지 않았다. 그래서 일단 for loop를 10억번 돌려서 강제로 기다리게 하였다.
   * Parsing된 인자들을 초기화된 esp Stack에 쌓는다. 이때 먼저 Argument의 값을 strlcpy()를 이용해 Stack에 쌓으면서 Stack의 길이를 알아낸다. Argument를 다 쌓았다면 Stack의 길이를 4의 배수로 맞춘다. 이후 NULL문자, 주소 값, argc, return address를 순서대로 Stack에 쌓는다. Return address는 가짜 주소를 이용한다.
   * 함수 구현 이후 hex\_dump()를 이용하여 Argument Passing이 정상적으로 작동하는 것을 확인하였다.
5. User Memory Access
   * Kerel Memory에 Access하는 것을 막기 위해 check\_vaddr()이라는 함수를 작성하였다. 이 함수는 pintos/src/thread/vaddr.h에 정의되어 있는 is\_user\_vaddr()함수를 호출한다.
   * is\_user\_vaddr()함수는 입력 받은 인자 vaddr이 PHYS\_BASE보다 작으면 true를return해준다. 이때 return된 값이 true가 아니라면 Invalid memory access이므로 exit()함수를 호출하여 process를 종료한다.
   * 이 함수를 만들고 난 후에 make check를 해보니 pass가 많이 늘어나긴 하였지만 모든 test case가 통과되지는 않았다.
   * src/userprog의 exception.c의 page\_fault()함수에서 출력으로 page\_fault가 뜨지 말아야할 조건을 걸어주었더니 해결되었다.
6. System Calls
   * src/userprog/syscall.c에서 pid\_t를 사용하려고 하니 에러가 발생했다. 그래서 lib/user/syscall.h을 include 시켜주었다.
   * **void halt (void)**

Pintos를 종료하기 위해 pintos/src/devices/shutdown.c에 정의되어 있는 shutdown\_power\_off()를 사용한다.

* + **void exit (int status)**

처음엔 종료문과 thread/thread.c에 정의되어 있는 process\_exit()만 호출하였다. 종료는 정상적으로 되었지만 Process의 이름인 thread\_name()이 Parsing되어있지 않은 상태로 출력되어서 userprog/process.c의 process\_excute()에서 file\_name을 Parsing해주었다. 또한 file open이 정상적으로 되지 않았을 때의 예외처리를 해주었다.

이후 wait()를 구현하고 나서 현재 thread의 exit status를 갱신하고 thread\_exit()함수를 호출하도록 수정했다.

* + **pid\_t exec (const char \*cmd\_line)**

src/userprog/process.c의 process\_execute()함수를 호출하고 생성한 process의 pid를 return해준다. 이때 적절한 synchronization이 필요하다.

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

src/thread/synch.h에 정의되어 있는 semaphore 구조체를 이용해 부모 process에서 호출 시 자식 process의 종료를 기다리도록 구현한다. 우선 thread.h에 정의되어 있는 thread 구조체에 semaphore와 자식 process를 저장하기 위한 list를 추가하고 thread.c의 init\_thread()에서 초기화한다. 이때 semaphore의 값은 0으로 초기화한다. wait()함수에서는 src/userprog/process.c에 정의된 process\_wait()를 호출한다. 이 함수에서는 현재 thread의 자식 thread의 tid를 모두 확인하고 만약 함수인자로 들어온 child\_tid와 같다면 src/threads/synch.c의 sema\_down()을 호출한다. 이 함수는 sema->value가 0인 동안에 thread\_block()을 호출한다. 이전에 0으로 초기화 했기 때문에 이후 자식 thread에서 process\_exit()의 sema\_up()이 호출되기 전까지 부모 thread는 thread\_block()이 실행된다. sema\_up()이 호출되면 exit\_status를 저장하고 list에서 자식 thread를 삭제한다. 자식 thread가 종료되기 전에 list에서 삭제되는 것을 보장하기 위해서 thread 구조체에 semaphore sema\_alive를 추가한다. 자식 thread가 종료되기 전 sema\_down()을 호출하고 부모 thread가 list\_child에서 자식 thread를 삭제할 수 있도록 한다. 이후 부모 thread에서 sema\_up()을 호출해서 자식 thread가 종료된다.

* + **int read (int fd, void \*buffer, unsigned size)**

fd가 0일 경우에 input\_getc() 함수를 이용해 키보드 입력을 받도록 구현했다. fd가 0이 아닌 경우엔 -1을 return한다.

* + **int write (int fd, void \*buffer, unsigned size)**

fd가 1일 경우에 putbuf() 함수를 이용해 console 출력을 하도록 구현했다. fd가 1이 아닌 경우엔 -1을 return한다.

1. Additional System calls
   * **src/lib/syscall-nr.h**
     + enum 추가: 새로운 System Call에 대한 number를 만들어주었다. (SYS\_FIBO, SYS\_MOF)
   * **src/lib/user**
     + syscall.h 변경: 새로운 System Call 함수 두개의 prototype을 작성하였다. (fibonacci(), max\_of\_four\_int())
     + syscall.c 변경: 새로운 System Call 함수 두개를 작성하고 syscall4에 대한 assembly code를 작성하였다. (fibonacci(), max\_of\_four\_int())
   * **src/examples**
     + additional.c 추가: 실행파일을 만들기위해 c파일을 생성하였다. 여기서 fibonacci와 max\_of\_four\_int를 호출한다. 이때 문자열을 정수로 바꾸기 위해서 str\_to\_int()를 정의해주었다.
     + Makefile 변경: make시에 additional.c도 포함시키기 위해 Makefile을 변경하였다.
   * **src/userprog/syscall.c**
     + int fibonacci(int n) 추가: 인자로 n을 받아서 피보나치의 수열의 n번째 숫자를 return해주는 함수를 작성하였다.
     + int max\_of\_four\_int(int a, int b, int c, int d) 추가: 인자로 정수 4개를 입력 받아서 그 중에 가장 큰 숫자를 return해주는 함수를 작성하였다.
     + void syscall\_handler() 변경: 새로운 System Call number에 대한 case문을 작성하고 함수를 호출하였다. 이때 함수의 인자로 넘기는 주소 값을 찾기 위해 시간이 많이 필요했다.
   1. **시험 및 평가 내용**

* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

