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

담당 교수 : 박성용 교수님

조 / 조원 :20181593 계인혜

개발 기간 : 20.10.20~20.11.01

1. **개발 목표**

Pintos OS에서 user program 구동을 가능하게 하는 것이다. 또한 pintos 내부에 구현되어 있는 부분을 이해하고, process, thread, synchronization을 적절하게 구현한다. Argument를 주어진 형식에 맞게 잘라 스택에 삽입하고 이를 이용하여 process와 thread를 execute한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

사용자가 실행하고자 하는 파일명과 argument를 입력하면, 파일명에 해당하는 프로그램을 로드하여 argument에 맞는 작업을 수행하고 종료한다**.**

1. Argument Passing

입력으로 들어올 수 있는 다양한 argument를 공백을 기준으로 잘라 80x86 calling convention에 맞게 스택에 넣는다. 이때 ESP 포인터를 이용하여 아래에서부터 값을 채워나간다. /userprog/process.c에 이를 구현하였다. 이를 이용하여 실행된 프로그램이 정상적으로 작동하도록 한다.

1. User Memory Access

User program은 kernel과 진행중인 프로세스를 방해하면 안 된다. 즉, 메모리에 적절하게 접근하는지 확인이 필요하다. 사용자 모드에서 커널 영역에 접근하거나, 프로세스가 할당 받은 영역의 바깥부분에 접근하는 경우 접근이 불가능하게 만들어야 한다. 그렇지 않으면 page fault가 발생하기 때문이다. 따라서 이 부분에서는 허용되지 않은 메모리의 접근을 방지한다.

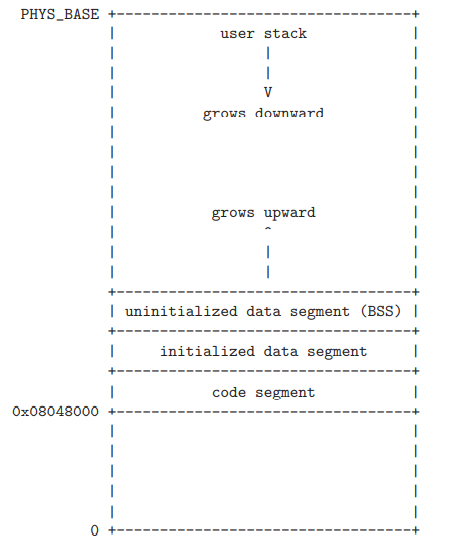
1. System Calls

User program 내의 system call 호출을 위한 system call handler를 구현한다. 또한, 명세서에 제시되어 있는 halt, exit, exec, wait, read, write system call function을 구현한다. 마지막으로, parent thread가 child thread의 exit 전까지 wait를 할 수 있도록 하는 synchronization도 semaphore 기법을 사용하여 구현했다.

* 1. **개발 내용**
* Argument Passing

Ppt에 있는 예시로 argument parsing에 대해 살펴보도록 하자. ‘/bin/ls -l foo bar’라는 argument가 들어오면 공백을 기준으로 잘라 각 단어를 저장한다. Argc는 문자열의 개수, Argv에는 문자열이 저장된다. 예시와 같은 경우 Argc = 4이고, Argv[0] = “/bin/ls\0”, argv[1] = “-l\0”, argv[2] = “foo\0”, argv[3] = “bar\0”으로 저장된다. 다음은 스택에 파싱한 argument를 쌓는 과정에 대하여 살펴보도록 하자. (manual 3.5 80x86 calling convention 참고)

* + 1. 스택의 제일 윗부분에 argv를 차례로 push한다. 이때, 스택의 증가방향은 아래이다.
    2. Word align을 위해 esp가 4의 배수가 아닌 경우, esp의 값을 4의 배수가 되도록 수정한다.
    3. ①에서 삽입한 argv의 주소를 차례로 스택에 push한다.
    4. Argc를 스택에 push 한다.
    5. Return address를 스택에 push한다. 이때, return address는 child process가 종료되었을 경우, parent process로 돌아가기 위한 것이다.
    6. 스택의 esp는 ⑤의 Return address를 담고 있는 스택의 주소를 가리킨다.
* User Memory Access

다음은 user virtual memory layout이다. 그림과 같이 user는 user stack에만 접근할 수 있다. Invalid memory access는 다음과 같은 경우를 의미한다.

1. User virtual memory가 아닌 부분에 접근하는 경우
2. Return address 보다 윗부분에 접근하는 경우
3. 0x08048000보다 아래의 메모리에 접근하는 경우

만약 프로그램 실행 중 위와 같은 invalid memory access를 하게 되면 프로세스를 종료시키고, exit status = -1로 설정한다. Parent process는 이를 보고 비정상적으로 종료되었음을 알 수 있다. 또한, SYSCALL\_HANDLER를 다루는 /userprog/syscall.c에서 메모리에 접근하기 전에 매번 isMemoryValid라는 함수를 이용하여 valid memory access인지 확인한다.

* System Calls

시스템 콜은 운영체제의 커널이 제공하는 서비스에 대해 API가 요청한 대로 커널에 접근하기 위한 것이다. 직접 시스템 호출을 사용할 수 없기 때문에 응용프로그램에서 커널의 서비스를 사용한다. 이 과정에서 system call이 필요하다.

1. SYS\_HALT : 프로그램을 종료시키는 함수로, shutdown\_power\_off를 수행한다.
2. SYS\_EXIT : 현재 진행중인 thread를 종료하는 함수로, parent에게 status를 알려주며 child thread가 끝났음을 semaphore를 이용하여 알린다. 이후 process\_exit을 호출한다.
3. SYS\_EXEC : process\_execute 함수를 이용하여 cmd\_line을 수행한다.
4. SYS\_WAIT : parent thread가 child thread를 기다리도록 하는 함수로 process\_wait을 사용한다.
5. SYS\_READ : 이번 프로젝트에서는 fd =0(stdin)의 경우만 고려하여 buffer의 처음부터 size 만큼 입력을 받아온다. 그 전에 엔터나 NULL이 입력되면 read를 종료한다. input\_getc 함수를 사용한다.
6. SYS\_WRITE : 이번 프로젝트에서는 fd =1(stdout)의 경우만 고려하여 buffer의 처음부터 size만큼을 출력한다. 그 전에 엔터나 NULL이 출력되면 write를 종료한다. putbuf 함수를 사용한다.
7. SYS\_FIBO : Fibonacci 수열의 n 번째 항을 리턴한다.
8. SYS\_MAXINT : 입력 받은 a, b, c, d 네 개의 정수 중 가장 큰 값을 리턴한다.

유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지의 과정은 다음과 같다.

1. User program이 system call function을 호출한다.
2. System call #와 argument가 stack에 쌓인다.
3. 0x30으로 system call을 위한 interrupt를 건다.
4. 스택을 설정하고, interrupt handler를 호출한다.
5. Intr\_handler가 system call handler를 호출한다.
6. Syscall\_handler가 control을 가지고, esp를 통해 스택에 접근할 수 있다. Return value를 eax 레지스터에 저장한다.
7. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/20~10/26 : ppt 및 Pintos 매뉴얼 분석

10/26~10/26 : Argument passing 구현

10/27~10/28 : User Memory Access & System call Handler 구현

10/29~10/30 : System call Implementation & Additional Implementation 구현

10/31~11/01 : Document 작성

* 1. **개발 방법**
* Argument Passing

/userprog/process.c의 load함수 내에서 strtok\_r()함수를 이용하여 공백을 기준으로argument를 parsing한다. 이후, 80x86 calling convention에 맞춰 스택을 구성한다.

* User Memory Access

Esp의 값이 NULL이거나, user virtual address의 시작인 0x08048000보다 작거나, PHYS\_BASE보다 크면 invalid memory access로 처리하였다. 이는 userprog/syscall.c의 isMemoryValid()라는 함수로 처리하였다.

* System calls

Userprog/syscall.c의 syscall\_handler라는 함수에서 struct intr\_frame \*f(인터럽트 프레임)를 받는다. \*(F->esp)는 system call #를 의미하며 이를 이용하여 switch 문으로 각각의 system call function을 처리하였다. 이때, system call #는 /lib/syscall-nr.h에 enum 으로 구현되어 있다.

System call function에 argument를 전달하기 위해서는 스택에 접근해야 한다. 위에서 말했듯 \*(f->esp) = system call #이기 때문에, \*(f->esp + 4)로 첫번째 argument에 접근할 수 있었다. 같은 방식으로 두 번째 argument는 \*(f->esp + 8)로 접근 가능하다. 이때 System call function의 결과 값은 f->eax에 저장한다.

각각의 system call은 /userprog/syscall.h, /userprog/syscall.c에 구현하였다. 추가 system call function인 fibonacci와 max\_of\_four\_int 역시 마찬가지이다. 추가 구현을 위하여 /lib/syscall-nr.h의 enum에 SYS\_FIBO, SYS\_MAXINT를 추가하였으며 /lib/user/syscall.c에 두 함수를 호출할 수 있도록 구현하였다. 이때, pintos에는 argument가 최대 4개까지 수용가능한데, max\_of\_four\_int 같은 경우에는 총 5개의 argument가 필요하다. 따라서 이를 위해 system call #를 invoke하는 syscall4라는 것을 추가하였다. 이 안에서 asm volatile 구문을 사용하여 argument의 parsing과 return value를 처리한다.

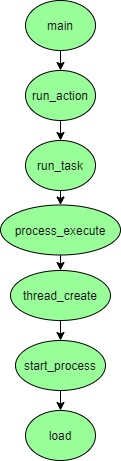
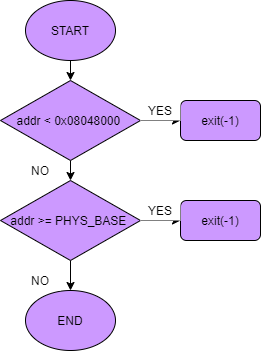
* Synchronization

부모 프로세스가 자식 프로세스가 terminate될 때까지 기다려야 한다. 이를 위해서는 부모 스레드가 자신의 자식 스레드를 구분할 수 있어야한다. 그러나 pintos에서는 이런 부분에 대해서는 구현되어 있지 않았다. 나는 semaphore를 이용하여 이를 구현하였다.

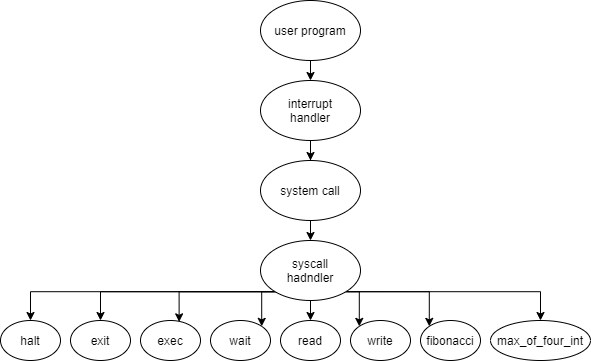
/userprog/threads/thread.h에 있는 struct thread에 다음과 같은 element를 추가하였다.

|  |  |
| --- | --- |
| int exit\_status; | 자식 프로세스의 exit status를 저장한다. |
| struct thread\* parent; | 부모 스레드의 포인터를 저장한다. |
| struct list child\_list; | 자식 스레드의 리스트이다. |
| struct list\_elem child\_elem; | 현재 스레드의 부모 스레드의 child\_list에 자신을 추가하기 위한 list\_elem이다. |
| struct semaphore sema; | 동기화를 위한 변수로, synch.h에 정의되어 있는 sema\_up, sema\_down 함수를 사용한다. |
| struct semaphore die\_sema; |

1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing 2. User Memory Access



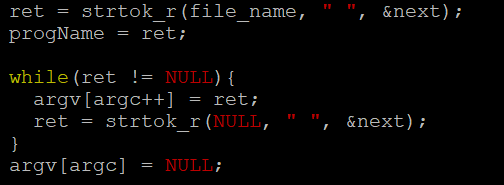
1. System Calls



* 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.**

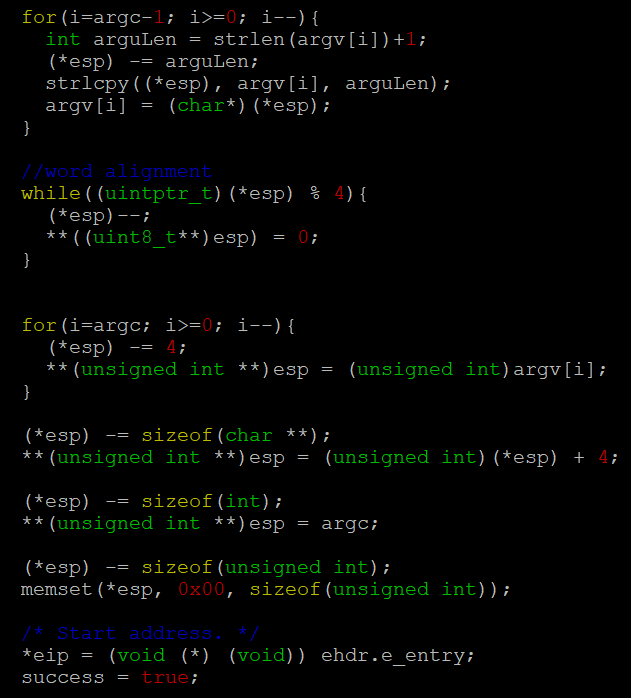
1. Argument Passing

* Userprog/process.c -> parsing(load 함수 내에 구현)



공백을 기준으로 자른 첫 부분을 progName에 저장하고, 남은 부분 역시 공백을 기준으로 argv에 저장한다. Argc에는 프로그램 이름을 제외한 argument 개수가 저장된다.

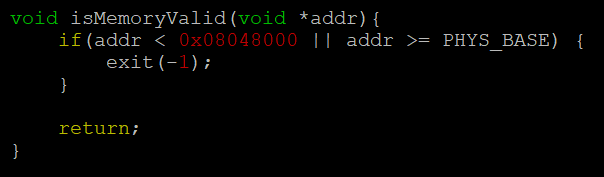
* Userprog/process.c -> setting stack(load 함수 내에 구현)



Argument의 끝에 NULL 문자를 넣어야 하기 때문에 길이를 맞추고, 스택 포인터를 argument의 길이에 맞게 조절한 다음 argument의 값을 스택에 넣는다. 이후 4의 배수가 아닌 경우에 word align을 실시하는데 이 값은 NULL로 설정한다. 그 다음으로는 각 argument의 주소 값을 저장한다. Argv의 주소를 저장하고, 마지막으로 argc의 값을 스택에 저장한다.

1. User Memory Access

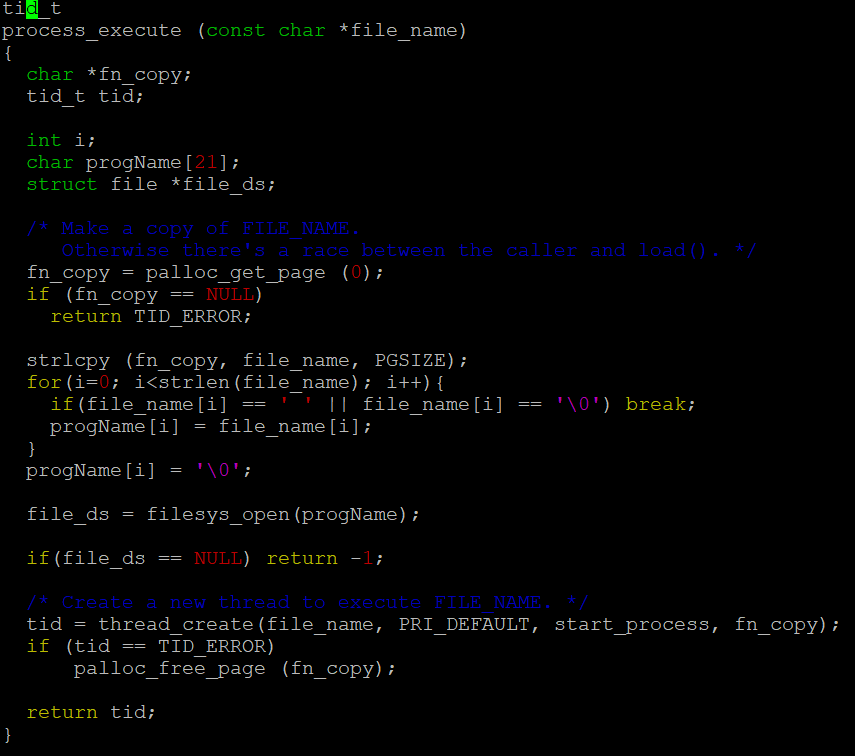
* Userprog/syscall.c -> isMemoryValid



Invalid memory access를 막기위한 함수이다.

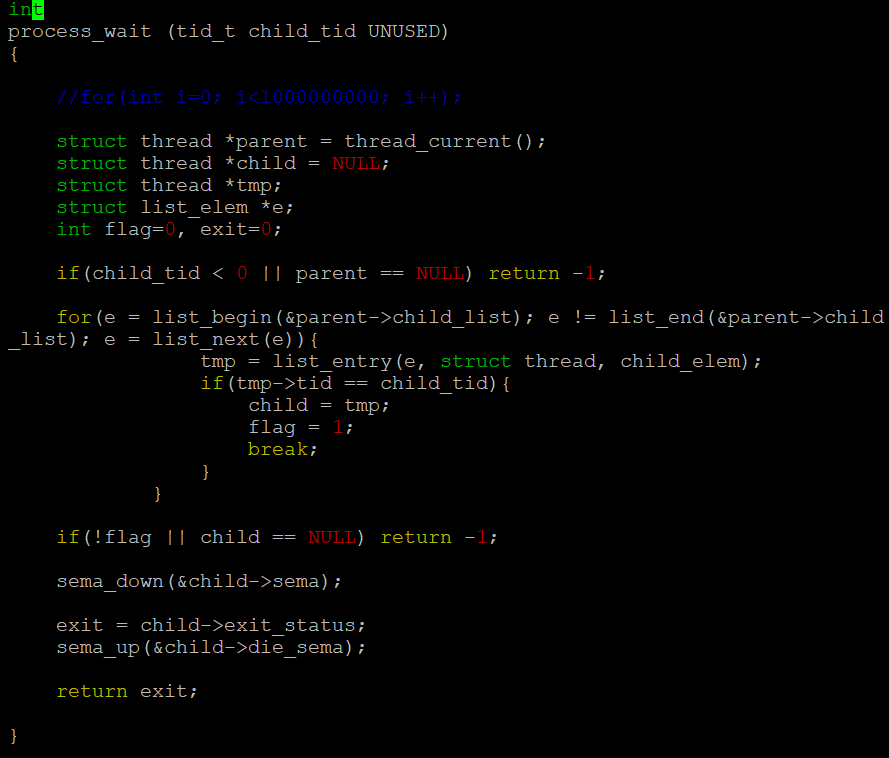
1. System Calls

* Userprog/process.c -> process\_execute



입력받은 file\_name을 이름으로 갖는 thread를 생성한다. 공백과 NULL문자를 기준으로 parsing하여 progName에 이름을 저장한 후 NULL문자를 삽입한다. 문제 없이 progName이 잘 열리면 thread를 생성하고, 지금 실행중인 thread의 tid를 새로 생성한 thread의 parent thread로 설정한다.

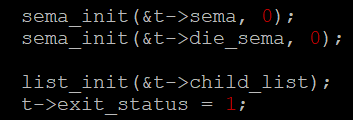
* Userprog/process.c -> process\_wait



Wait를 구현하기 위해서는 synchronizaion이 필수적이다. 이를 semaphore를 이용하여 구현하였으며, pintos 내부에 저장되어 있는 synch.c 파일을 이용하였다. 기다리고자 하는 child thread의 tid를 받아 자신의 자식 thread들 중에 해당 tid가 있는지 확인한다. Parent thread는 자신이 직접 생성한 자식 thread만을 기다릴 수 있으므로 확인 후 발견되지 않으면 -1을 리턴한다.

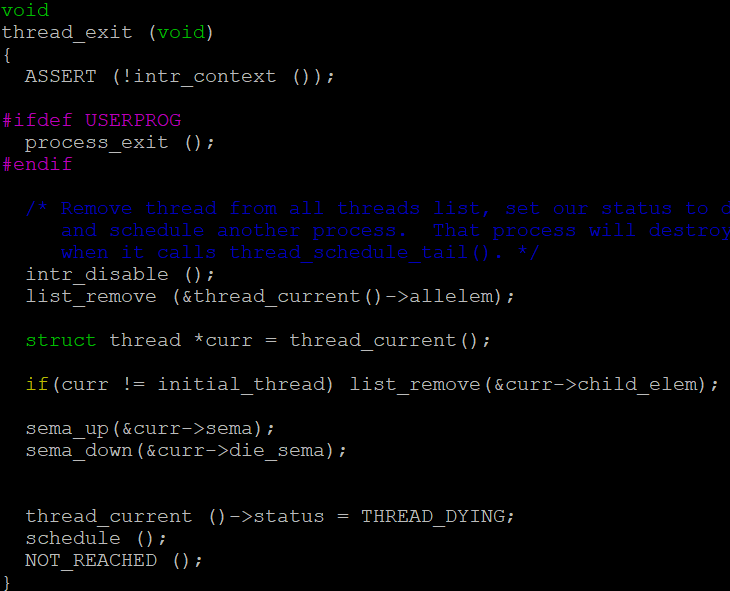
기다리고자 하는 child thread를 찾은 경우, child thread가 종료되기 전까지 parent thread를 block한다. 만약 child thread가 끝난다면, sema\_up함수를 사용하여 parent thread를 다시 풀어준다. 이때, child thread의 exit status를 parent thread에도 전달하였다.

* threads/thread.c -> init\_thread



Struct thread에 새로 추가한 element 들을 초기화 해준다.

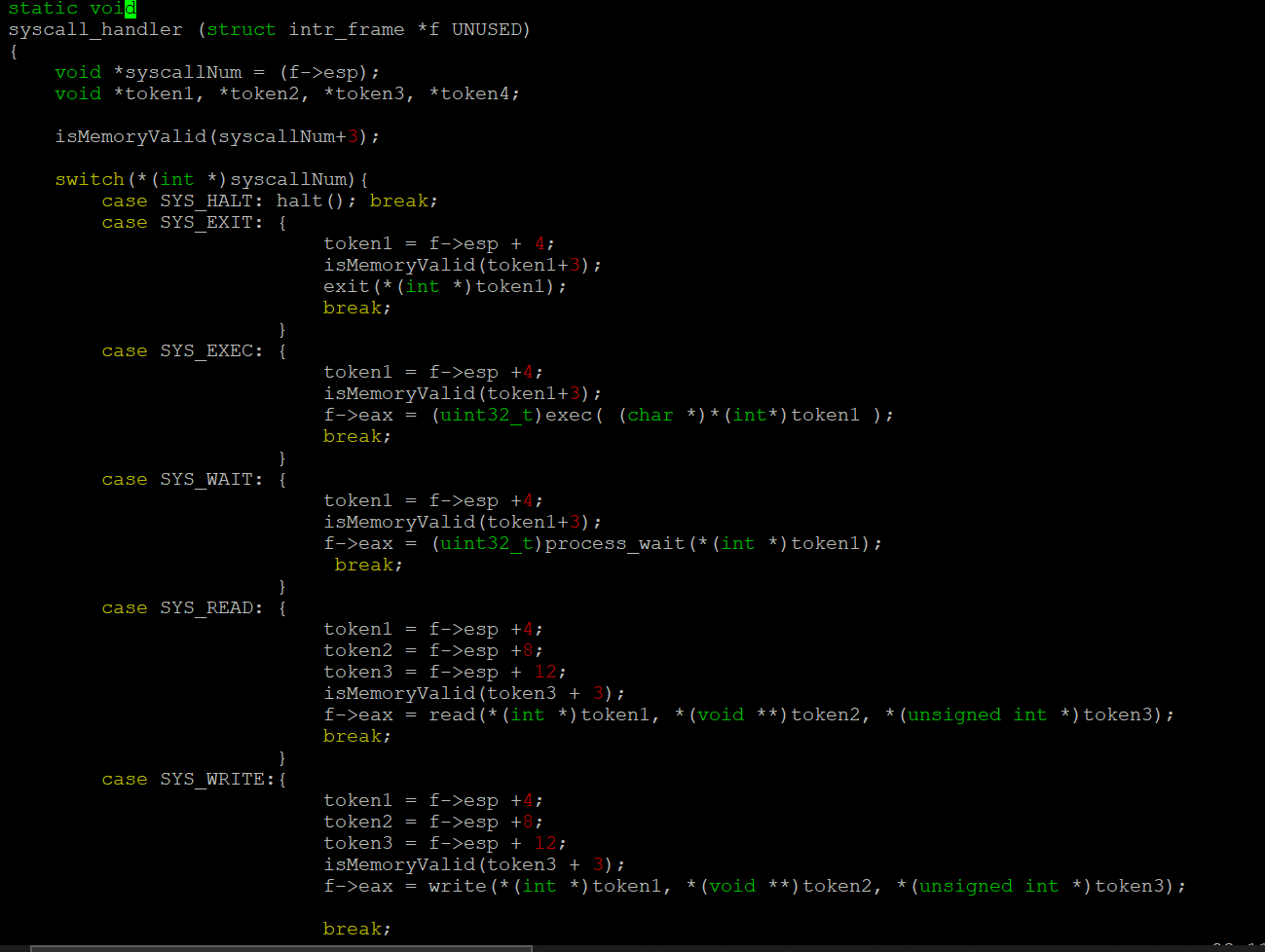
* threads/thread.c -> thread\_exit



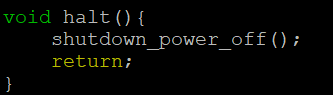
스레드가 종료될 때, 부모 스레드의 wait을 종료하기 위하여 semaphore의 값을 조정하는 코드를 추가하였다.

* Userprog/syscall.c ->syscall\_handler

Interrupt가 걸리면 \*(f->esp)의 값을 확인하여 해당하는 system call을 호출한다. 또한, f->esp를 이용하여 스택에 저장된 argument를 system call의 parameter로 넘겨준다. 이때, 결과 값은 pintos manual에 의해 eax 레지스터에 저장한다. 마지막으로 각각의 system call function을 호출하기 전에 invalid memory access가 아닌지 isMemoryValid 함수로 확인해준다.



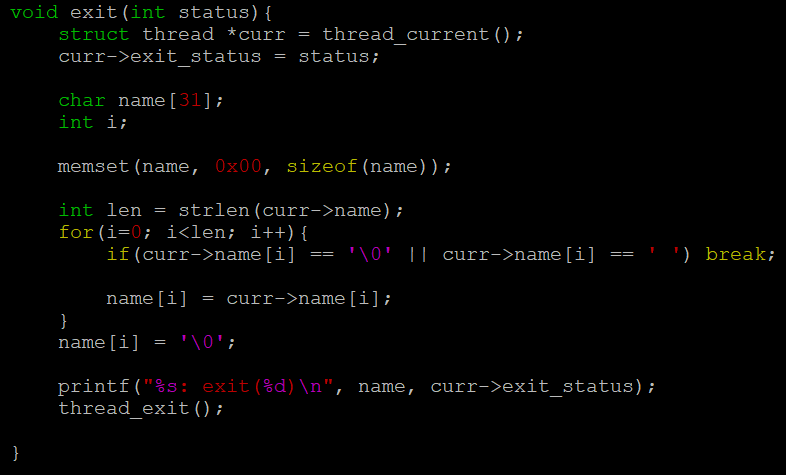
* Userprog/syscall.c ->halt



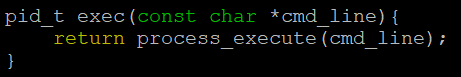
Pintos 프로그램을 종료하는 system call로, 내장 함수인 shutdown\_power\_off를 호출한다.

* Userprog/syscall.c ->exit

현재 수행중인 스레드를 종료하며 종료 시, 스레드의 이름과 status를 출력한다. 종료 전에 현재 수행중인 스레드의 부모 스레드를 찾아 부모 스레드의 child\_status를 자신의 status로 바꿔준다.

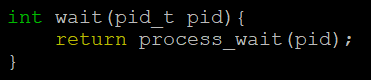


* Userprog/syscall.c ->exec



Cmd\_line을 이름으로 갖는 thread를 새로 생성하는 함수로, 리턴 값은 새로 생성한 thread의 tid값이다. Process\_execute 함수를 이용한다.

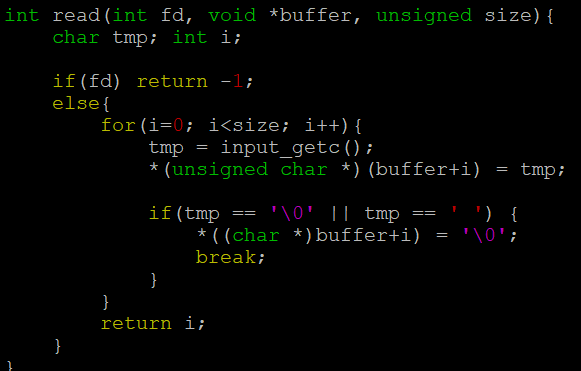
* Userprog/syscall.c ->wait



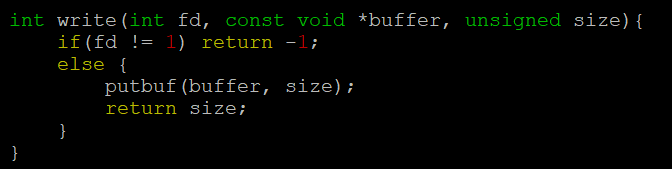
Child thread가 수행되는 동안 parent thread는 이를 기다리기 위한 함수로, child thread가 종료되면 child thread의 exit staus를 리턴값으로 갖는다.

* Userprog/syscall.c ->read

표준 입출력(fd=0)을 사용하여 문자를 읽어온다. 입력 받은 size만큼 읽어오기 전에 엔터나 NULL 값을 만나면 읽기를 종료한다. 읽어온 문자를 buffer에 하나씩 넣는다. 리턴 값은 종료 전까지 읽은 문자의 개수이고, size를 넘을 수 없다.



* Userprog/syscall.c ->write



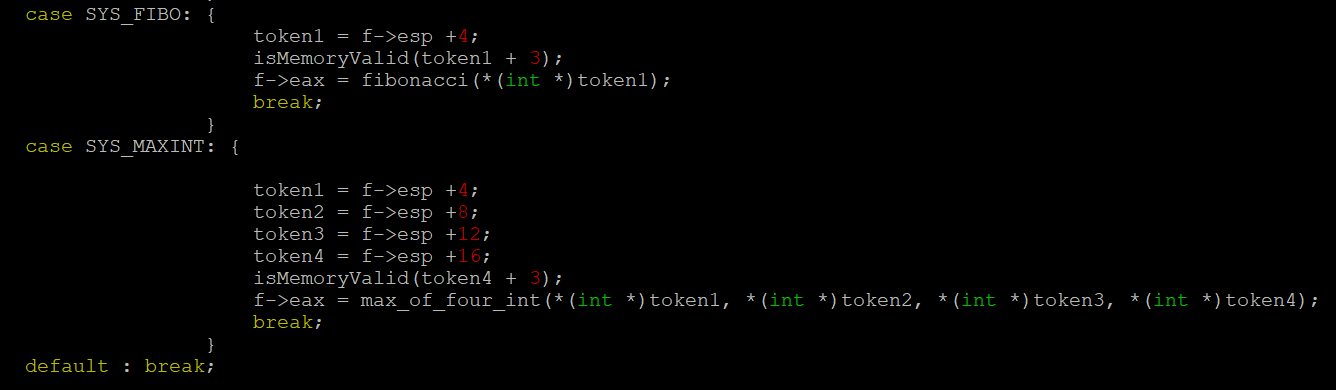
표준 입출력(fd=1)을 사용하여 버퍼의 값을 출력한다. 버퍼에 저장된 내용을 size만큼 출력한다. 리턴 값은 종료 전까지 출력한 문자의 개수이고, size를 넘을 수 없다

1. Additional System calls

* Userprog/syscall.h

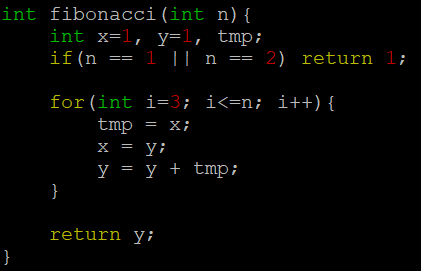


* Userprog/syscall.c ->syscall\_handler



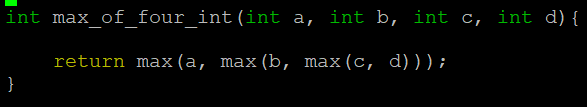
Interrupt가 걸리면 \*(f->esp)의 값을 확인하여 해당하는 system call을 호출한다. 또한, f->esp를 이용하여 스택에 저장된 argument를 system call의 parameter로 넘겨준다. 이때, 결과 값은 pintos manual에 의해 eax 레지스터에 저장한다. 마지막으로 각각의 system call function을 호출하기 전에 invalid memory access가 아닌지 isMemoryValid 함수로 확인해준다.

* Userprog/syscall.c ->Fibonacci



피보나치 수열의 n번째 항을 계산하여 리턴한다.

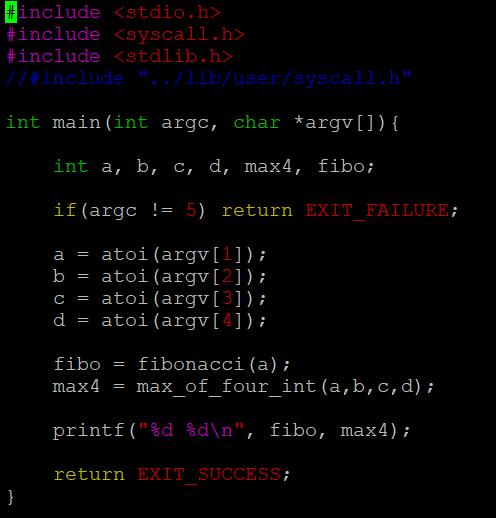
* Userprog/syscall.c ->max\_of\_four\_int



#define을 이용하여 max 값을 찾는 매크로를 만들었다. 이를 이용하여 네 개의 정수 중 가장 큰 값을 찾아 리턴한다.

* Examples/additional.c

네 개의 정수를 차례로 입력 받아 각각 a, b, c, d에 저장한다. 피보나치 수열의 a 번째 항과, a~d까지의 정수 중 가장 큰 값을 찾아 출력한다. Argument의 개수가 5개인지 파악하고 그렇지 않다면 system call을 호출하기 전에 exit(-1) 한다.



* Examples/Makefile





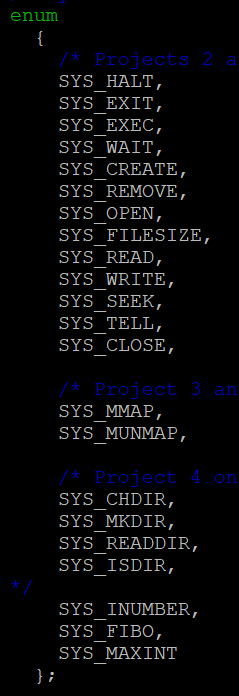
기존에 있던 makefile을 수정하여 additional.c도 실행 파일이 생성되도록 하였으며, 실행 파일의 이름은 additional이다.

* Lib/user/syscall.h ->

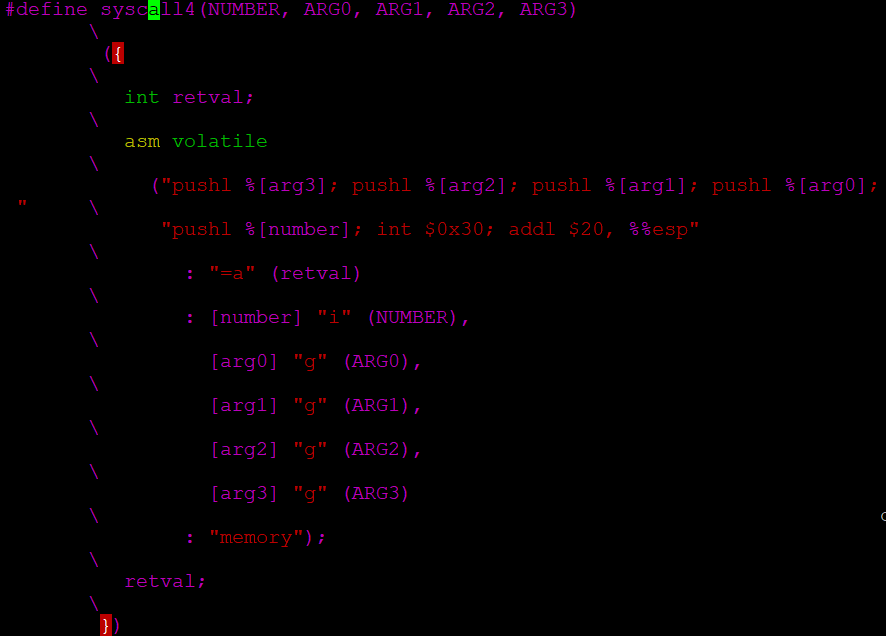


* Lib/syscall-nr.h -> enum

새로운 system call을 다룰 수 있도록 system handler의 enum 자료형에 SYS\_FIBO, SYS\_MAXINT를 추가하였다.

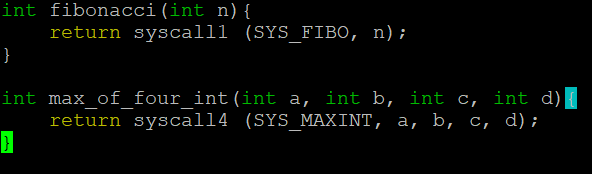


* Lib/user/syscall.c -> #define syscall4

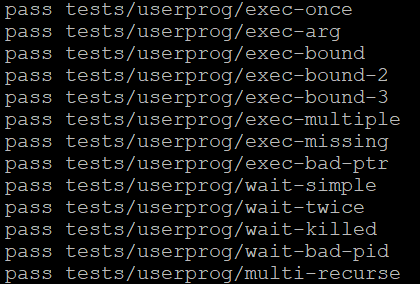
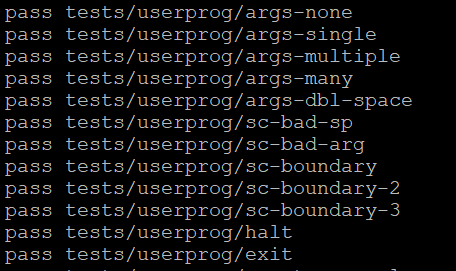


Max\_of\_four\_int system call은 argument가 4개이다. 그러나 pintos에서는 argument를 3개 까지 받는 system call만 구현되어 있다. 따라서 추가 함수를 구현하기 위해 argument를 4개까지 받을 수 있는 syscall4 매크로를 구현하였다. 어셈블리어의 asm volatile 구문을 사용하였다.

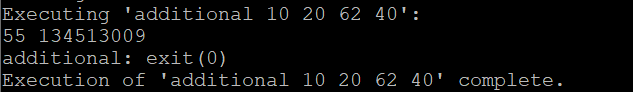
* Lib/user/syscall.c -> Fibonacci, max\_of\_four\_int



각각의 함수에 해당하는 system call #를 호출하였다.

* 1. **시험 및 평가 내용**

test case 목록에 있는 21개에서 전부 pass를 받았다.



Fibonacci system call은 올바른 결과를 내지만, max\_of\_four\_int는 결국 끝까지 구현하지 못했다. Src/lib/user/syscall.c 에서 추가한 #define syscall4가 제대로 작동하지 않는 것 까지 확인하였으나, 시간 부족으로 오류를 고치지 못하였다.