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

담당 교수 : 김영재 교수님

조 / 조원 : 박성우

개발 기간 : 9.10~10.02

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

현재의 pintos는 적절한 기능들이 아직 구현이 되어 있지 않은 상태라 user program을 적절하게 수행해 주지 못한다. user program이 어떠한 과정을 거쳐 실행되는지에 대한 이해를 바탕으로 pintos에서 user program을 올바르게 실행시키는 것이 이번 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

Argument Passing 은 사용자에게서 입력 받은 명령어를 공백을 기준으로 잘라서 실행하고자 하는 파일 이름과 argument들로 나눈다. 그런 뒤 그 파일을 load 하고 argument들과 그 주소들 등 필요한 데이터를 user stack에 추가해준다.

1. User Memory Access

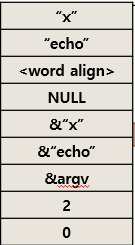
User program이 사용할 포인터를 확인하여 이것이 만약 kernel memory 영역을 가리키거나 null pointer 이거나 unmapped 된 가상 메모리를 가리키는 경우에는 프로그램을 종료 시키는 기능을 구현한다.

1. System Calls

User program이 사용하는 여러 system call 함수를 system call handler를 통해 실제 kernel api 들과 연결을 해주어야 한다. 이번 프로젝트에서는 halt, exit, exec, wait, read, write system call 이 실행되었을 때 적절히 동작하도록 system call handler를 구현한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing

“echo x” 라는 명령어가 들어왔다고 가정하면



위 그림처럼 스택에 argument 의 뒤부터 쌓는다. 그 다음 스택의 주소가 4의 배수에 맞게 0을 채워 word align 해준다. “x\0” “echo\0” 은 길이의 합이 7이므로 이 경우에 7보다 큰 4의 배수는 8 이므로 1칸을 0으로 채운다. 다음으로 선을 긋듯이 4칸을 0을 채운다. 그리고 argument 들의 주소를 넣어주고 &argv 부분에는 바로 위칸의 주소를 넣고, 다음으로 argument의 개수를 넣고 마지막으로 return address로는 0을 넣어준다.

* User Memory Access

Pintos에는 user memory 영역과 kernel memory 영역이 있다. 이때 user program 에서 포인터가 가리키는 곳이 kernel address 영역이거나 null pointer 거나 unmapped virtual memory 이면 invalid memory access 가 된다.

Invalid memory access를 막기 위해서는 user program 의 pointer가 valid 한지 invalid 한지 확인해야한다. 다행스럽게도 pintos 에서는 이를 위해 is\_user\_vaddr 과 is\_kernel\_vaddr 함수를 제공한다. 이를 통해 확인하고 invalid pointer 면 프로그램을 종료시킨다.

* System Calls

User program은 kernel 에 있는 메모리에 마음대로 접근할 수 가 없지만 접근해야 하는 경우들이 있어 이를 위해 직접 접근하는 대신 시스템 콜을 이용해 간접적으로 접근할 수 있게 된다. 이번에 구현한 system call 은 halt, exit, exec, wait, read, write이다. 따라서 user program이 위 6가지 system call 을 하면 적절한 동작이 수행이 된다.

Halt(): shutdown\_power\_off() 함수를 호출하여 pintos를 끝낸다.

Exit(): current thread 의 exit status 를 반환해주고 current thread 를 끝낸다.

Exec(): process\_execute() 함수를 호출하여 child process를 만들어준다.

Wait(): child process 가 끝날 때 까지 기다리는 시스템 콜이다. Child process가 끝나면 exit status를 받아오고 만약 child process 가 비정상적으로 끝나도 여전히 끝난 것을 알고 exit\_status 를 받아야한다.

Read(): STDIN 에 대해서만 구현한다. Fd 값이 0이면 stdin 이므로 input\_getc() 함수를 호출하여 구현한다.

Write(): STDOUT 에 대해서만 구현한다. Fd 값이 1 일때 putbuf() 함수를 호출하여 구현한다.

User program 이 시스템 콜을 한다. -> lib/user/syscall.c 에서 그에 맞는 함수를 실행한다. -> syscall1~syscall4 중 하나가 실행된다. -> syscall 숫자와 받은 인자를 스택에 쌓고 30번 interrupt 를 건다. -> threads/intr-stubs.S 에 intr\_entry 가 실행되고 threads/interrupt.c 가 호출되고 여기서 intr\_handlers[30] 에 있는 userprog/syscall.c 의 syscall\_handler() 가 실행된다! 여기서 우리가 각각 system call 에 맞는 kernel api 들을 연결해주어 처리해주고 eax 레지스터에 리턴값을 저장해주면 유저 레밸로 돌아간다!

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

9/11 ~ 9/18: 큰 흐름을 파악 후 argument passing 작성

9/18 ~ 9/30: 고통의 User memory access, syscall handler, additional 구현

10/1 : 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

Command string 을 공백을 기준으로 자른 뒤 80x86 calling convention에 맞게 user stack 에 추가해주는 코드를 process.c 안에 load 함수 안에 작성한다.

Process\_execute() 함수에서도 command string을 공백을 기준으로 자르고 thread\_create 가 호출될 때 파일 이름만 들어갈 수 있도록 수정해준다. 이를 위해 strtok\_r 함수를 사용하였다.

1. User Memory Acess

Userprog/syscall.c 에 valid\_pointer 함수를 만들어 is\_user\_vaddr 함수를 통해 pointer 가 user address space 에 있는지 확인하고 아니면 exit(-1)로 프로그램을 종료하는 부분을 추가한다.

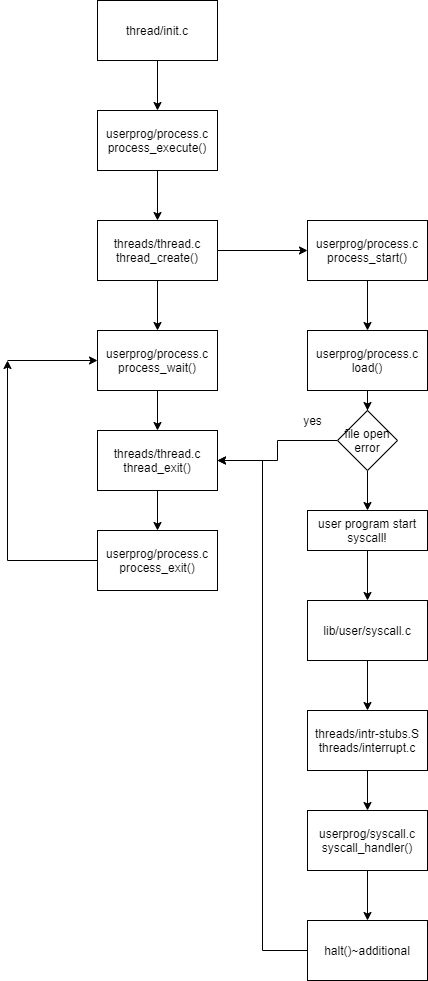
Userprog/exception.c 의 page\_fault() 에서 user 변수가 false 이거나 is\_kernel\_vaddr 이 true 인 경우 exit(-1)로 프로그램을 종료하는 부분을 추가한다.

1. System call

Userprog/syscall.c 의 syscall\_handler() 에서 들어오는 syscall 에 대해 switch문에서 적절한 함수를 실행할 수 있게 구현한다. 추가적으로 fibonacci 와 max\_of\_four\_int 가 실행될 수 있도록 lib/user/syscall.c,h 파일에 함수를 추가하고 lib/syscall-nr.h enum 에도 적절한 상수이름을 추가해준다. 모든 syscall 을 적절히 처리하고 eax에 반환값을 저장한다.

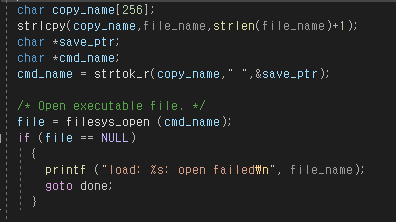
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

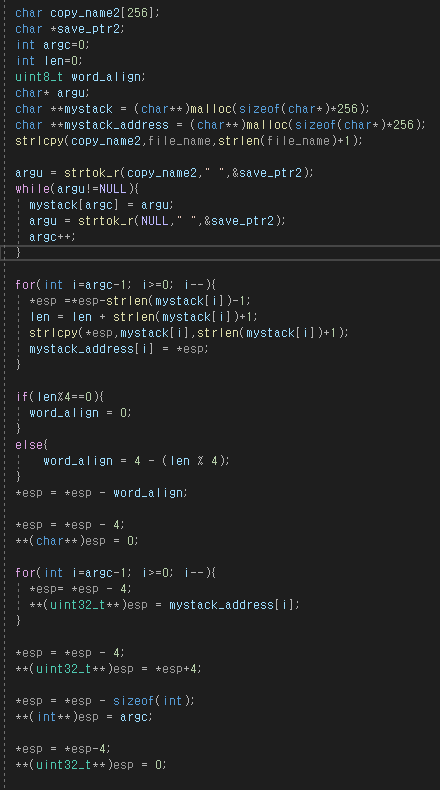


* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

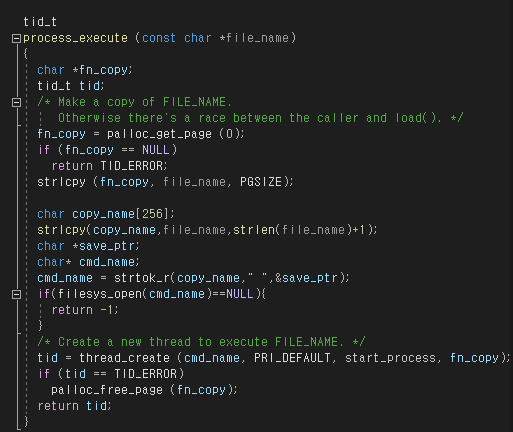
1. Argument Passing



먼저 파일이 잘 열리도록 파일 이름부분을 parsing 해준다.

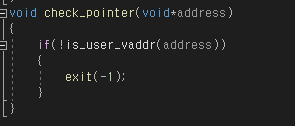


Parsing 한 것들을 mystack에 쌓는다. 쌓으면서 몇 개 인지 argc 에 넣어준다. 인자들을 넣어줄 때 밑에서 인자들의 주소값도 넣어야 하므로 주소값을 저장하는 mystack\_address 배열을 만들었다. 이후는 calling convention 에 맞춰서 esp 포인터를 줄여가면서 차근차근 넣어주었다.

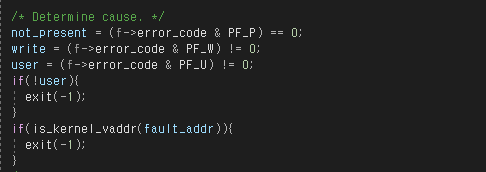


Process\_execute 부분에서도 쓰레드를 생성할 때 쓰레드 이름이 제대로 들어가도록 들어온 명령어를 잘 parsing 하였다.

1. User Memory Access



Userprog/syscall.c 안에 check\_pointer 함수를 만들어서 address 가 유효한지 매번 검사한다.



Userprog/exception.c. 에서 user 변수가 false 이거나 커널 주소를 참조하고 있을 때 또한 확인한다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
* static void
* syscall\_handler (struct intr\_frame \*f)
* {
* switch (\*(uint32\_t \*)(f->esp)) {
* case SYS\_HALT:
* shutdown\_power\_off();
* break;
* case SYS\_EXIT:
* {
* check\_pointer(f->esp+4);
* exit(\*(uint32\_t \*)(f->esp + 4));
* break;
* }
* case SYS\_EXEC:
* {
* check\_pointer(f->esp+4);
* const char\* cmd\_line = (const char\*)\*(uint32\_t\*)(f->esp+4);
* f->eax=exec(cmd\_line);
* break;
* }
* case SYS\_WAIT:
* {
* check\_pointer(f->esp+4);
* pid\_t pid = (pid\_t)\*(uint32\_t\*)(f->esp+4);
* f->eax = process\_wait(pid);
* break;
* }
* case SYS\_READ:
* {
* int fd = (int)\*(uint32\_t\*)f->esp+4;
* unsigned size = (unsigned)(\*(uint32\_t\*)(f->esp+12));
* if(fd==0){
* unsigned i;
* for(i=0; i<size; i++){
* if(input\_getc()=='\0'){
* break;
* }
* }
* f->eax= i;
* }
* else{
* f->eax = -1;
* }
* break;
* }
* case SYS\_WRITE:
* {
* int fd = (int)\*(uint32\_t\*)(f->esp+4);
* const void\* buffer= (void\*)\*(uint32\_t\*)(f->esp+8);
* unsigned size = (unsigned)(\*(uint32\_t\*)(f->esp+12));
* if(fd ==1){
* putbuf(buffer,size);
* f->eax = size;
* }
* else{
* f->eax = -1;
* }
* break;
* }
* case SYS\_FIBO:
* {
* int n = (int)\*(uint32\_t\*)(f->esp+4);
* f->eax = fibonacci(n);
* break;
* }
* case SYS\_MAX:
* {
* int a = (int)\*(uint32\_t\*)(f->esp+4);
* int b = (int)\*(uint32\_t\*)(f->esp+8);
* int c = (int)\*(uint32\_t\*)(f->esp+12);
* int d = (int)\*(uint32\_t\*)(f->esp+16);
* f->eax = max\_of\_four\_int(a,b,c,d);
* break;
* }
* }
* // thread\_exit ();
* }

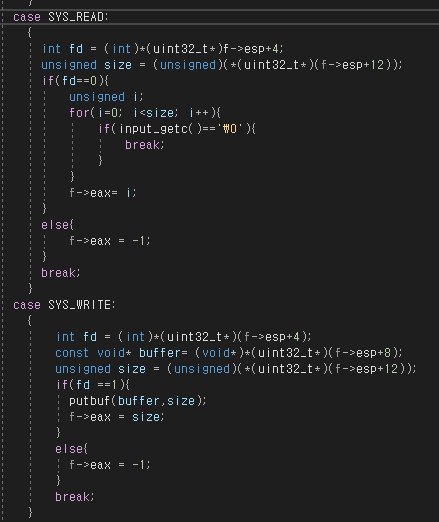
이것이 syscall.c 의 핸들러이다. User program 이 호출한 system call 에 따라서 switch 문에 내에 그것에 맞는 부분을 실행한다. Case 문 안에서 바로 구현한 system call 도 있고 추가구현 함수들과 exit, exec system call은 함수를 따로 만들어서 구현하였다. 각 system call 마다 적절한 개수의 인자를 f->esp 포인터를 이용해 가져와야 한다. 이제 각 system call을 자세히 보면

Halt()



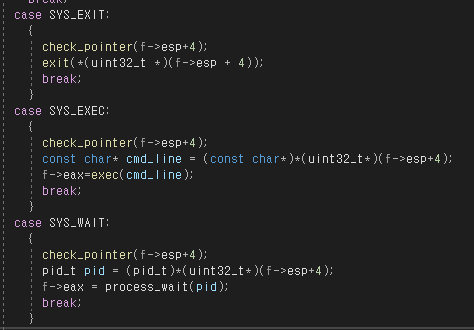
Halt 는 shutdown\_power\_off kernel api 를 부르고 끝낸다. 이 함수가 devices/shutdown.h 에 정의되어 있어서 #include “devices/shutdown.h” 해주었다.

Read, Write

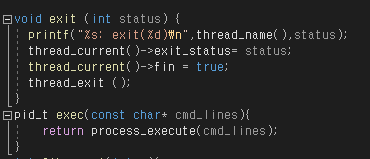


Read, write 은 stdin,stdout 에 대해서만 구현하였다. 각 system call에 대해서 적절한 인자를 받고 fd 에 따라서 input\_getc, putbuf 함수를 호출하고 f->eax 에 size 를 저장하고 끝냈다.

Exit(), exec(), wait()



각 인자를 받고 exit system call 은 직접 만든 exit() 함수를 호출하고 exec system call 은 직접 만든 exec() 함수를 호출하고 wait system call은 userprog/process.c 에 있는 process\_wait 함수를 호출한다. 따라가보면

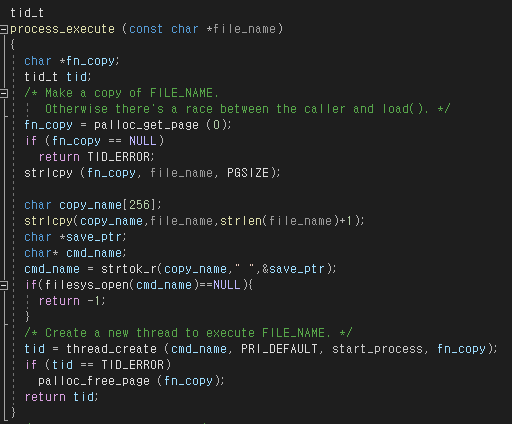


Exit 함수는 요구사항에 따라 현재 thread 이름과 exit status 를 출력한다. 현재 스레드 상태를 나타내는 변수에 실제 상태를 넣고 현재 스레드가 끝났는지 안끝났는지를 체크하는 변수인 fin 에 true 를 넣고 thread\_exit() 을 호출한다. Threads/thread.c 에 있는 thread\_exit() 함수를 보면 userprog/process.c 에 있는 process\_exit() 함수를 호출한다.

Exec 함수는 userprog/process.c 에 있는 process\_execute 함수를 실행한다.

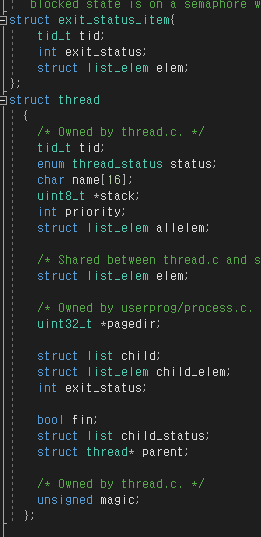
정리하면 exit, exec, wait system call 은 systemcall handler 에서 적당한 변수 처리를 한 후 모두 userprog/process.c 에 있는 process.exit(), process\_execute(), process\_wait() 함수를 실행하니 이제 userprog/process.c 를 확인하러 가야한다.

Exec는 process\_execute 를 실행한다.



Wait 과 exit 는 밀접한 연관이 있고 process.c 를 보기전에 process\_wait 과 process\_exit 에서 사용한 구조체를 확인하러 src/threads/threads.c,h 에 먼저 들러야 한다.

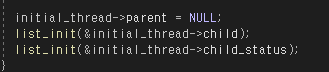
Thread.h



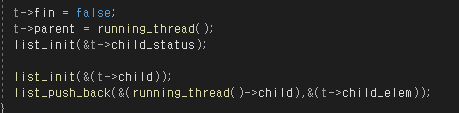
기존의 thread structure에 필요한 멤버를 추가한다. Child thread 에 쭉 저장하는 struct list child, 부모의 child 리스트에 추가되기 위한 struct list\_elem child\_elem, 자신이 종료되었다는 것을 부모에 알라기 위한 fin, 자신의 exit status 를 저장하는 exit\_status, 자신의 부모를 가리키는 struct thread\* parent 이다. 추가로 child thread 가 종료되면 부모가 child thread 의 exit status 를 알아야 하는 데 이미 child thread가 종료되어 알 수가 없어서 그것을 따로 저장하는 exit\_status\_item 구조체를 만들었다.

멤버를 새로 추가했으므로 threads/thread.c 에서 초기화를 해줘야 한다.

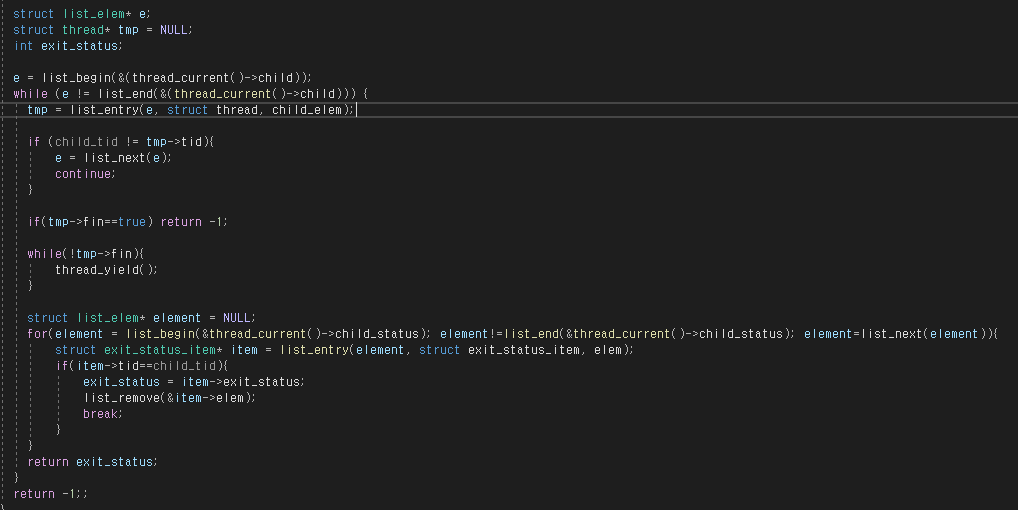
Thread\_init 함수에서

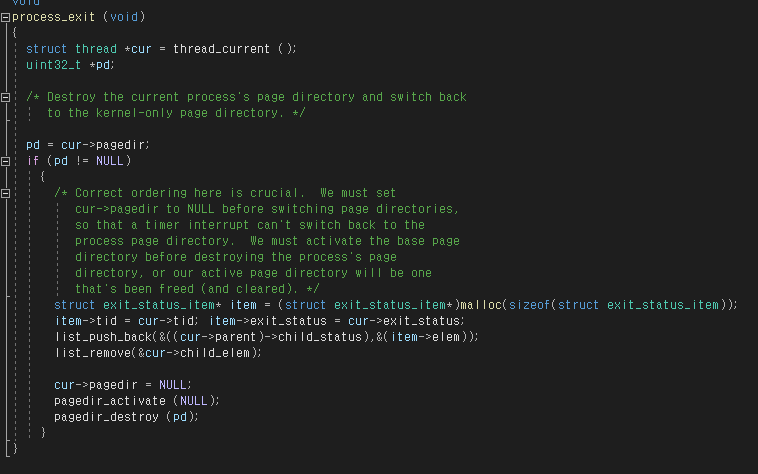


Init\_thread 함수에서



이제 드디어 userprog/process.c 로 간다. 위: wait , 아래: exit





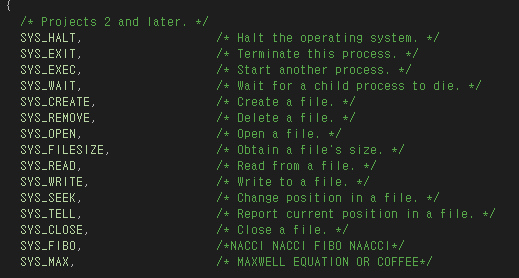
현재 실행중인 thread의 child 리스트 중에 tid가 process\_wait 함수가 실행될 때 인자로 넘어온 child\_tid 와 같은 thread 가 있으면 이 thread 가 끝날 때까지 기다린 뒤, exit\_status를 받아 리턴해주는 함수가 바로 wait 함수이다. Tid가 같으면 혹시 모를 비정상적인 종료를 확인하기 위해 이미 죽었는지 확인을 해준다. 만약 이미 죽었으면 기다릴 필요 없이 바로 -1 을 return 해준다. 만약 정상적으로 살아있으면 프로젝트1 ppt 에 나와있는 thread\_yield 함수를 사용하여 죽을 때까지 기다려준다. Child thread 가 죽고 나면 parent 는 child 의 exit status를 확인해야하는데 그것을 위해 thread 가 죽을 때 process\_exit() 함수안에서 exit\_status\_item 구조체를 만들고 여기에 현재 thread id 와 exit\_status를 넣고 리스트에 넣어준 뒤 죽는다. 그러면 부모 thread 에서 이 리스트를 탐색하여 exit\_status를 얻을 수 있다!. 얻었으면 exit\_status 를 return 해주고 만약 child\_tid와 동일한 id 를 가지는 child thread 가 없으면 -1을 return 해준다.

1. Additional System calls

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

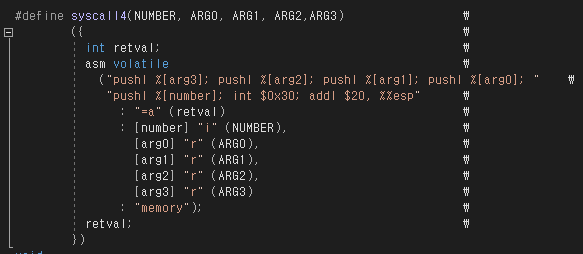
새로운 system call 인 Fibonacci, max\_of\_four\_int는 새로운 system call 이라 lib/syscall-nr.h, lib/user/syscall.c 부터 시작해야한다.

Lib/syscall-nr.h

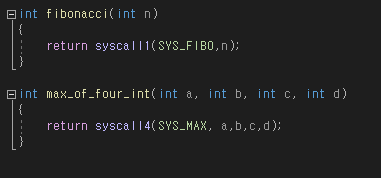


새로운 enum 타입 변수를 만들어야 한다 SYS\_FIBO, SYS\_MAX 이다.

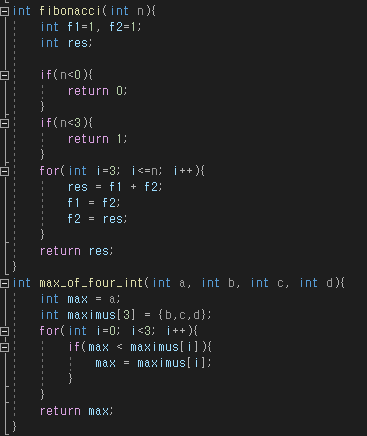
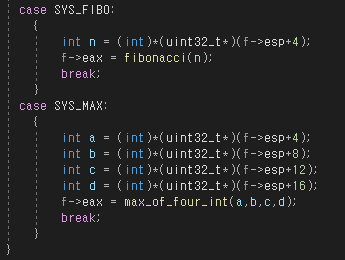
Max\_of\_four\_int 는 인자를 4개 받아야 하므로 lib/user/syscall.c 에 새로운 syscall4 를 만들어야 한다.



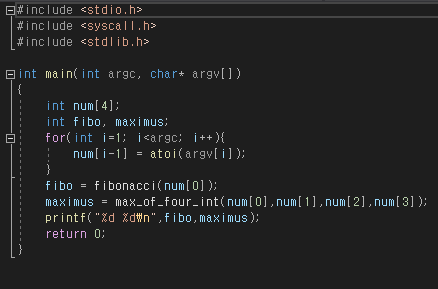
그리고 이 syscall 을 불러주는 함수를 만들어 준다.



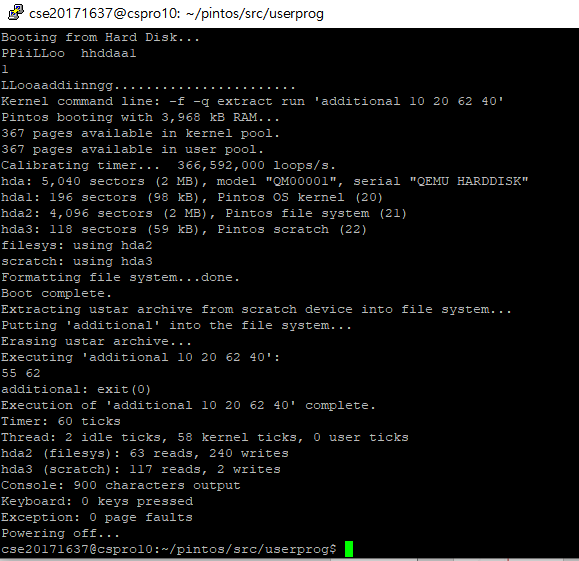
이제 타고 타고 가서 userprog/syscall.c 에 있는 syscall\_handler 가 실행이 되니 거기서 처리를 해주면 된다.



새로운 system call 을 만들었으니 실제 user\_program 에서 새로 만든 system call 을 사용해야 한다. Src/examples 에 user program 을 만들고 makefile 을 수정해서 컴파일을 하면 이제 이 user program을 실행해주면 새로 만든 system call이 제대로 수행된다.



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



제대로 동작함을 확인할 수 있다.