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

담당 교수 : 김영재

조 / 조원 : 한석기 / 한석기

개발 기간 : 2023.09.18 ~ 2023.10.08

1. **개발 목표**

Pintos에서 입력 받는 명령어들을 Parsing하고 Stack에 Passing하는 과정을 진행한다. 그리고 Kernel 메모리와 User 메모리를 신경 쓰면서 halt, exit, exec, wait, read, write의 system call과 additional system call인 fibonacci, max\_of\_four\_int를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

Argument로 받은 file\_name을 parsing하고 stack에 passing한다. 그 후 hex\_dump()함수로 stack에 잘 passing 됐는지 확인한다.

1. User Memory Access

Kernel 영역을 침범하지 않게 address를 확인하고 침범했다면 exit system call로 비정상 종료한다.

1. System Calls

Halt, exit, exec, wait, read, write system calls를 실행할 수 있다. additional system call(fibonacci, max\_of\_four\_int) 또한 실행할 수 있고 실행하면 arguments에 따른 결과값이 나온다.

* 1. **개발 내용**
* Argument Passing

입력으로 받은 Arguments를 Parsing한다. 그리고 Parsed arguments를 stack에 차곡차곡 역순으로 쌓는다. Stack의 포인터를 의미하는 esp 포인터를 초기화하고 주소를 넣을 배열을 만든다. 전체 byte 수를 셀 수 있는 변수를 선언하고 byte수를 세면서 Arguments를 순회한다. 순회하면서 esp로 주소 계산을 하며 Arguments를 esp에 넣어주고 마지막에 word align 과정을 거친다. 그리고 각각의 Argument에 따른 주소를 넣어주고 Arguments의 수, return address까지 넣는 과정을 마치면 끝이다.

* User Memory Access

Pintos 상에서의 invalid memory access는 user mode에서 kernel address로 접근하는 것을 의미한다. Src/threads/vaddr.h에 있는 is\_kernel\_vaddr을 사용해 kernel이면 exit로 비정상 종료해 valid한지 확인한다.

* System Calls

System Calls는 간단하게 말해 인터페이스다. 즉 user mode에서 kernel mode의 실용적인 API를 이용하게 하는 것이다. 위에 User memory access에서와 마찬가지로 user mode는 user process만 kernel mode는 kernel process만 실행될 수 있는데 System Calls를 통해 각각의 mode에서 다른 process에 요청하는 일이 가능하다. 이번 프로젝트에서 개발할 시스템 콜은 아래와 같다.

Halt: Pintos를 종료한다.

Exit: thread의 이름과 exit\_status인 종료 id를 출력하고 process를 종료한다.

Exec: process를 실행한다.

Wait: parent process가 child process가 완료될 때까지 대기한다.

Read: file descriptor가 stdin일 때, 명령어를 한 char씩 읽는다.

Write: file descriptor가 stdout일 때, 출력한다.

Fibonacci: argument의 첫번째 인자에 해당되는 수를 n으로 하고 n번째 피보나치 수를 출력한다.

Max\_of\_four\_int: 네 개의 수 중에서 가장 큰 수를 출력한다.

User mode에서 system call을 호출하면 interrupt handler가 interrupt를 발생시킨다. 그러면 context switch가 일어나고 kernel mode로 전환된다. Kernel mode에서는 전달받은 system call number를 참고해서 system call table의 명령어와 매칭시킨다. 매칭된 system call을 kernel mode에서 실행하고 interrupt를 발생시켜 context switch과정을 거쳐 user mode로 돌아온다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2023.09.18 ~ 2023.09.25: argument passing 방법 생각, memory access control 공부

2023.09.26 ~ 2023.10.03: system call 공부

2023.10.04 ~ 2023.10.07: 이전에 공부한 내용을 토대로 argument parsing 및 passing, memory access control, system calls 구현 및 test

2023.10.08: 보고서 작성

* 1. **개발 방법**

**\*Parameter parsing**

수정 경로: userprog/process.c

prj1 ppt의 설명에 나와 있듯이 load 함수의 filesys\_open 전에 수행한다. Thread\_safe함수인 strlcpy와 strtok\_r을 이용해 parsing을 진행한다.

**\*Parameter passing**

수정 경로: userprog/process.c

Prj1 ppt의 설명에 나와 있듯이 load 함수의 setup\_stack 이후에 수행한다. 위에서 parameter parsing을 수행하고 난 결과인 parsed arguments를 이용해 주소를 계산하며 stack에 쌓는다. 쌓을 것은 각각의 argument와 각 argument의 주소, arguments의수, return address다.

**\*User memory access**

수정 경로: userprog/syscall.c, userprog/exception.c

Userprog/syscall.c에서 valid한 address인지(user 영역인지) 확인하는 함수를 만들고 각 system call을 실행하기 전에 확인한다.

Userprog/exception.c에서 page\_fault 함수에 printf 전에 kernel에 의해 접근됐다면 비정상 종료, falut address가 kernel 영역이면 비정상 종료한다.

**\*System Call Handler**

수정 경로: userprog/syscall.c

Switch문을 통해 lib/syscall-nr.h에서 정의한 SYS\_NAME으로 System call의 종류를 확인하고 그에 해당하는 함수를 실행한다.

**\*System Call implementation**

수정 경로: userprog/syscall.c, userprog/process.c, threads/thread.c, threads/thread.h

각각의 System call을 함수로 구현하고 userprog/process.c에 존재하는 것을 이용해 수정하는 과정을 거친다. 특히 wait에서 사용될 semaphore를 threads/thread.h에 추가한다. 자세한 내용은 아래 제작 내용에서 다루겠다.

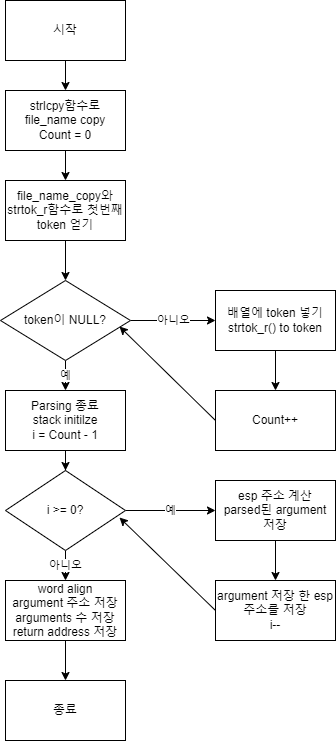
**\*Additional Implementation**

수정 경로: userprog/syscall.c, examples/additional.c, lib/user/syscall.c

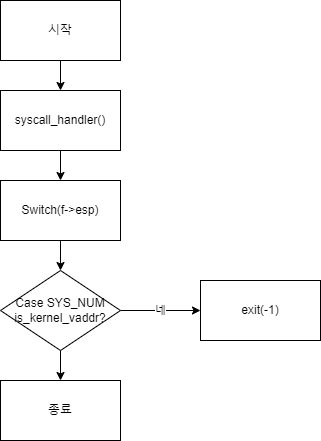
lib/syscall-nr.h, lib/user/syscall.h

Fibonacci와 max\_of\_four\_int 함수를 syscall.c에 구현하고 system call로 연결시킨다. 연결 과정에서 SYS\_FIBO와 SYS\_MAXNUM의 enum값 주기 및 lib/user/syscall에서 arguments의 low level handling의 과정을 거쳐 완성한다.

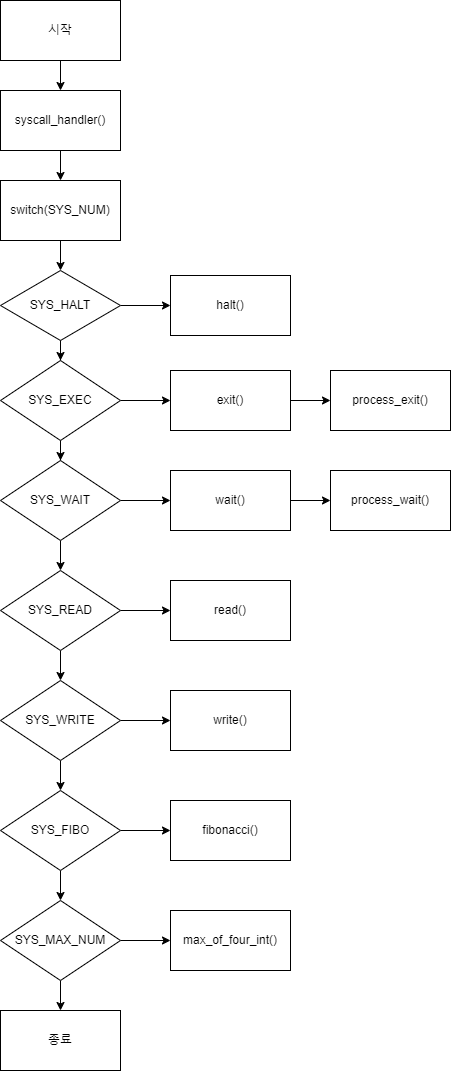
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access

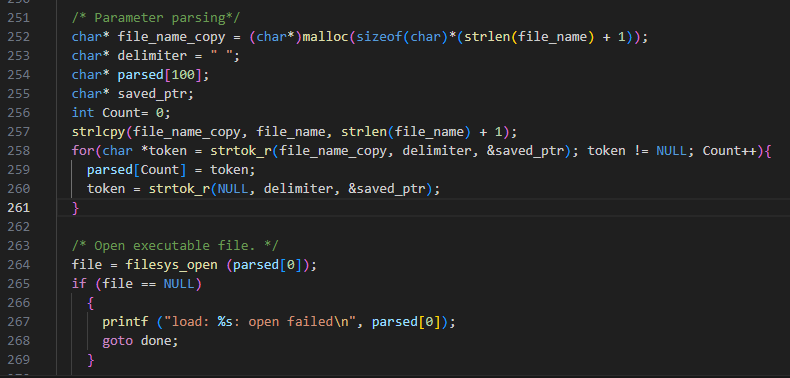


1. System Calls

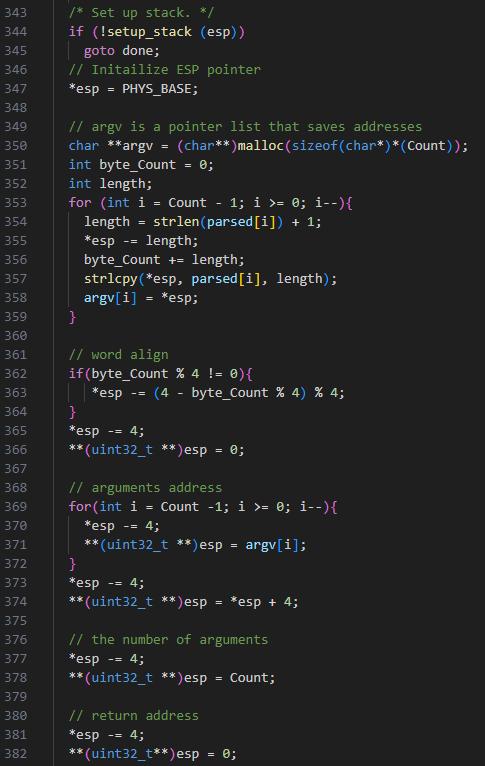


* 1. **제작 내용**

1. Argument Passing

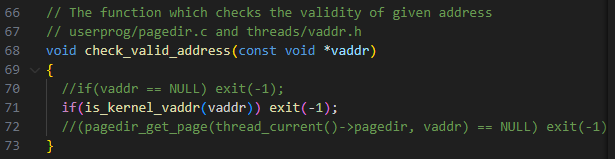


Userprog/process,c의 load함수의 parameter parsing하는 부분이다. File\_name, 즉 arguments가 전부 포함되어 있는 string을 string.h의 thread\_safe한 함수인 strlcpy를 사용해 복사하고 string.h에 있는 thread\_safe한 strtok\_r함수를 사용해 “ “(공백)을 기준으로 나눈다. 나눈 token들을 parsed라는 배열에 저장한다.



Userprog/process.c의 load함수에서 stack construct하는 부분이다. Esp 포인터를 초기화하고 malloc을 이용해 argument가 들어있는 곳의 주소를 저장할 배열을 만든다. 그리고 아까 parsing한 parsed배열을 이용해 stack에 차곡차곡 쌓는다. 쌓을 때, 총 byte수를 세주고 주소 또한 argv 배열에 저장한다. 이 과정이 다 끝나면 총 byte수를 이용해 word align을 하고 argv에 들어있는 주소, arguments의 수, return address순으로 쌓는다.

1. User Memory Access



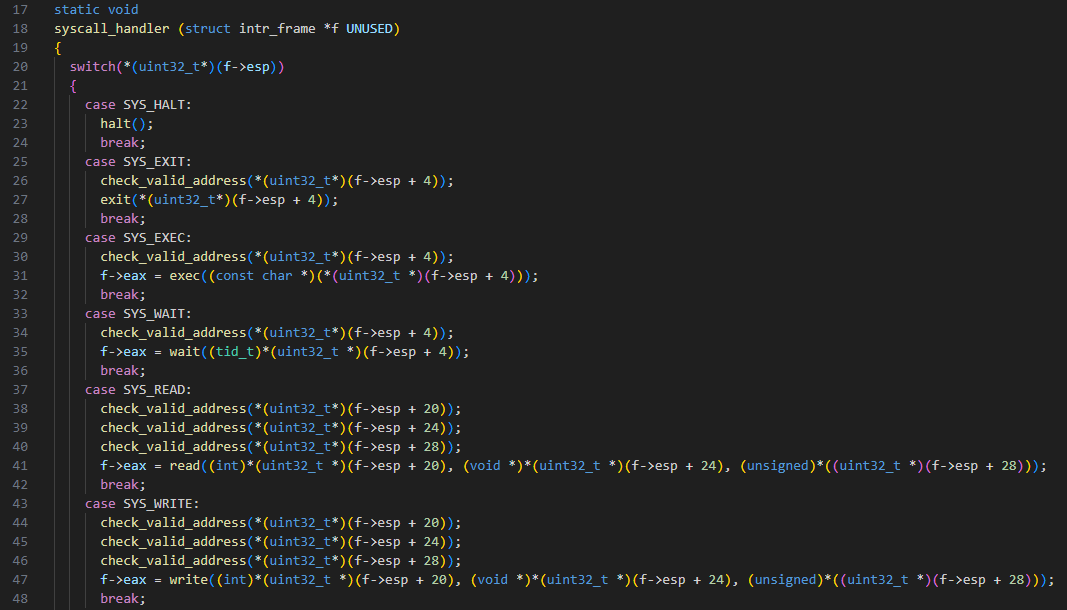
Threads/vaddr.h에 있는 is\_kernel\_vaddr 함수를 이용해 kernel 영역인지 확인하고 kernel영역이면 비정상 종료하는 함수다. //로 주석처리 되어 있는 부분은 address가 NULL일 때, 비정상 종료하는 것과 user virtual address에 대해 kernel virtual address를 구했는데 그것이 NULL이면 비정상 종료하는 것이었는데 이 처리에서 버그가 많이 발생했다. 계속 주석부분에서 비정상 종료가 되어 주석처리를 해줬고 vaddr이 kernel virtual address인지 확인하는 코드만 남겼다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

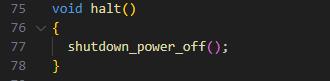
자동 생성된 설명

Userprog/exception.c에서 kernel virtual address인지 확인하는 코드를 추가해주고 kernel이라면 비정상 종료한다.

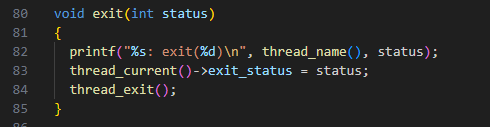
1. System Calls

****

Userprog/syscall.c부분에서 syscall\_handler 부분이다. SYS\_HALT, SYS\_EXIT, SYS\_EXEC, SYS\_WAIT, SYS\_READ, SYS\_WRITE는 lib/syscall-nr.h에 enum으로 선언되어 있다. 각 system call에 맞는 것을 switch문을 통해 찾아 들어간다. 그리고 각 system call마다 arguments를 고려해 주소를 잘 찾아주고 Type casting을 통해 각 함수를 실행시켜준다.



Halt 함수다. Devices/shutdown.h의 shutdown\_power\_off()함수를 이용해 pintos를 종료한다.

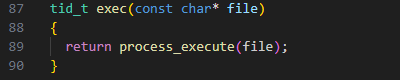


Exit 함수다. 현재 thread의 이름과 status를 출력하고 현재 thread의 exit\_status 변수에 status를 저장한다. 그리고 thread를 종료한다.

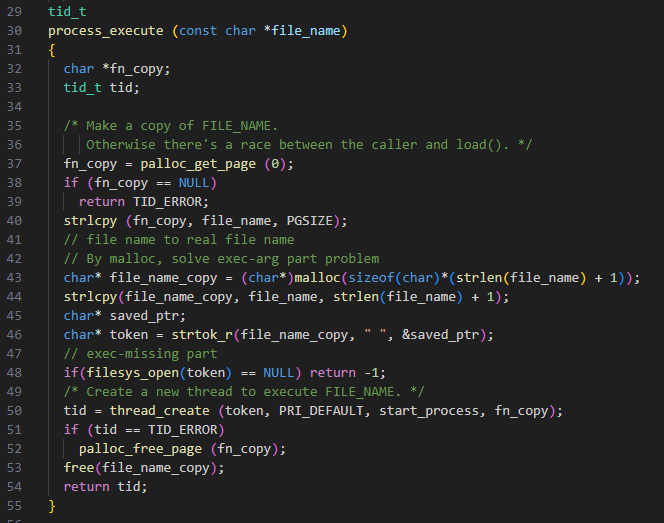
텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

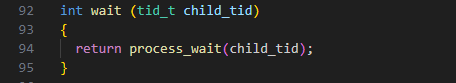
Exit\_status는 threads/thread.h에 thread 구조체에 추가해줬다.



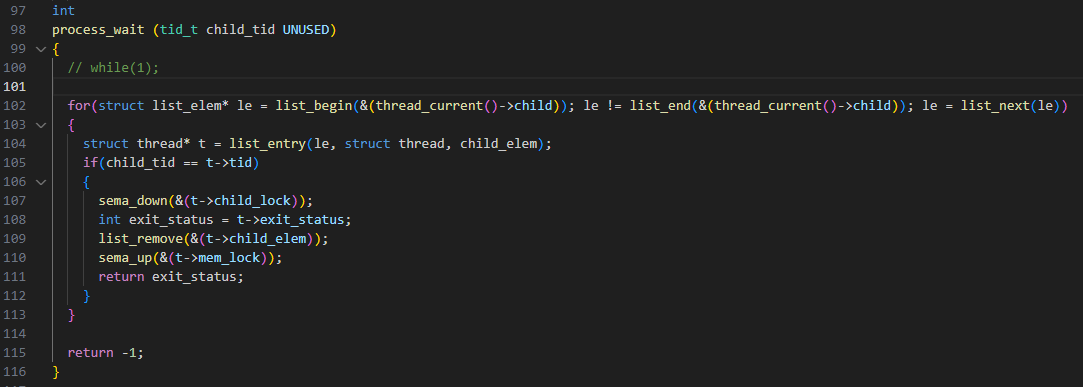
Exec 함수다. Process.c의 process\_execute 함수를 실행한다.



Userprog/process.c의 process\_execute 함수다. Child process를 생성할 수 있기에 malloc을 통해 thread safe하게 동작하게 file\_name을 copy한다. 그리고 copy한 file\_name을 맨 앞 argument만 token으로 받아낸다. 그것이 명령어의 이름이기에 만약 해당 명령어가 파일로 존재하지 않는다면 비정상 종료하게 -1을 리턴하고 존재한다면 해당 이름으로 thread를 만들고 thread id를 리턴한다. 리턴된건 f->eax에 저장된다.



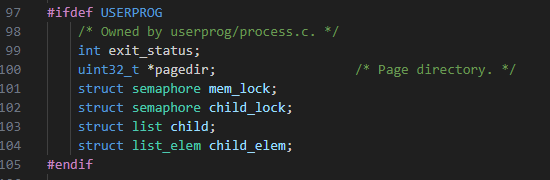
Wait 함수다. Userprog/process.c의 process\_wait함수를 실행한다.



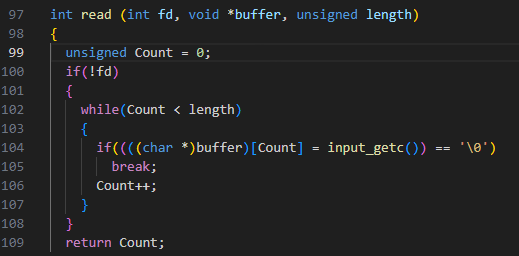
Userprog/process.c의 process\_wait함수다. Semaphore를 사용하여 wait system call을 구현했다. Semaphore는 threads/synch.c에 관련 함수가 있다. Child\_tid가 끝날 때까지기다린 후 thread의 exit\_status를 리턴한다. 모든 child process가 종료되지 않았다면 계속 for문에서 해당 자식 프로세스들 및 부모 프로세스가 남아있다. 이걸 가능하게 하는 것이 semaphore다. 만약 모든 child process가 종료된다면 -1을 리턴해 부모 프로세스도 종료한다.



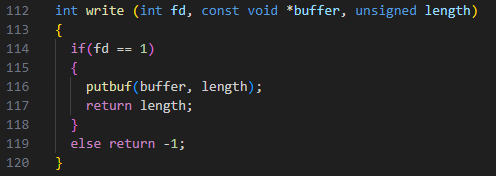
Userprog/process.c의 process\_exit함수의 마지막 부분이다. 프로세스가 종료될 때, semaphore관리가 필요하므로 해당 코드를 추가했다.



Semaphore는 threads/thread.h의 exit\_status추가한 부분에 추가해줬고 list와 list\_elem도 추가해줬다.

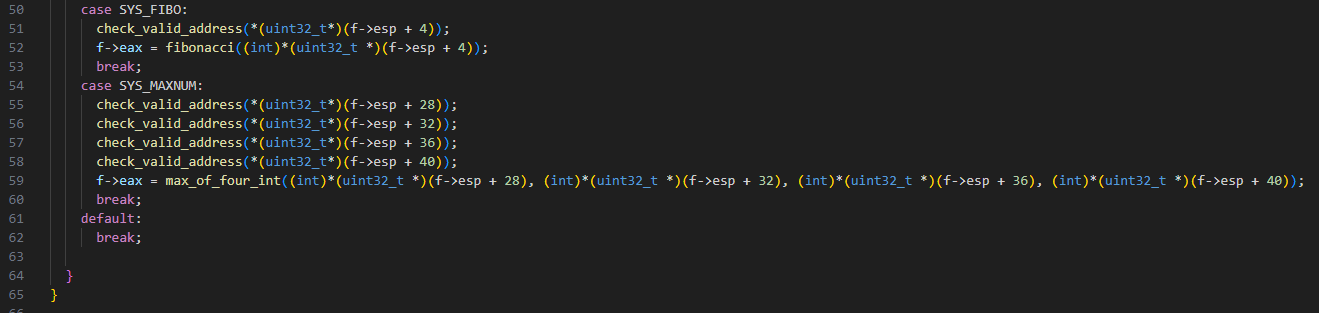


Read함수다. File descriptor가 0일 경우 널문자가 나올 때까지 devices/input.c의 input\_getc 함수를 이용해 한 문자마다 센다. 그리고 센 값을 return해 f->eax에 저장한다.



write함수다. File descriptor가 1일 경우 putbuf함수를 이용해 출력하고 길이를 리턴해 f->eax에 저장한다.

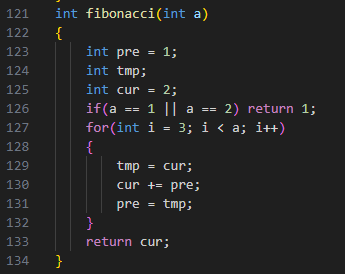
1. Additional System calls



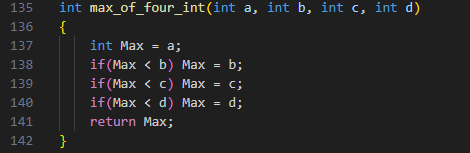
Userprog/syscall.c의 SYS\_FIBO와 SYS\_MAXNUM에 해당하는 명령어를 수행하는 부분이다. 주소를 잘 계산해서 구현한 각 함수에 인자로 전달하고 반환되는 값을 f->eax에 저장한다.



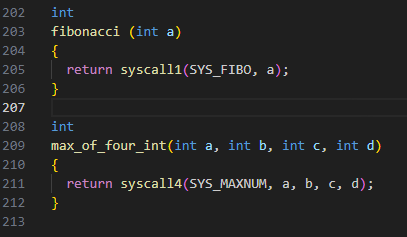
Lib/syscall-nr.h에 enum부분에 해당 것들을 추가해준다.



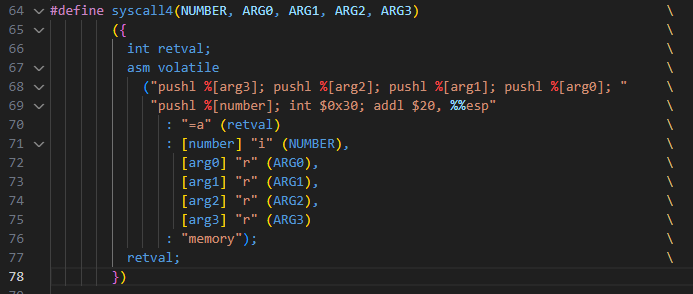
SYS\_FIBO에 해당되는 system call의 실행결과인 Fibonacci 함수를 구현한 것이다. 초기값을 설정하고 이전 값을 계속 저장하면 현재 값을 업데이트 시킨다. 그렇게 a번째 피보나치 수를 구한 후 리턴한다.



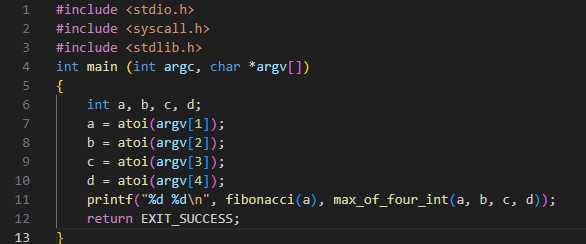
SYS\_MAXNUM에 해당되는 system call의 실행결과인 max\_of\_four\_int 함수를 구현한 것이다. 첫번째 인자를 일단 Max로 두고 두번째, 세번째, 네번째 차례로 비교하며 가장 큰 값을 Max에 저장한다. 그리고 Max값을 리턴한다.



Lib/user/syscall.c에 추가한 함수다. 위에 다른 함수들의 양식에 따라 비슷하게 구현했다.

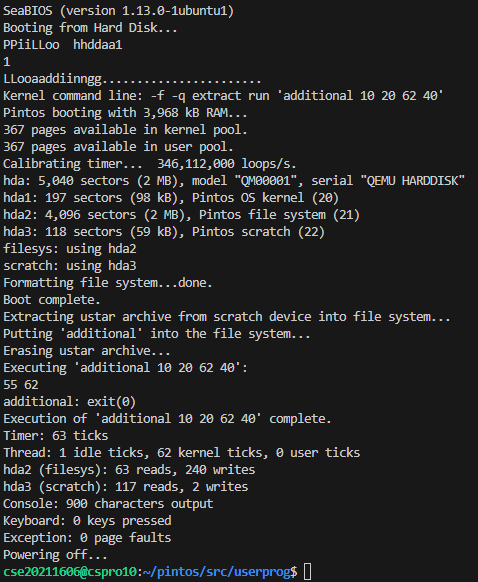


위의 과정에서 인자를 4개 받는 syscall4 부분이 구현이 되어있지 않아서 syscall1~syscall3 부분을 확인해 양식에 따라 syscall4 부분을 정의해줬다.

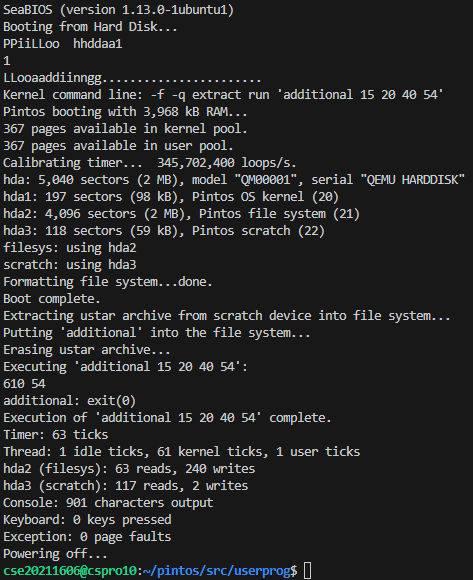


Examples/additional.c의 파일을 만들었다. 사용되는 header 파일을 추가해주고 인자로 들어오는 string을 int로 변환 후 userprog/syscall.c에서 구현한 함수를 실행시켰다.

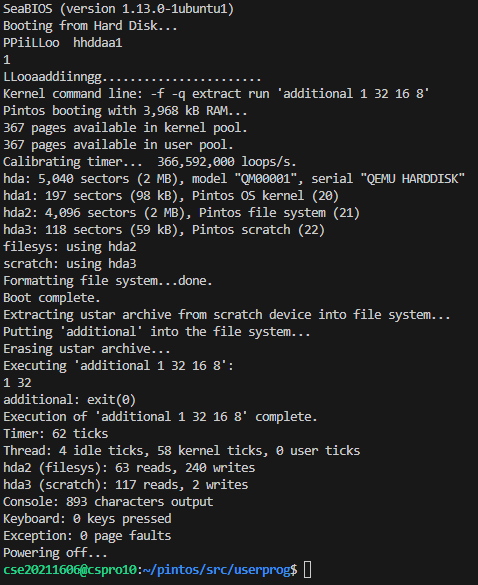
* 1. **시험 및 평가 내용**



10 20 62 40을 값으로 주었을 때, 10번째 fibonacci수와 4개의 값 중 최댓값을 출력했다. 각각 55와 62를 출력했다.



위 사진은 15 20 40 54를 값으로 주었을 때다. 15번째 fibonacci수인 610과 4개 값 중 최댓값 54를 출력했다.



위 사진은 1 32 16 8를 값으로 주었을 때다. 첫번째 fibonacci수인 1과 4개 값 중 최댓값 32를 출력했다.