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

담당 교수 : 박성용 교수님

조 / 조원 : 20211513 김소현

개발 기간 : 9.18~10.8

1. **개발 목표**

- 핀토스에서 프로그램이 실행되도록 하는 것이 이 프로젝트의 목표이다. 이를 달성하기 위해 아래의 항목들을 구현해야 한다.

가. 프로세스가 실행될 때 커널에 argument passing하기

나. 인터럽트 벡터 안의 시스템콜 핸들러 구현하기

시스템콜 번호에 따라 함수를 호출하는데, 이 떄 halt, exit, wait, read, write, exec, fibonacci, max\_of\_four\_int 로 총 8개의 함수를 구현하여 호출하도록 한다.

다. 프로세스가 접근 불가한 주소에 접근하는 것을 막기

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

사용자가 입력한 명령어에 있는 프로그램이 실행되고, 함께 입력된 변수들도 파싱되어 스택에 쌓인다.

1. User Memory Access

User가 잘못된 메모리 주소에 접근하려고 할 때 이를 감지하고 프로세스가 종료된다. 이 때 exit(-1)으로 종료한다.

1. System Calls

유저 프로그램이 시스템콜 API를 호출하고 트랩을 발생시키면 커널의 시스템콜 핸들러에서 대응하는 시스템콜 함수가 호출된다. Halt, exec, exit, read, write, wait, fibonacci, max\_of\_four\_int 시스템콜이 호출된다.

* 1. **개발 내용**
* Argument Passing

1) 입력받은 (프로그램 명 + 입력 변수)를 띄어쓰기 단위로 파싱한다

2) 1에서 파싱하여 얻은 토큰들을 마지막 것부터 스택에 집어넣는다.

3) word-align을 해주고, 0도 하나 넣어주어 토큰 영역을 표시한다.

4) 각 토큰이 저장된 주소를 가져와 스택에 넣는다. 이 역시 마지막 토큰의 주소부터 집어넣어준다.

5) 첫번째 토큰의 주소가 저장된 주소, 즉 4에서 첫번째 토큰의 주소가 저장된 주소를 스택에 넣는다.

6) argc, 즉 토큰의 개수를 스택에 넣는다.

7) return address로 0을 스택에 넣는다.

* User Memory Access

User program은 null 포인터나, 매핑되지 않은 virtual memory, 커널 영역을 가리키는 포인터 등의 invalid 포인터를 넘길 수도 있다. 이런 경우를 invalid memory access라고 한다.

가상 메모리 매핑은 프로그램 로딩 과정에서 막을 수 있다. Load() 함수 내부를 살펴보면 Install\_page()라는 user의 가상 메모리와 커널의 가상 메모리를 매핑해주는 함수가 있는데, 내부에서 pagedir\_get\_page()를 호출한다. 이는 user의 가상 메모리를 받아 이의 피지컬 주소를 찾고, 여기에 매핑된 kernel의 가상 메모리를 찾아주는 함수이다. 만약 매핑된 kernel의 가상 메모리가 없다면 NULL을 리턴하여 install\_page의 리턴값을 false로 만들어 프로그램의 loading을 실패하게 만든다. 커널 영역의 메모리를 참조하는 것을 검사하기 위해서는 is\_user\_vaddr(), is\_kernel\_vaddr()함수를 사용하면 된다. NULL 포인터 입력은 시스템콜 핸들러에서 시스템콜 함수들을 호출할 때 NULL 여부를 검사하면 된다.

* System Calls

시스템콜은 현재 실행 중인 프로그램이 허용되지 않은 작업을 수행하는 것을 막기 위해 필요하다. 사용자가 직접적으로 자원에 접근하지 못하고 반드시 OS를 거치도록 만든다.

이를 위해 운영체제는 user 모드(하드웨어 자원에 대한 접근 권한이 제한)와 kernel 모드(모든 접근 권한 가짐) 2개로 나뉘어져 있는데, 실행 중인 프로그램은 user 모드에 해당된다. User 모드에서 open, write, exec 등의 시스템콜 API를 호출하면 인터럽트가 발생하고 kernel 모드에서 인터럽트 벡터 안의 시스템콜 핸들러를 실행하여 해당하는 시스템콜을 호출한다.

Halt(): pintos 자체를 종료하기 위해 사용된다. Shutdown\_power\_off()를 호출한다.

Exit(): 실행중인 프로세스를 종료시키고, exit status를 부모 프로세스에게 리턴한다. 부모 프로세스가 없다면 커널에게 리턴된다.

Exec(): 실행할 프로그램의 이름을 받아 Process\_execute()를 사용해서 새로운 child 프로세스를 만들고 입력받은 프로그램을 실행한다. Child 프로세스의 아이디를 리턴한다.

Wait(): 입력받은 아이디를 가진 프로세스가 종료될 때까지 기다리고, 이의 exit status를 리턴한다. 해당 프로세스가 정상 종료되었고, 정상적으로 wait 했으면 이의 exit status가 리턴되지만 아닐 시 -1을 리턴한다.

Write(): 파일디스크립터 fd, write할 문자열이 담긴 버퍼, 문자열 길이 size를 입력받아 putbuf()를 이용해 fd에 출력한다. 이번 프로젝트에서는 STDOUT에 대해서만 구현하면 된다. Write한 길이를 리턴한다.

Read(): 파일디스크립터 fd, read해온 문자열이 담길 버퍼, read할 길이 size를 입력받아 input\_getc()를 이용해 fd에서 읽어온다. 이번 프로젝트에서는 STDIN에 대해서만 구현하면 된다. Read한 길이를 리턴한다.

Fibonacci(): 정수 n을 입력받아 n번째 피보나치 수를 리턴한다.

Max\_of\_four\_int(): 4개의 정수를 입력받아 그 중 가장 큰 수를 리턴한다.

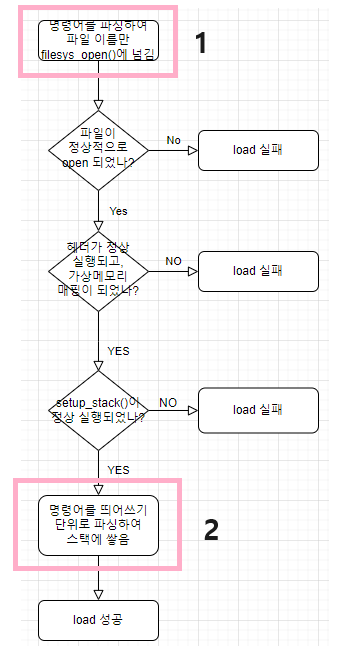
유저 레벨에서 시스템콜 API를 호출하면 스택에 각종 파라미터와 시스템콜에 해당하는 번호, 그리고 시스템콜에 해당하는 번호(핀토스에서는 0x30)을 담고 trap을 발생시킨다. 이 때, 커널 스택에 프로세스의 각종 값들을 저장해둔다. 이제 커널모드로 전환되는데, 이 때 하드웨어의 mode bit가 0으로 바뀐다. 이제 커널은 인터럽트 벡터에 저장된 인터럽트 핸들러 중 시스템콜 핸들러로 가서 해당하는 시스템콜을 호출한다. 이 과정이 끝나면 다시 return from trap 명령을 통해 유저모드로 돌아오고, mode bit 역시 1로 바뀐다. 이 때 커널 스택에 저장했던 값들도 pop되어 다시 프로세스에게 돌아온다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 첫주: argument passing 구현. Halt, read, write 등 간단한 시스템콜 구현. User memory access 구현
* 둘째주: wait, exec 등 process.c를 수정해야 하는 복잡한 시스템콜 구현.
* 셋째주: 불통과된 test 체크, 보고서 작성, fibonacci, max\_of\_four\_int 구현
  1. **개발 방법**
* Argument passing을 구현하기 위해 process.c의 load() 내부를 수정한다. 우선 프로그램을 실행하기 위해 입력된 명령어 중 프로그램명에 해당하는 부분만 파싱하여 filesys\_open()의 인자로 넣어준다. 이후 setup\_stack()까지 실행되면 입력된 명령어를 파싱하여 스택을 쌓아주면 된다.
* 시스템콜 핸들러를 구현하기 위해 userprog/syscall.c에 switch문을 추가해 호출된 번호에 대응하는 시스템콜이 실행되도록 한다. Halt, exit, exec 등의 시스템콜 함수들도 이 파일에 구현해준다.
* Invalid memory에 접근하는 것을 막기 위해 userprog/syscall.c에서 시스템콜 함수를 호출하기 전에 파라미터로 넘기는 주소값들에 대해 is\_user\_vaddr()을 통해 invalid memory인지 검사한다. Exception.c에서는 is\_kernel\_vaddr()을 사용해서 페이지폴트가 발생했을 때 커널영역의 메모리에 접근하는지를 검사하여 그렇다면 exit(-1)을 호출하도록 한다.
* Wait() 시스템콜은 userprog/process\_wait()를 사용해 구현한다. 자식 프로세스의 아이디를 저장하기 위한 자료구조가 필요하고, 자식의 종료를 기다리기 위한 방법도 필요하다. 따라서 threads/thread.h에서 struct thread에 list 타입 child\_list와 자신이 누군가의 자식임을 표시하기 위한 list\_elem 타입 Iamyourchild를 추가해준다. threads/thread.c의 init\_thread()를 수정하여 새 프로세스가 생성될 때 child\_list를 초기화하고 부모 프로세스의 child\_list에 Iamyourchild를 추가하는 로직을 추가한다. 그리고 struct thread에 세마포어 state를 추가해 자식의 exit 신호를 기다리도록 한다. 자식이 종료된 후 reaping되기 전 부모의 child\_list에서 자식 원소를 제거해야 하므로 세마포어 mem도 추가해준다.

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

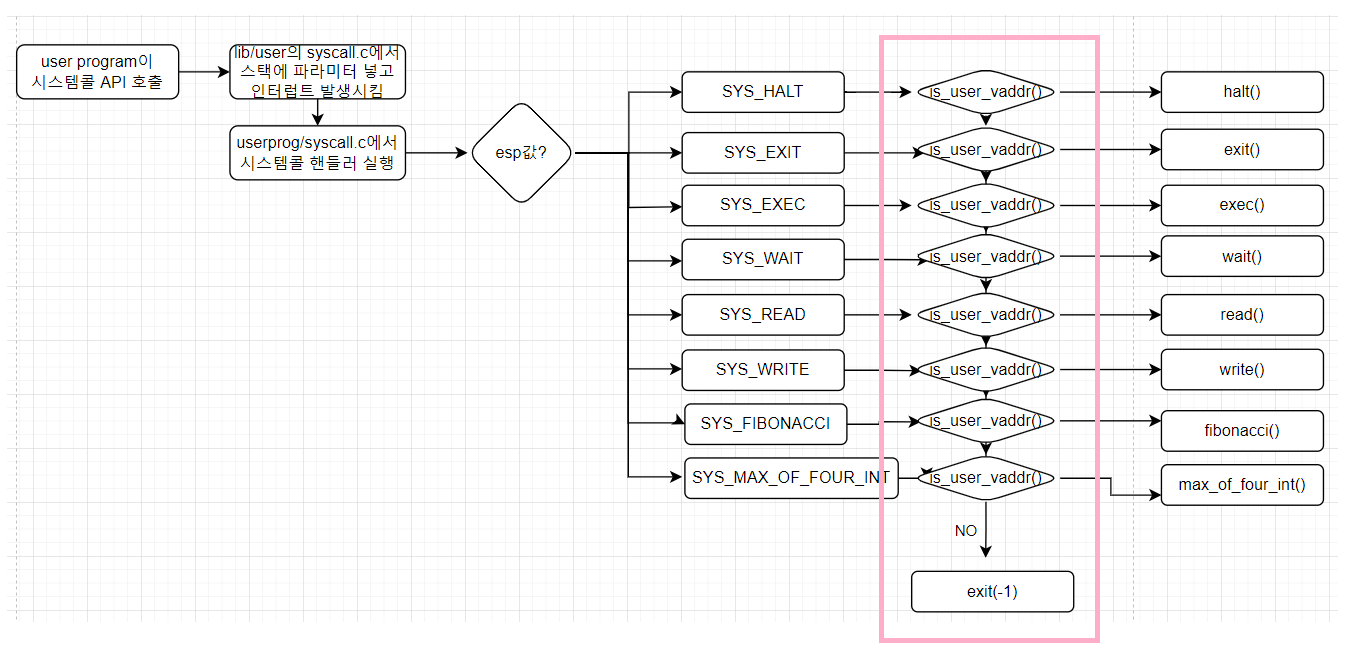
Load() 내에서 argument passing이 일어나는 과정은 아래와 같다.



1. User Memory Access

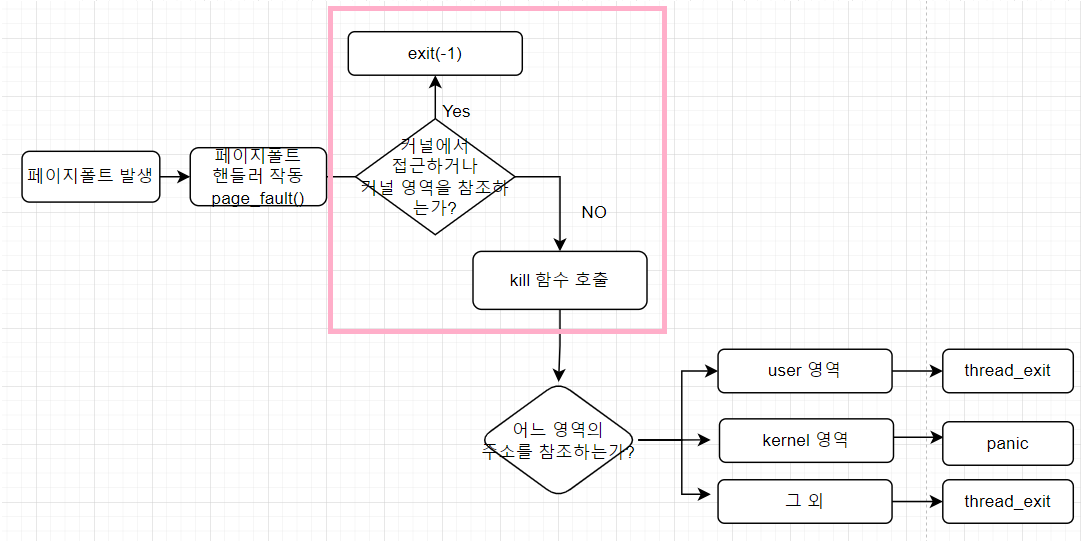
1) 시스템콜 호출

Userprog/syscall.c에서 시스템콜을 호출한다. 이 때 is\_user\_vaddr()을 통해 파라미터로 넘겨지는 주소를 검사해준다. 잘못된 주소가 넘겨지면 exit(-1) 한다.



2) 페이지 폴트 발생

Exception.c의 page\_fault()함수는 페이지 폴트가 발생했을 때 실행되는 일종의 핸들러이다.

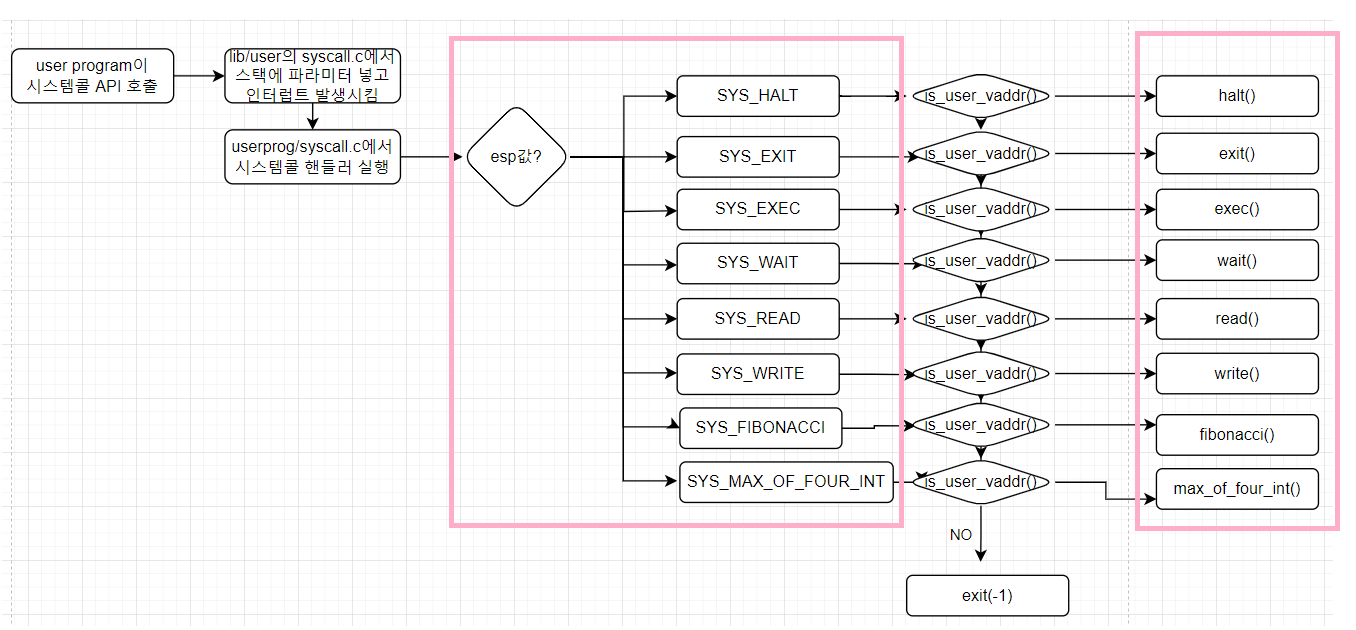


3) 가상 메모리 매핑

이 부분은 프로젝트에서 내가 추가한 것이 없어서 다이어그램을 그리지 않았다.

1. System Calls

userprog/syscall.c에서 switch문을 사용하여 esp에 담긴 시스템콜 번호에 해당하는 시스템콜을 호출한다.



**제작 내용**

1. Argument Passing

- 파일 이름 파싱하기

while문으로 널문자나 띄어쓰기를 만날 때까지 명령어를 한 글자씩 탐색하여 다른 배열에 복사한다. 이를 만나면 그 위치에 널문자를 넣어준다.

- 스택 쌓기

1) 입력받은 (프로그램 명 + 입력 변수)를 띄어쓰기 단위로 파싱한다

Strtok\_r()을 사용하여 파싱했다. Strtok\_r은 파싱된 문자열을 리턴하고, 더 이상파싱할 것이 없으면 NULL을 리턴하므로 While문을 사용하여 명령어를 전부 파싱하여 이들을 cmd\_parsed[]에 저장했다. 전체 토큰의 개수도 체크하여 cmd\_num에 저장한다.

2) 1에서 파싱하여 얻은 토큰들을 마지막 것부터 스택에 집어넣는다.

for문을 사용해 cmd\_parsed[]의 마지막 인덱스부터 0번째 인덱스까지 방문한다. esp에서 각 위치에 저장된 토큰(문자열)의 길이만큼을 빼주어 이들이 저장될 공간을 확보한다. 그리고 strlcpy()를 사용해 esp가 가리키는 메모리에 이를 저장한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

3) word-align을 해주고, 0도 하나 넣어주어 토큰 영역을 표시한다.

2 과정에서 모든 토큰의 길이를 더한 값을 저장해둔다. 이를 4로 나눈 나머지를 체크하고 그만큼 0을 채워넣어 Word-align을 해준다. 그리고 마지막으로 0을 하나 더 넣어준다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

4) 각 토큰이 저장된 주소를 가져와 스택에 넣는다. 이 역시 마지막 토큰의 주소부터 집어넣어준다.

1 과정 전에 esp\_copy에 esp의 값을 저장해두었다. for문을 사용해 cmd\_parsed[]의 마지막 인덱스부터 0번째 인덱스까지 방문하며 esp\_copy에서strlen( cmd\_parsed[i] +1 )만큼 빼면 1과정에서 i번째 토큰이 저장된 주소값이 나온다. 이를 esp가 가리키는 곳에 저장한다. 주소값은 4바이트이므로 esp에서는 4만큼씩만 빼주면 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

5) 첫번째 토큰의 주소가 저장된 주소, 즉 4에서 첫번째 토큰의 주소가 저장된 주소를 스택에 넣는다.

esp에서 4를 빼주고, 여기에 esp+4에 저장된 주소값을 넣어준다.

6) argc, 즉 토큰의 개수를 스택에 넣는다.

1에서 구한 cmd\_num을 넣어준다.

7) return address로 0을 스택에 넣는다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

Syscall.c의 switch문에서 각 케이스별로 유저 메모리를 접근하는 것이 맞는지 검사하는 과정을 추가한다. Is\_user\_vaddr()로 각 시스템콜에 넘길 파라미터를 모두 검사하고, 이들 중 하나라도 false가 된다면 exit(-1)을 하도록 한다. 아닐 경우에는 정상적으로 시스템콜을 호출하도록 했다.

Exception.c에서 페이지폴트가 발생했을 때 user의 접근이 아니거나, user가 커널 영역의 메모리를 참조하려고 했을 때는 exit(-1)을 하도록 해주었다. page\_fault()가 호출하는 kill()을 보면 커널영역의 메모리에 접근할 때만 커널이 패닉하기 때문에 유저가 커널에 접근하거나, 처음부터 커널이 접근한 경우는 미리 종료시켜주어야겠다고 판단했다. 테스트케이스 중 Sc-bad-sp에서 esp에 유저 메모리를 벗어난 주소를 담고 시스템콜을 호출했을 때 이 파일의 page\_fault()에서 에러 메시지를 띄어주는 것을 보고 이 부분을 수정할 수 있었다.

1. System Calls

가. halt

핀토스 자체를 종료하면 되기 때문에 복잡한 처리 없이 shutdown\_power\_off()만 호출해주었다.

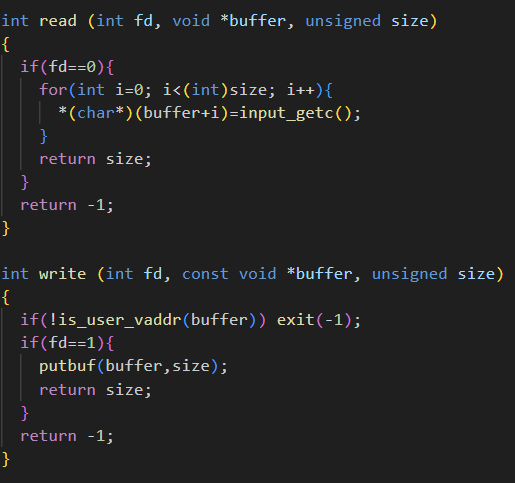
나. read, write

Read: for문을 사용하여 입력된 size만큼 (buffer+i)가 가리키는 위치에 input\_getc()로 읽어온 값을 저장해준다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

Write: for문을 사용하여 입력된 size만큼 putbuf()를 사용하여 buffer에 있는 값을 출력해준다.

****

이번 프로젝트에서는read, write 모두 STDIN, STDOUT에 대해서만 구현했다.

다. exec

입력된 file을 새로운 프로세스에서 실행하기 위해 process\_execute()에 넣어준다. Process\_execute()가 리턴해준 새로운 프로세스의 아이디를 리턴해주는데, 이는 tid\_t 타입이므로 pid\_t 타입으로 변환하여 리턴한다.

라. wait

Process\_wait()에 입력된 프로세스 아이디를 넣어 wait해준다.

1) thread.h에서 struct thread 구조 수정

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

자식프로세스를 저장할 child\_list와 거기에 저장될 Iamyourchild, 자식의 종료를 기다리기 위한 세마포어 state와 자식의 메모리해제를 방지하기 위한 세마포어 mem을 추가했다. 그리고 종료 상태를 저장하기 위한 exit\_status도 추가했다.

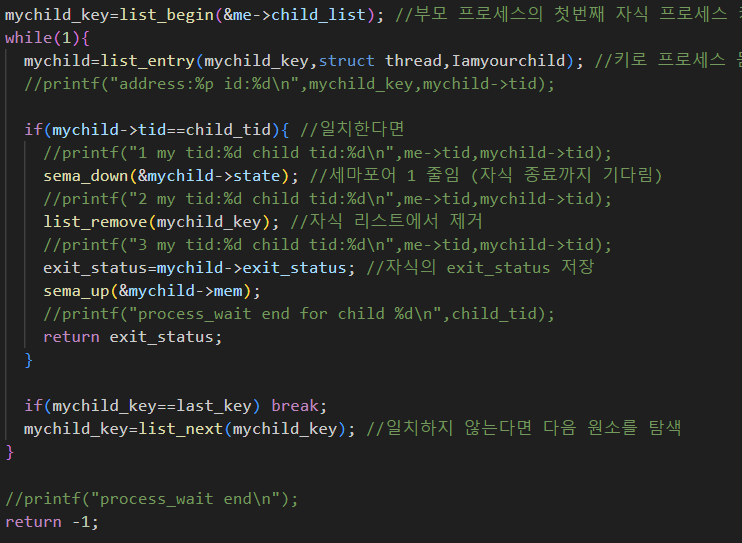
2) thread.c에서 init\_thread() 수정

1에서 struct thread에서 새로 추가한 원소들이 있기 때문에 이를 초기화시키는 코드를 추가했다. List\_init()으로 child\_list를 초기화해주고, state와 mem도 sema\_init()으로 0으로 초기화해준다. 부모 프로세스의 child\_list에 현재 생성중인 프로세스의 Iamyourchild를 추가하여 부모-자식 관계를 저장한다.

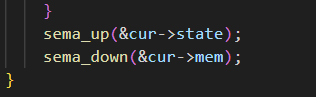
**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

3) process.c에서 process\_wait(), process\_exit() 수정

****

While()을 사용해 부모 프로세스의 child\_list에서 자식 프로세스를 하나씩 꺼내와 입력받은 자식 프로세스의 id와 일치하는 id를 가진 것을 찾는다. 만약 찾았다면, 자식은 종료될 때 세마포어 state를 0에서 1로 만들기 때문에 sema\_down()을 사용해 자식의 종료를 기다리고 child\_list에서 제거한다.

****

Child\_list에서 list\_remove()하기 전에 자식 프로세스에게 할당된 메모리가 없어져버리면 안된다. 처음에 mem 세마포어를 추가하지 않고 테스트를 해보았더니 exec 테스트들에서 에러가 떠서 위 while문에서 일일이 printf를 찍어가며 디버깅을 해보았더니 list\_remove에서 코드가 더 진행되지 않는 것을 확인할 수 있었다. 그래서 단순 종료와 메모리 해제는 다를 수 있겠다고 생각했고, 세마포어 mem을 추가한 후 process\_wait()에서 mem을 sema\_up()하고 process\_exit()에서 mem을 sema\_down하여 문제를 해결했다.

**마. exit**

종료될 프로세스의 이름과 입력된 status를 함께 출력해주었다. 그리고 프로세스의 exit\_status에 입력된 status를 저장하고 thread\_exit()을 통해 프로세스를 종료시킨다.

1. Additional System calls

1) lib/user/syscall.c, syscall.h 수정

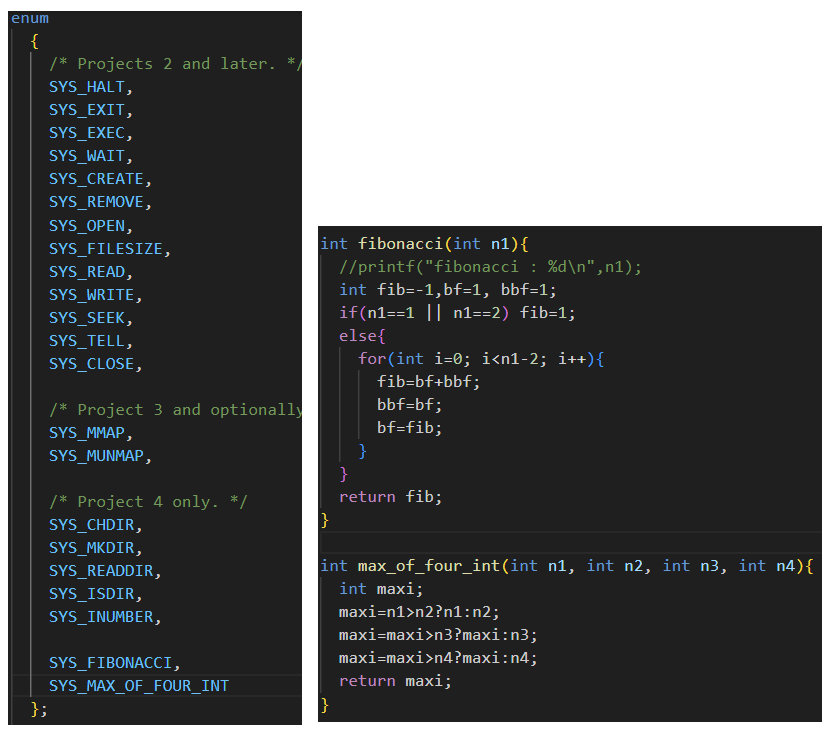
Max\_of\_four\_int는 파라미터가 4개인 시스템콜이기 때문에 이에 해당하는 어셈블리 코드를 syscall4로 하여 추가로 define했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

2) syscall\_nr.h, userprog/syscall.c, syscall.h 수정

syscall\_nr.h에 SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INT에 해당하는 enum을 추가했다. Syscall.c에는 해당 시스템콜 함수를 구현하고, switch문에 이들에 대한 케이스를 추가해주었다.



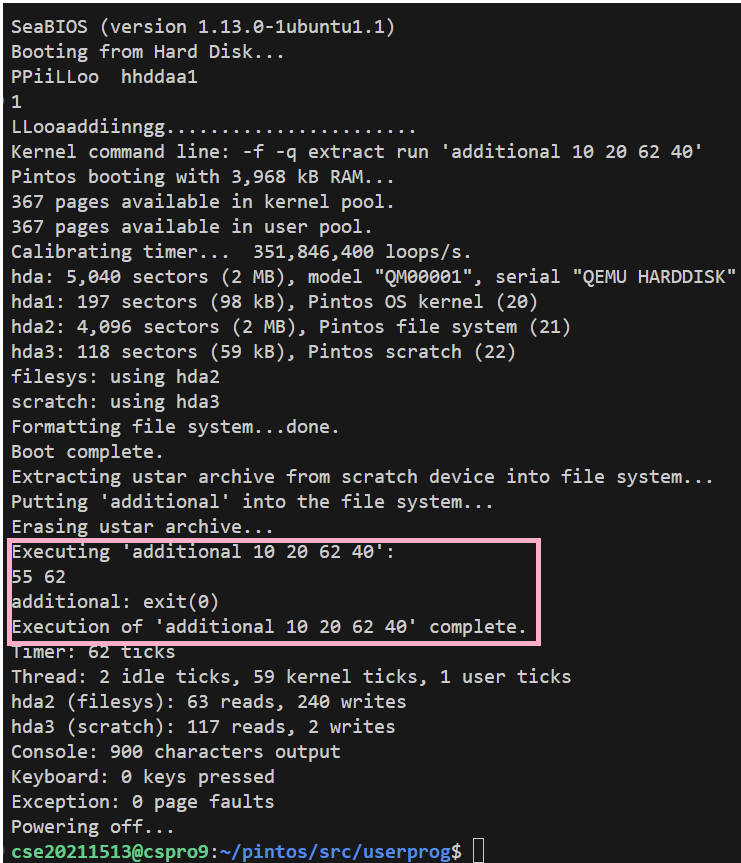
3) examples폴더에 additional.c 추가, makefile 수정

Examples 폴더에 fibonacci와 max\_of\_four\_int 시스템콜을 호출하는 프로그램을 추가하고, makefile을 수정하여 컴파일시켰다.

텍스트, 스크린샷, 폰트, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* 1. **시험 및 평가 내용**
* fibonacci 및 max\_of\_four\_int 시스템 콜이 정상 작동한다.

****