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

담당 교수 : 김영재 교수님

조 / 조원 : 20211584 장준영

개발 기간 : 2023.09.25 ~ 2023.10.05

1. **개발 목표**

앞으로의 pintOS project를 수행하기 위해, OS가 갖춰야 할 가장 기본적인 기능을 pintOS 내에 구현한다. 유저의 입력을 넘겨받는 argument passing, virtual address를 적합한 physical address에 할당하여 유저가 메모리를 사용할 수 있게 하는 virtual memory access validation, 유저가 커널의 기능을 사용 위탁할 수 있게 하는 system call 등을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

Project 1에서 요구하는 21가지의 평가 기준에서 모두 pass를 받을 수 있도록 구현하였고, additive system call 두 가지(fibonacci, max\_of\_four\_int)도 성공적으로 구현하였다.

1. Argument Passing

유저의 요청을 필요한 구간 별로 나눠 스택에 순서대로 집어넣어야만 커널이 요청을 인식할 수 있다. 이 과정을 argument passing이라고 한다. 유저의 커맨드 라인을 구간에 맞게 잘라 tokenize하고, 이를 pintOS가 인식하는 방법대로 쌓아 올리는 과정을 성공적으로 수행하여 커널이 명령어 인식을 할 수 있도록 하였다.

1. User Memory Access

OS는 유저가 사용할 수 있는 메모리 영역만을 유저에게 전달해야 하고, 그렇지 않으면 커널 메모리 영역에 접근해 시스템에 치명적인 오류를 야기할 수 있다. 이를 막기 위해 사용하고자 하는 메모리 영역이 유저 영역의 메모리인지 확인해야만 한다. is\_user\_vaddr, is\_kernel\_vaddr 등의 함수를 사용하여 user memory access validation이 잘 작동하도록 구현하였다.

1. System Calls

System call은 유저 프로그램이 OS의 기능을 사용하기 위해 요청하는 것으로, 주로 OS가 독자적인 권한을 갖고 실행할 때 시스템이 올바르고 공평하게 작동하는 동작들로 이루어져있다. 이번 프로젝트에서 구현해야 하는 여섯 가지 기본적인 system call(halt, exit, exec, wait, read, write)을 제대로 구현하였고, argument를 네 개 갖는 syscall4를 추가적으로 작성하여 fibonacci, max\_of\_four\_int 도 구현하였다.

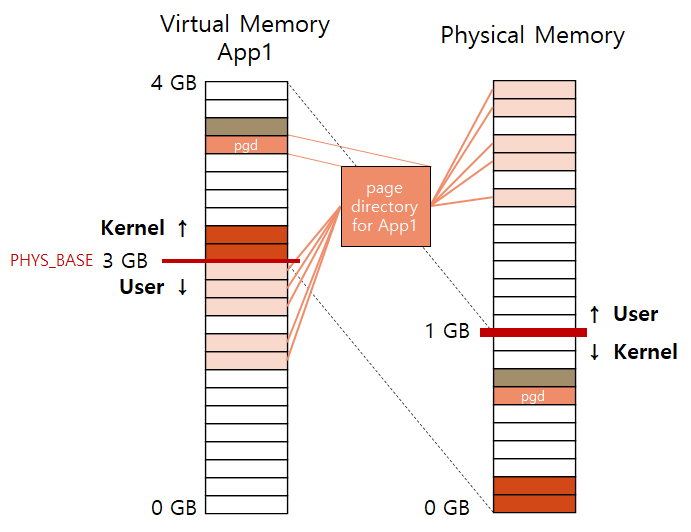
* 1. **개발 내용**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

가장 먼저, 유저가 입력한 command line 문자열을 argument로 tokenizing하는 과정이 필요하다. argument 별로 parsing을 마치면, 이를 스택에 쌓는 stack set up 과정을 거친다. ~/pintos/src/userprog/process.c에 구현된 setup\_stack() 함수를 통해 스택을 위한 메모리 할당이 이루어지고, 이 과정에서 스택의 top인 esp(extended stack pointer)를 조작하여 스택에 argument를 쌓을 수 있다.



프로젝트 자료에 나온 “/bin/ls -l foo bar” 명령어를 스택에 쌓은 모습이다. 우선 공백을 기준으로 parsing한 /bin/ls, -l, foo, bar을 역순으로 쌓는다. char 개수만큼의 byte를 차지하는데, 이때 OS에서 따르는 word alignment를 위해 홀수 개의 byte가 쌓였다면 padding으로 0을 넣어준다. 다음으로는 각각의 argument data를 가리키는 pointer를 쌓는다. 이 때 argv[argc]의 pointer는 NULL로 하여 쌓아야 한다. 마지막으로 argc를 쌓고 return address를 쌓으면 argument 스택이 완성된다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명



유저 프로세스가 커널에 system call을 보내 특정 기능을 요청한다고 설명하였는데, 이 과정에서 유저 프로세스는 OS에게 본인의 data나 pointer를 전달하기도 한다(read, write를 생각하면 편하다). 이 과정에서 전달받은 data(pointer)가 커널 영역에 침범한다면 시스템에 치명적인 오류가 발생할 수 있다. 또, 전달받은 pointer가 NULL이라면 system call이 올바르게 작동할 수 없다. 이를 OS 차원에서 사전 방지해야 시스템을 유지할 수 있다.

* + Invalid memory access를 어떻게 막을 것인지 설명

OS의 memory/address virtualization, page table 등에 대해서는 아직 자세히 배우지 않아 설명하기 힘들지만, ~/pintos/src/threads/vaddr.h에서 제공하는 PHYS\_BASE macro 기반 함수를 이용하여 간편하게 메모리 영역을 구별할 수 있다. is\_user\_vaddr 함수를 호출하면 해당 virtual address가 유저 메모리 영역에 있을 때만 true를 return한다. 유저 프로세스가 system call의 인자로 전달한 argument들을 해당 함수로 모두 체크한다면, invalid memory access를 막을 수 있다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

앞에서도 간략하게 설명하였듯이, system call은 시스템 내에서 권한이 있는 커널만이 동작할 때 시스템을 보호할 수 있다. 예를 들어, I/O나 File system에 대한 request를 어떤 프로세스도 처리할 수 있다면 시스템의 보안에 결함이 생기는 등 다양한 문제가 발생할 수 있다. 또, 메모리나 CPU는 한정된 자원이기 때문에 특정 규칙에 따라 모든 프로세스가 공평하게 사용해야 하고, 이 규칙을 세워 자원을 공평하게 분배할 수 있는 위상이 높은 분배자(커널)가 필요하다. 따라서, 유저 모드와 커널 모드를 나누어 유저가 system call이 필요한 시점에 커널을 호출하고, interrupt를 통해 커널 모드로 전환되어 커널이 대신 이 작업을 마치고 return하는 방식으로 system call이 동작한다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

1. halt) 시스템 자체를 절차 없이 강제로 종료할 때 사용하는 시스템 콜이다. 별도의 구현 없이 pintOS에서 지원하는 shutdown\_power\_off() 함수를 호출하면 된다.

2. exit) exit call을 한 유저 프로세스를 종료하는 시스템 콜이다. 추가적으로 exit status를 커널에 전달한다. 이는 프로세스가 어떻게 종료되었는지에 대한 정보를 커널에 제공한다.

3. exec) 유저가 전송한 커맨드 라인(file\_name)을 통해 새로운 프로세스(thread)를 생성해 그 파일을 실행하는 시스템 콜이다. 시스템프로그래밍에서 배운 exec과 달리 fork까지 atomic하게 실행한다는 차이가 있다.

4. wait) 어떤 프로세스가 자기 child 프로세스의 exit status를 기다리도록 요청하는 시스템 콜이다. child 프로세스가 종료된 이후에 suspension이 해제되어야 하므로 synchronization을 신경써서 구현해야 한다.

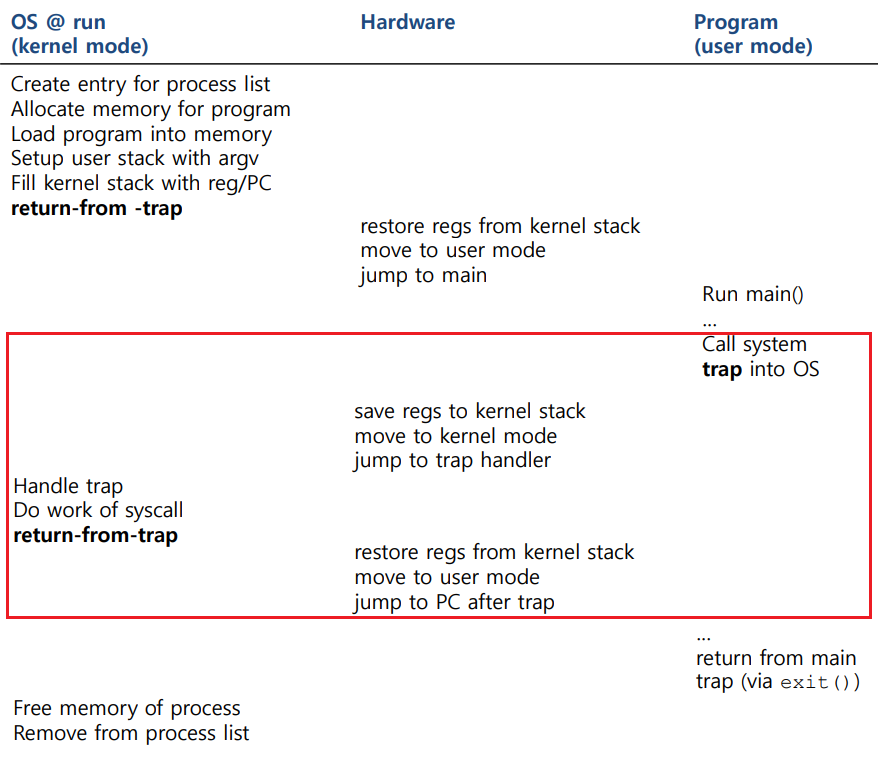
5. read) fd(file descriptor), buf(buffer), nbytes 세 개의 인자를 전달해 fd에서 nbytes 만큼 읽어 buf에 쓰고 싶을 때 사용하는 시스템 콜이다. 이번 프로젝트에서는 stdin에 대해서만 구현하였다.

6. write) read와 반대로 fd에 buf에 있는 nbytes를 쓰고 싶을 때 사용하는 시스템 콜이다. 역시 이번 프로젝트에서는 stdout에 대해서만 구현하였다.

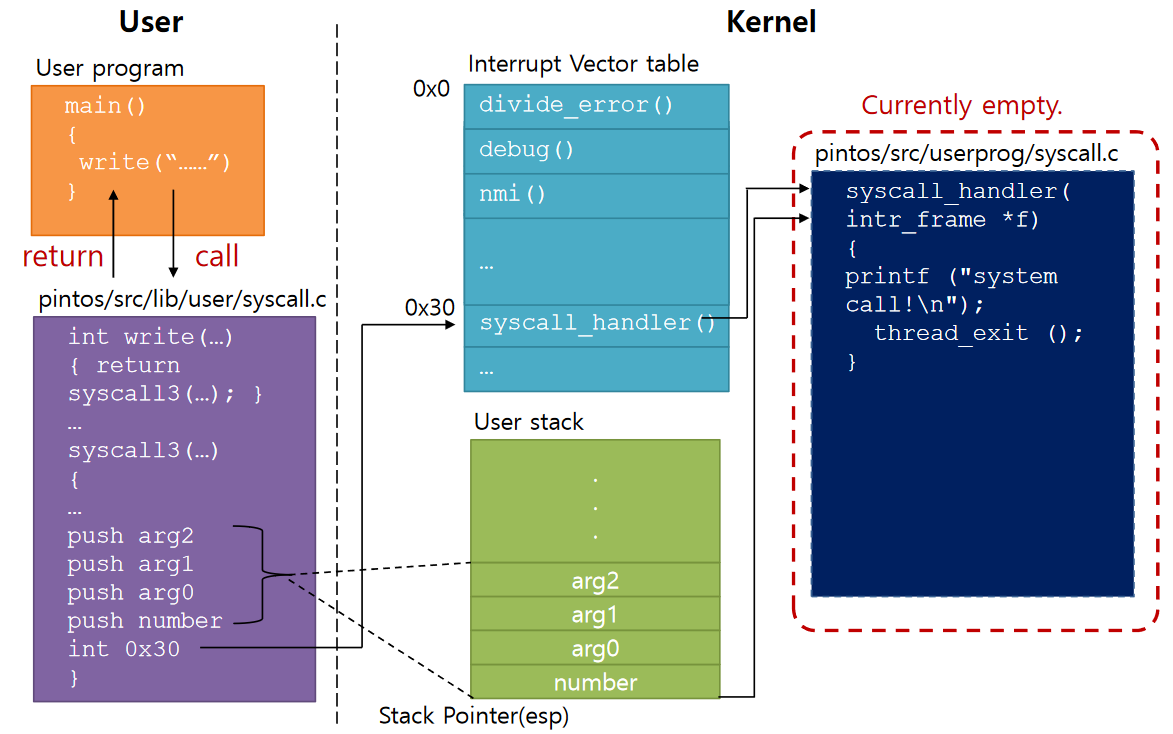
7. fibonacci) n번째 피보나치 수열 항을 반환하는 시스템 콜이다.

8. max\_of\_four\_int) 네 개의 정수 중 가장 큰 정수를 반환하는 시스템 콜이다. argument가 네 개이기 때문에, 기존에 구현되어 있지 않은 syscall4를 새로 만들어야 한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

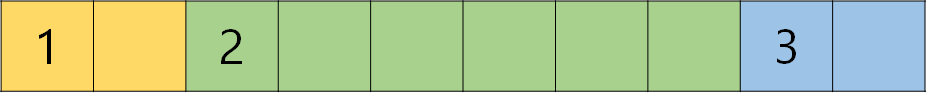


운영체제 강의 자료에 있는 흐름표이다. 유저가 시스템 콜을 호출한 시점부터 확인하면, 시스템 콜이 호출되면 trap instruction으로 인해 kernel mode로 전환될 준비를 시작한다. 이후 하드웨어에선 시스템 콜을 이용하기 위해 필요한 정보들(syscall number, arguments, parameters, …)을 OS로 전달한다. trap instruction을 받은 OS는 시스템 콜 핸들러를 통해 시스템 콜에 맞는 동작을 하고, 결과 값을 반환한다(이는 eax 레지스터에 반환 값을 저장하는 형식으로 보통 구현된다). OS가 핸들링을 마치면 다시 유저 모드로 전환되어 유저는 반환값을 이용한다.



write의 예시를 확인해보자. 유저 프로그램에서 write가 호출되면 라이브러리 내의 write 래퍼 함수는 argument와 함께 syscall3이라는 시스템 콜을 호출한다. syscall3이 전달해야 하는 정보를 스택에 삽입하고 int 0x30이라는 trap instruction을 호출히면, 이후 interrupt table에서 0x30에 맞는 interrupt, 즉 syscall\_handler를 호출한다. 커널의 syscall\_handler는 이후 interrupt와 함께 전달된 정보를 통해 해야 할 일을 하고, 반환 값을 eax 레지스터에 넣은 후 동작을 마친다(그림에서는 구현해야 하는 부분이 비어있다).

1. **추진 일정 및 개발 방법**
   1. **추진 일정**



프로젝트를 시작하고 이틀 정도는 너무 막막해서 수정해야 하는 pintOS의 소스 코드를 모두 읽어보았다. 주로 ~/pintos/src/userprog, threads의 코드와 헤더 파일을 전부 읽어 내가 채워야 할 부분과 수정해야 할 부분, 사용해야 할 함수 등을 완전히 이해하기 위해 노력하였다. 이 과정에서 시스템 콜 흐름에 따라 구현해야되는 부분, 순서 등을 파악하게 되었다.

이후 6일정도 코딩하였다. 먼저 ~/pintos/src/userprog/syscall.\*과 exception.c를 수정하면서, 시스템 콜 종류에 따른 브런칭을 진행해 시스템 콜 핸들러의 윤곽을 잡고, 핸들러 차원에서 처리해야 할 invalid memory access validation process를 짰다. 윤곽을 잡은 이후, ~/pintos/src/threads/thread.\*와 ../userprog/process.c를 수정하면서 시스템 콜이 올바르게 작동할 수 있도록 하였다. synchronization을 위한 semaphore를 thread structure에 추가하거나, process\_exec, process\_wait 등의 함수를 구현하였다. argument passing을 위한 parsing 함수도 작성하였다. 마지막으로, additional system call을 위해 ~/pintos/src/lib/user/syscall.\*, ../lib/syscall-nr.h 등을 수정하고 additional.c를 작성하였다. 특히, max\_of\_four\_int를 구현하기 위해선 기존 코드엔 없는 syscall4를 구현해야 했다. 최종적으로, System calls -> User Memory Access -> Argument Passing 순으로 구현하였다.

마지막 이틀은 보고서를 작성하였다. 추석 연휴에 코딩에 몰두하여 보고서 작성은 빠르게 마무리할 수 있었다.

* 1. **개발 방법**

1. Argument Passing

Argument passing 과정은 대략 parsing(tokenizing)을 마친 arguments를 setup\_stack으로 마련한 공간에 쌓는 걸로 이루어져 있다. 이 모두 ~/pintos/src/userprog/process.c에서 구현할 수 있다. load 함수 내에서 이 과정을 전부 처리하기 때문이다.



먼저 parsing은 반복문을 돌려 단순히 공백에 따라 문자열을 분해하는 과정이다. 처음에는 strtok 함수를 사용하려고 했지만, 버퍼 오버플로/언더플로 문제가 있어 컴파일 과정에서 여러 오류 메시지를 보기도 하였고, 외부 함수를 사용하니 디버깅이 어려웠다. 따라서, 반복문을 직접 돌려 parsing 하는 int parsing(char \*\*argv, char \*buf) 함수를 만들었다. argv 문자열 배열에 arguments를 삽입하고, 그 개수인 argc를 return하는 함수이다.

이후 스택을 쌓는 과정은 앞서 설명한 일련의 절차를 거친다. 이는 시스템프로그래밍 과제인 malloc-lab때처럼 정교한 포인터 연산이 필요하다. 정확한 바이트 수를 위해 적재적소에 포인터 자료형 변환(uint8\_t, uint16\_t, uint32\_t, …)를 사용해야 한다.

2. User Memory Access

User memory access validation은 ~/pintos/src/userprog/syscall.\*에서 간단하게 구현할 수 있다. 이 내부엔 syscall\_handler가 있는데, 여기서 유저가 넘긴 interrupt frame을 통해 argument와 syscall type(num)을 알 수 있다. syscall number에 따라 필요한 argument의 개수와 자료형이 정해져 있으므로, esp를 적절히 조절하여 내부 값을 참조하고 이 주소가 적절한지만 확인하면 된다. 반복문을 통해 모든 argument의 pointer를 확인하고, is\_user\_vaddr을 사용하여 커널 영역을 침범하진 않았는지, NULL로 누락되진 않았는지를 모두 확인한다. 만약 문제가 있다면 exit(-1) call을 한다. 이를 통해 제대로 전달된 메모리에만 시스템 콜이 작동되로록 할 수 있다.

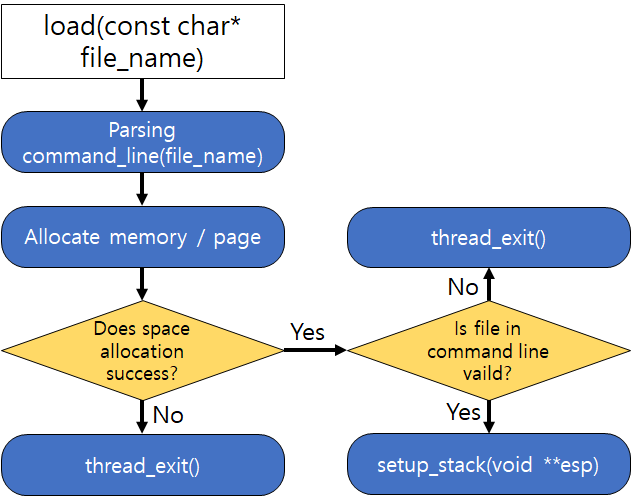
하지만 아무리 이 부분을 구현해도 \*-bad-ptr에서 계속 FAIL을 받을 때가 있었다. 이는 page fault가 발생하였을 때의 처리가 누락되어서였다. 코드를 읽어보니 page fault handler는 exception.c 파일에 있었고, 여기에서도 마찬가지로 invalid memory access가 감지되면 exit(-1)을 통해 프로세스가 중지되도록 구현해주어야 한다.

3. System Calls

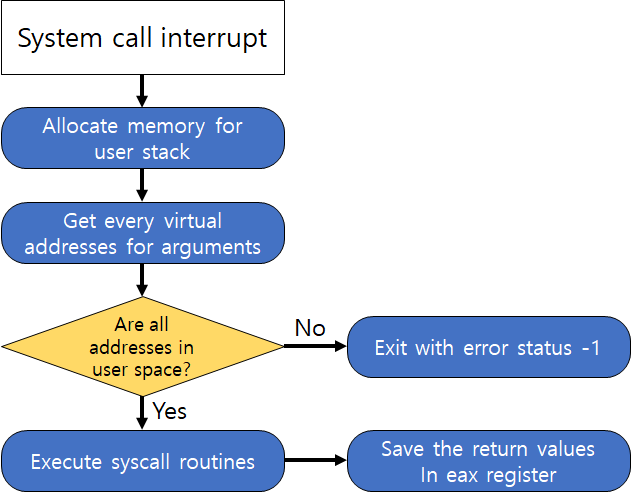
시스템 콜 역시 ~/pintos/src/userprog/syscall.\*, process.\*에서 구현한다. 또, synchronization이 필요한 경우 ~/pintos/src/threads/thread.\*도 수정할 필요가 있다. 우선 syscall\_handler를 syscall.c에 구현하기 위해 사용자의 interrupt frame에서 정보를 얻어 해당 routine을 호출하는 과정을 구현해주어야 한다. 이를 switching으로 구현하였으며, 그 과정에서 invalid memory access도 막아낸다. process.c에 틀은 구현되어있는 process\_execute, process\_wait, process\_exit 등을 제외하고 exit, read, write, halt, fibonacci, max\_of\_four\_int는 syscall.c 내에서 스스로 subroutine을 구현해야 한다(additional syscall은 ~/pintos/src/lib/user/의 관련 파일에 정보를 추가해주어야 한다는 것을 잊으면 안된다. 또, max\_of\_four\_int는 syscall4를 user 디렉토리의 파일에 새로 만들어주어야 한다. 이는 후술하겠다).

synchronization을 위해 thread.\* 파일을 손봐야 한다는 것을 좀 더 자세히 설명하자면, 앞서 시스템프로그래밍 시간에 배운 semaphore가 해당 폴더의 synch.\*에 구현이 되어있으므로 그걸 사용할 것이다. 또, 각각 프로세스의 성질이 담긴 thread structure도 수정해야 한다. thread structure에는 synchronization을 위한 semaphore 두 개와 프로세스의 child list, exit status 등이 추가된다. 이를 통해 wait, exec이 제대로 수행될 수 있다. 자세한 내용은 뒤의 코드 리뷰에서 설명한다.

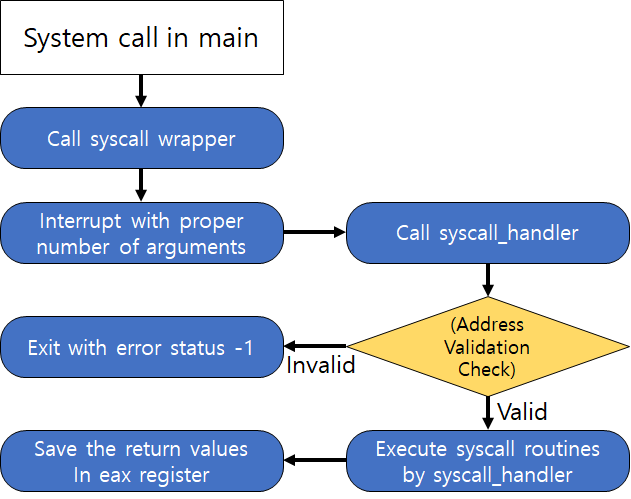
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



1. System Calls



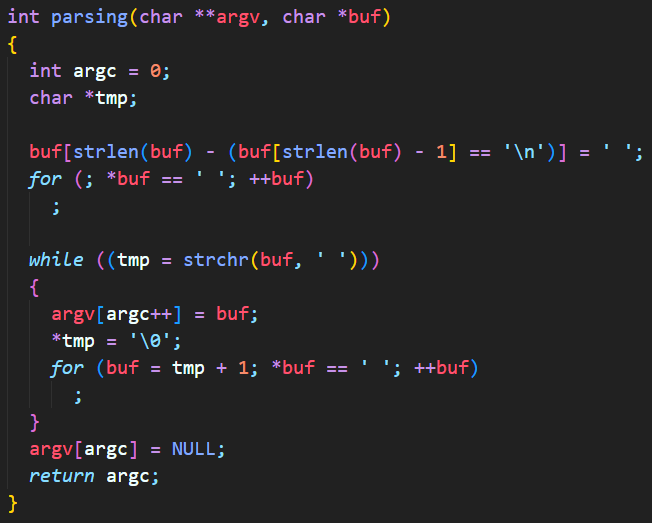
* 1. **제작 내용**

1. Argument Passing

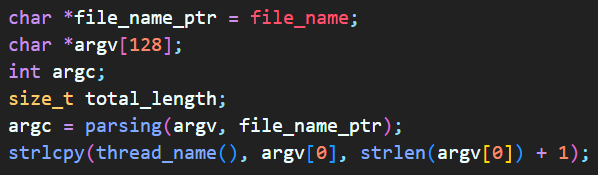
Argument passing은 앞서 설명한 parsing, load(set stack arguments)로 이루어진다. 이는 process.c에서 구현된다.



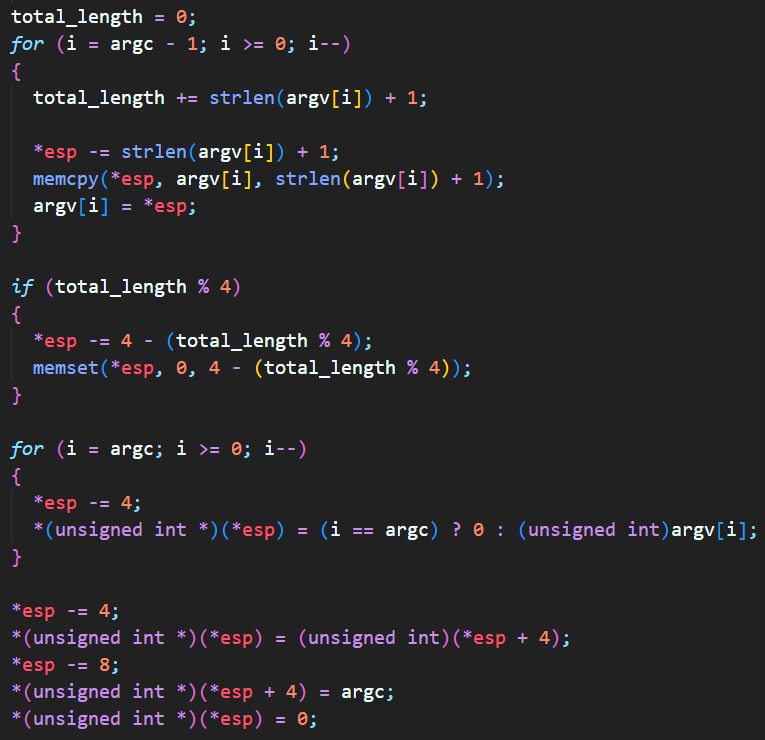
parsing 함수는 command line을 받고 argv에 parsing된 argument를 넣은 다음 그 개수인 argc를 return한다. 처음에는 위 사진과 같이 strtok을 이용하여 작성하였는데, 컴파일 과정에서 warning이 많이 뜨고, 실제로 알 수 없는 이유로 작동이 안되기도 하였다. 이에 따라, 문자열을 모두 확인하면서 나이브하게 parsing하는 함수로 아래와 같이 수정하였다.



strchr과 함께 버퍼에 담긴 모든 문자를 확인하면서, 공백을 발견할 때마다 단어로 나누어서 담아준다. 이 과정을 통해 스택에 넣어야 할 argument와 그 개수를 정리할 수 있다.



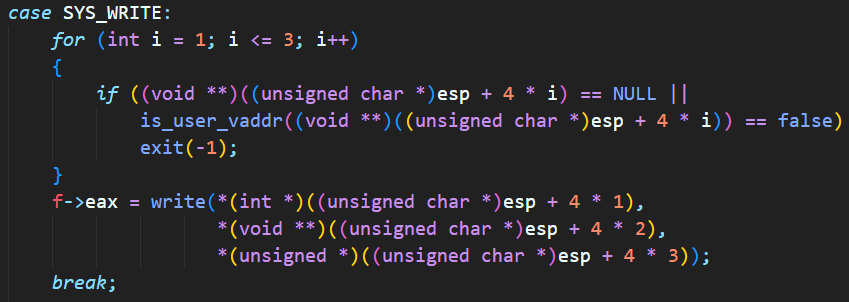
기존 load 함수에 구현된 부분 이외에 추가로 수정한 부분이다. parsing에 이용하기 위한 argv, argc, 스택 설정을 위한 total\_length 등을 추가로 선언하였다. 또, 이후 system call에서 이용하기 위해 유저의 명령어를 thread\_name에 넣어주었다(후술).



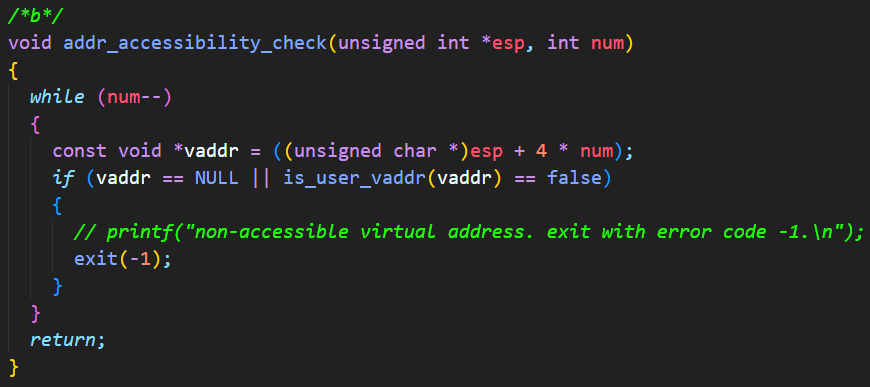
복잡한 포인터 연산으로 스택에 내용물을 쌓는 과정이다. 이 과정은 앞서 설명한 스택 쌓는 법을 그대로 따르고 있다. void \*\*esp에 대해 \*esp를 4바이트씩 확장하며 데이터를 쌓는다. argv를 역순 순회하며 스택에 요소를 넣고, 그 길이를 기록한다. 길이를 통해 argument가 word alignment가 안되었다는 것을 파악하면 한 칸 더 늘려 0으로 된 padding을 넣어준다. 이후 argument pointer를 쌓고, 마지막으로 argv pointer, argc, 0을 넣으면 스택이 만들어진다. 단위와 크기를 고려하는 포인터 연산인 만큼 상당히 복잡하고 정교한 자료형 변환이 요구되었다.

1. User Memory Access

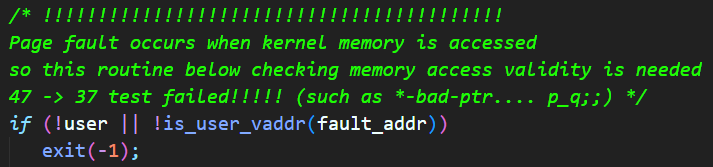
System call로 넘어온 user memory를 체크하기 위해, ~/userprog/syscall.c에서 argument를 하나씩 확인한다.



syscall\_handler 함수에서 SYS\_WRITE를 처리하는 부분이다. Interrupt frame의 eax에 결과값을 넣기 위해 write를 호출하기 이전에, 반복문을 돌며 유저의 argument가 valid한지 확인하는 부분이 있다. write는 세 개의 argument를 받으므로, 스택을 1칸, 2칸, 3칸 탐색하며 포인터가 NULL은 아닌지, 유저 영역의 메모리가 맞는지를 확인한다.



맨 처음의 구현 이후 루틴이 반복되는 것을 발견하여 위와 같은 함수를 만들어 처리하려 했지만, 알 수 없는 오류가 계속 발생하여 다시 나이브하게 라인을 작성하였다. 시간이 충분하다면 다시 매크로나 함수를 이용하여 이 부분을 가독성 좋게 고쳐보고 싶다.

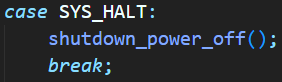


이전에 얘기한 것처럼, syscall\_handler에서 메모리를 체크하는 것만으로는 부족하다. 처음에는 계속되는 FAIL의 이유를 알 수 없었고, 내가 실패한 메모리 관련 채점 기준 항목을 검색해보니 exception.c의 page fault handler에서 잘못된 메모리 영역을 참조하는 것이 원인이라는 사실을 알게 되었다. 이후 page\_fault() 함수가 원인이 되는 메모리의 특정을 마쳤을 때 위와 같은 체크 루틴을 추가해주니 10개의 FAIL을 pass로 바꿀 수 있었다.

1. System Calls

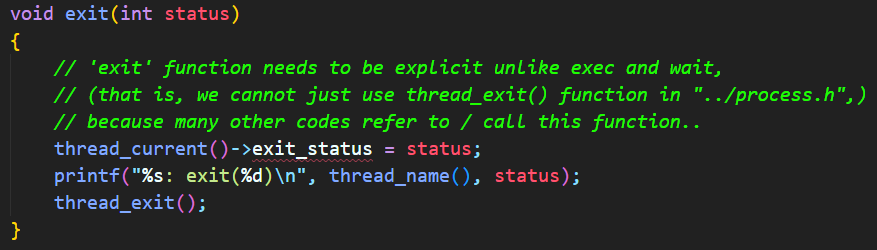
시스템 콜이 호출되면 syscall\_handler에서 각각의 루틴을 진행한다. 이미 설명한 핸들러의 형식은 건너뛰고, 각각 시스템 콜의 루틴을 코드 레벨에서 자세하게 확인하여 동작을 알아본다.

[1] halt

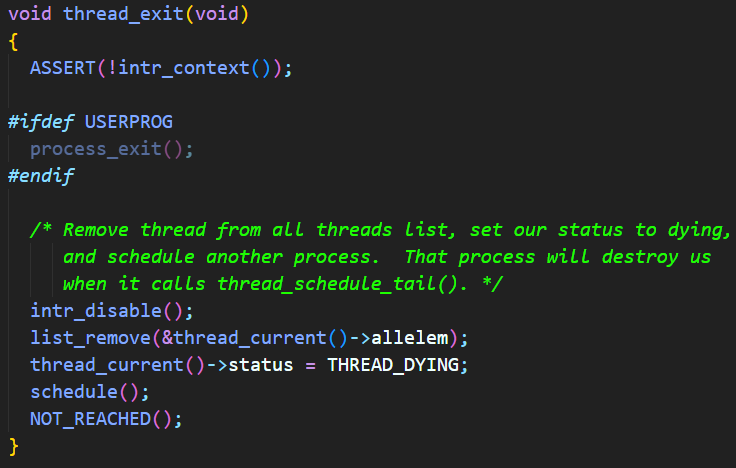


halt의 처리는 간단하다. pintOS에서 지원하는 shutdown\_power\_off() 함수를 호출하면 유저 프로세스를 강제로 종료한다. 프로세스의 상태나 경과를 저장하지 않고 즉시 종료한다.

[2] exit

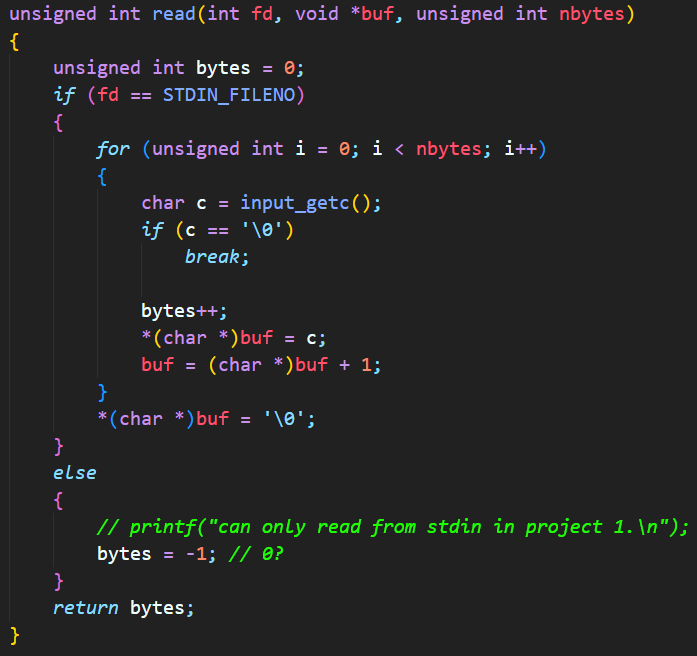


exit이 호출되면 현지 실행되는 프로세스(쓰레드)의 exit status를 정해주고 종료한다(후술할 시스템 콜에서 사용된다). 처음에는 halt와 마찬가지로 switch branching 선에서 끝내려고 했는데, 다른 코드나 함수에서 exit(status)를 호출하는 경우가 많아 그렇게 하면 컴파일 에러가 발생한다. 따라서 명시적으로 exit 함수를 만들어주어야 한다.



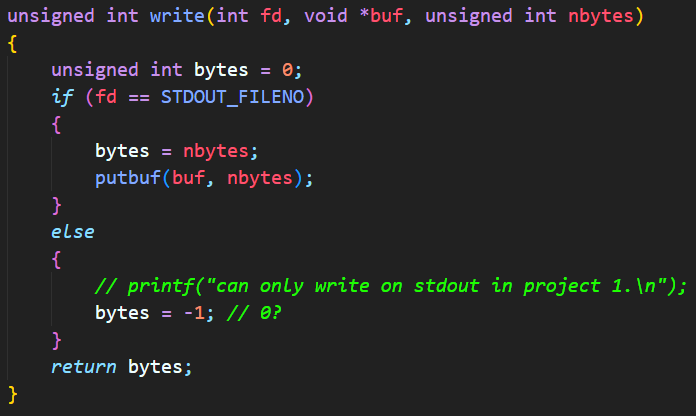
단순히 thread\_exit을 호출하긴 하였지만, 이 함수를 thread.c에서 확인하면 USERPROG(user process)인 경우 process\_exit의 래퍼 함수의 형태인 것을 확인할 수 있다. 이는 이후 process\_exit, process\_wait, process\_execute 등의 함수를 아우르는 semaphore lock synchronization과 연관이 있으므로 확인해 둘 필요가 있다.

[3] read



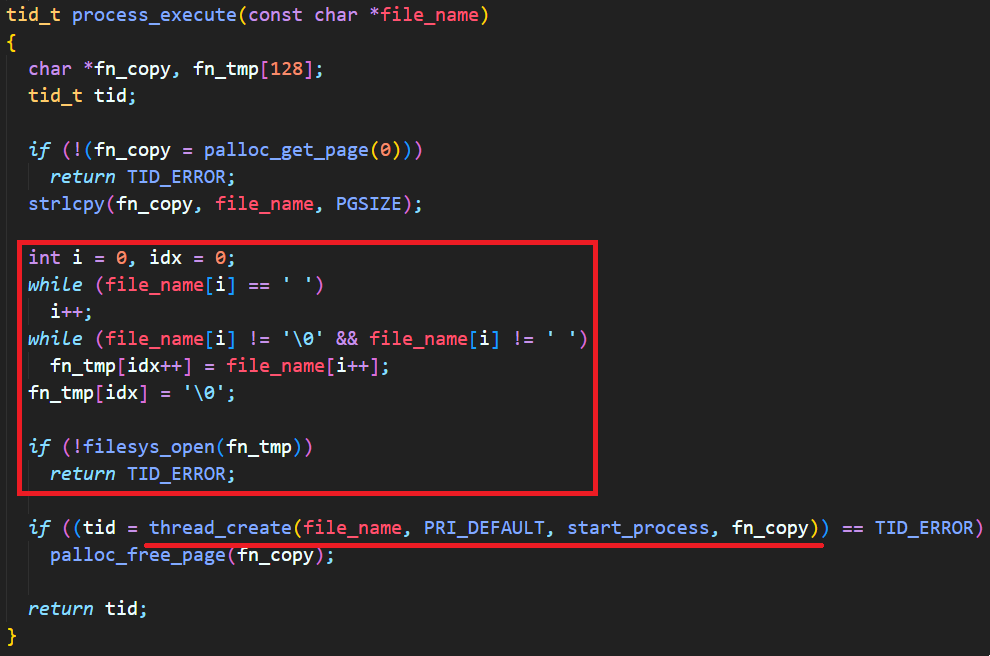
이번 프로젝트에서는 stdin에 대해서만 read하기 때문에 구현이 간단하다. fd에서 읽고 유저가 넘긴 버퍼에 nbytes 이하로 쓴 후 쓴 바이트 수를 return하면 되는데, 이 경우 ~/pintos/src/devices/input.c에 정의된 input\_getc라는 함수를 이용해 한 글자씩 읽는다. 한 글자씩 읽어 버퍼에 삽입하고, stdin에서 널캐릭터를 만나면 버퍼 맨 뒤에 널캐릭터를 삽입한다.

[4] write



write에선 ~/pintos/src/lib/kernel/console.c에 정의된 putbuf 함수를 사용한다. putbuf는 크기와 버퍼 포인터를 넘겨주면 통째로 그만큼 쓰는 함수이므로, return할 bytes를 1씩 더하지 않고 nbytes를 그대로 할당하는 것이 특징이다.

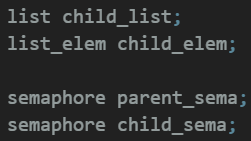
[5] exec



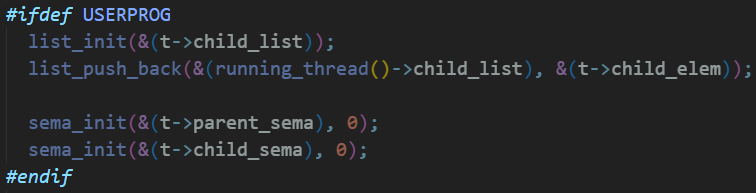
지금부터는 ~/pintos/src/userprog/process.c를 사용한다. exec의 경우 여기의 process\_execute 함수를 호출한다. 기존 구현부를 제외하고 추가로 제외한 부분에서, file\_name(command line)을 normalize해주었다. 이후 ~/pintos/src/filesys/filesys.c의 함수인 filesys\_open을 호출해 file\_name이 잘못되었다면 -1을 return하도록 하였다. 모든게 정상회되면 이후 ~/pintos/src/threads/thread.c의 함수인 thread\_create를 한다. 이는 시스템프로그래밍에서 배운 fork와 동일한 역할로, 새로운 프로세스(쓰레드)에서 요청이 수행될 수 있도록 한다. 쓰레드가 생성되어 어떤 동작을 하는지는 synchronization issue와 밀접한 관련이 있으므로 wait에서 후술한다.

[6] wait

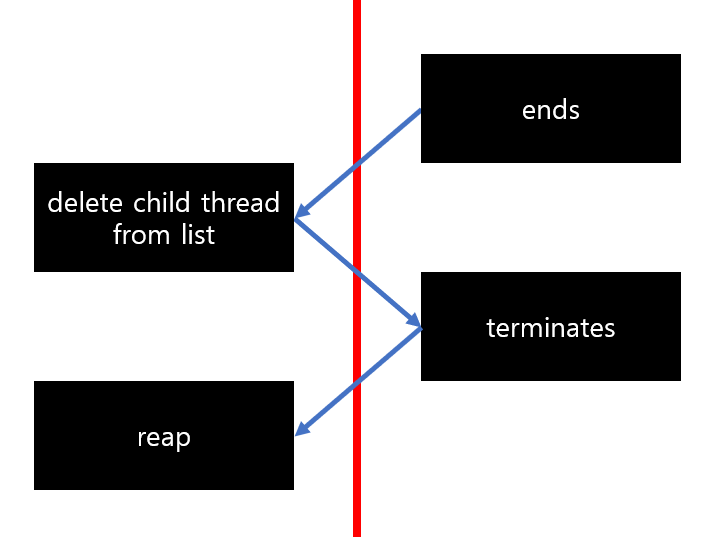
exec에서 확인했듯 pintOS에서 프로세스는 하나의 쓰레드로 관리된다. wait call의 구현을 위해선 1) 자신의 child 쓰레드가 정지되었다는 걸 알아낼 수 있어야 하고, 2) 그때까지 종료되지 않도록 suspend되어야 한다. 이를 위해 쓰레드별로 1) child 쓰레드 정보를 저장할 수 있는 자료 구조와 2) mutex(semaphore)가 필요하다.



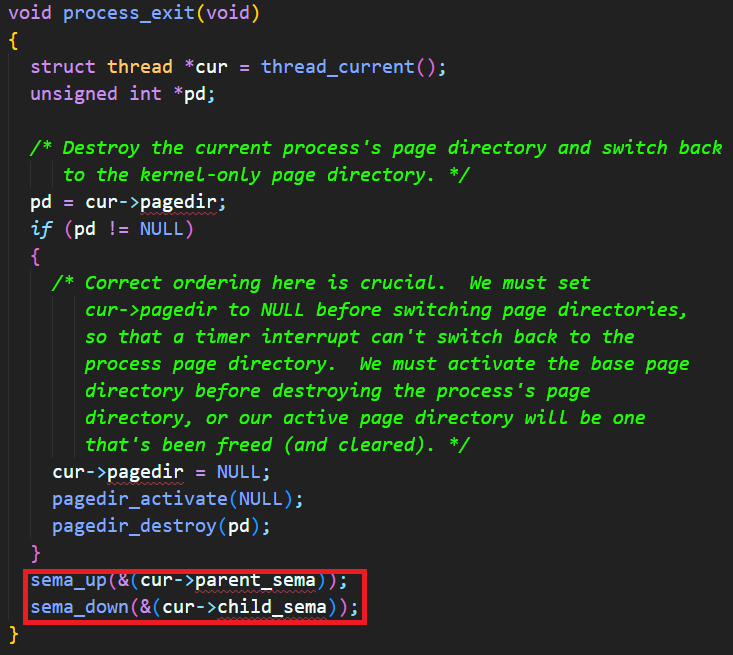
thread.h의 struct thread에 추가한 요소이다. child 쓰레드를 관리하기 위한 list와 syncronization을 위한 semaphore가 각각 선언되었다.



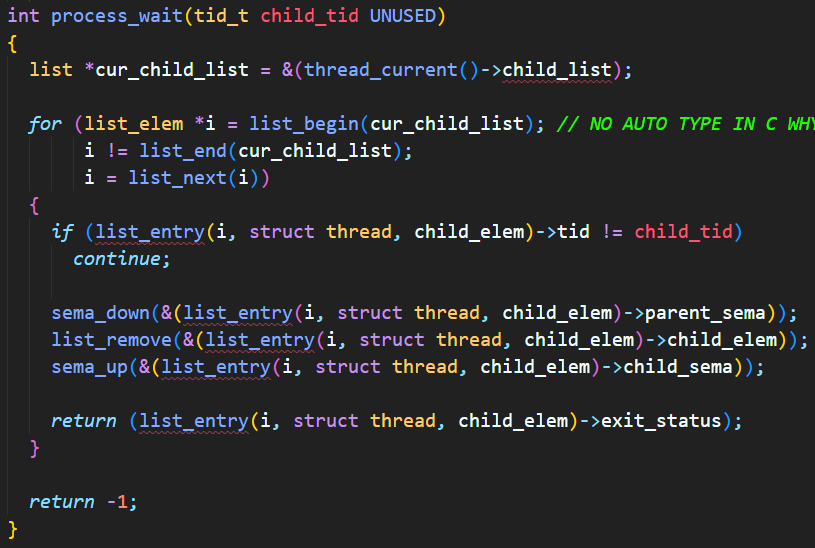
이후 쓰레드가 생성되면 초기화를 해주는 함수인 init\_thread에 다음과 같은 초기화 과정을 거친다. 이 과정에서 list와 semaphore가 모두 초기화되어 사용할 수 있는 상태가 된다.



semaphore lock은 다음과 같은 단계로 진행된다. wait 중인 parent는 lock된 상태로, child 쓰레드가 실질적인 동작을 종료하면, child를 lock하면서 parent를 unlock한다. 이후 parent는 child\_list에서 child 쓰레드를 제거하고 다시 child를 깨운다. exit/return까지 마친 child 쓰레드를 parent가 reap하고 절차가 종료된다. 이를 위해 process.c의 process\_wait과 process\_exit을 수정할 것이다.



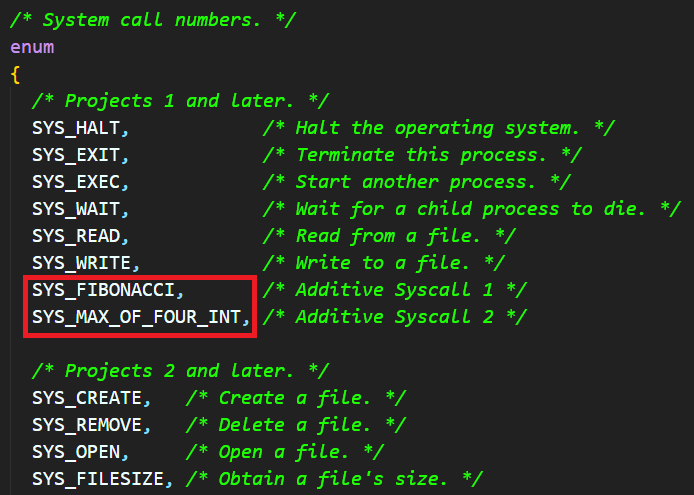
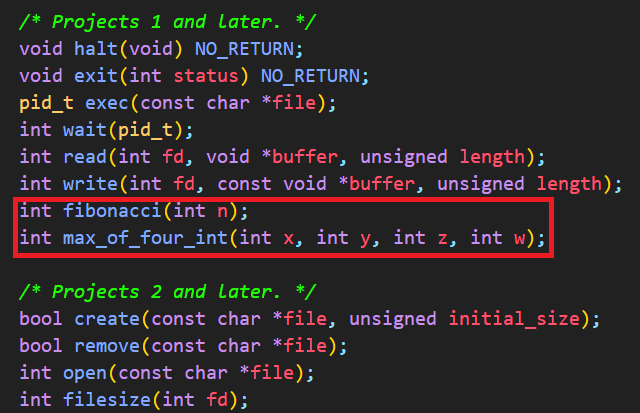
먼저 process\_exit에서의 과정이다. 앞선 과정을 마치면 자고있는 parent를 깨워야 하므로 parent\_sema를 up한다. 이후 자신은 down하여 리스트에서 사라지기 이전에 완전히 종료되는 것을 막는다.



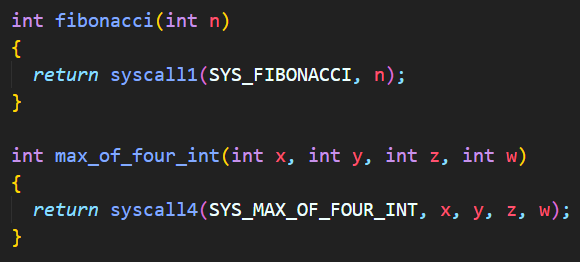
process\_wait에서는 child\_list를 순회하며 wait하고자 하는 child의 차례가 되었을 때 semaphore를 down하여 기다린다. child의 exit에 의해 잠에서 깨면 child\_list에서 해당 child를 지우고 자고있는 child를 깨워 종료시킨다. 이후 child의 exit status를 return하면 wait 과정이 끝난다.

1. Additional System calls

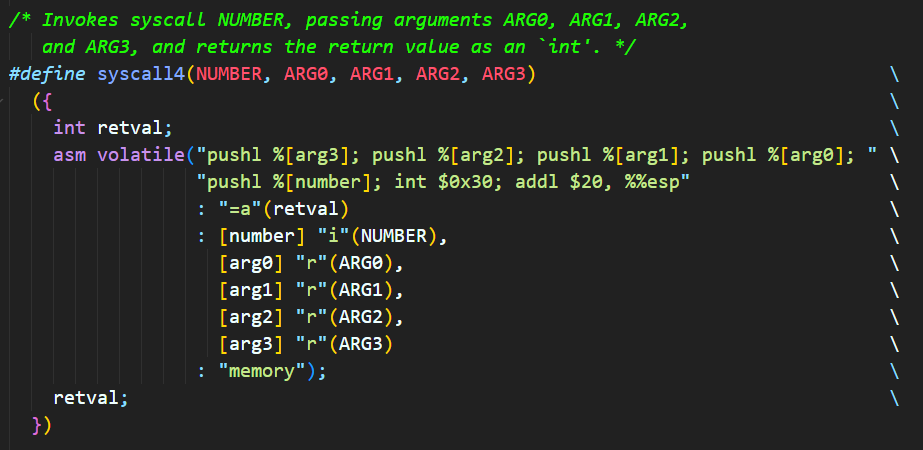
두 시스템 콜을 구현하기 이전에, 이런 시스템 콜을 추가할 것이라는 정보를 OS가 알아야 한다.

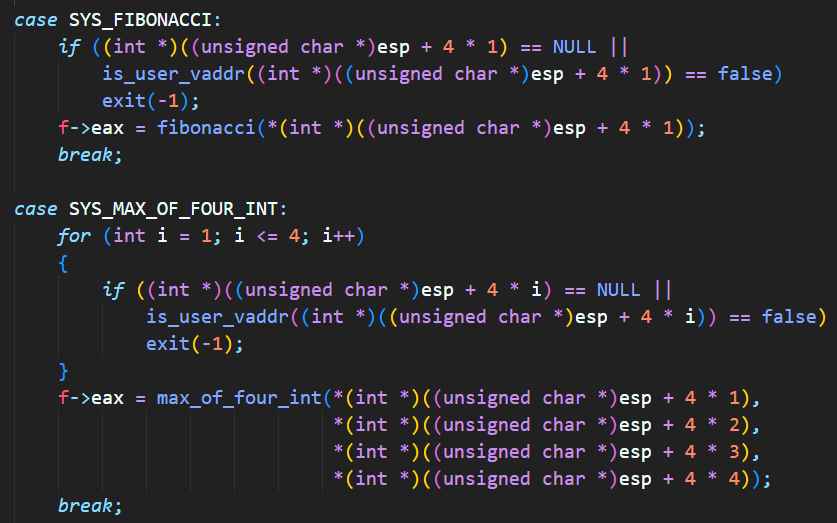
우선 이용의 용이함을 위해 ~/lib/syscall-nr.h의 열거형에 사용할 이름을 추가해준다. 또, ~/lib/user/syscall.h에 구현한 시스템 콜 래퍼 함수의 선언도 추가해준다.



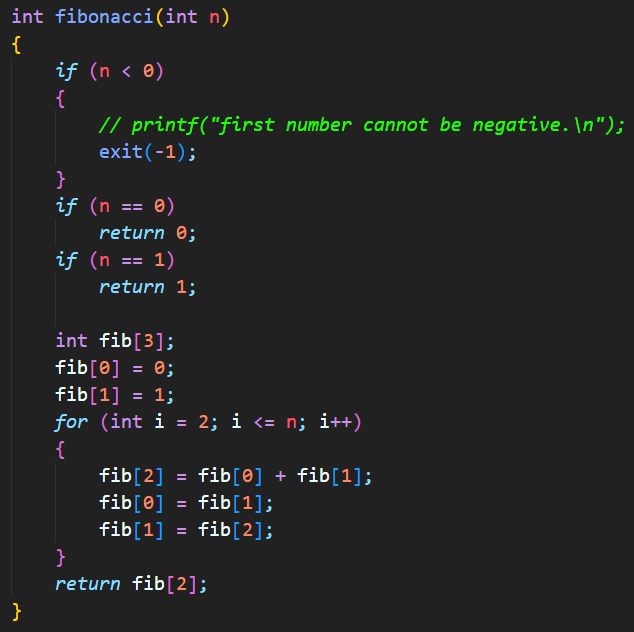
이는 ~/lib/user/syscall.c에 위와 같이 작성되어 있다. 이때, syscall1은 이미 구현되어 있지만 syscall4는 그렇지 않다. 따라서, syscall4 매크로를 따로 구현해주어야 작동시킬 수 있다.



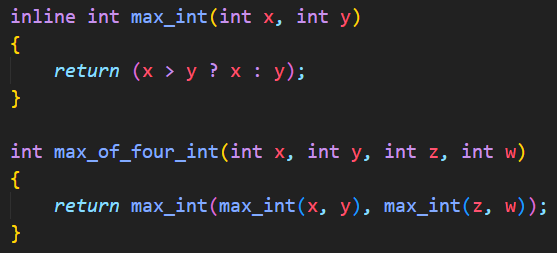
매크로는 이미 구현된 syscall1, 2, 3을 참고하면 손쉽게 구현할 수 있다. arg가 하나 늘어났고, 그에 따라 add1 뒤의 숫자도 4만큼 늘어난다. 나머지는 다른 매크로와 완전히 동일하게 작성하면 된다.



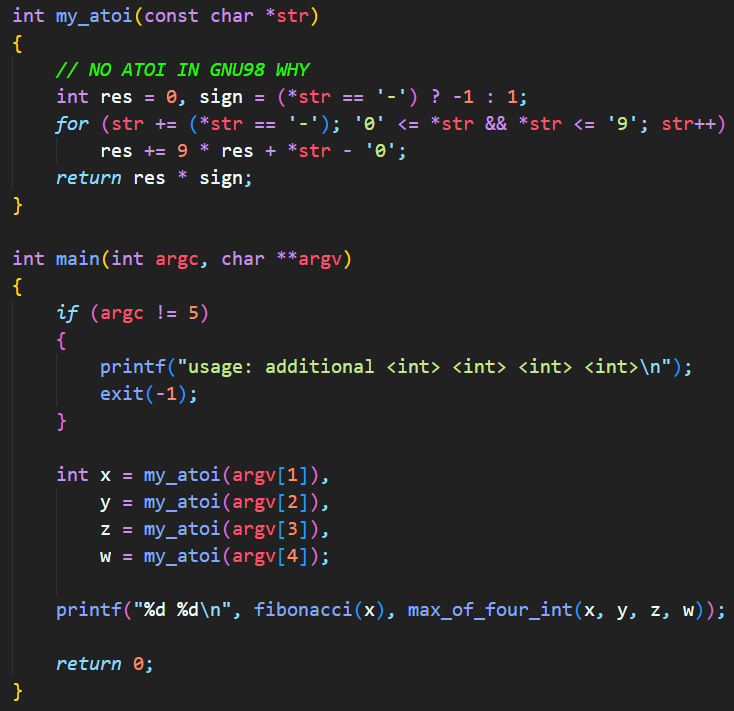
이제 해당 시스템 콜이 호출되면, syscall\_handler가 정해진 동작을 수행한다. 앞서 설명한 user memory check도 동일하게 수행되고, frame의 eax 레지스터에 결과값을 저장하는 것도 똑같다. 이제 실제 동작(fibonacci(n), max\_of\_four\_int(x, y, z, w))만 구현해주면 된다.



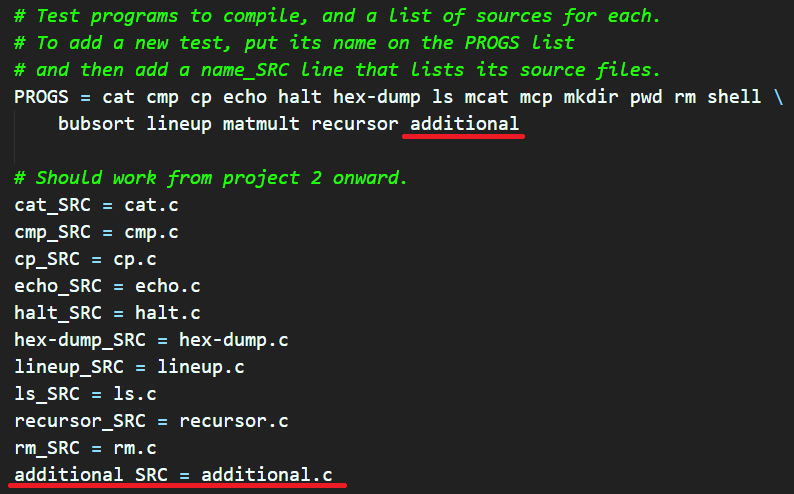
fibonacci의 구현은 다음과 같다. 기본적인 피보나치 항을 구하는 공식인데, 수행 시간을 줄이고 추가적인 메모리를 사용하지 않기 위해 이전 항을 계속하여 toggling하는 iterative한 bottom-up 방식을 사용하였다. 또, 만약 음수가 들어온다면 에러 처리를 하는 부분도 작성하였다.



max\_of\_four\_int의 구현은 다음과 같다. 토너먼트 방식의 단순 비교로 최대값을 간편하게 구할 수 있다. 추가적인 메모리 선언이 필요한 정렬 방식을 사용하지 않기 위해 이런 식으로 구현하였다.

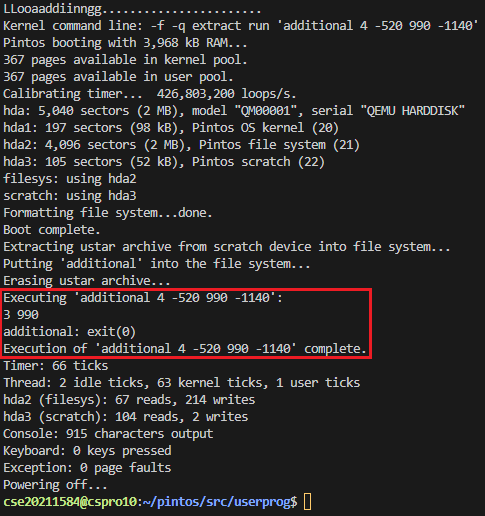
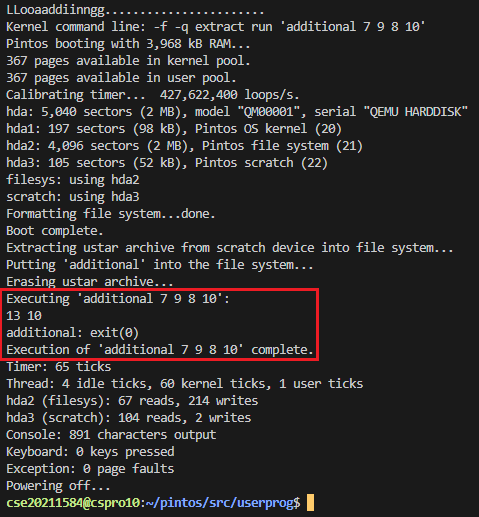


마지막으로 ~/examples/additional.c이다. 특별한 점은 없고, 컴파일 표준에 atoi 함수가 없어서 스스로 작성하였다. 부호 처리 및 10진 정수화를 제대로 수행한다.



make를 통한 일괄 컴파일을 위해 폴더 내 Makefile도 다음과 같이 수정해야 한다.

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

다양한 입력에 대한 올바른 출력을 확인할 수 있다.