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

담당 교수 : 김영재

조 / 조원 : 20171666 이예은

개발 기간 : 10/3 ~ 10/31

1. **개발 목표**

사용자의 기본적인 시스템 명령을 수행하는 환경을 만드는 것이 목표이다. 현재 pintos는 System call, Argument passing등이 되어있지 않아 사용자의 명령을 수행하지 못하므로 이를 제대로 동작할 수 있게 해준다. 이를 위해 argument passing으로 argument를 분석, stack에 할당하고, 명령에 따라 기능을 수행하는 system call을 구현한다. 또한 추가적인 system call도 구현한다.

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

/bin/ls -l foo bar 과 같은 명령이 들어오면, 이를 “/bin/ls”, “-l”, “foo”, “bar”로 나누어 형식에 맞게 stack에 차례로 쌓이게 된다. 그림으로 표현하면 다음과 같다.

 즉 stack에 차례로 쪼개진 arguments들, 이후 word를 맞추기 위한 값과 null을 넣어주고 해당 argument가 쌓인 주소를 다시 넣어준다. 그리고 argument의 개수와 마지막 return address를 스택에 쌓은 형태가 된다.

1. User Memory Access

User 코드가 유효하지 않은 포인터로 접근해 커널에 문제를 일으키거나, 해당 실행중인 다른 process에 영향을 줄 수 있으므로 이를 막아야 한다. 이를 위해 먼저 유효한 포인터인지 userprog/exception.c의 page\_fault() 함수에서 검사하고 이후 다른 파일에서 is\_user\_vaddr(), is\_kernel\_vaddr()을 사용해 잘못된 메모리 영역에 침범했을 시 에러처리를 해준다. 이를 통해 결론적으로 올바른 접근만 정상적으로 실행되게 된다.

1. System Calls

Syscall\_handler()에서 어떤 syscall인지 확인하고 적절한 syscall 함수를 불러 해당 system call의 기능을 수행하게 된다. System call 함수는 각각 halt, exit, exec, wait, read, write, Fibonacci, max\_of\_four\_int이며 이 system call 구현으로 인해 해당 명령어를 pintos에서 사용할 수 있게 한다.

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

Esp가 현재 스택의 가장 윗 부분을 가리키고 있다. 그러므로 이 esp를 움직이며 해당 위치에 원하는 값을 넣어준다. 문자열이 들어오면 그 문자열을 “ “ 단위로 나누고 이를 위 그림처럼 하나씩 넣어준다. Esp의 위치에서 넣을 문자열의 길이만큼 – 하여 문자열의 시작 주소를 esp가 가리키게 한다. 이후 strlcpy를 통해 해당 esp의 주소에 문자열을 복사해 넣어준다. 문자열을 넣는 과정이 끝났다면, word-align을 맞추기 위해 전체 문자열의 길이가 4로 나뉘는지를 확인한다. 4로 나뉘지 않는다면 word를 맞추기 위해 적절한 개수의 0 값을 넣어준다. 이후에는 null 값을 넣는다. 여기까지 한 후에는 이제 esp를 4씩 줄여가며 해당 위치에 저장했던 문자열의 스택 위치 주소를 차례대로 넣는다. 그리고 이 주소들을 저장한 가장 마지막 위치를 다시 스택에 넣어준다. 마지막으로 return address를 스택에 넣으면 유저가 입력한 명령을 저장한 스택이 만들어진다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명

핀토스는 virtual memory를 사용해 메모리를 관리하며, 이 virtual memory는 또한 user memory 영역과 kernel memory 영역으로 나뉜다. Invalid memory access라는 것은 user의 프로그램이 kernel memory 영역에 침범하게 되는 것으로, 이렇게 침범하게 되면, 각 process를 제대로 실행하지 못하거나 kernel code에 변형이 생길 수 있는 문제점이 있다. 이를 막기 위해 system call을 이용하는데, 이 system call을 이용해서 kernel에 접근할 수 있게 하는 것이다. 또한 실행 중 다른 memory 영역에 접근하는 것을 막기 위해 is\_kernel\_vaddr, is\_user\_vaddr등의 내장함수를 사용해 검사해주는 방식을 사용한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

시스템 콜은 user program이 kernel에 직접적으로 접근해 발생할 수 있는 문제들을 막고, os가 프로그램을 수월하게 관리할 수 있게 해준다. 이번 프로젝트에서 개발한 시스템 콜은 다음과 같다.

Halt(): shutdown\_power\_off 함수를 불러 pintos를 종료한다.

Exit(): 현재 user program을 종료한다. Kernel에 status를 반환한다.

Exec(): 새로운 child process를 만든다.

Wait(): child가 끝날 때까지 기다리게 해준다.

Write(): stdout으로 buffer에 있는 정보를 쓴다.

Read(): stdin으로, 정보를 읽어 buffer에 넣는다.

Fibonacci(): 받은 숫자의 피보나치 수를 반환한다.

Max\_of\_four\_int(): 받은 숫자 중 가장 큰 값을 반환한다.

유저 레벨에서 해당 system call 함수가 불리면, 먼저 lib의 syscall.c의 해당 형식에 맞는 define된 syscall을 사용해 stack에 arguments 들을 push 해준다. 이후 interrupt를 건다. Intr\_handler는 다시 syscall\_handler를 부르고, syscall\_handler가 스택에 저장된 해당 system call number를 보고 이에 맞는 system call 함수를 부른다. 이후 이에 맞는 동작을 하고 다시 유저 코드로 돌아가게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/24: 구현해야 할 내용 파악 및 Argument passing 구현.

10/30: system call handler 및 틀 구성. Additional system call 구현.

10/31: system call 구현 및 에러 처리.

11/1: 보고서 작성

* 1. **개발 방법**

1. Argument passing

Process.c의 load() 함수 내부를 수정한다. Load 함수에서 filesys\_open을 하기 전, strtok 함수를 사용해 받은 file\_name을 “ “ 단위로 쪼개 argv[] 배열에 넣는다. 그리고 이를 stack이 setup 된 이후에 스택에 알맞게 정보를 넣는 코드를 load 끝부분에 추가한다.

1. User memory access

접근할 주소가 user address 범위 내인지 is\_kernel\_vaddr함수를 이용해 판단하는 함수를 syscall.c에 추가한다. 그리고 syscall\_handler 함수에서 주소에 접근해 값을 system call에 넘겨줄 때 마다 이 함수를 불러 유효성을 체크한다.

또한 exception.c 코드에서 page\_fault함수에 유효한 페이지인지 체크하는 코드를 추가한다. Pagedir\_get\_page함수를 이용해 unmapped한 memory를 접근하려 하는지 확인하고 그렇다면 에러 처리를 해준다.

1. System call

Syscall.c에 해당 system call에 맞는 함수를 불러오도록 syscall\_handler() 코드를 수정하고 각 system call의 코드를 추가한다.

Sys\_halt(): shutdown\_power\_off()를 호출한다.

Sys\_exit(): termination message를 출력하고 thread의 상태를 업데이트 한 후, thread\_exit 함수를 불러 종료한다.

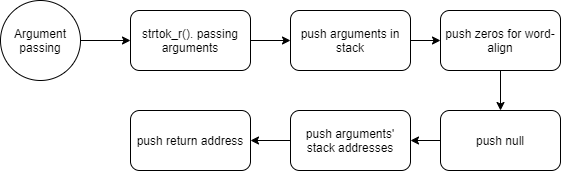
Sys\_exec(): process\_execute()를 호출해 실행되도록 한다. 또한 process\_execute함수에서는 올바르게 실행되기 위해 여기서도 filename을 파싱해 명령어만 thread name으로 들어가게 수정한다. 또한 파싱한 이름의 파일이 없는지 확인하는 코드를 추가해 없다면 에러 처리를 해주어 exec가 알맞게 돌아가게 한다.

Sys\_wait(): process\_wait()를 호출한다. Process\_wait는 먼저 받은 child\_tid가 유효한지 확인하고, 해당 thread에서 이 child\_tid를 가진 child thread를 찾아 해당 child thread가 종료될 때까지 기다린다. 이 때 기다리는 부분은 semaphore로 구현하였으며, process\_exit에 실행이 끝나면 lock을 푸는 코드를 추가해 wait의 lock을 풀어준다. 그리고 다시 작동하는 부모 thread가 실행이 끝난 child thread를 정리하고 상태를 업데이트 한 후에 wait를 종료한다. 이를 위해 child의 상태 등을 저장하는 변수를 thread 구조체에 추가한다.

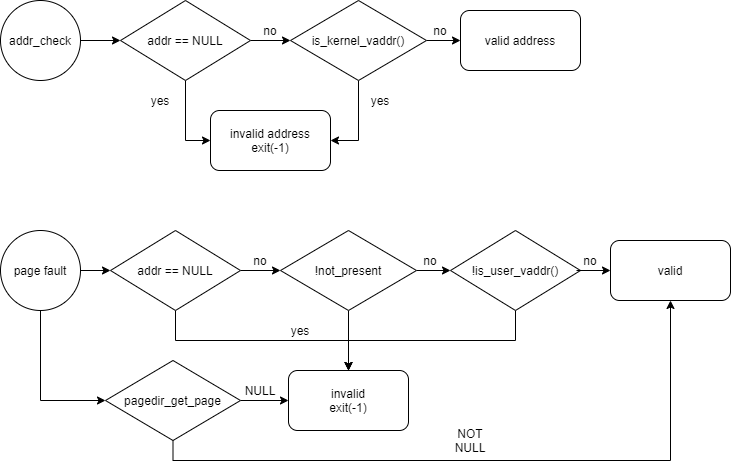
Sys\_read(): fd가 0일 때 읽기 이므로 이 때 input\_getc 함수를 사용해 STDIN을 읽는다. 잘못된 read면 -1, 올바르게 읽었다면 읽은 size를 반환한다.

Sys\_write() fd가 1일 때 쓰기 이므로 이 때 putbuf 함수를 사용해 STDOUT으로 쓴다. 잘못된 쓰기면 -1, 올바르게 write 했다면 write한 size를 반환한다.

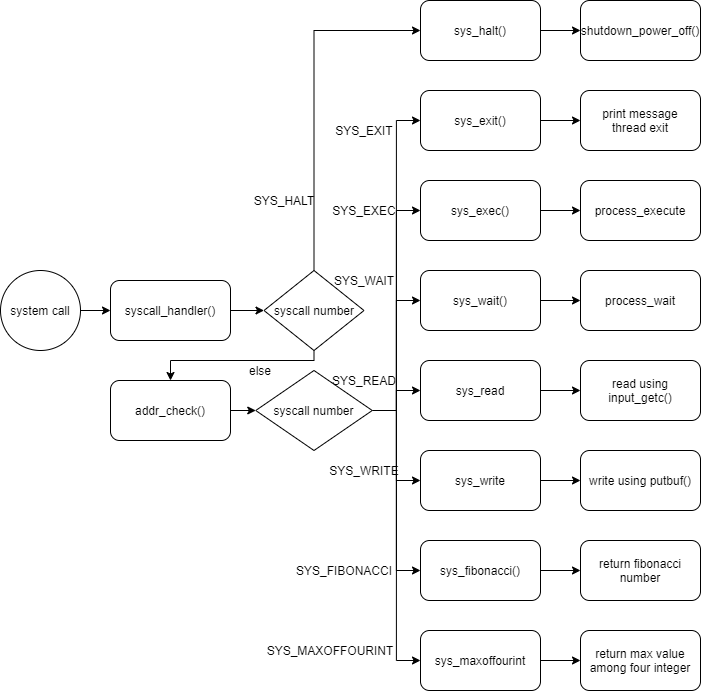
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



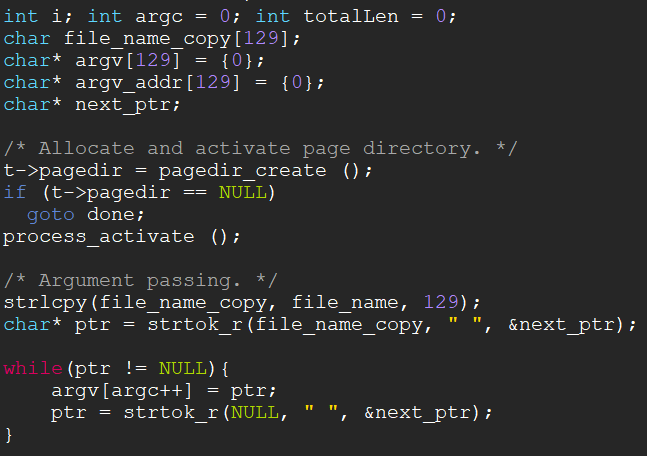
1. System Calls



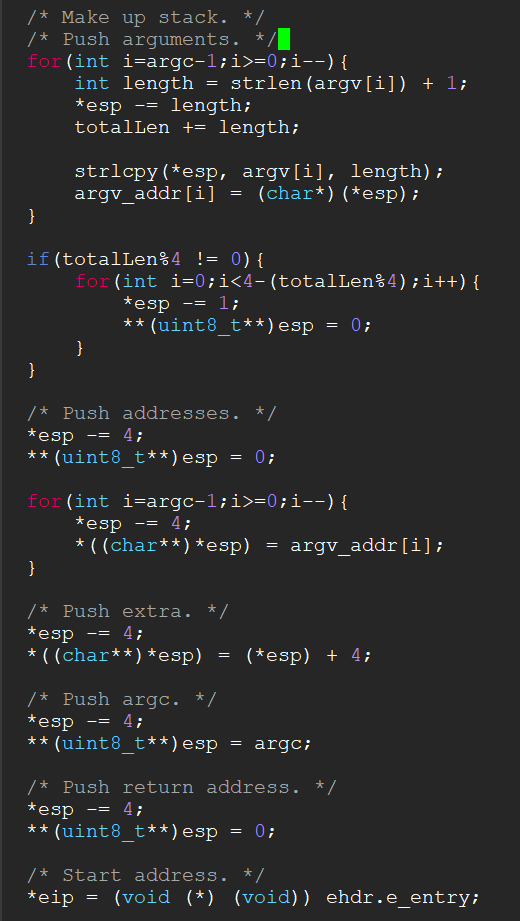
* 1. **제작 내용**

1. Argument Passing

Load()



Load 함수에 다음과 같은 코드를 추가한다. Strlcpy로 우선 file\_name을 복사해 file\_name\_copy에 넣어주고, 이 file\_name\_copy를 strtok\_r로 쪼갠다. 쪼갠 문자열을 argv[]안에 차례대로 넣어주며, argc를 넣을 때 마다 +1을 하며 전체 문자열의 개수를 저장한다. 그리고 아래 stack을 초기화 한 후에는 다음과 같은 코드를 추가한다.

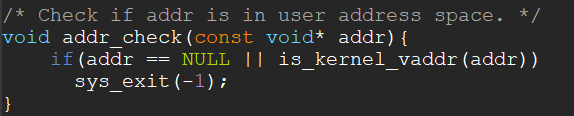


Esp가 현재 스택을 가리키는 포인터기 때문에 이를 조정해주며 스택에 넣는다. Argv에 저장한 문자열에 따라 문자열의 길이만큼 esp를 – 하여 위치를 조정하고 해당 위치에 strlcpy로 argv 문자열을 넣는다. 또한 이후에 저장한 stack의 주소를 다시 넣어야 하기 때문에 argv\_addr이라는 배열에 그 때의 주소 값을 저장한다. 또한 word-align을 맞추기 위해 totalLen이라는 변수에 모든 argv 문자열의 길이를 더하면서 저장한다. Argv를 다 넣었다면, 이 totalLen을 사용해 길이가 4로 떨어지지 않는다면 그만큼 0을 넣어 word를 맞춰준다. 그리고 esp에 4를 빼고 null 값을 넣어준다. 이제 다시 esp를 4씩 – 하며 위치를 조정하고 해당 위치에 argv\_addr에 저장했던 주소 값들을 넣어준다. 그리고 다시 esp-4의 위치에 주소값 저장한 주소를 다시 넣어주고, argc를 넣어준다. 마지막으로 return address를 넣어 stack을 완성한다.

Esp에 값을 넣어주는 과정에서 \*\*(char\*\*)esp와 같이 포인터 연산을 하자 주소 값이 끝부분만 들어가는 문제가 발생했지만, \*((char\*\*)\*esp)로 형변환을 바꾸어 전체 주소가 제대로 들어가게 수정하였다.

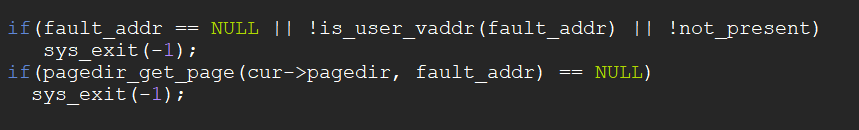
1. User Memory Access

Addr\_check()



Syscall.c에서 다음과 같은 함수를 만들어 올바른 주소 접근인지 확인한다. 인자로 받은 addr 주소 값이 NULL이거나, 내장함수인 is\_kernel\_vaddr() 함수를 사용해 kernel address 공간에 있는 주소 값이라는 것이 확인되면 sys\_exit(-1)을 불러 thread가 종료되게 해준다.

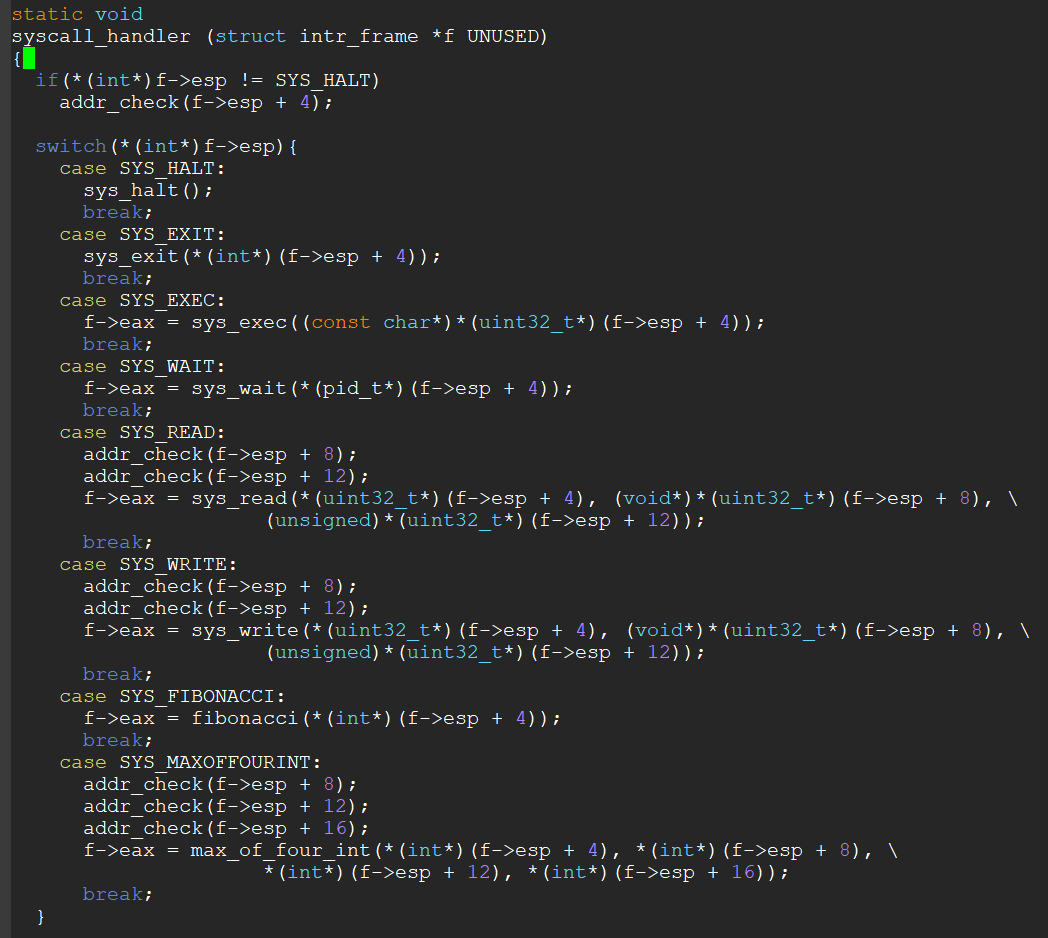
Exception.c / page\_fault()



Exception.c의 page\_fault() 함수에서 User address 공간 포인터가 아니거나 unmapped page인지 확인하고 에러 처리를 해준다. Fault\_addr이 NULL이거나, is\_user\_vaddr() 함수를 사용해 user address 공간에 있는 주소 값이 아니거나, 존재하지 않는 페이지라면 sys\_exit(-1)을 부르고, 내장 함수인 pagedir\_get\_page() 불러 반환값이 NULL이라면 unmapped page이므로 이를 확인해 sys\_exit(-1)을 불러준다.

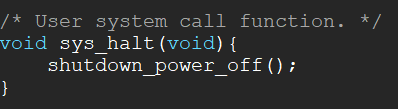
1. System Calls

Syscall\_handler()



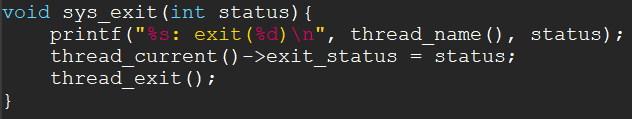
f->esp가 system call number를 저장하고 있으므로 이를 이용해 switch 문을 구성한다. Number에 따라 각각에 맞는 함수들을 불러준다. 또한 만약 stack의 다른 arguments를 접근해야 할 경우 주소가 유효한지 확인하기 위해 addr\_check()를 사용해 검사를 한 후 syscall 함수를 불러주도록 한다.

Sys\_halt()



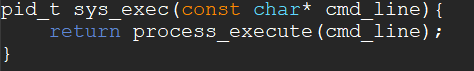
Shutdown\_power\_off 내장함수를 불러 halt system call을 구현한다.

Sys\_exit()



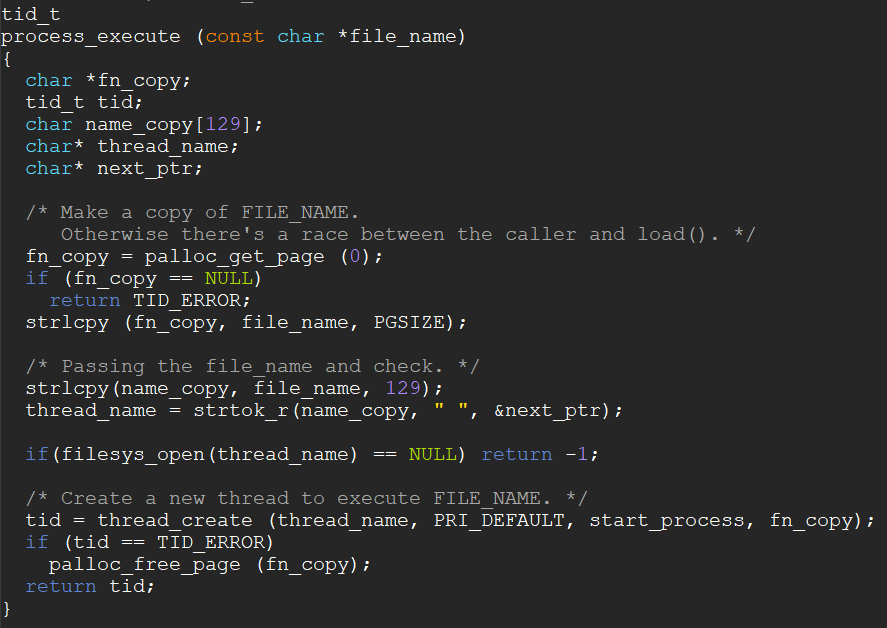
먼저 실행중인 thread의 이름과 상태를 이용해 termination message를 출력한다. 이후 현재 thread 구조체의 exit\_status를 status로 하여 업데이트 해주고, thread\_exit() 함수를 불러 thread를 종료한다.

Sys\_exec()



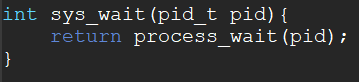
Process\_execute() 함수를 불러 exec system call을 구현한다.

Process\_execute()



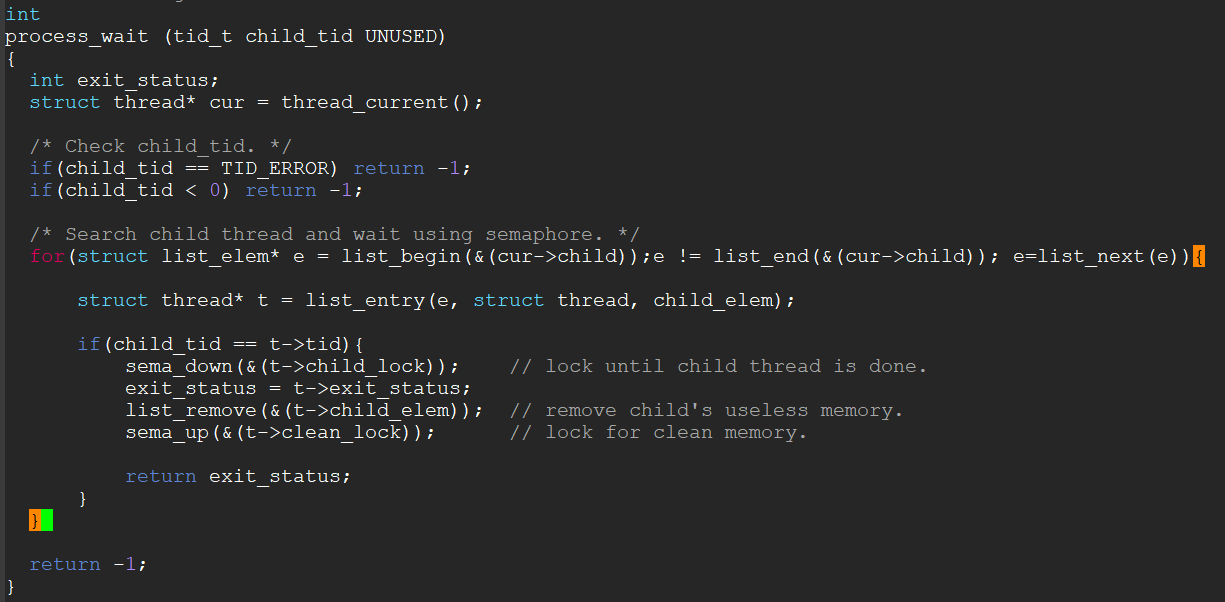
Process\_Execute는 다음과 같으며, 입력 받은 이름으로 thread를 만들어 실행한다. 이 때, 파라미터로 받는 file\_name은 아직 파싱하기 전의 이름이 넘어오므로, load 함수에서 구현한 것 처럼 맨 앞의 argument만 파싱하여 이를 thread name으로 넘겨주는 부분을 추가한다. 또한 해당 thread\_name의 파일이 없을 수 있으므로 filesys\_open 내장함수를 불러 NULL을 반환하는지 체크 후 NULL이라면 -1을 반환해 에러를 확인할 수 있게 한다.

Sys\_wait()



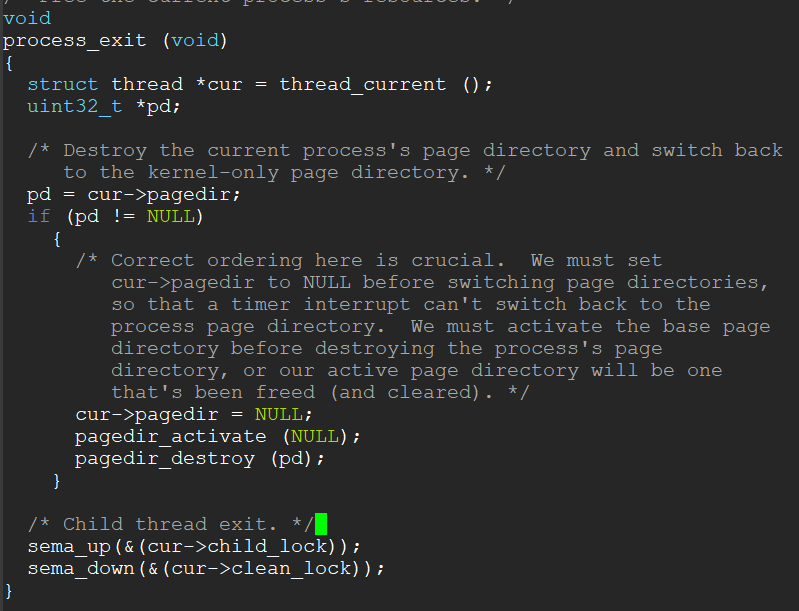
Process\_wait()를 불러 wait system call을 구현한다.

Process\_wait()



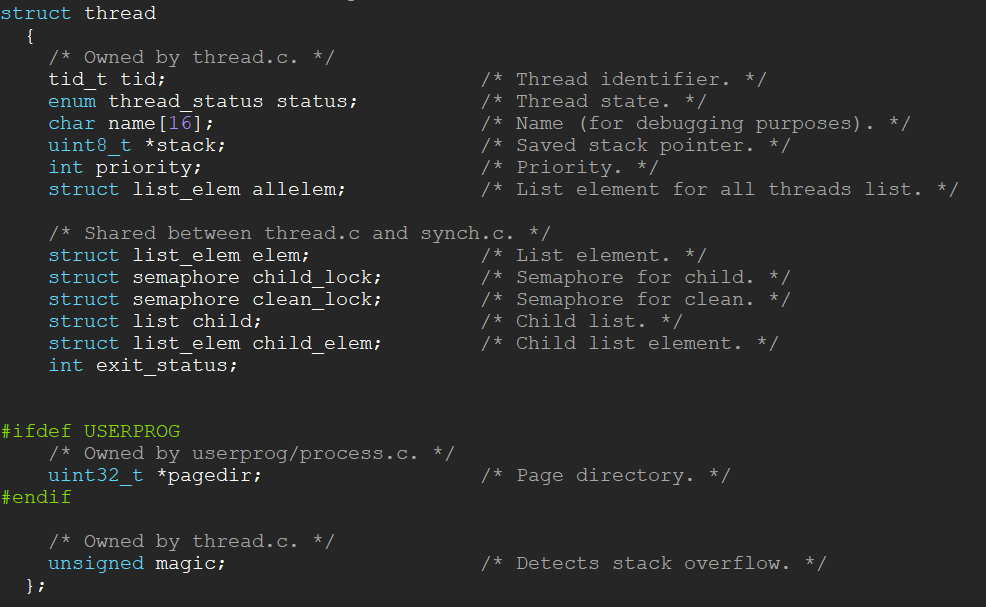
Process\_wait는 부모가 자식 thread가 끝날 때까지 기다리는 것을 구현한 함수이다. 먼저 넘겨 받은 child\_tid가 유효한지 확인한다. 이후 thread 구조체 안의 child들이 저장된 list를 돌면서 child\_tid를 가지는 child thread를 찾는다. 찾았다면, sema\_down을 통해 lock을 걸고, 해당 child thread가 끝나 sema\_up을 해서 lock을 풀어줄 때까지 기다린다. Child thread가 종료되어 lock이 풀렸다면, list\_remove를 이용해 child thread가 차지하고 있던 메모리를 정리해준다. Clean\_lock은 메모리 정리를 위한 semaphore로, parent가 child의 메모리를 정리해주기 전까지 child를 완전히 끝내는 것을 막는 lock이다. Process\_wait에서는 list\_remove로 메모리 정리가 끝나면, sema\_up으로 이 lock을 풀어 child가 완전히 종료되게 해준다. 그리고 해당 status를 반환해 wait를 끝낸다. 만약 child\_tid를 갖는 child thread를 찾지 못했거나 하는 등의 에러가 발생하면 -1을 반환한다.

Process\_exit()



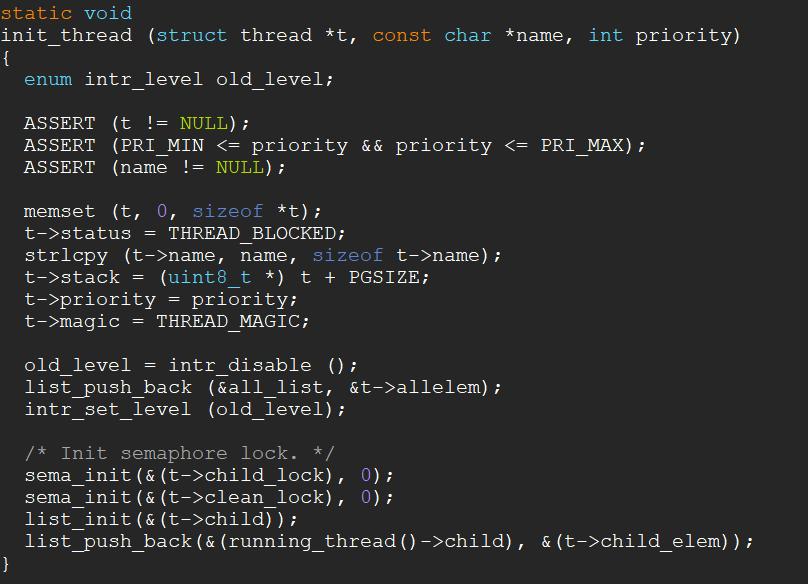
Thread가 끝날 때, 부모 thread가 더 이상 기다리지 않도록 sema\_up을 코드에 추가한다. 또한 parent가 내 메모리를 다 정리해줄 때까지 기다려야 하므로 sema\_down을 통해 lock을 걸고, 메모리 정리가 끝나면 해당 함수가 끝나게 구현한다.

Struct thread



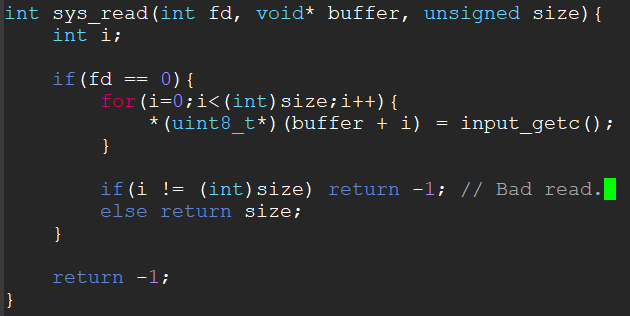
위와 같은 wait를 구현하기 위해 struct thread에 다음과 같은 변수를 추가한다. Child\_lock, clean\_lock은 semaphore 를 사용하기 위한 변수들이고, child는 child thread들을 저장하기 위한 list, child\_elem은 각 child thread의 list 원소들이며, exit\_status는 종료 status를 저장한다.

Init\_thread()



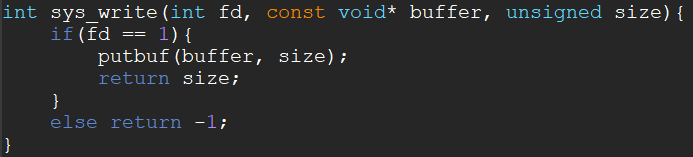
해당 thread 구조체를 다른 함수들에서 문제없이 사용하기 위해 init\_thread() 함수에 초기화 부분을 추가해준다.

Sys\_read()



Fd가 0일 때 STDIN이고, 이번 프로젝트에서는 이것만 판단하므로, fd가 0일 때, size만큼 돌며 buffer에 input\_getc() 함수를 사용해 읽은 값을 넣어준다. 다 넣은 후 만약 넣은 size가 받은 size가 아니라면 제대로 읽은 것이 아니므로, -1을 반환하고, 아니라면 size를 반환한다.

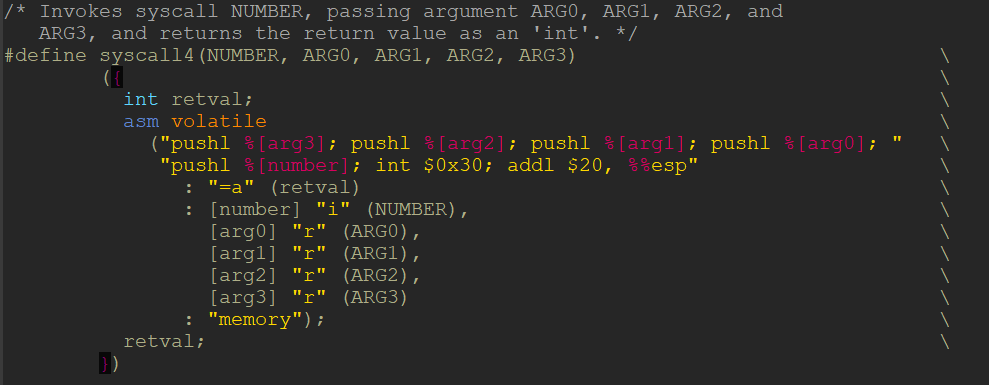
Sys\_write()



Fd가 1일 때, STDOUT으로, 1인 경우에 putbuf() 함수를 사용해 buffer에 있는 값들을 stdout으로 출력해준다. 출력한 size를 반환하고 아니라면 -1을 반환해준다.

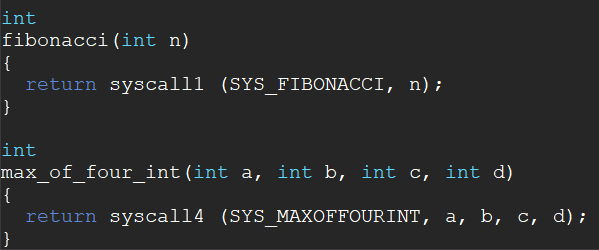
1. Additional System calls

Syscall4



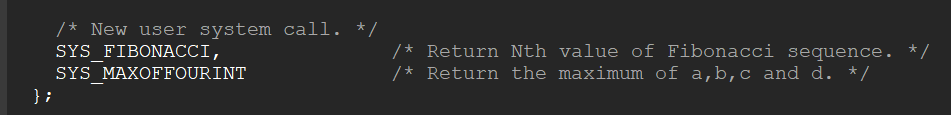
Max\_of\_four\_int는 4가지 parameter를 받는 함수이고 이를 처리해주기 위해 syscall4를 define 해준다.

Lib/syscall.c



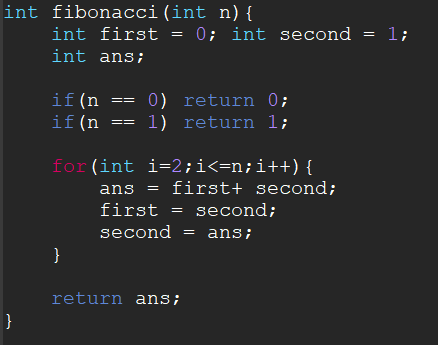
Fibonacci와 max\_of\_four\_int가 각각 맞는 syscall 형식을 부르도록 코드를 추가한다.

syscall-nr.h



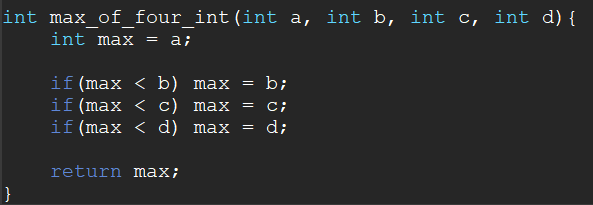
Fibonacci와 max\_of\_four\_int에 대한 enum system call 값을 추가한다.

Fibonacci()



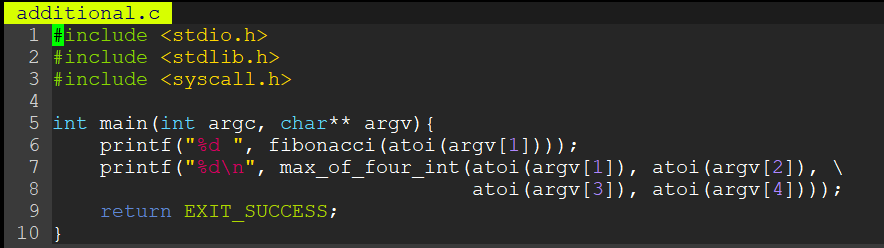
피보나치 수를 구하는 함수를 구현한다. 0,1일 때는 그냥 0, 1을 반환하고 아니라면 for문을 돌며 이전 n-1, n-2 값의 합을 구한다. 구한 피보나치 수를 반환한다.

Max\_of\_four\_int()



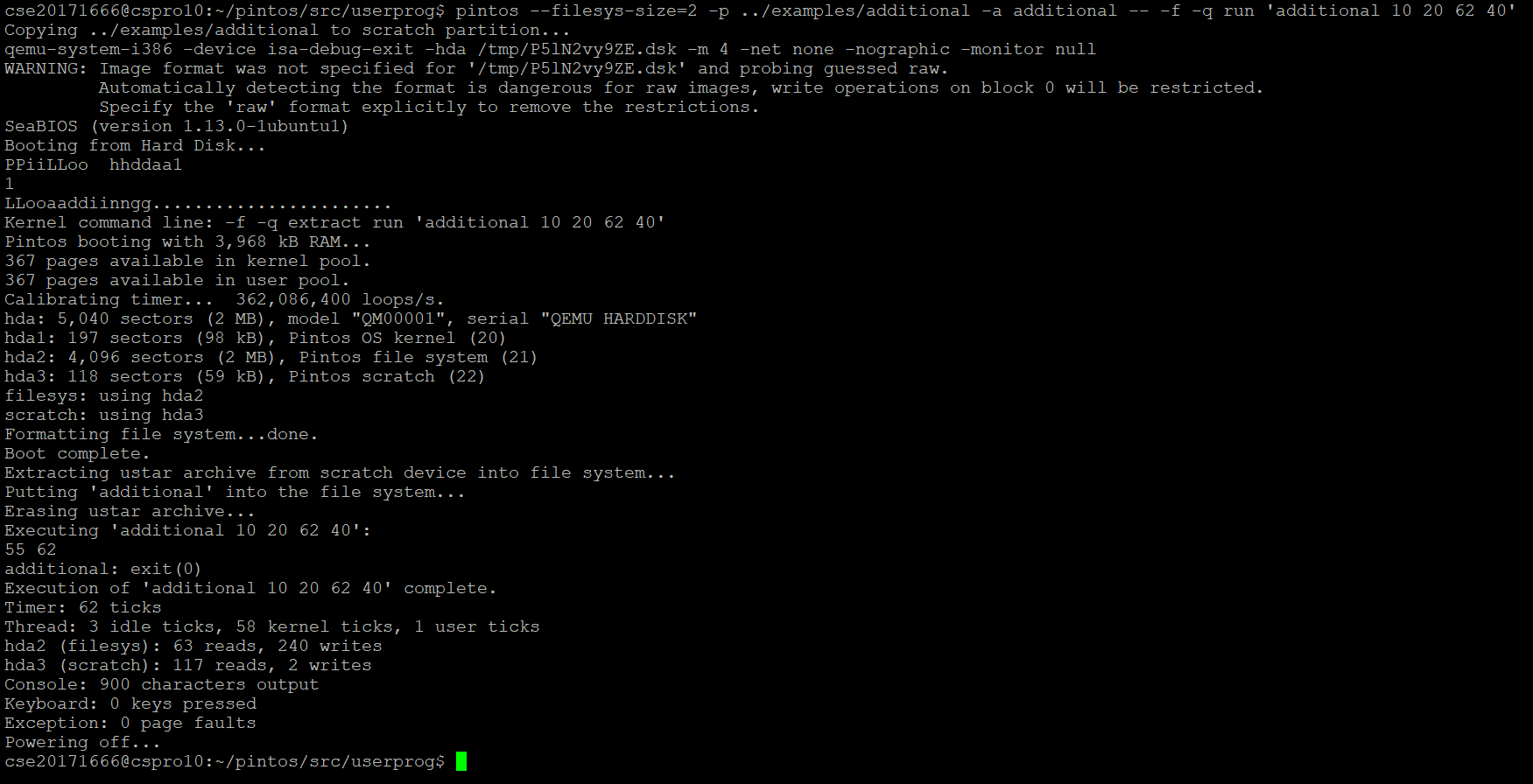
A,b,c,d 수 중 가장 큰 값을 반환한다.

Additional()



Test를 위해 다음과 같은 함수를 examples에 추가해 pintos 내에서 테스트할 수 있게 한다.

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

****