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

담당 교수 : 박성용

조 / 조원 : - / 20161559 강민석

개발 기간 : 2020.10.10 ~ 2020.10.24

1. **개발 목표**

pintos에서 기본적인 운영체제 동작을 할 수 있도록 각종 기능들을 구현한다. 함수를 호출 할 때 적절한 방식으로 argument를 넘길 수 있도록 argument passing을 구현한 후, user memory 접근을 보호하는 기본적인 기능을 구현한다. 이후에 write(stdout), read(stdin), exit, halt, exec, wait등의 system call을 구현한다. 추가 구현으로 사용자로부터 input을 받아 해당 input에 대한 fibonacci 수, 최댓값을 구하는 system call을 구현한다.

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

Argument Passing을 구현함으로써 함수 호출을 할 때 해당 함수에 대한 argument들을 적절하게 넘기고, 이로써 함수가 호출됐을 때 정상적으로 작동할 수 있다.

1. User Memory Access

User program은 지정된 memory 범위 안에서 작동해야 한다. User Memory Access 기능을 구현함으로써 User program이 적절치 못한 영역을 가리키거나 범위를 벗어난 경우에 대한 접근을 방지해주고 에러처리를 해줄 수 있다.

1. System Calls

요구하는 System call을 구현함으로써 process가 실행되었을 때 표준 입출력, 다른 process가 동시에 실행되거나 child process가 생성되었을 때의 synchronization을 맞춰줄 수 있다.

* 1. **개발 내용**
* Argument Passing

스택에 argument를 쌓을 때는 다음과 같은 순서로 쌓는다.

(1) 각 argument의 ‘내용물’들을 하나씩 뒤에서부터 차례로 집어넣는다.

(2) (1)의 결과로 쌓았을 때, word-alignment를 위해 현재 주소값이 4의 배수가 아니라면 4의 배수가 되게끔 추가로 의미없는 데이터를 쌓아준다.

(3) argv[argc], 즉 null값을 stack에 하나 쌓는다.

(4) 차례로 각 argument의 ‘stack에서의 주소값’들을 하나씩 뒤에서부터 차례로 집어넣는다.

(5) 마지막 argv[0](첫 번째 argument)의 stack에서의 주소값을 쌓을 때의 주소값을 집어넣는다.

(6) argc(argument의 개수)를 집어넣는다.

(7) fake return address인 0을 집어넣는다.

* User Memory Access

kernel은 user program이 제공하는 pointer를 통해 memory에 접근하게 되는데 해당 pointer가 invalid한 경우는 다음과 같다.

(1) Null Pointer

(2) Pointer unmapped to virtual memory

(3) Pointer to kernel virtual address

위와 같은 pointer들은 적절히 reject되어야 하며 그러기 위해 다음과 같은 방법을 이용한다.

(1) pointer의 validity를 검사하고 dereference한다. 이 때 userprog/pagedir.c와 threads/vaddr.h에 있는 함수들을 참조한다.

(2) user pointer가 PHYS\_BASE밑(invalid한 위치)에 있는지 확인한 후 dereference한다. 이 때 page fault가 발생하므로 이에 따라 userprog/exception.c에 있는 page\_fault()함수를 적절히 변형하여 에러처리를 해준다.

해당 프로젝트를 수행하면서 위의 두 가지 방법 중 (1)을 주 방법으로 사용하되 (2)의 방법도 이용하였다.

* System Calls

운영체제에서 I/O등의 operation은 kernel mode에서만 수행 가능하다. User mode에서 해당 작업을 수행하기 위해 system call을 호출하여 kernel mode로 진입한 후, 다시 user mode로 돌아오는 방식을 이용할 수 있다. 이번 프로젝트에서 구현해야 할 system call은 다음과 같다.

halt : pintos를 shutdown\_power\_off() 함수를 호출함으로써 종료시킨다. 웬만해서는 이용 안 하는 것이 좋다.

exit : 현재 user program을 종료하고, 종료 status를 kernel에게 전달한다. 혹은 process의 parent가 wait을 호출했다면, status가 parent에게 전달된다.

read : stdin(keyboard)을 이용하여 buffer 입력을 받는다. 입력을 받은 개수를 return한다.

write : stdout(console)을 이용하여 buffer를 출력한다. 출력을 한 개수를 return 한다.

exec : cmd\_line에 있는 executuable한 파일을 실행한 후, 해당 파일을 실행함으로써 생성된 process id를 return한다. 만약 valid하지 않거나 존재하지 않는 파일 등의 원인으로 정상적으로 execute하지 못하면 -1을 return 한다.

wait : parent process가 child process의 종료를 기다리게끔 하기 위한 기능을 수행한다. 해당 child가 exit할 때 return 한 status를 return한다. 만약 비정상적으로 child가 종료되었을 시, -1을 return 한다.

fibonacci : n 번째 Fibonacci 수를 콘솔에 출력해준다.

max\_of\_four\_int : 4가지 수 중 최대값에 해당되는 수를 콘솔에 출력해준다.

모든 system call은 다음과 같은 과정을 거친다.

(1) user program이 system call 호출

(2) 해당 syscall에 대응되는 만큼 argument를 stack에 쌓고 int 0x30을 통해 system call interrupt 호출 (kernel mode 진입)

(3) interrupt vector table 중 0x30에 대응되는 syscall\_handler()호출

(4) stack에 쌓아두었던 argument를 통해 작성한 system call 호출

(5) system call이 종료되면서 다시 user mode 진입

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

Argument Passing : (10.10 ~ 10.11) 제일 초반 부분이지만 코드를 읽고 이해하는데 시간을 많이 소비하였다. 코드를 작성하고 hex\_dump()를 통해 적절히 argument가 stack에 쌓였는지 확인하였다.

User Memory Access : (10.12 ~ 10.23) invalid한 pointer는 일절 처리해주어야 하므로 모든 구현을 끝마친 뒤에도 보수작업을 하였다. 10.12 ~ 10.16까지는 userprog/pagedir.c와 threads/vaddr.h에 있는 함수들로 해당 pointer가 valid한지 판단하는 기능을 구현하였고, 10.23 test case 중 exec-bad-ptr.c를 처리하기 위해 exception.c에 있는 page\_fault() 함수 부분에 코드를 추가하였다.

System Calls(read, write, halt, exit) : 우선 system call 구현 부분을 진행하면서 구현해야 할 6가지의 system call 중 앞에 기술한 4가지가 난이도가 비교적 쉽고 나머지 두 개인 exec, wait가 어렵다고 판단하였다. read, write의 경우 stdin, stdout에 대한 부분만 구현하면 되고, halt는 shutdown\_power\_off()호출, exit는 thread\_exit()함수를 호출하면 되므로 10.17 ~ 10.18에 걸쳐 구현을 완료하였다.

System calls(exec, wait) : exec, wait system call의 경우 process간, parent-child간 적절한 synchronization이 이뤄져야 하므로 보다 유의하여서 구현해야 했다. semaphore에 대한 내용을 학습하고 적용하는데 보다 오랜 시간이 걸렸으며 10.19 ~ 10.23에 걸쳐 구현을 완료하였다.

Additional System calls : fibonacci, max\_of\_four\_int system call을 구현하였다. 이전 system call을 구현하였다면 똑같은 방식으로 구현하면 되므로 하루 안에 구현을 완료할 수 있었다. (10.24)

* 1. **개발 방법**

- Argument Passing

우선 Argument passing을 하기 위해 argument를 parsing하는 과정이 필요하다. 즉 parsing + passing(stack 쌓기) 작업을 구현하여야 하는데, 이는 userprog/process.c에 있는 load 함수에 구현하였다. load함수의 parameter인 file\_name에는 아직 parsing되지 않은 결과가 담겨있다. (ex) “echo x”와 같은 형태) 이를 위해 우선 argv[0] = “echo”, argv[1] = “x”와 같은 형태로 parsing을 구현한다. parsing에는 strtok\_r함수를 이용하였으며, argument의 개수를 저장하는 argc 변수를 선언한다. 이후에 parsing 구현을 완료한 뒤, stack에 쌓기 위해 동일한 load 함수 내부에서 381번째 줄부터 argument를 쌓는 기능을 구현한다.

- User Memory Access

해당 기능은 system call을 구현하기 이전에 해당 포인터가 적합한지 판단하는 기능을 수행한다. 해당 기능은 모두 userprog/syscall.c 파일에 구현하였다. check\_address()라는 함수를 호출함으로써 해당 포인터와 syscall에 필요한 argument들이 모두 valid한 포인터인지를 심사하는 기능을 한다. 만약 valid 하다면 이후에 구현한 system call을 호출하고, 그렇지 못한 경우 exit(-1)함수를 호출하여 비정상적인 종료를 한다. valid한지 check하는 기능으로 threads/vaddr.h 파일에 있는 is\_user\_vaddr()함수, userprog/pagedir.c에 있는 pagedir\_get\_page 함수를 호출하고, 추가적으로 NULL pointer인지 check하는 기능을 한다.

- System call

모든 system call은 호출 이전에 앞서 기술한 check\_address()함수를 호출하여 해당 pointer가 valid한지 check한다. 이후에 get\_arguments라는 함수를 호출하여 stack으로부터 argument를 읽어온다. 또한 각 system call을 호출하였을 때, 반환 값이 있다면 eax 레지스터에 저장해 준다.

halt : userprog/syscall.c에 halt 함수를 구현한다. 함수 내부에서는 shutdown\_power\_off()를 단순 호출한다.

exit : userprog/syscall.c에 exit 함수를 구현한다. 명세서에서 요구하는 종료되는 thread\_name, 종료 status를 출력하기 위해 해당 정보를 출력하는 printf문을 추가하고 thread\_exit()함수를 호출함으로써 종료한다.

read : userprog/syscall.c에 read 함수를 구현한다. input\_getc()함수를 통하여 한 글자씩 콘솔로부터 읽어 들인 후, 읽어들인 개수만큼 반환한다.

write : userprog/syscall.c에 write 함수를 구현한다. putbuff함수를 호출하여 size 만큼 console에 출력한 뒤, 출력한 개수만큼 반환한다.

exec : userprog/syscall.c에 exec 함수를 구현한다. exec 함수는 다른 system call과는 다르게 synchronization 작업을 수행 해야하기 때문에 보다 조심해야 한다. 프로젝트를 수행하면서 synchronization을 구현하기 위해 semaphore를 이용한다. 0으로 초기화된 semaphore에서 exec함수를 호출할 때, sema\_down을 이용하여 parent가 대기하다가 child 쪽에서 load과정까지 수행되었을 때 sema\_up을 하게끔 구현한다.

wait : userprog/syscall.c에 wait 함수를 구현한다. wait 함수 역시 exec와 마찬가지로 synchronization 작업이 필요하다. wait 함수를 구현하면서 두 개의 semaphore를 이용하였다. syscall.c의 wait함수는 단순히 userprog/process.c에 정의되어 있는 int process\_wait()함수를 호출한다. 즉, 세부적인 구현은 userprog/process.c에 구현한다. 이에 앞서 thread에 부모 자식간의 관계를 기술하기 위해 thread 구조체에 다음과 같은 부분을 추가한다.

- struct thread\* parent : thread의 부모 thread를 가리킨다.

- struct list child : thread의 child를 list로 관리한다.

- struct list\_elem child\_elem : child\_elem을 저장하는 list\_elem이다.

- struct semaphore exit\_lock : wait에 사용되는 semaphore1이다.

- struct semaphore load\_lock : exec에 사용되는 semaphore다.

- struct semaphore memory\_lock : wait에 사용되는 semaphore2이다.

- int exit\_status : 프로세스의 종료 status를 기술한다.

- bool load\_succeded : 프로세스가 load에 성공했는지를 기술하는 Boolean 변수

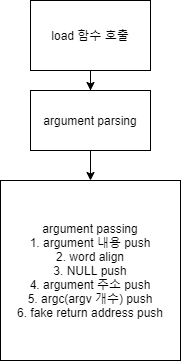
또한 tid를 통해 자신의 자식 thread를 찾아주는 함수 get\_child\_thread를 선언한다.

process\_wait은 다음과 같이 작동하게끔 한다. 우선 exit\_lock과 memory\_lock은 둘다 0으로 초기화 되어있다. 이 때 get\_child\_thread를 호출한 뒤, sema\_down(exit\_lock)을 통해 child process가 종료되기를 기다린다. 프로세스가 종료된 후, child 정보를 유지하고 있던 list에서 child를 제거해야하는데, 이 때 만약 child가 memory에서 없어진 상황이라면 비정상적으로 프로그램이 작동할 수 있으므로, child process는 process\_exit함수에서 sema\_down(memory\_lock)을 통해 exit 되기 전에 list에서 child가 제거 되기를 기다린다. list에서 child가 삭제된 후, sema\_up(memory\_lock)연산을 수행하여 정상적으로 process가 list에서 삭제됨을 보장할 수 있다.

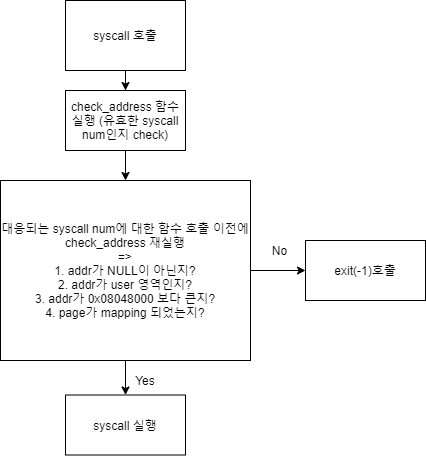
- fibonacci : userprog/syscall.c에 Fibonacci 함수를 구현한다. 또한 함수에 대한 원형을 lib/user/syscall.c에 선언해준다. Fibonacci 함수는 반복문을 통해 n 번째 피보나치 수를 계산한 후, 반환하도록 한다.

- max\_of\_four\_int : userprog/syscall.c에 max\_of\_foru\_int 함수를 구현한다. fibonacci와는 다르게 해당 system call은 parameter가 4개이다. 즉 이전 system call들은 최대 3개 까지만 argument를 받던 것에 반해 4개이므로 추가적으로 assembly macro를 lib/user/syscall.c에 선언해준뒤, 함수의 원형 또한 선언해준다. userprog/syscall.c에서는 조건문을 연달아 3번 선언하여 4개의 정수중 가장 큰 정수를 반환 값으로 넘겨준다.

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

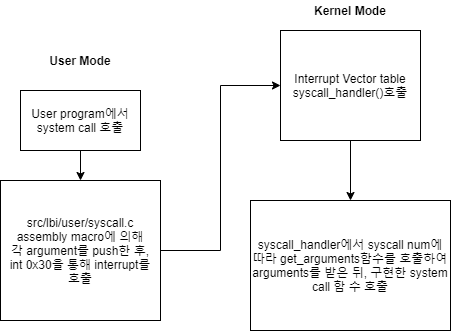


1. User Memory Access

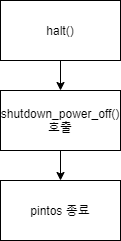


1. System Calls

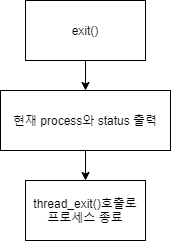
- 전체 흐름



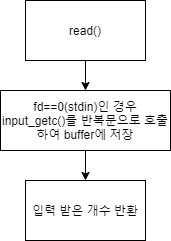
- halt



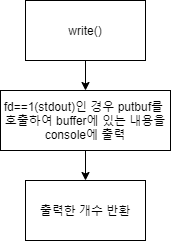
- exit



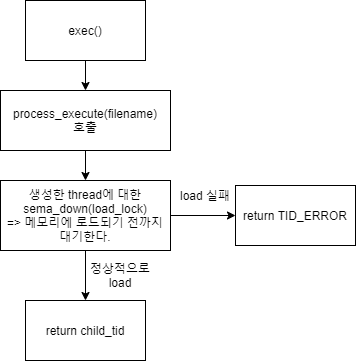
- read



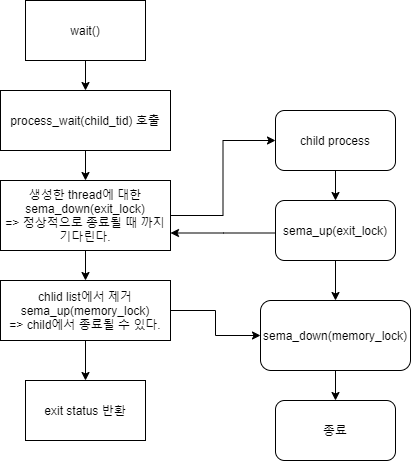
- write



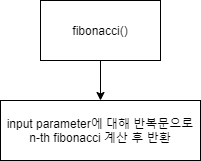
- exec



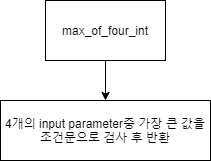
- wait



- Fibonacci

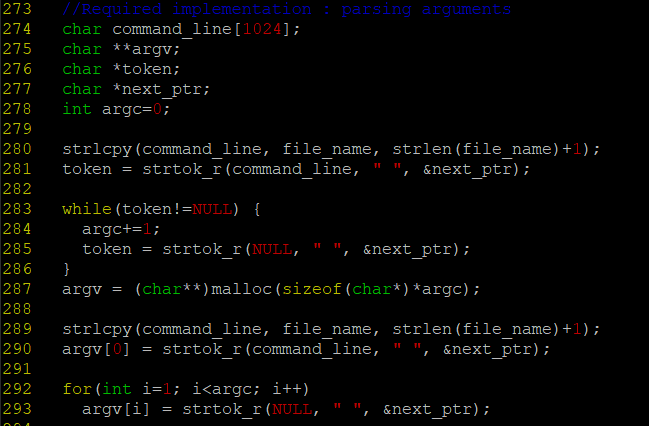


- max\_of\_four\_int



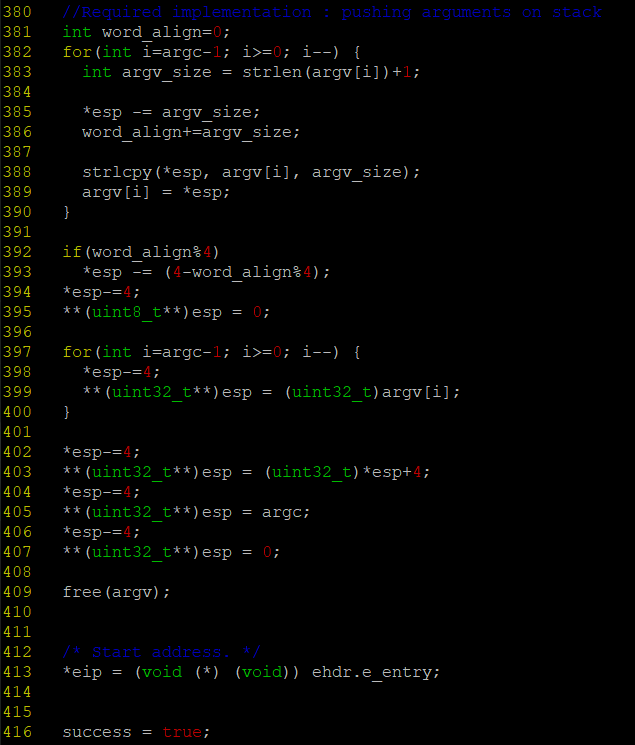
* 1. **제작 내용**

1. Argument Passing



(userprog/process.c load function 내부)

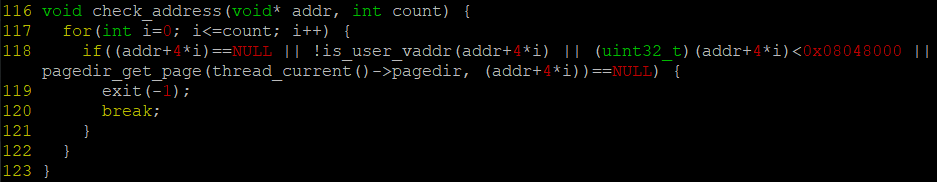
위 코드는 argument passing을 하기 이전에 argument parsing을 하는 과정을 기술한다. userprog/process.c 파일의 load함수 내부의 273번 째 줄이며, 아직 parsing안 된 parameter file\_name을 command\_line 변수에 복사한 뒤, strtok\_r 함수를 이용하여 공백을 기준으로 parsing을 진행한다. 이로써 293까지 코드를 실행하였을 때, argc에는 file\_name에 있던 argument의 개수가, argv 배열에는 처음부터 차례로 argv내용이 담긴다.



(userprog/process.c load function 내부)

위 코드는 argument parsing을 마친 후, argument를 하나씩 stack에 집어넣는 작업을 거친다. 앞에서 기술하였듯이 우선 stack에 ‘argument 내용’을 하나씩 집어넣는다. 이때 가장 마지막 argument 부터 차례대로 집어넣는다. 382 ~ 390 줄에서는 해당 작업을 수행한다. 392 ~ 395에서는 word alignment를 위해 추가적으로 null data를 push 하는 과정이다. stack은 크기가 4단위로 (4byte) 자라므로, argument들을 push 했을 때, 4의 배수가 아닌 경우 word alignment를 통해 4의 배수가 되게끔 만들어준다. 397 ~ 400에서는 각 argv의 ‘주소값’을 차례로 집어넣어준다. 마지막으로 402 ~ 407에서는 argv[0]이 stack에 push 되었을 때의 stack 주소값을 push해주고, argc를 push, 마지막으로 fake return addresss인 0을 push 해주고 마무리된다.

1. User Memory Access



(userprog/syscall.c syscall\_handler 내부)

User memory access는 위와 같은 함수를 호출함으로써 이뤄진다. system call이 이뤄질 때, stack pointer가 가리키는 곳으로부터 시스템 콜이 필요로 하는 argument의 개수(count) 만큼 address 검사를 진행한다. 검사 조건은 다음과 같다.

(1) 해당 포인터가 NULL가 아닌지

(2) 해당 포인터가 user virtual address를 가리키고 있는지

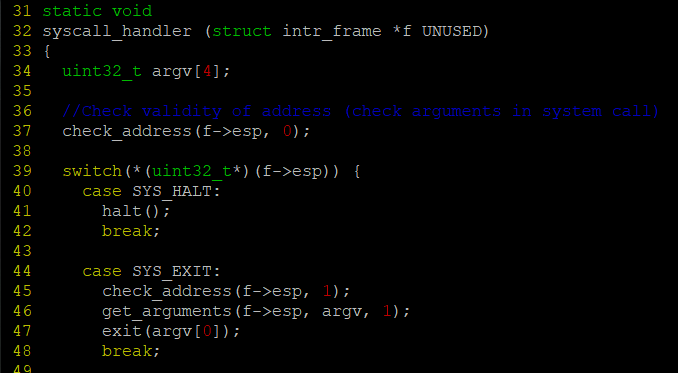
(3) 해당 포인터가 0X08048000이상인지

(4) 해당 포인터가 page mapping이 이뤄졌는지

만약 위 중 하나의 조건이라도 만족하지 않는 경우에는 exit(-1)을 호출함으로써 오류 처리를 하였다.

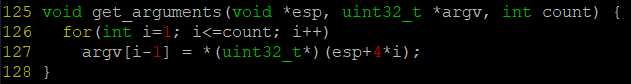
1. System Calls

System call은 기본적으로 userprog/syscall.c에 있는 syscall\_handler를 구현함으로써 이뤄진다. 기본적인 틀은 다음과 같다.

****

(userprog/syscall.c syscall\_handler 일부)

위와 같이 syscall\_handler 내부에 argv[4]를 선언한 뒤, (구현해야 하는 syscall은 최대 4개의 argument를 갖는다. check\_address(f->esp, 0)을 호출함으로써 syscall number가 유효한지 먼저 check 한다. 이 후에 39줄의 switch문을 통해 각 경우마다 만약에 추가적으로 argument를 갖는 system call이라면 check\_address를 통해 argument에 대해서도 pointer가 valid한지 검사해주고, get\_arguments 함수를 호출하여 이전에 선언한 argv에 argument를 저장한다. get\_arguments함수는 다음과 같다.



(userprog/syscall.c get\_arguments함수)

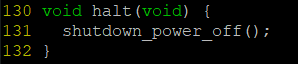
위는 get\_arguments 함수이다. count parameter 만큼 esp로부터 읽어들여 argv에 저장하는 반복문을 구현하였다. 추가적으로 만약에 syscall 호출 이후 return 값이 있는 경우 f->eax에 그 값을 저장하도록 코드를 작성하였다. 각 syscall의 구체적인 구현 방법은 다음과 같다.

- halt

C:\Users\seven\Desktop\4.PNG

(userprog/syscall.c syscall\_handler 내부)

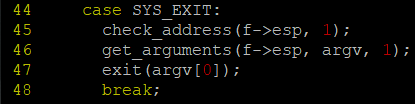
halt syscall의 호출은 system call number가 SYS\_HALT인 경우 호출된다. halt는 따로 argument pass를 하지도 않고 반환도 하지 않으므로 호출된뒤 break문을 통해 바로 종료된다.



(userprog/syscall.c halt)

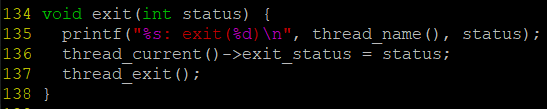
halt의 구현 내용은 위와 같다. 단순히 shutdown\_power\_off()함수를 호출함으로써 pintos가 종료된다. 위 함수는 devices/shutdown.h 파일에 있기 때문에 이를 호출하기 위해 syscall.c 상단부에 해당 헤더파일을 include하였다.

- exit



(userprog/syscall.c syscall\_handler 내부)

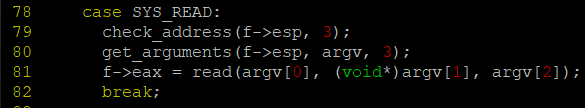
exit 호출은 system call number가 SYS\_EXIT인 경우 이뤄진다. exit는 한 개의 argument(status)를 필요로 하기 때문에 check\_address 및 get\_arguments를 호출하여 argument를 stack으로부터 읽어 들인 후, exit(argv[0])호출을 해준다.



(userprog/syscall.c exit)

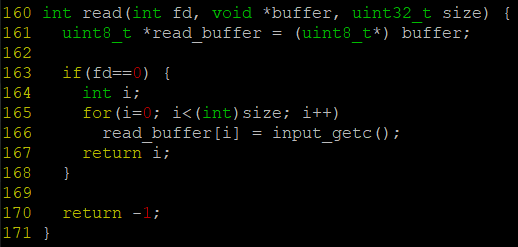
exit 함수 내부는 위와 같다. exit를 호출하게 되면 exit 될 때의 thread\_name과 종료 status를 printf문을 통해 호출한다. 이후에 thread\_current->exit\_status를 status로 갱신하고 thread\_exit()함수를 호출하여 스레드를 종료한다. 이 때 exit\_status는 thread 구조체에 추가로 정의한 변수이다.

-read



(userprog/syscall.c syscall\_handler 내부)

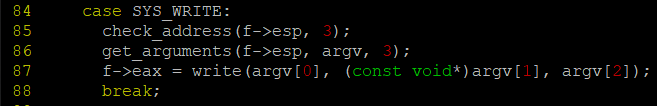
read system call은 3개의 argument를 필요로 한다. 이를 위해 check\_address, get\_arguments함수를 호출하고, return 값이 있으므로 read함수를 호출한 후 f->eax에 저장한다.



(userprog/syscall.c read)

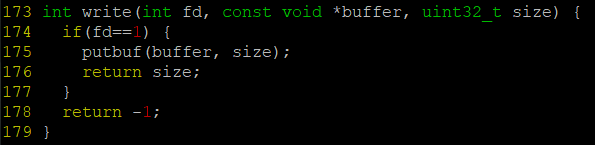
read system call은 우선 stdin에 대해서만 구현한다. input\_getc()를 호출하여 키보키보부터 한 문자씩 입력을 받고 그것을 buffer에 저장한 뒤, 읽어들인 만큼 그 값을 반환한다.

- write



(userprog/syscall.c syscall\_handler 내부)

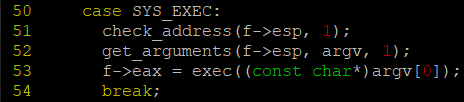
write system call은 3개의 argument를 필요로 한다. 이를 위해 check\_address, get\_arguments함수를 호출하고, return 값이 있으므로 write함수를 호출한 후 f->eax에 저장한다.



(userprog/syscall.c write)

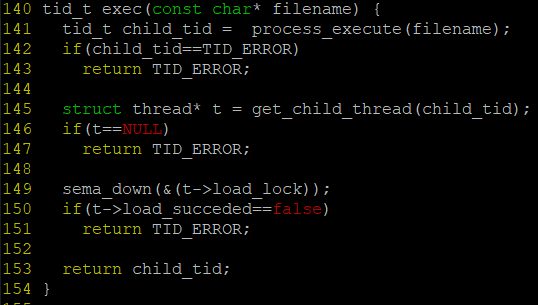
write system call은 putbuf 함수를 이용하여 buffer에 값을 저장한다. putbuf 함수를 호출하기 위해 devices/input.h 헤더파일을 include 하였다.

- exec



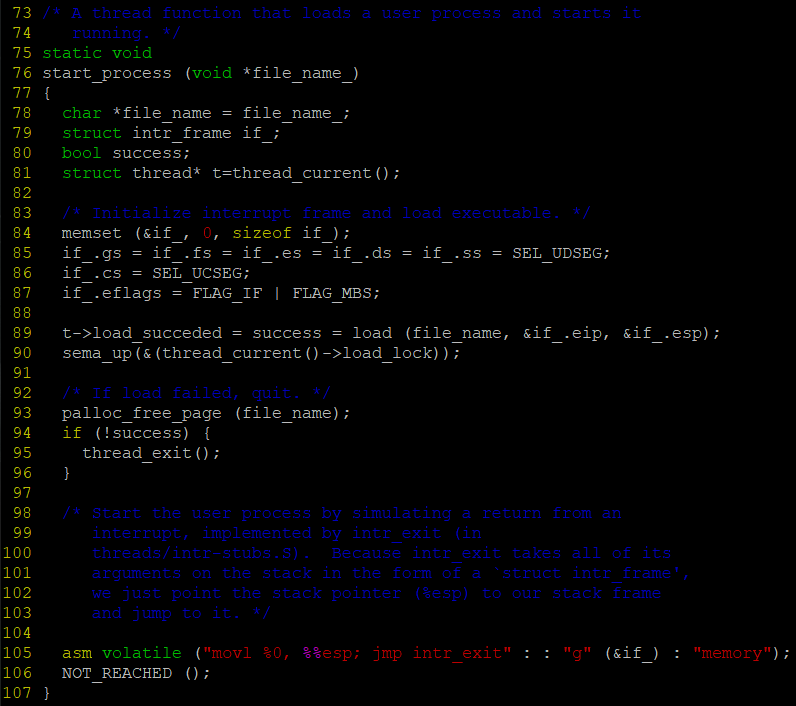
(userprog/syscall.c syscall\_handler 내부)

exec system call은 1개의 argument를 필요로 하므로 우선 check\_address, get\_arguments함수를 호출하고, return 값이 있으므로 f->eax에 저장한다.



(userprog/syscall.c exec)

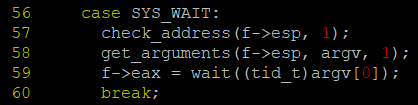
exec system call은 filename에 해당하는 파일을 실행시켜주는 기능을 한다. 이 때 파일을 실행시키면서 child process가 생성이 되고, exec을 호출한 process는 부모 process가 된다. 부모 process는 child가 정상적으로 종료되기까지 기다려야 하므로 semaphore를 이용하여 synchronization을 구현하였다. 만약 정상적으로 child가 종료되었다면 해당 child id를 반환하고, 그렇지 않은경우 TID\_ERROR를 반환하도록 구현하였다. 여기서 load\_lock이라는 semaphore를 이용하였는데 load\_lock은 0으로 초기화되며 sema\_down을 통해 부모는 child가 load 되기를 기다린다.



(userprog/process.c start\_process 내부)

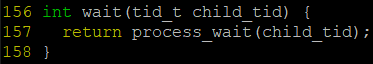
sema\_up은 위와 같이 start\_process에서 load함수가 호출된 직후 수행된다. 이로써 부모 process는 child process가 정상적으로 load 될 때까지 synch를 맞춰줄 수 있다. 또한 thread 구조체에 load\_succded라는 변수를 추가하여 load가 정상적으로 되었는지 판별할 수 있도록 하였다. exec하려는 파일이 없는 경우에 대처하기 위함이다.

- wait



(userprog/syscall.c syscall\_handler내부)

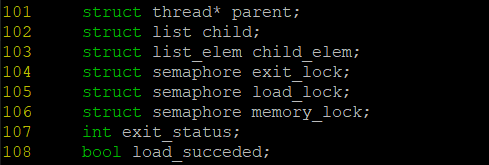
wait system call은 1개의 arguement를 가지므로 check\_address 및 get\_arguments 함수를 호출하여 argument를 저장한다. 이후에 wait함수를 호출하고 return 값이 있으므로 f->eax에 그 값을 저장한다.



(userprog/syscall.c wait)

wati 함수는 단순히 userprog/process.c에 있는 process\_wait함수를 호출함으로써 수행된다.

process\_wait 함수를 설명하기에 앞서 thread 구조체에 추가적으로 선언한 변수 및 함수들에 대해 설명하겠다.



(threads/thread.h struct thread선언 부분)

struct thread\* parent : 현재 thread의 부모 thread를 가리킨다.

struct list child : 현재 thread의 child thread를 list로 관리한다.

struct list\_elem child\_elem : list를 이용하기 위해 선언한 list\_elem 변수이다.

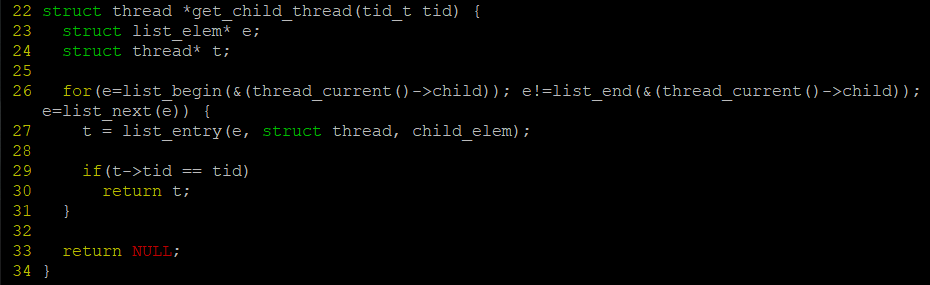
struct semaphore exit\_lock : wait에 이용되는 semaphore이다.

struct semaphore load\_lock : exec에 이용되는 semaphore이다..

struct semaphore memory\_lock : wait에 이용되는 semaphore이다.

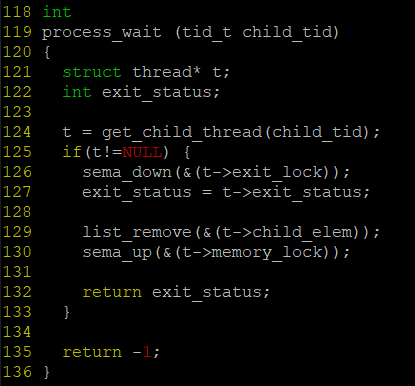
int exit\_status : thread의 exit\_status를 나타낸다.

bool load\_succded : exec에 이용되는 boolean value이다.



(userprog/process.c 상단부에 선언한 함수)

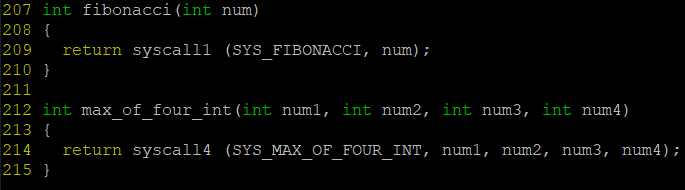
위 함수는 tid가 주어졌을 때 그 tid에 해당되는 child thread를 찾는 함수이다.



(userprog/process.c process\_wait 함수)

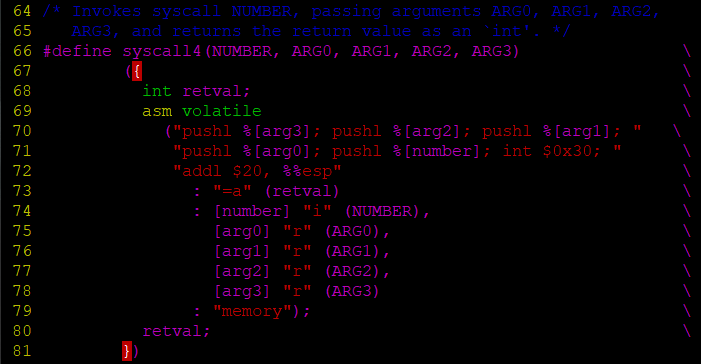
process\_wait 함수는 위와 같다. 우선 parameter child\_tid에 대해 get\_child\_thread라는 함수를 호출하여 해당 tid에 대한 thread를 검색한다. 만약 해당 child 가 없다면 -1을 return 하고 그렇지 않은 경우 sema\_down(exit\_lock)을 통해 process가 종료되기를 기다린다. 이후에 list에서 해당 child process를 삭제해야하는데 child process가 완전히 삭제된후 remove를 호출하면 문제가 발새아할 가능성이 있으므로 child process가 삭제되기 이전에 해당 부분에서 sema\_down(memory\_lock)을 통해 기다린후, list\_remove가 호출된 후 sema\_up(memory\_lock)을 해주어 정상적으로 작동하도록 synchronization을 구현하였다.

1. Additional System calls

****

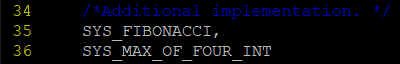
(lib/user/syscall.c)

우선 위와 같이 lib/user/syscall.c에 system call 선언을 해준다. 또한, 위와 별개로 syscall4에 대한 정의를 해주어야 한다. (이전 system call은 최대 3개 까지의 argument만 받을 수 있는 syscall3만 있었으므로 syscall4를 아래와 같이 정의해준다.)



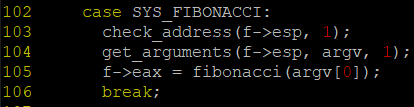
(lib/user/syscall.c)

또한 각 system call에 대한 새로운 syscall number를 정의해주어야 한다. syscall-nr.h에 다음과 같이 enum 변수에 선언해준다.



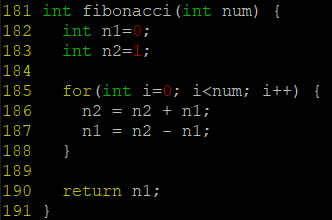
(lib/syscall-nr.h)

- fibonacci



(userprog/syscall.c syscall\_handler 내부)

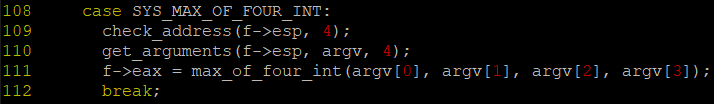
syscall\_handler 내부에 다음과 같이 syscall number가 SYS\_FIBONACCI인 경우에 대해 추가해준다. 우선 check\_address 및 get\_arguments를 수행해준 뒤, fibonacci를 호출하고 그 return 값을 f->eax에 저장한다.

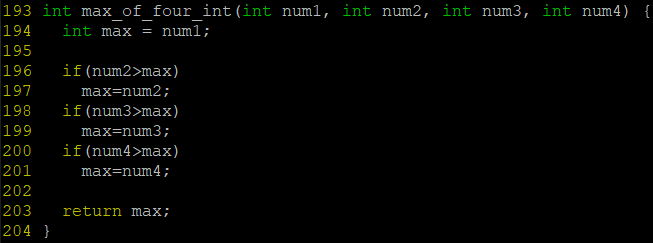


(userprog/syscall.c fibonacci)

parameter num에 대해 num-th 피보나치 수를 반복문으로 계산한 후, return 해주는 구조를 지녔다.

- max\_of\_four\_int

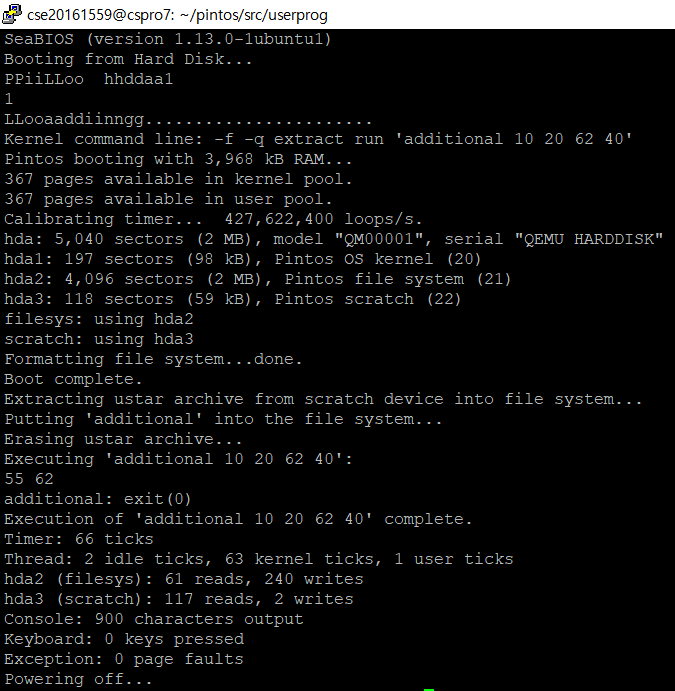




(userprog/syscall.c max\_of\_four\_int)

4개의 paramter에 대해 조건문을 3번 호출하여 가장 큰 수를 return 한다.

* 1. **시험 및 평가 내용**

****