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

담당 교수 : 박성용 교수님

학번 / 이름 : 20161211 / 김동빈

개발 기간 : 2021. 10. 10 ~ 2021. 10. 31

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* Pintos 내에 user program을 수행할 수 있도록 process의 기본적인 halt, execute, wait, exit 등의 system call을 직접 구현한다.
* stdin, stdout에 대한 read, write system call을 구현한다.
* Fibonacci, max\_of\_four\_int system call을 직접 만들어 추가한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

입력 받은 프로그램 수행 명령을 blank를 기준으로 parsing하고 각 argument들을 80x86 calling convention에 따라 stack을 construct한다.

1. User Memory Access

User program의 경우 kernel영역의 메모리를 침범하지 않아야 하기 때문에 user program에서 kernel memory를 access하는 경우 프로그램을 exit한다.

1. System Calls

Halt, execute, exit, wait read(stdin에만 한정), write(stdout에만 한정) 등의 system call을 구현하고 fibonacci, max\_of\_four\_int system call을 새로 만들어 구현하여 user program이 동작할 수 있도록 한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

argument passing은 기본적으로 80x86 calling convention을 따르는데 먼저 blank를 기준으로 parsing된 argument가 순서대로 char\* 타입 배열 argv에 들어 있을 때, ‘\0’을 포함하기 위해 argv[i]의 length + 1만큼 esp에서 뺀 다음 memcpy함수를 이용해 현재 esp위치에 argv[i]를 저장한다.

다음 esp 메모리를 word의 size로 맞추어 주기 위해 필요한 만큼 stack에 null값을 추가하는 word align 과정을 거친다.

이후 esp를 word size만큼 감소시키면서 각 argv의 주소를 저장하고 다음 argument의 개수 argc를 저장한다. 마지막으로 return address를 저장하기 위한 위치에 null을 저장해 stack construction을 완료한다.

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

OS는 각 process들이 서로의 메모리 공간을 침해하거나 kernel의 메모리 공간을 침해하는 것을 방지하기 위해 virtual memory를 이용하는데 pintos도 이와 같이 모든 process는 각자 4GB의 virtual memory를 가지고 있다. 이때 0GB ~ 3GB(PHYS\_BASE)는 user memory, 3GB ~ 4GB를 kernel memory로 설정되어 있다. 이때 user program이 kernel memory를 접근하는 경우 kernel code들을 보호하기 위해 이러한 접근을 invalid memory access로 규정하고 아래의 과정을 통해 처리해준다.

* + Invalid memory access를 어떻게 막을 것인지 설명

User program이 kernel의 메모리 영역에 접근하는 경우 현재 수행중인 thread를 바로 종료하여 이를 방지하도록 하는데 project 1에서 이를 구현한 방법은 userprog/syscall.c의 check\_address함수에서 threads/vaddr.h의 is\_user\_vaddr()함수를 호출하여 현재 접근하는 메모리가 user memory가 아니라면 exit(-1)을 이용해 프로그램을 동료한다. 또한 userprog/execption.c의 page\_fault함수에서도 user memory를 접근하는 경우에 exit(-1)호출해 비정상 종료시킨다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

User program은 직접적으로 kernel memory에 접근할 수 없고 위의 과정처럼 kernel memory에 접근하는 경우 올바르게 작업을 수행하지 못하고 종료될 수 있는데 이렇기 때문에 user program은 kernel memory에 존재하는 memory나 disk에 접근하거나 하는 등의 여러 유용한 함수들을 직접 호출해서 사용하지 못한다. 이때 user program에서 kernel code를 사용하기 위해서 여러 OS는 system call이라는 것을 제공함으로써 해당 문제를 해결하고자 한다. User program에서 system call을 호출하면 해당 system call과 관련된 kernel code를 이용할 수 있는 것이다.

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

halt() : devices/shutdown.h의 shutdown\_power\_off()함수를 호출함으로써 pintos를 종료한다.

exit() : 현재 실행중인 program을 종료하고 exit status를 kernel에 전달하며 현재 program의 이름과 exit status를 출력한다. 올바르게 종료가 되었다면 status는 0이 되고 그 이외의 status는 비정상 종료를 의미한다.

exec() : userprog/process.c의 process\_execute()을 호출하고 그 안에서 새로운 thread를 생성하고 생성된 thread의 tid를 반환한다.

wait() : parent process의 경우 child process가 exit을 호출하여 exit status를 반환할 때까지 기다려야한다. Busy waiting과 thread\_yield()를 이용해 child process가 종료될 때까지 기다리는 과정을 구현했다.

read() : stdin으로 입력이 들어오는 경우에 대해서만 구현되었으며 devices/input.h의 input\_getc()함수를 이용해 입력을 받고 받는 문자의 수를 반환한다.

write() : stdout으로 buffer의 내용을 출력하는 경우에 대해서만 구현하였으며 putbuf()함수를 이용해 반환한다.

fibonacci() : Fibonacci수열의 n번째 항을 반환한다.

max\_of\_four\_int() : 네 개의 정수 중 최댓값을 찾아 반환한다.

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

먼저 user program에서 system call API를 호출하면 lib/user/syscall.c에서 해당 system call에 대한 함수가 수행되어 interrupt가 발생한다. 이후 threads/intr\_stubs.S내의 intr\_entry에서 intr\_handler를 호출하면 threads/interrupt.c의 intr\_handler가 수행되고 이는 userprog/syscall.c의 syscall\_handler를 호출한다. 이때 parameter인 intr\_frame f에서 f->esp가 syscall number을 알고 있고 이를 이용해 해당 system call을 호출하면 kernel API가 호출되어 수행되고 return value가 있는 경우 f->eax에 저장한다. 이후는 지금까지 온 경로를 되돌아가며 반환한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

2021. 10. 10 ~ 2021. 10. 12 : Argument parsing 과정 및 stack construction 과정 구현, user memory access 관리 부분 구현

2021. 10. 13 ~ 2021. 10. 14 : system call 중 halt, read, write 구현

2021. 10. 15 ~ 2021. 10. 19 : system call 중 wait, exit 구현

2021. 10. 22 ~ 2021. 10. 23 : system call 중 exec 구현

2021. 10. 26 : Additional(Fibonacci, max\_of\_four\_int) 구현

2021. 10. 27 ~ 2021. 10. 31 : 구현 내용 수정, 보완 및 마무리

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* Argument Passing

get\_command() : parsing 전 argument로부터 명령어만 parsing해내는 함수, blank를 기준으로 첫번째 argument만 반환한다.

set\_argv() : argument parsing을 이용해 blank를 기준으로 argument들을 구해 argv배열에 저장한다. Argument의 개수 argc를 반환한다.

set\_stack() : 80x86 calling convention에 맞춰 argument들을 esp에 저장하는 과정을 수행하는 함수.

* User Memory Access

check\_address() : 현재 접근하고자 하는 메모리가 user memory가 아니라면 exit(-1)로 프로그램을 종료한다.

userprog/exception.c의 page\_fault함수에서 user가 false인 경우 exit(-1)로 프로그램을 종료한다.

* System Call

userprog/syscall.c의 syscall\_handler함수에서 syscall-nr.h의 syscall 번호를 이용해 switch문을 이용해 각 system call마다 halt, exit, exec, wait, read, write 중 해당하는 함수를 호출해준다.

특히 exit와 wait system call을 구현하는 과정에서 threads/thread.h의 thread struct에 추가해 주어야할 부분이 있는데 먼저 thread가 exit할 때의 status의미하는 exit\_status, 각 thread의 부모 thread를 의미하는 parent, 현재 thread가 exit될 준비가 되었는지 여부를 의미하는 ready\_to\_exit, 현재 thread가 parent의 child\_list로부터 삭제되었는지를 의미하는 deleted\_from\_parent, 현재 thread의 자식 thread list를 의미하는 child\_list, 현재 thread가 부모의 child\_list에 속하기 위해 필요한 list\_elem을 의미하는 child\_elem을 추가한다.

* Additional Implement

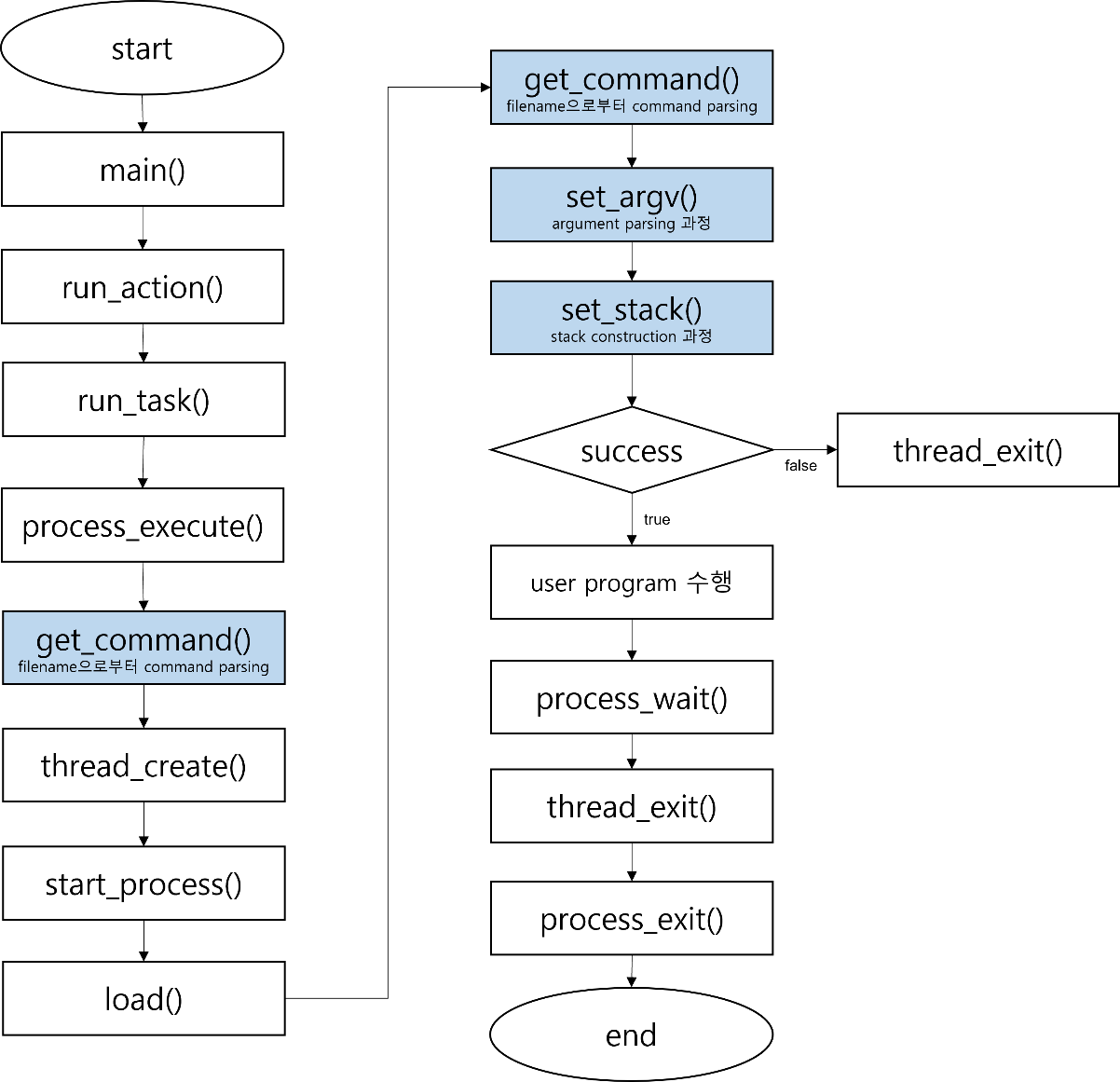
새로운 system call fibonacci와 max\_of\_four\_int를 lib/user/syscall.h와 syscall.c에 추가하고 syscall-nr.h에도 두 system call에 대한 system call 번호를 지정한다. 그리고 argument 4개를 입력받을 수 있는 syscall4도 작성해준다. 또한 examples폴더 하위에 additional.c를 작성하여 additional.c로부터 fibonacci와 max\_of\_four\_int를 호출하여 올바르게 동작하는지 확인한다.

**연구 결과**

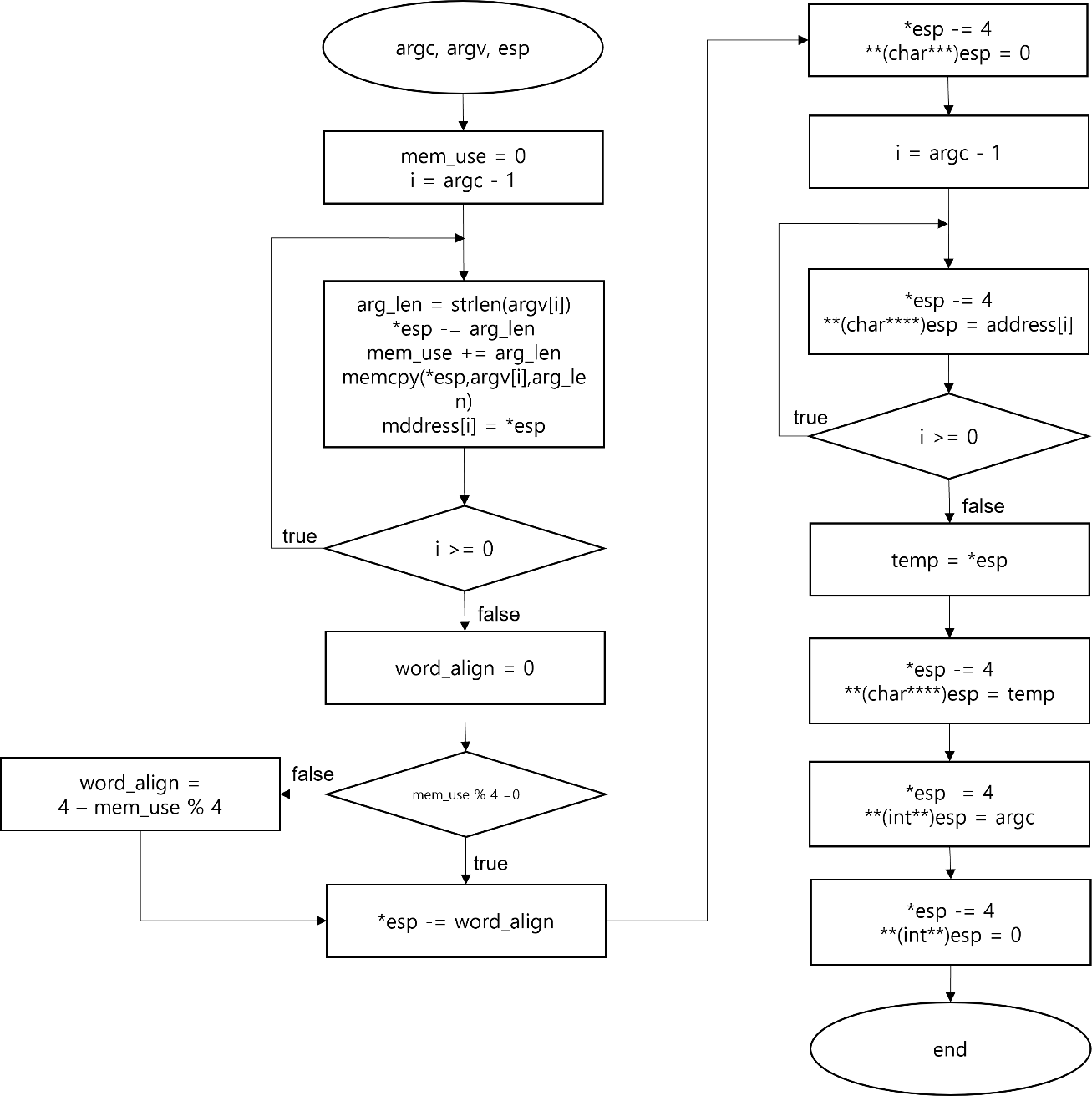
* 1. **Flow Chart**
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

전체 프로그램이 동작하는 방식은 아래와 같고 그 중 파란색으로 표시한 부분이 argument passing과정을 구현하기 위해 추가된 부분이다.

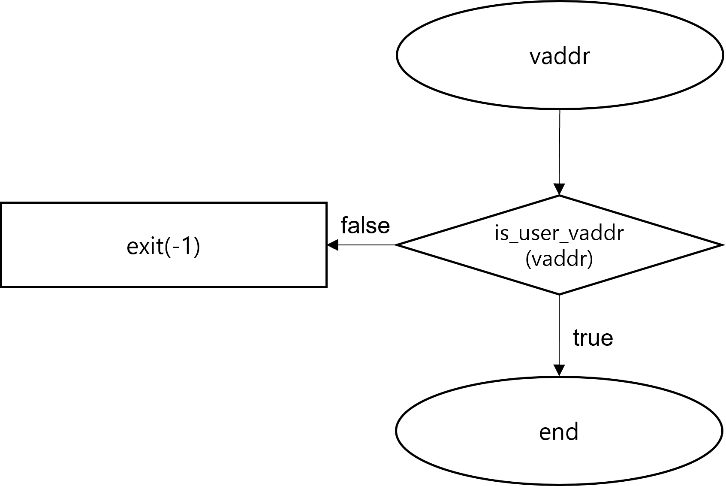


또한 set\_stack()과정만 따로 flow chart를 그려보면 아래와 같다.



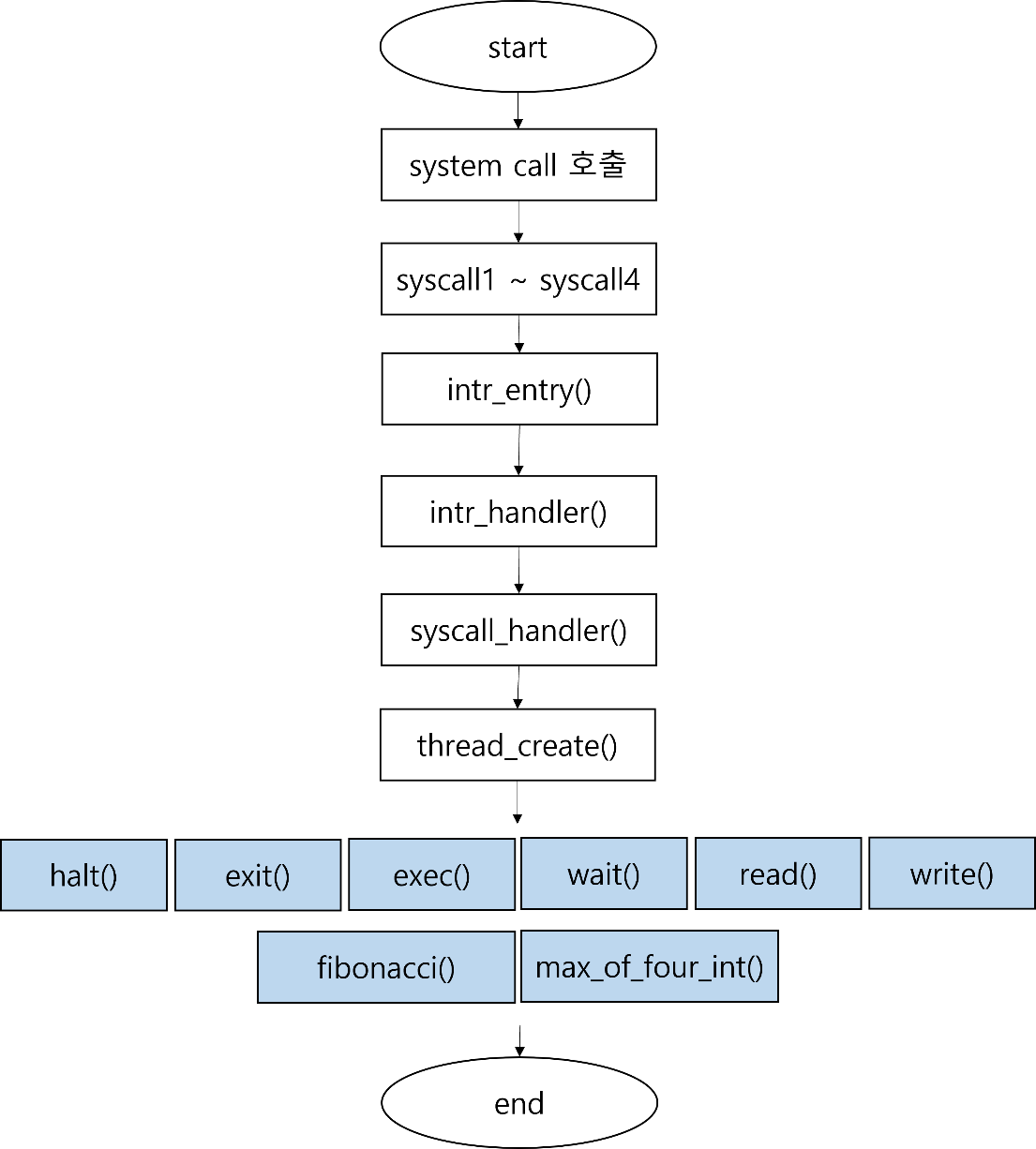
1. User Memory Access

현재 접근하는 메모리가 user memory인지 여부를 판단하는 flow chart는 아래와 같다.



1. System Calls

전제 system call 호출 과정의 flow chart는 아래와 같다. 이 때 구현한 system call은 파란색으로 나타내 주었다.

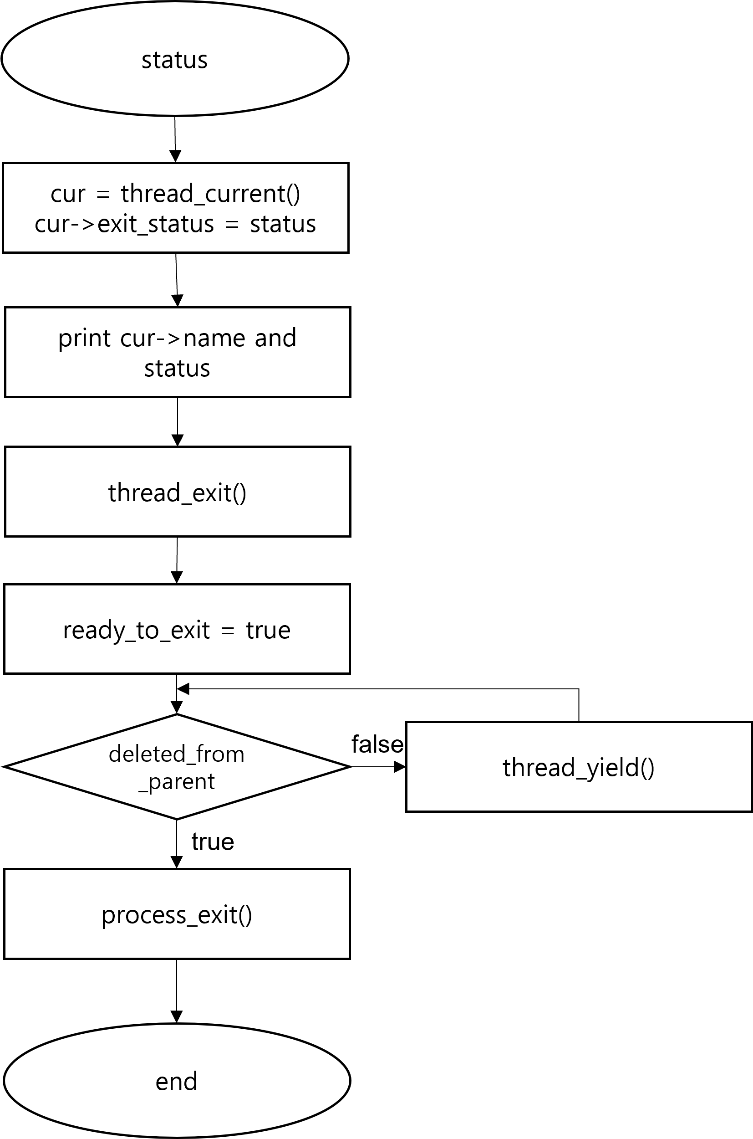


halt()의 flow chart는 아래와 같다.

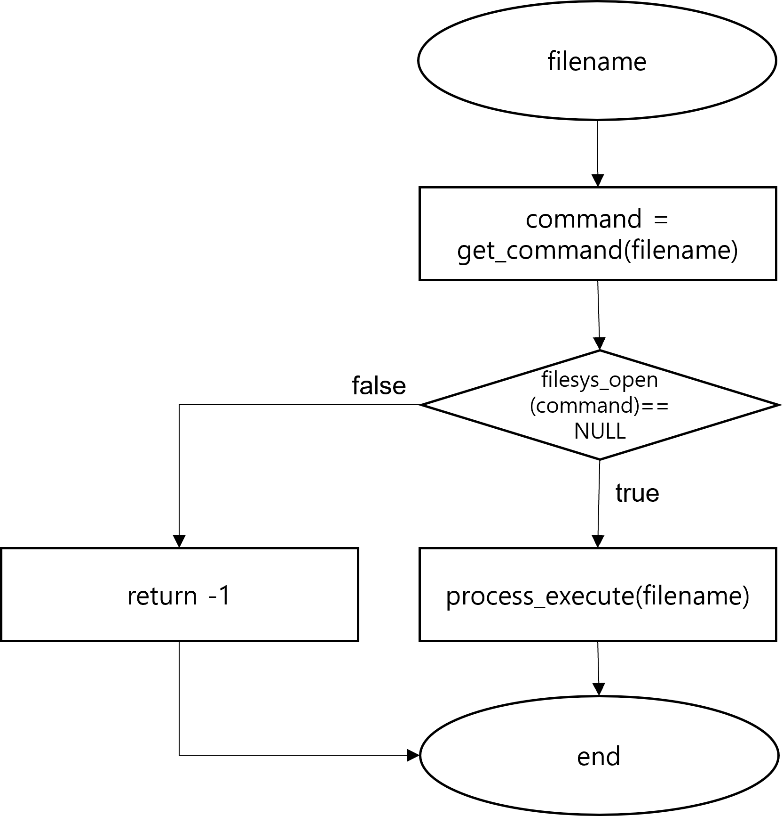
텍스트, 풀볼, 벡터그래픽, 콤팩트디스크이(가) 표시된 사진

자동 생성된 설명

exit의 flow chart는 아래와 같다.



exec의 flow chart는 아래와 같다.

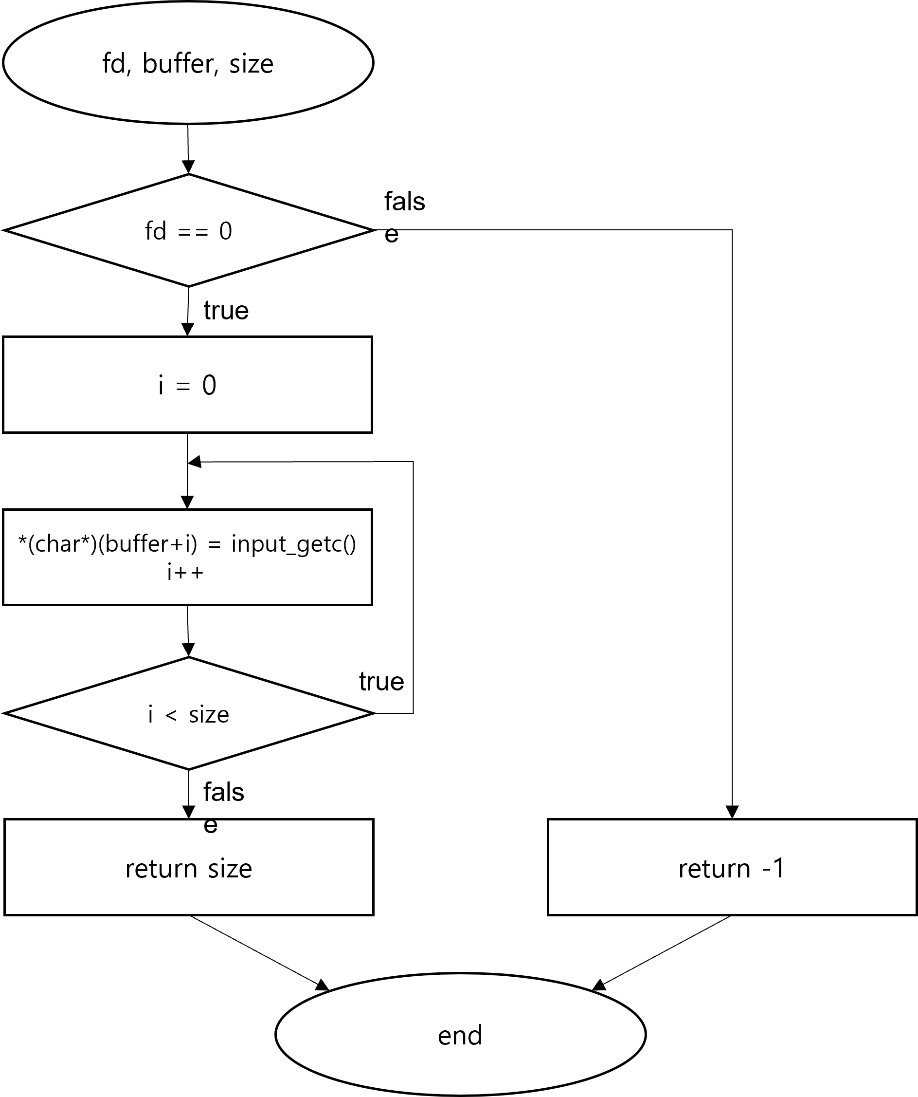


wait의 flow chart는 아래와 같다.

텍스트, 명함, 벡터그래픽, 스크린샷이(가) 표시된 사진

자동 생성된 설명

read의 flow chart는 아래와 같다.

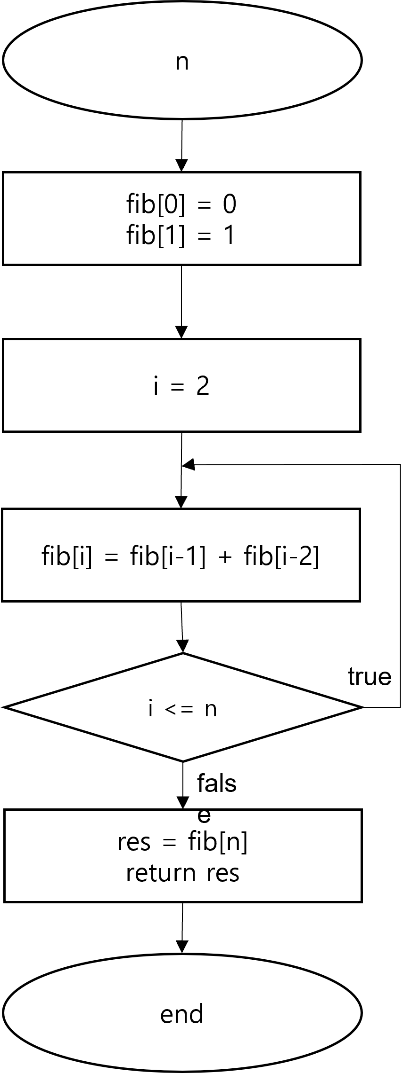


write의 flow chart는 아래와 같다.

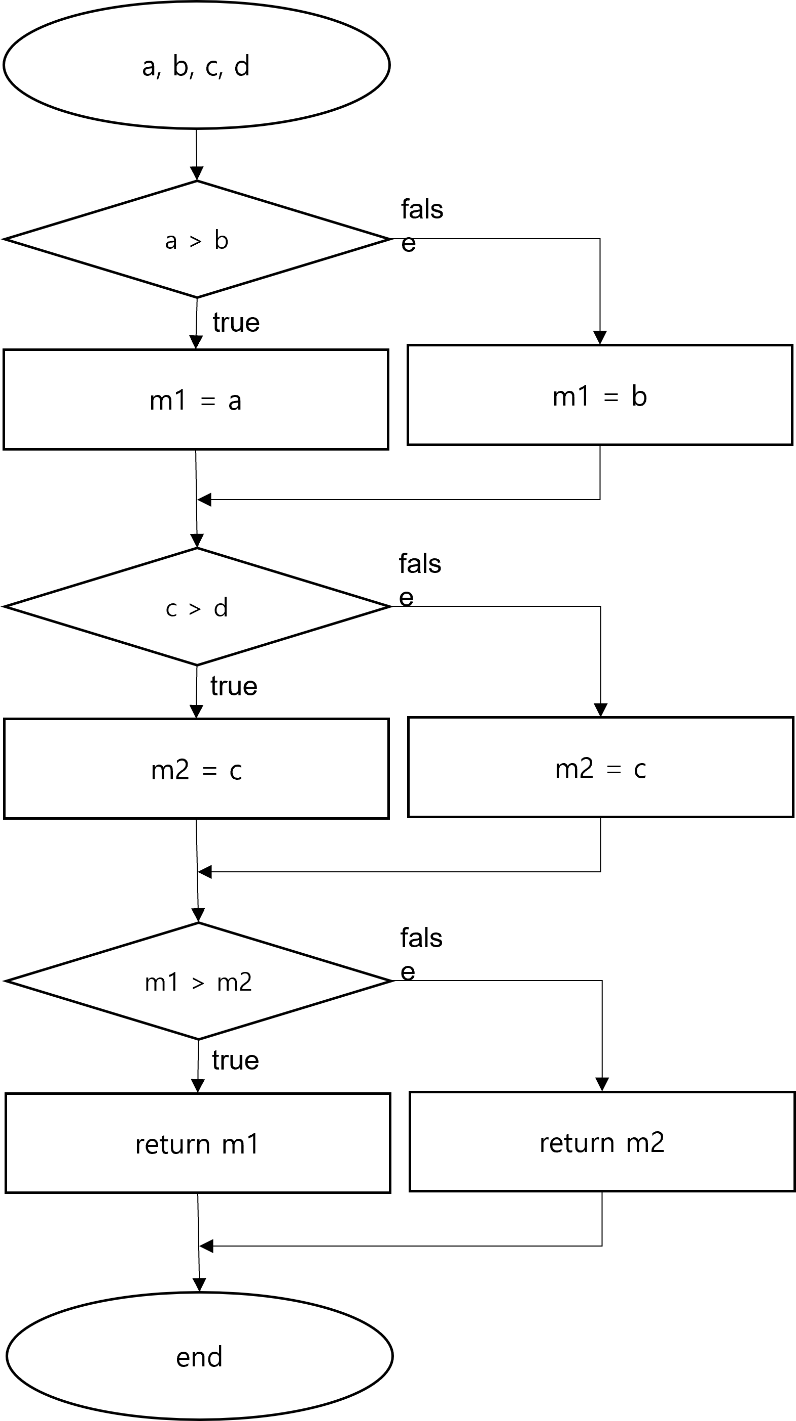
텍스트, 명함, 벡터그래픽, 스크린샷이(가) 표시된 사진

자동 생성된 설명

fibonacci의 flow chart는 아래와 같다.



max\_of\_four\_int의 flow chart는 아래와 같다.



* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

get\_command() : 전체 argument중에서 blank를 기준으로 처음 command만 반환하는 함수이다. 특히 load함수에서 filesys\_open함수를 호출할 때 해당 parsing을 진행하지 않으면 전체 argument가 filename으로 전달되어 올바른 명령 수행이 되지 않아 구현하게 되었다.

set\_argv() : argument parsing 과정과 stack에 저장하는 과정을 구분하기 위해 먼저 argument parsing을 이용해 blank를 기준으로 argument들을 구해 argv배열에 저장한다. 이때 pintos manual에서 언급한 strtok\_r을 사용해주었다. Parsing 과정에서 argument의 개수 argc를 계산하고 이를 반환한다.

set\_stack() : 먼저 set\_argv에서 argv배열에 저장한 argument를 esp에 저장하는데 각 argument의 length에 1을 더한만큼 먼저 esp는 낮춰주고 그 자리에 argv값을 저장한다. 다음은 위의 argument가 총 얼마만큼의 메모리를 차지하는지 확인한 후 4의 배수가 아니라면 4의 배수가 될 수 있도록 null을 추가한다. 이때 4인 이유는 WORD size가 4 byte이기 때문이다. 이를 word align과정이라 한다. 다음은 esp를 word size인 4씩 감소시키면서 argument들의 주소, argv의 주소, argc, return address등을 저장한다.

1. User Memory Access

check\_address() : threads/vaddr.h의 is\_user\_vaddr()함수를 이용하여 현재 접근하고자 하는 메모리가 user memory가 아니라면 아래에서 구현한 system call 함수 exit(-1)로 프로그램을 종료한다.

또한 userprog/exception.c의 page\_fault함수에서 user, not\_present, write중 하나라도 false인 경우 exit(-1)로 프로그램을 종료한다. 이때 page fault 이유를 출력하는 instruction 후에 exit를 호출하면 test과정에서 해당 오류 메시지를 출력하지 않는 것을 요구하고 있어서 해당 instruction전에 exit를 호출해주었다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

먼저 intr\_frame\* f의 esp는 syscall-nr.h의 system call 번호를 저장하고 있으므로 이 값을 이용해 switch문을 통해 각각의 아래의 함수들을 호출해주는데 f->esp+4부터 argument를 하나씩 저장하고 있으므로 해당 값을 이용해 argument를 구하고 함수의 parameter로 전달해준다. 이때 위의 check\_address함수를 이용해 kernel memory를 접근하고 있는 것은 아닌지 판단해준다.

halt() : devices/shutdown.h의 shutdown\_power\_off()함수를 호출함으로써 pintos를 종료한다.

exit() : 현재 실행중인 program을 종료하고 exit status를 kernel에 전달하며 현재 program의 이름과 exit status를 출력한다. 이때 thread\_exit()함수를 호출하는데 thread\_exit()함수 안에서는 process\_exit()함수를 호출한다. 이때 process\_exit()함수를 호출하기 전에 ready\_to\_exit를 true로 바꾸어 아래의 wait과정에서 부모 thread가 자식 thread가 exit될 차례임을 알린다. 이후 부모 thread의 child\_list에서 child thread를 제거하기를 기다려야 하는데 기다리지 않을 경우 부모 thread의 child\_list에서 child thread를 삭제하기 전에 child thread의 메모리가 다 해제되어버리는 경우가 발생할 수도 있기 때문이다. 따라서 busy waiting을 통해 deleted\_from\_parent가 true가 될 때까지 thread\_yield()를 호출하며 기다린다. 위의 과정에서 thread\_exit()이전에 busy waiting과정을 수행하도록 구현하니 bad jump2를 통과는 하지만 굉장히 시간이 오래 소요되어 이를 thread\_exit()내의 process\_exit()함수 호출 이전에 수행되도록 해주었다.

exec() : userprog/process.c의 process\_execute()을 호출하고 그 안에서 새로운 thread를 생성하고 생성된 thread의 tid를 반환한다. 먼저 인자로 주어지는 argument에서 get\_command함수를 이용해 command만 구한 다음 filesys\_open()함수의 결과가 null인지 체크해보았다. 위 부분을 판단하지 않으면 일부 test를 통과할 수 없어서 이 부분을 체크하는 instruction을 추가했고 null인 경우 -1을 바로 반환하도록 했다. 하지만 process\_execute로는 filenam을 전부 전달해야 이후 stack construction이 가능해 filename을 그래도 넘겨주었고 이후 process\_execute 내부에서 thread의 이름을 argument전체가 아닌 command만으로 만들어주어야 exit과정에서 올바르게 원하는 thread이름을 출력할 수 있어 이를 위해 위의 get\_command함수를 이용해주었다. 이후에는 process\_execute함수의 과정을 따라 thread를 생성해주었다.

wait() : parent process의 경우 child process가 exit을 호출하여 exit status를 반환할 때까지 기다려야한다. 따라서 process\_wait()함수를 호출하고 이를 구현해주었는데 먼저 현재 thread의 child\_list안에서 parameter로 전달받은 child\_tid를 tid로 가지는 child가 존재하는지 linked list를 이용하여 찾아본다. 발견하지 못하는 경우는 오류가 있다는 것으로 판단하여 -1을 반환했다. 해당 child thread를 발견한 경우 그 child thread의 ready\_to\_exit가 true인지 살펴본다. 아닌 경우에는 busy waiting을 이용해 true가 될 때까지 thread\_yield()를 호출하며 기다린다. 이후 부모 process의 child\_list에서 child thread를 제거해주는데 이 경우에 child\_list에서 child thread를 제거하기 전에 이미 child\_thread의 메모리 해제과정이 다 끝나버렸다면 list에서의 제거가 안되기 때문에 위에서 살펴보았듯이 exit과정에서 child thread의 deleted\_from\_parent가 true가 되기를 기다리고 있다. 따라서 list에서의 제거가 완료되었다면 deleted\_from\_parent를 true로 바꿔준다. 또한 child thread가 가지고 있던 exit\_status를 반환한다.

read() : fd=0 즉, stdin으로 입력이 들어오는 경우에 대해서만 구현되었으며 devices/input.h의 input\_getc()함수를 이용해 한 문자 씩 buffer에 입력을 받고 받는 문자의 수를 반환한다. 이외의 경우에는 모두 -1을 반환한다.

write() : fd = 1 즉, stdout으로 buffer의 내용을 출력하는 경우에 대해서만 구현하였으며 putbuf()함수를 이용해 buffer에 존재하는 내용을 출력하고 출력된 내용의 size를 반환한다. 이외의 경우에는 모두 -1을 반환한다.

struct thread에 list\_elem child\_elem 변수를 추가해주는 과정에서 처음에는 thread와 list\_elem을 변수로 갖는 child\_item이라는 새로운 struct를 구현하려고 하였으나 list\_entry를 이용해서는 thread로부터 list\_elem을 구할 수 없었기에 이를 구현하는 방법을 고민하다가 thread 내부의 변수로 list\_elem을 두기로 했다.

위의 구현을 수행하면서 thread 구조체에 parent, ready\_to\_exit, deleted\_from\_parent, child\_list, child\_elem을 추가해주었을 때 이를 초기화 해주는 과정이 필요한데 thread\_create에서 init\_thread함수를 호출하면서 thread를 초기화해주는 작업 후에 새로운 thread의 parent는 현재 running\_thread()가 되도록 해주었는데 이때 thread\_current()로 정하는 경우 thread status가 running이 아니라는 오류가 발생해서 runnin\_thread()로 해주었다. 그리고 ready\_to\_exit와 deleted\_from\_parent는 false로 초기화하고 list\_init을 통해 child\_list를 초기화하고, parent의 child\_list에 child\_elem을 추가해주었다.

이때 처음 프로그램이 수행될 때 호출되는 thread\_init()에서는 thread\_create()가 아니라 init\_thread()만 호출해주어 오류가 발생하기에 thread\_init()에서도 init\_thread() 이후에 위의 초기화 과정을 추가해주었다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

먼저 lib/user/syscall.h와 syscall.c에서 fibonacci와 max\_of\_four\_int를 추가로 구현해주었는데 fibonacci의 경우는 syscall1을, max\_of\_four\_int의 경우는 syscall4을 반환하였는데 이를 위해 syscall3을 참고해 syscall4를 추가로 구현해주었다.

userprog/syscall.c에서의 fibonacci()함수에서는 먼저 size n+1의 정수 배열 fib에 메모리를 할당하고 fib[0] = 0, fib[1] = 1로 초기화 한 다음 for문을 이용해 fib[i] = fib[i-1] + fib[i-2]를 통해 fib[n] 값을 찾고 이를 반환한다.

max\_of\_four\_int()함수에서는 먼저 처음 두 정수 중 큰 숫자를 m1에 저장하고 마지막 두 수 중 큰 정수를 m2에 저장하여 m1과 m2중 더 큰 수를 반환하는 방식으로 최댓값을 반환했다.

마지막으로 examples 폴더 아래 additional.c을 작성하는데 4개의 argument를 받아 첫번째 argument를 인자로 하여 fibonacci를 호출하고, 4 argument를 모두 인자로 하여 max\_of\_four\_int를 호출한다. 두 결과 값을 blank를 두고 출력한다.

**시험 및 평가 내용**

* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

아래 첨부된 결과와 같이 additional을 수행해주었을 때 올바르게 Fibonacci와 max\_of\_four\_int의 결과를 출력하는 것을 살펴볼 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명