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

담당 교수 : 김영재 교수님

이름 : 이승형

1. **개발 목표**

이번 프로젝트는 argument passing, user stack, system call handler, system call을 구현하여 Pintos에서 사용자의 프로그램들을 실행할 수 있도록 개선하는 것을 목표로 한다. 그 과정에서 halt, exit, exec, wait, read, write의 system call들과 fibonacci, max\_of\_four\_int와 같은 추가적인 system call을 원할하게 수행할 수 있도록 프로그램을 구현하게 된다.

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

Argument Passing을 구현했을 때에는 실행하고자 하는 프로그램의 이름과 실행에 필요한 인자들이 공백을 기준으로 잘 parsing이 되어야 한다. 그 뒤, 프로그램명을 이용하여 프로그램이 잘 load되어 process가 생성되어야 하고, 이 정보들이 user stack에 올바른 순서로 쌓여야 한다.

1. User Memory Access

User Memory Access를 구현했을 때에는 user program에서 사용, 접근하는 메모리의 주소가 valid한지 잘 판별이 되어야 한다. User program에서 사용/접근을 요청하는 메모리 주소가 NULL값이거나, mapping이 안 된 상태이거나, kernel이 점유하고 있는 메모리라면 이를 invalid하다고 판별하여 exit(-1)로 처리해주어야 한다.

1. System Calls

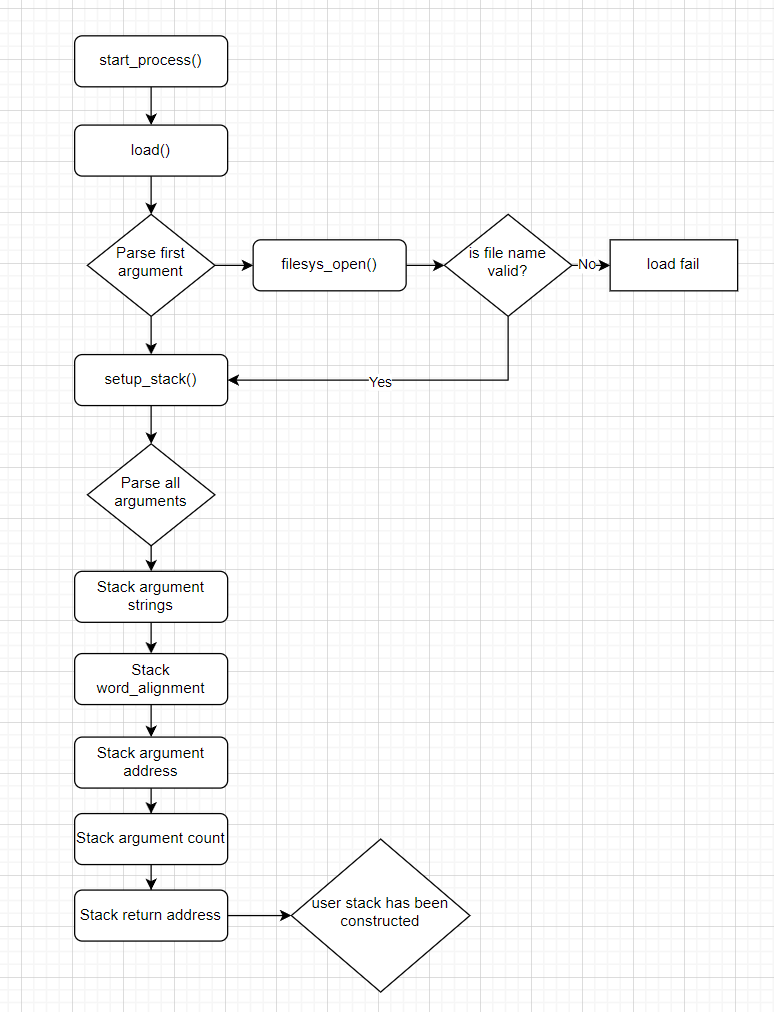
System Calls를 구현했을 때에는 system call이 요청되었을 때, syscall\_num에 따라 동작을 분리하는 handler가 잘 작동하여야 한다. 이번 프로젝트에서 구현하여야 하는 system call들은 halt, exit, exec, wait, read, write으로, 이 중 read와 write은 stdin/stdout에 대해서만 잘 작동하게끔 프로그램을 구현하면 된다. 추가로 피보나치 수열의 n번째 수를 반환하는 fibonacci와 4개의 정수 중 최댓값을 반환하는 max\_of\_four\_int system call도 지원해야 한다.

* 1. **개발 내용**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명  
    스택에 argument를 쌓기 위해 공백을 기준으로 파싱을 진행한다. 그 중, 첫 번째 인자는 파일명으로 filesys\_open을 위해 사용한다. 그 뒤, 파싱된 인자들을 역순으로 80x86 convention에 의거하여 스택에 쌓아준다. 이 때 스택이 위에서 아래로 자라므로, 스택 포인터를 미리 감소시킨 후, 인자들을 복사하여 스택에 넣어준다. 그 뒤, 4 word alignment, 인자들의 주소, 인자의 개수, return address를 추가로 스택에 넣어준다.
* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명  
    Invalid memory access란, 접근을 요청하는 메모리가 invalid함을 말한다. 이 때, 주소 자체가 NULL 값이거나, 메모리가 virtual memory에 mapping되어 있지 않은 상태이거나, user memory 영역을 벗어나 다른 영역의 메모리에 접근하려고 할 때 invalid하다고 한다.
  + Invalid memory access를 어떻게 막을 것인지 설명  
    System call을 요청할 때, intr\_frame이라는 인자를 통해 syscall\_handler가 호출된다. 이 때, intr\_frame->esp라는 스택 포인터를 통해 system call에 필요한 값들이 들어있는 메모리 공간에 접근하게 된다. 이 스택 포인터들이 invalid memory에 접근하고 있는지 판단하여 Invalid memory access를 막는다. 우선, 포인터가 NULL값인지 판단하고, is\_user\_vaddr이라는 함수를 이용하여 user 공간의 메모리에 접근하고 있는지 파악하며, pagedir\_get\_page 함수를 이용하여 메모리가 virtual memory에 mapping되어 있는지 차례로 확인한다. 이 중 하나라도 만족하지 않는다면 비정상 종료인 exit(-1)을 호출한다.
* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명  
    User program에서는 I/O와 같이 프로그램 자체에서 수행할 수 없는 명령을 요청하기도 한다. 이 명령들은 kernel이 제어권을 가지고 user program 대신 수행해주어야 하는데, 이렇게 kernel에게 특정한 동작을 요청하는 것이 system call이다.
  + 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: shutdown\_power\_off() 함수를 통해 Pintos를 종료한다.  
    2. exit: 현재 실행중인 userprogram을 종료하고, 종료한 결과를 kernel에 전달한다. 실행에 성공하면 0을 반환하고, 실패하면 0이 아닌 값들을 전달한다.  
    3. exec: 실행이 가능한 program을 실행한다. exec를 실행한 부모 프로세스는 프로그램을 실제로 실행하는 자식 프로세스가 종료할 때까지 wait 상태에 접어든다.  
    4. wait: 자식 프로세스가 종료될 때까지 부모 프로세스로 하여금 기다리게 하는 system call이다. 자식의 exit status를 반환한다. 자식이 kernel에 의해 실행이 종료되었을 때는 -1을 반환한다.  
    5. read: fd의 파일로부터 size만큼 buffer로 바이트를 읽어들인다. 이번 프로젝트에서는 stdin(fd = 0)의 케이스만 처리한다.  
    6. write: fd의 파일로 size만큼 buffer로부터 바이트를 쓴다. 이번 프로젝트는 stdout(fd = 1)의 케이스만 처리한다.  
    7. fibonacci: 피보나치 수열의 n번째 항을 반환한다.  
    8. max\_of\_four\_int: 인자로 전달된 4개의 정수에 대해 최댓값을 반환한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명  
    유저 레벨에서 시스템 콜 API를 호출하면, 유저가 요청한 syscall\_number와 system call에 필요한 인자들을 스택에 저장한 뒤, int 0x30을 통해 interrupt를 걸어준다. 이 때, interrupt handler가 작동하여 kernel 모드로 진입하게 되고, system call의 인자들에 따라 syscall handler가 작동하여 알맞은 동작을 수행한다. 이후, 반환 값이 있는 system call이라면 레지스터에 이 값을 저장한 뒤, 유저 레벨로 돌아오기 위해 interrupt 이전에 진행되고 있었던 user program의 정보를 load한다. 이제, 유저는 syscall의 결과값을 레지스터로부터 읽어와 다음 로직을 진행할 수 있게 된다.

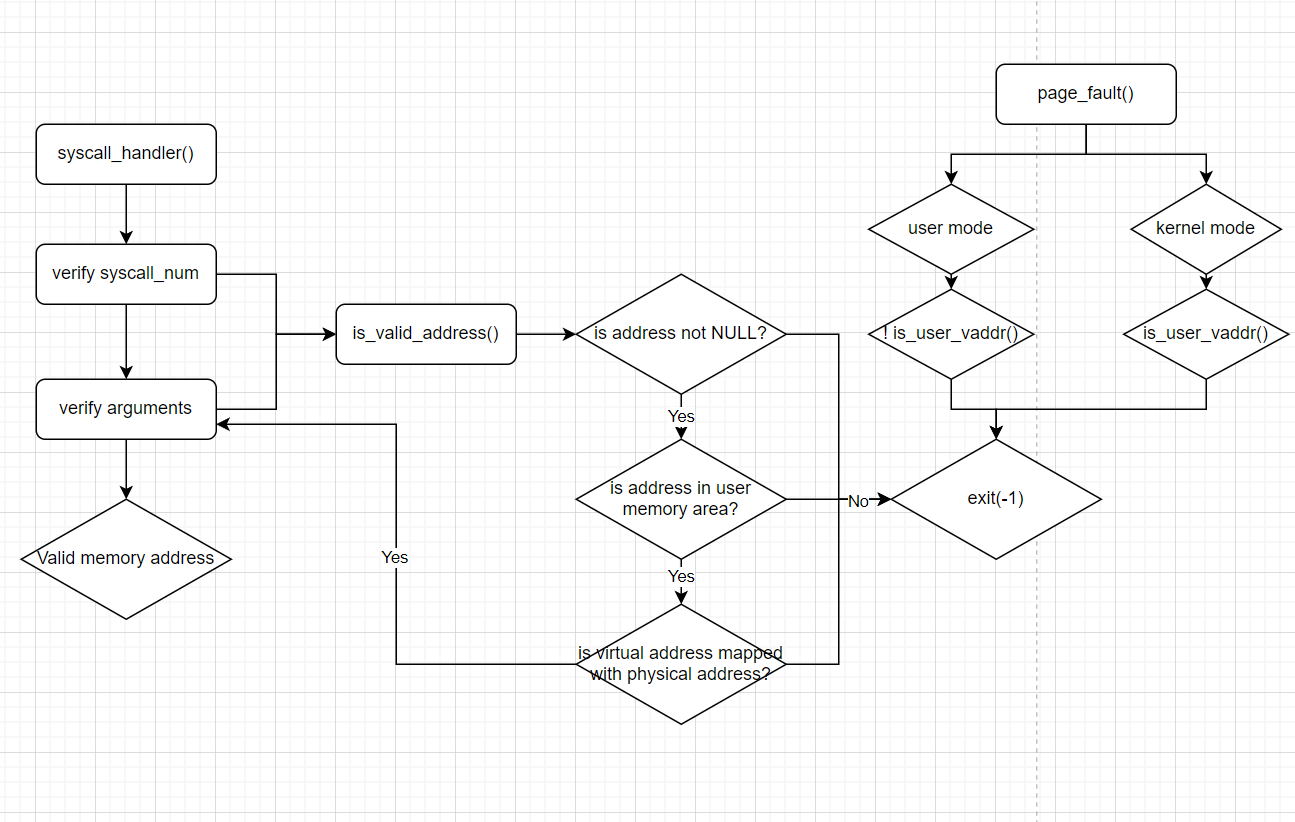
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* Argument Passing: 2024-09-21
* User Memory Access: 2024-09-21
* System Call Handler: 2024-09-22
* System Calls: 2024-09-22 ~ 2024-09-24
* Docs: 2024-09-25 ~ 2024-09-26
  1. **개발 방법**
* Argument Passing  
  process.c의 load라는 함수에서 load가 원활하게 되기 위해 들어온 file\_name을 공백을 기준으로 파싱한다. 그 중 첫 번째 공백 이전을 파일명으로 간주하여 file load를 진행한다. 그 뒤, 파일명을 포함한 모든 인자를 user stack에 넣어주기 위한 작업을 진행한다. 80x86 convention에 의거하여 스택에 인자들을 역순으로 넣어주어야 하기 때문에 파싱을 진행하면서 각 인자를 저장하고, 인자의 개수 argcnt를 함께 저장한다. 그 뒤, 인자를 역순으로 순회하며 esp를 메모리 공간에 맞게 감소시킨 후 인자를 저장한다. Word alignment를 맞추어 준 뒤, 인자들의 주소, argcnt, return address를 차례로 쌓아 user stack 구성을 마친다.
* User Memory Access  
  syscall.c의 syscall\_handler에 들어온 intr\_frame의 포인터에 대한 메모리 접근의 유효성을 판별하기 위해 is\_valid\_address라는 함수를 새로 구현할 것이다. 이 함수에서는 앞서 말한 3가지 요소를 판별한다.   
    
  1. 주소값이 NULL이 아닌지 판별.  
  2. 주소값이 user\_vaddr이 아닌지 판별. 이때, threads/vaddr.h에 구현된 is\_user\_vaddr이라는 함수를 이용한다.  
  3. 주소값이 virtual memory와 mapping 되어있지 않은지 판별. 이때, userprog/pagedir.c에 구현된 pagedir\_get\_page라는 함수를 이용한다.  
    
  이러한 요소들을 syscall에 필요한 인자에 맞게 판별할 것이다. 예를 들어, write는 3개의 인자를 필요로 하기 때문에, syscall\_num, arg1, arg2, arg3에 해당하는 4개의 포인터에 대한 validity 검사를 진행한다. 만약 validity 검사를 통과하지 못하였다면, 오류로 인한 user program 종료를 의미하는 exit(-1)을 수행한다.  
  또한, page\_fault가 발생하였을 때, userprog/exception.c에 존재하는 page\_fault라는 함수가 호출되는 데, 이 공간에서도 user mode인 상태에서 user\_vaddr을 참조하고 있지 않은 경우와 user mode가 아닌 상태에서 user\_vaddr을 참조하고 있는 경우를 판별하여 exit(-1)을 통해 알맞게 종료하여 준다.
* System Calls  
  System Calls를 위해서는 syscall\_handler와 각각의 syscall에 대해 개발을 진행한다. Syscall\_handler에서는 syscall\_num과 syscall에 필요한 arguments에 대한 user memory access에 대한 검사를 진행하는 함수를 호출하고, 해당 syscall\_num에 따라 해당하는 syscall 함수를 호출하는 동작을 수행한다. 만약, 반환값이 있는 syscall이라면, 이를 intr\_fram의 eax 레지스터에 저장하여 user mode로 전환이 되었을 때 그 값을 사용할 수 있도록 한다.   
  그 뒤, 각각의 system call을 수행하는 함수 wait(), halt(), exec()…. 를 구현한다. 이 때, 추가적인 구현인 fibonacci와 max\_of\_four\_int의 system call은 해당하는 syscall\_num이 존재하지 않기 때문에 syscall\_num들이 enum 타입으로 정의되어 있는 syscall-nr.h에 SYS\_FIBO, SYS\_MAX을 추가하여 준다. 또한, 아직 Pintos에는 argument가 4개인 system\_call을 수행하는 syscall4를 추가로 구현하여야 한다.

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



1. User Memory Access



1. System Calls

텍스트, 도표, 평면도, 평행이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**

1. Argument Passing  
   Argument Passing의 구현은 userprog/process.c의 load함수에서 이루어졌다. file\_name으로 주어진 인자에 대해, 첫 번째 공백 이전의 문자열을 파일명으로 간주하여 fname이라는 변수에 저장한 뒤, filesys\_open의 인자로 넘겨주었다.  
   

그 뒤, 실행 가능한 file의 open이 정상적으로 이루어지고, 본격적으로 user stack을 구성하는 부분을 구현하였다. 공백을 기준으로 인자의 개수를 우선적으로 세어준 다음에, 인자들을 저장할 배열인 args와 현재 인자에 대한 정보 now를 동적할당하였다. 이러한 구현에서는, 공백이 여러 개 연속으로 들어오는 인자에 대해서 빈 문자열을 저장하는 문제점이 발생하였다. 따라서, 빈 문자열을 저장하지 않도록 double space와 같은 케이스를 처리하여 정확한 인자의 개수인 aptr을 저장하였다.  
텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명  
그 뒤, 변수에 저장한 인자들을 실제로 user stack에 쌓는 부분을 구현하였다. 앞선 과정에서 args에서 저장한 문자열을 역순으로 차례로 스택에 넣어주고, word alignment을 수행한 뒤, 인자의 주소, 인자의 개수, return address를 차례로 스택에 넣어 user stack의 구성을 마쳤다.  
텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

User Memory Access의 구현은 userprog/syscall.c에서 이루어졌다. syscall에서 호출되는 주소의 정합성을 검사하는 함수 is\_valid\_address를 선언하여 사용하였다.  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
이 함수에서는 세 가지를 판별한다. 우선, address 자체가 NULL 값인지 판별하고, 그 뒤 address에서 접근을 요청하는 메모리 주소가 user 영역의 메모리인지 판별한다. 이 때, threads/vaddr.h에 포함된 is\_user\_vaddr이라는 함수를 사용한다. 마지막으로, 주소의 virtual address가 실제 physical address와 mapping이 되어 있는지 확인하기 위해 userproc/pagedir.c에 구현된 pagedir\_get\_page 함수를 사용하여 검사를 해준다. 이렇게 구현된 함수는 syscall\_handler에서 호출되는 인자에 대해 사용된다.  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명  
 이러한 방식으로도 발생하는 올바르지 못한 접근(file\_open 등)을 한 번 더 검사해주기 위해 userprog/exception.c의 page\_fault 함수에 대한 수정도 진행하였다.   
텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

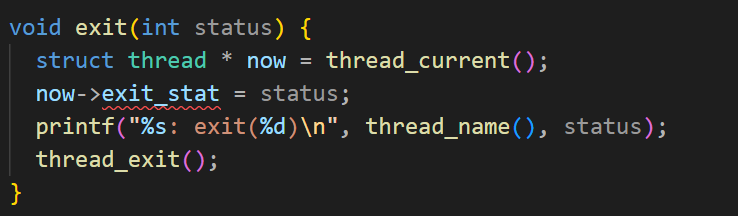
자동 생성된 설명  
user mode이면서 user\_vaddr이 아닌 경우와 kernel mode이면서 user\_vaddr인 경우에 exit(-1)을 하는 방식을 채택하였다.

1. System Calls

우선 interrupt handler로 인해 호출되는 syscall\_handler를 syscall.c에 구현하였다. syscall\_handler에서는 syscall에 필요한 syscall\_num과 인자들에 대한 주소 정합성을 판별하는 is\_valid\_address의 호출이 이루어지고, syscall\_num에 따라 조건 분기를 태운 뒤, 알맞은 함수로 연결된다.  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명syscall\_num에 따라 필요한 개수의 인자를 검사하고, 반환 값이 있는 system call의 경우 레지스터에 저장하게끔 구현을 진행하였다. 각 syscall에 대한 구현은 다음과 같다. 모든 syscall 함수는 syscall.c에 구현되었다.  
1. halt:  
텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명  
halt는 shutdown\_power\_off를 call해주고 끝난다. 이는 Pintos를 종료시킨다.  
2. exit:  
  
exit은 현재 thread를 종료시킨다. 이 때, thread의 종료 상태를 저장하는 exit\_stat에 status를 저장한 뒤, thread의 이름과 종료 상태를 출력하고 thread\_exit()을 통해 종료시킨다. 이를 위해 threads/thread.h의 thread라는 struct에 exit\_stat라는 멤버 변수를 추가하였다.  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
3. exec:  
텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명  
exec는 userprog/process.c의 process\_execute라는 함수를 실행시킨다. 이 경우, 전달된 cmd\_line 전체를 program의 이름으로 인식하는 오류가 발생하여, process\_execute 함수 내에 첫 번째 공백 이전의 문자열만을 프로그램 이름으로 간주하게끔 하는 파싱 과정을 추가하였다.  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
4. wait:  
폰트, 텍스트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명  
wait은 userprog/process.c의 process\_wait이라는 함수를 실행시킨다. process\_wait이라는 함수는 전혀 구현되어 있지 않아 내용을 새로 구현하였다. 현재 thread의 자식 thread의 list를 순회하여 인자로 들어온 chid\_tid와 일치하는 thread를 찾아 종료를 기다리는 로직이다.  
텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명  
이 과정에서 앞서 exit\_status를 새로 정의한 것처럼, thread라는 struct에 child와 child\_element라는 list와 list\_element를 정의하였고, wait 로직에 필요한 semaphore 두 개를 새로 선언하였다. 먼저 sema\_down된 child의 semal\_all은 process.c의 process\_exit이라는 함수에서 up된다.   
텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명  
즉, 자식 thread가 종료되기 직전까지 sema\_all이라는 semaphore은 down된 상태이기 때문에, 부모 thread가 wait 상태에 접어들게 된다. 자식 프로세스가 종료되기 직전 sema\_all이 up되고, 부모가 자식 thread를 list에서 삭제하기 전까지 sema\_child가 down되어 wait 상태가 된다. 그 후, 부모가 자식을 list에서 삭제하고 그제서야 자식 thread가 종료된다.  
5. read:  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
이번 프로젝트에서는 stdin의 fd에서만 read를 지원하므로, fd가 0이 아닐 때에는 종료해주었다. stdin의 경우에는 size만큼 devices/input.c의 input\_get() 함수를 이용하여 바이트를 읽어들여왔다.  
6. write:  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
이번 프로젝트에서는 stdout의 fd에서만 write를 지원하므로, fd가 1이 아닐 때에는 종료해주었다. stdout인 경우에는 size만큼 lib/kernel/console.c의 putbuf() 함수를 이용하여 바이트를 쓰기 하였다.

1. Additional System calls

1. fibonacci:  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
fibonacci의 경우에는 인자로 들어온 n번째 피보나치 수열의 항을 구하는 것이다. sliding window 기법을 이용하여 변수 두 개를 관리하여 피보나치 값을 구하는 로직을 구성하였다.   
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
또한, 새로운 syscall을 위하여 lib/syscall\_nr.h에 SYS\_FIBO라는 syscall\_num을 새로 추가하였다. 그리고 lib/user/syscall.c에 fibonacci syscall의 system call API를 추가하였다.  
폰트, 텍스트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명  
fibonacci는 한 개의 인자를 이용한 system call이기 때문에 syscall1을 이용하여 스택을 구성한다.  
2. max\_of\_four\_int:  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
max\_of\_four\_int의 경우에는 단순히 3번의 비교를 통해 가장 큰 값을 가지는 인자를 반환하는 로직으로 구성하였다.   
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
또한, 새로운 syscall을 위하여 lib/syscall\_nr.h에 SYS\_MAX라는 syscall\_num을 새로 추가하였다. 그리고 lib/user/syscall.c에 fibonacci syscall의 system call API를 추가하였다.  
폰트, 텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명  
max\_of\_four\_int는 네 개의 인자를 이용한 system call이기 때문에 syscall4를 이용하여 스택을 구성한다. 그러나, syscall4는 아직 구현되어 있지 않기 때문에 syscall3를 참고하여 syscall4를 새로 구현하였다.  
텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.  
  텍스트, 스크린샷, 폰트이(가) 표시된 사진

  자동 생성된 설명**실행 결과, 피보나치 수열의 10번째 항인 55와, 4개의 정수 중 가장 큰 수인 62가 잘 출력되는 것을 확인할 수 있었다.