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

담당 교수 : 박성용

조 / 조원 : 이선호

개발 기간 : 2022.09.18 ~ 2022.10.03

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

Pintos에서 사용자가 user mode에서 실행 가능한 user program 명령어를 구현해야 하며, 이를 위해 argument passing, system call, memory access의 내용을 이해하고 필요한 부분을 완성한다. 더 나아가 두 가지의 새로운 user program을 추가로 만드는 것을 목표로 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

사용자가 Pintos console에서 입력한 명령어를 space로 구분 지어서 argument를 token 단위로 나누고, Pintos에서 사전에 정의한 형식에 맞게 stack에 저장한다.

1. User Memory Access

사용자가 kernel 영역에 해당되는 메모리 주소를 접근하지 못하도록 exit를 발생시켜서 실행 중인 프로세스를 강제로 종료한다.

1. System Calls

User mode에서 실행할 수 없는 일을 kernel mode에서 처리하기 위해 user program에서 대응되는 system call을 호출함으로써 system call handler라는 interrupt를 발생시킨다. 이 interrupt handler에서 system call 호출 시 stack에 쌓았던 argument를 가져와서 명령어 값에 대응되는 명령을 수행한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 입력 받은 명령어 줄에서 space를 기준으로 문자열을 인자로 나누는 tokenizing을 수행하며, 명령어 줄을 복사한 문자열 메모리를 strtok\_r 내장함수를 사용하여 인자를 구분하였다.
  + 나눈 인자들을 pintos에서 정의한 형식에 맞게 setup\_stack 함수로 생성한stack에 push 하는데, stack의 가장 윗부분을 가리키는 esp 포인터를 4 bytes 단위로 조절해가면서 뒤의 인자부터 차례대로 stack에 넣는다. 넣은 인자 값이 차지하는 영역이 4 bytes 단위로 끊어질 수 있도록 word alignment를 수행하며, 이후 첫 번째 인자 값을 가리키는 메모리 주소, 인자 개수, 그리고 반환 주소를 차례로 stack에 넣는다.
* User Memory Access
  + System call을 호출할 때 kernel 영역 등 사용자가 접근하지 말아야 할 메모리 주소를 요청할 수도 있고, 가상 메모리에 mapping되지 않는 물리 메모리에 접근하려고 할 수도 있다. 또한 아예 NULL pointer를 요청할 수도 있는데, 이러한 경우가 invalid memory access이다.
  + Invalid memory access로 인해 발생하는 문제를 사전에 예방하는 여러 함수가 존재하며, 기본적으로 어떤 메모리 주소가 user 영역 또는 kernel 영역인지를 확인할 수 있도록 is\_user\_vaddr와 is\_kernel\_vaddr 함수가 있다. 또한 어떤 가상 메모리 page가 실제 물리적인 메모리에 mapping 되는지를 알 수 있는 page\_dir\_get\_page()를 통해 확인할 수 있다.
* System Calls
  + User program이 운영체제에서 허락하지 않은 작업을 함부로 처리할 수 있도록 안전성을 보장하고자 user program은 kernel mode와는 분리된 user mode에서 실행되는데, I/O 작업과 같이 운영체제의 기능을 사용하고자 할 때는 kernel mode에서 원하는 작업을 수행할 수 있어야 한다. 이를 위해 user mode에서 system call을 호출하여 user program이 직접 접근하거나 실행할 필요 없이 원하는 작업을 수행하도록 한다.
  + (1) halt: “devices/shutdown.h”의 shutdown\_power\_off 함수를 통해 pintos를 종료한다.  
    (2) exit: 실행 중인 user program을 종료하면서 thread의 상태를 출력하고, thread의 멤버 변수인 exit\_status를 업데이트한 후 thread\_exit을 통해 process\_exit가 수행된다.  
    (3) exec: 자식 process를 실행하기 위해 process\_execute를 통해 file\_name의 맨 앞 인자인 명령어 이름 부분만 받아서 해당 이름을 지니는 thread를 생성하며, 생성한 thread의 id를 반환 받는다.  
    (4) wait: 기다리고자 하는 자식 process의 id를 통해 id가 유효한지를 확인하고, 해당 process를 찾아서 exit\_status를 넘겨 받는다. 현재 실행 중인 부모 프로세스가 자식 프로세스가 기다릴 때까지 완전히 종료되지 말아야 하므로 자식 process를 제거할 때까지 종료되지 않도록 semaphore를 통해 synchronization 한다.  
    (5) read, write: stdin에 대응되는 기본적인 file descriptor인 0인지를 확인한다. File descriptor가 가리키는 파일을 buffer로 읽으면서 몇 바이트를 읽었는지를 반환한다.  
    (6) write: stdout에 대응되는 기본적인 file descriptor인 1인지를 확인한다. File descriptor가 가리키는 파일을 buffer로 쓰면서 몇 바이트를 썼는지를 반환한다.  
    (7) fibonacci: 요청으로 온 n번째 피보나치 수를 변수 세 개를 사용하여 구한다.  
    (8) max\_of\_four\_int: 명령어에서 입력 받은 네 개의 수 중에서 가장 큰 값을 반환한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명  
    User level에서 system call을 호출할 때 system call number와 argument가 stack에 쌓이고, interrupt가 발생하면서 mode bit가 user mode에서 kernel mode로 변경된다. Interrupt vector에서 system call을 처리하는 system call handler가 호출되고, 앞서 stack에 쌓은 인자들을 받아서 요청 받은 system call에 대응되는 번호를 찾아 해당 번호의 작업을 처리한다. 작업이 완료된 후 eax에 반환값을 저장하여 다시 mode bit를 변경하고 user mode의 system call을 호출한 곳으로 돌아간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

09/18 ~ 09/20: 구현에 필요한 개념과 userprogram 작동 flow를 이해하고, argument passing을 구현한다.

09/21 ~ 09/24: system call handler, user memory access, synchronization을 구현한다.

09/25 ~ 09/27: additional에 속하는 추가적인 system call인 fibonacci와 max\_of\_four\_int를 구현한다.

09/28 ~ 10/01: debugging을 하고 보고서를 작성한다.

* 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 passing은 process.c의 load 함수에서 parse\_file\_name과 construct\_stack 함수를 통해 구현한다. parse\_file\_name은 명령어로 온 복사된 문자열을 space로 tokenizing하여 나눈 token과 그 개수를 저장하는 구조체로 정의한 parsed\_args에 포인터 배열로 저장한다. 이후 construct\_stack 함수에서 setup\_stack 함수로 생성된 stack에 가장 top을 가리키는 ESP 포인터를 이동하면서 스택에 쌓는다.

Memory access 접근 제한을 위한 함수는 userprog의 exception.c의 page\_fault 함수에서 threads의 vaddr.h의 is\_kernel\_vaddr 함수를 이용해 현재 요청 받은 메모리 주소가 유효한지를 판단한다. 또한 system call handler에서도 stack으로 넘겨 받은 메모리 주소가 유효한지를 판단하기 위해 is\_user\_vaddr 함수를 내부에서 사용하는 check\_valid\_address 함수를 구현했다.

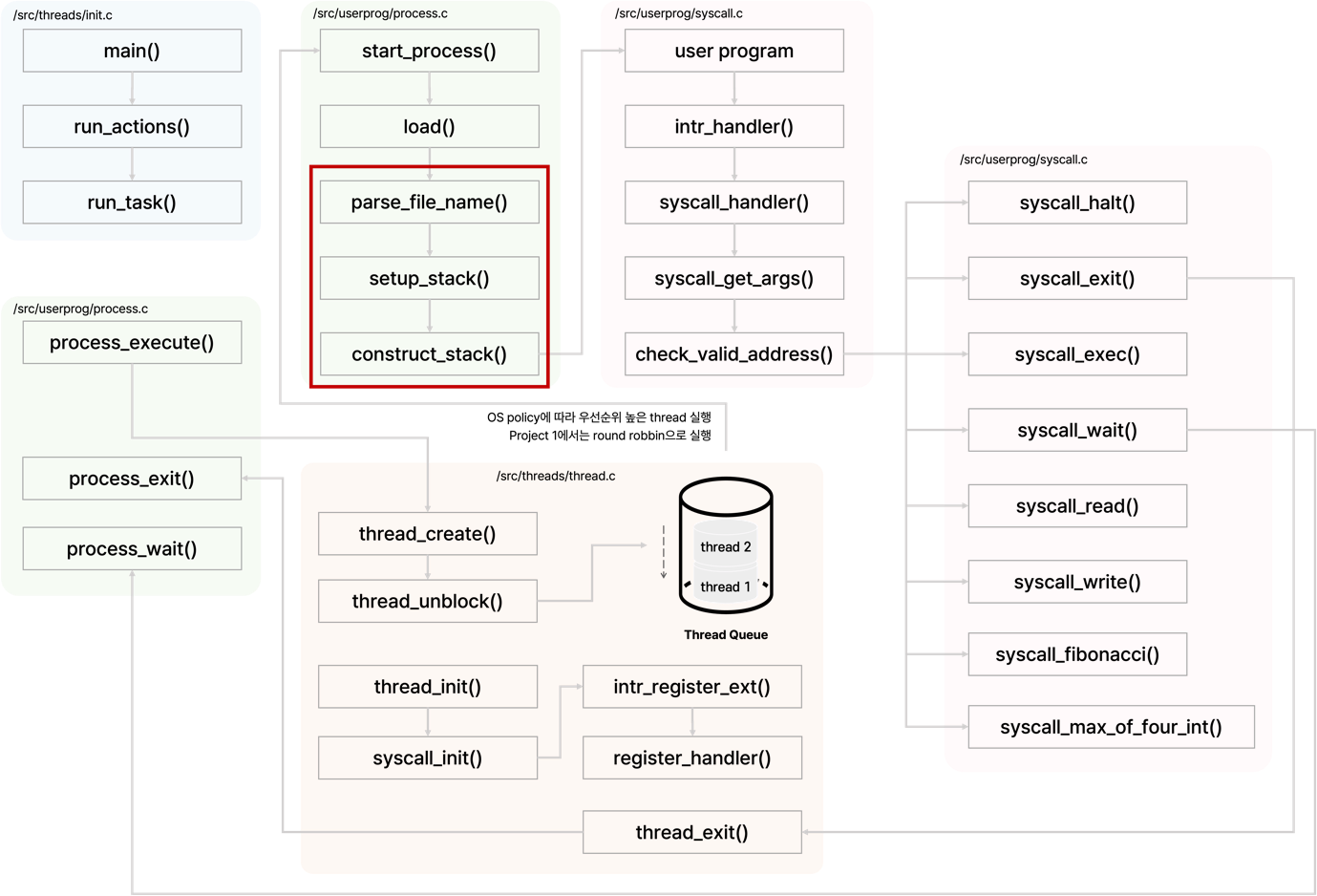
System call이 호출되었을 때 interrupt handler에 필요한 정보를 넘겨줄 수 있도록 assembly code로 stack에 인자를 push하고, system call handler에서 frame의 esp라는 포인터 변수를 통해 앞에서 stack에 넣은 인자를 차례로 꺼내서 switch문으로 요청으로 온 작업을 분기하여 적합한 system call 함수를 호출한다. 이때, lib 폴더의 syscall-nr.h에서 사전에 정의한 작업 번호를 확인하여 switch문을 분기한다. 여기에 새롭게 만든 fibonacci와 max\_of\_four\_int user program를 위한 작업 번호와 system call 함수를 추가했다.

부모 process는 자기 자신이 fork하거나 exec한 자식 process가 죽을 때까지 기다리고 난 후 지워주는 synchronization을 구현했다. 자식이 현재 실행 중인지와 자식이 점유하는 메모리가 사용 가능한지를 semaphore로 관리하고, 처음에 thread를 초기화할 때 semaphore와 구조체 list를 초기화하는 작업을 추가한다. 그리고 자식 process가 exit 할 때는 자기 자식의 실행이 죽었음을 알리기 위해 실행을 위한 semaphore는 1로, 자식 process가 사용 중인 메모리를 남겨두기 위해 메모리를 위한 semaphore를 0으로 설정한다.

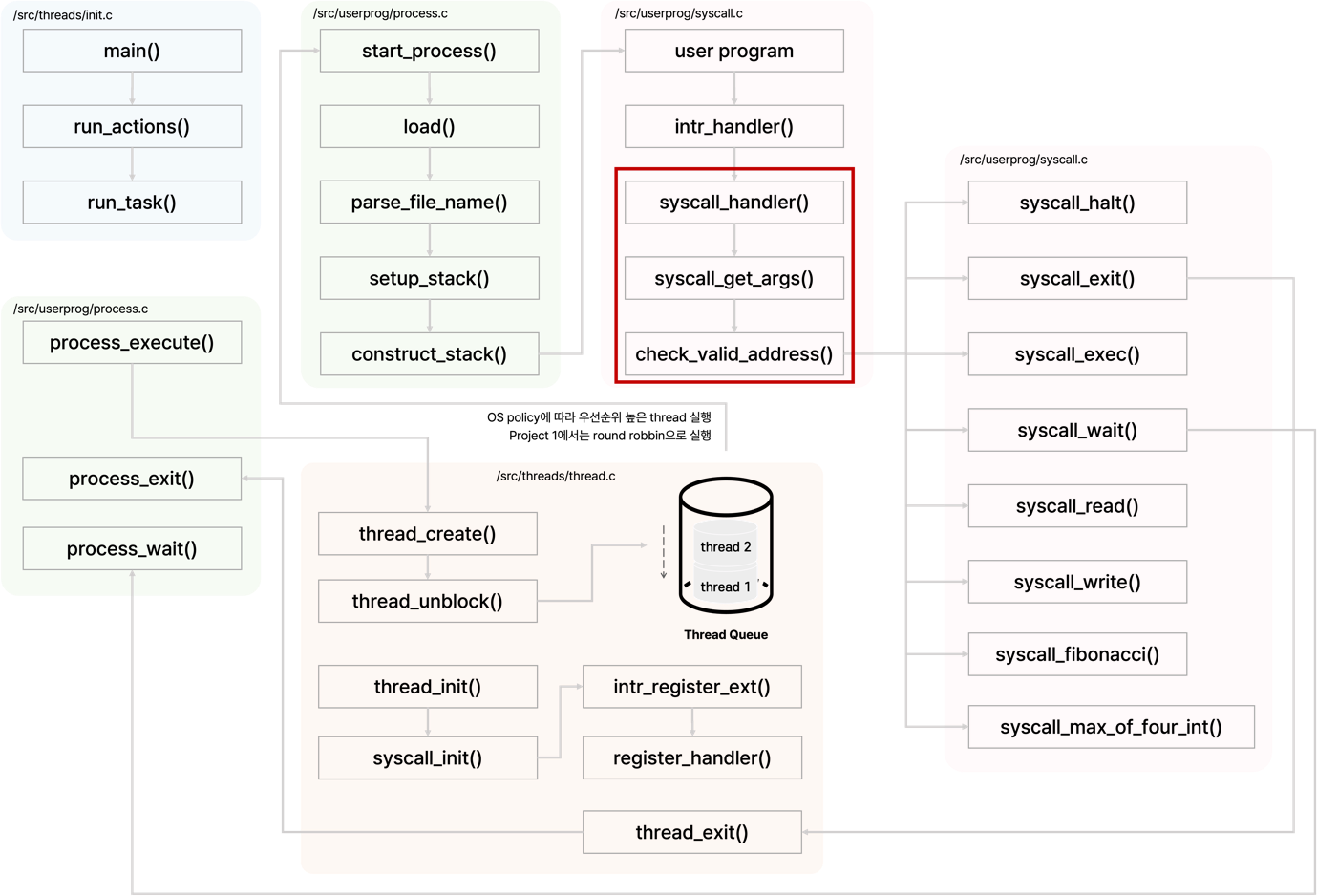
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

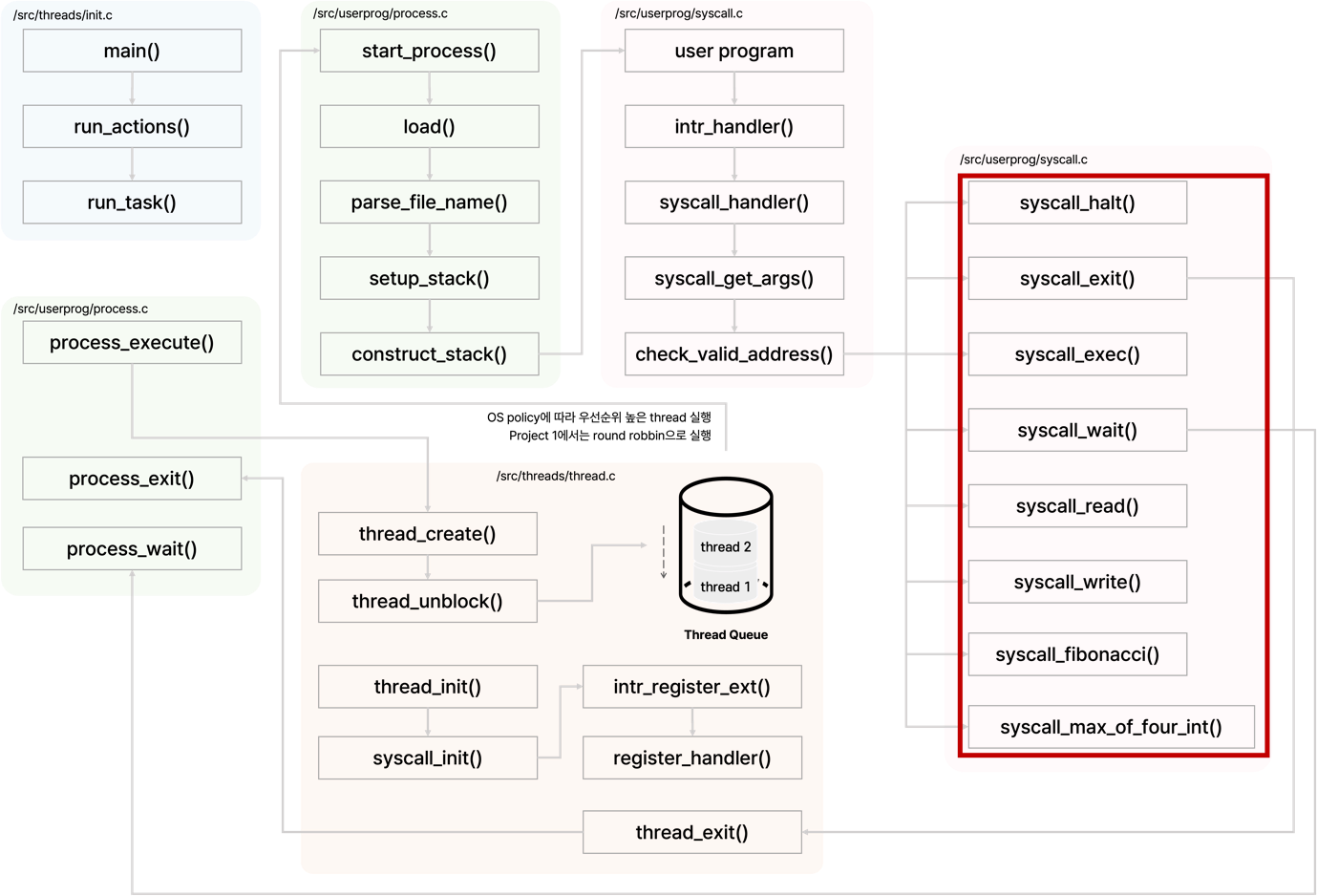
1. Argument Passing



1. User Memory Access



1. System Calls



* 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

함수의 argument로 넘어 온 file\_name을 먼저 file\_name\_to\_parse에 strlcpy로 복사하고, 구현한 parse\_file\_name를 호출한다. 이 함수 내에서 strtok\_r 내장함수를 가지고 space를 기준으로 구분하고, 포인터로 인자로 넘긴 구조체 parsed\_args의 멤버변수인 argc에 tokenizing된 인자의 개수를 저장하고, argv 배열에는 나눈 인자를 원소에 하나씩 저장한다. argv 첫 번째 원소에는 명령어 자체가 담기게 되는데, 이를 가지고 filesys\_open 함수의 인자로 넘긴다.

setup\_stack 함수 실행 다음에 construct\_stack 함수를 호출하여 앞서 parsing한 인자를 담은 구조체 변수인 parsed\_args를 넘긴다. 해당 함수에서는 ESP 포인터 변수를 조절해 가면서 인자를 저장하는데, 인자 값들의 합이 WORD인 4 바이트 단위로 떨어지는지 확인하고, 떨어지지 않으면 4 바이트 단위로 떨어지도록 memset으로 Null을 채운다. 그리고 parsed\_args의 멤버 변수인 argv 배열의 원소를 하나씩 복사할 때 memcpy를 사용했다.

1. User Memory Access

syscall.c 내의 check\_valid\_address 함수에서 vaddr.h에서 import한 is\_user\_vaddr 함수를 이용하여 해당 메모리 주소가 유효한지를 확인하고, 마찬가지로 exception.c의 page\_fault 함수에서도 is\_kernel\_vaddr 함수를 사용한다. Stack으로 넘어 온 인자들이 유효한 메모리 주소인지를 확인하기 위해 addr 포인터 배열에 인자를 담아서 check\_valid\_address 함수를 for loop으로 돌면서 배열의 각 원소에 관해 유효한 메모리인지 확인한다.

process\_execute 함수 내에서 file\_name을 지니는 파일이 이미 열려 있는지 filesys\_open 함수를 통해 확인한다.

1. System Calls

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

syscall\_handler 함수에서 앞서 system call에서 쌓았던 stack의 정보를 interrupt stack frame 구조체 데이터 인자로 받아와서 구조체의 멤버 변수인 esp 포인터 변수의 첫 번째 값을 system call number로 읽어서 switch 문으로 적절한 system call 함수를 실행하도록 분기 처리한다. 그전에 stack에 쌓은 인자를 syscall\_get\_args 함수를 통해 받아오는데, 각 system call 함수마다 필요로 하는 인자 개수를 미리 static한 syscall\_num\_args 배열로 저장해서 관리한다. 필요한 인자 개수만큼 esp 포인터 변수를 4 바이트씩 조절해 가며 해당 인자가 담긴 메모리에 접근이 가능한지 확인한다.  
System call handler 각각의 구현 내용을 정리하면 다음과 같다.

* + syscall\_halt: shutdown\_power\_off 함수를 호출하여 pintos를 중지한다.
  + syscall\_exit: 현재 실행 중인 thread의 정보를 thread\_current()로 받아와서 thread 이름이 담긴 name 변수 값을 가져와 status를 같이 출력하고 thread의 eixt\_status 값을 status 값으로 변경한다.
  + syscall\_exec: process\_execute를 호출하고 그 결과를 반환한다. process\_exec 함수에서 명령어 문장인 file\_name에서 명령어 부분만 가져오기 위해 get\_command\_from\_file\_name 함수를 통해 명령어 문자열을 복사하여 strtok\_r 내장함수를 사용해서 명령어 부분만 분리한다. 분리한 명령어 부분만으로 filesys\_open 함수를 통해 이미 실행 중인 파일인지를 확인하고, 실행 중이지 않은 파일이면 해당 명령어 이름으로 thread\_create를 통해 thread를 생성한다.
  + syscall\_wait: process\_wait 함수를 통해 실행이 완료될 때까지 기다리고자 하는 child process id를 갖는 자식 process를 찾는데, 이를 위해 현재 thread의 정보를 thread\_current 함수로 가져오고, child process list를 가리키는 곳의 리스트 처음부터 끝까지 탐색하면서 해당 process id를 찾으면 child process를 정상적으로 종료하기 위해 child process를 기다렸다가 종료된 이후에 sema\_down으로 fork\_lock을 0으로 설정하고, exit\_status를 읽어오며, child process list에서 삭제하고, 다시 child가 점유했던 메모리 semaphore 값인 free lock을 sema\_up을 통해 1로 설정한다.
  + syscall\_read: file descriptor인 fd 값이 0일 때 stdin을 구현해야 하므로 devices의 input.c의 input\_getc 함수를 가져와서 buffer에 할당한다. length 길이만큼 다 할당하거나 문장 종료 문자를 만나는 순간 break 하여 종료한다.
  + syscall\_write: file descriptor인 fd 값이 1일 때 stdout을 구현해야 하므로 lib의 kernel 폴더의 console.c에 정의된 putbuf 함수를 통해 buffer에 있는 데이터를 size만큼 standard output으로 출력한다.

1. Additional System calls

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

먼저 lib의 user 폴더의 syscall.c에서 새로운 user program에 관해 system call을 호출해서 interrupt를 발생시키는 코드를 추가했는데, max\_of\_four\_int는 명령어로부터 네 개의 인자를 받으므로 이를 stack에 넣기 위한 syscall4 매크로를 어셈블리 코드로 작성했다. fibonacci와 max\_of\_four\_int system call 각각에 관해 받은 인자를 syscall1과 syscall4의 매크로 함수로 넘겨서 실행한다. system call handler에서 요청 받은 system call을 처리하기 위해 syscall\_fibonacci와 syscall\_max\_of\_four\_int를 구현했다. syscall\_fibonacci에서는 세 개의 변수를 이용해 x번째 피보나치 항의 값을 구하고, syscall\_max\_of\_four\_int에서는 인자로 온 네 개의 변수를 하나씩 비교해 가면서 이중 가장 최댓값을 반환하도록 했다.

또한 examples의 additional로 fibonacci와 max\_of\_four\_int system call이 실행되도록 구현했는데, make 시 pintos에서도 제대로 additional 명령어를 사용할 수 있도록 makefile에 additional을 추가했다.

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

****