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

담당 교수 : 박 성 용 교수님

조 / 조원 : 20161595/배성현

개발 기간 : 2020/10/03 ~ 2020/10/31

1. **개발 목표**

* User Program을 구동할 수 있는 Pintos 환경을 구축한다. 이를 위하여 Process를 로드하고 이 때 filename과 같이 들어온 Argument를 Parameter로 passing하는 기능을 stack을 이용하여 구현한다. 또 halt, wait, exit, execute, read(stdin), write (stdout), fibonacci, max\_of\_four\_int의 system call 기능을 구현한다. 또한 User program이 잘못된 Memory 영역(Kernel address space, Unmapped virtual memory, Null pointer)에 접근하지 않도록 하는 기능을 구현한다.

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

Stack에 Argument들에 대한 정보가 저장되어 있게 된다. (stack의 top에서부터 return address, argument의 개수, 첫 argument의 주소를 담고 있는 stack 공간의 주소, argument들의 stack에서의 주소, word alignment, argument들이 저장되어 있음)

1. User Memory Access

User program이 잘못된 Memory 영역(Kernel address space, Unmapped virtual memory, Null pointer)에 접근하는 경우에, exit code -1과 함께 프로세스가 종료된다.

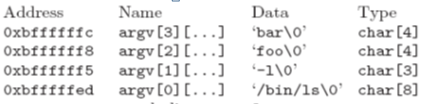
1. System Calls

유저 프로그램이 System call API를 통해 다양한 System Call을 요청하였을 때에 아래와 같이 해당 System call에 맞는 기능이 수행되게 된다.

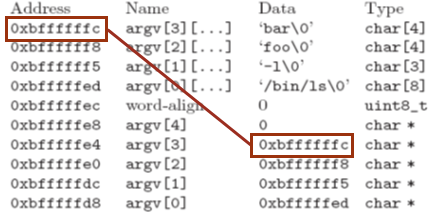
halt(pintos 종료), exit(현재 유저 프로그램을 종료하고 종료 status를 kernel로 반환), exec(새로운 프로세스를 실행), wait(특정 프로세스가 끝날 때까지 기다림), write(STDOUT으로 내용 출력), read(STDIN으로 입력 읽어 들임), fibonacci(피보나치 수 반환), max\_of\_four\_int(4개의 정수 중 가장 큰 수 반환)

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

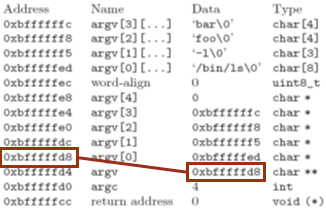
먼저 명령으로부터 argument들을 각각 parsing하여, 아래와 같이 argument들을 들어온 순서의 역순으로 stack에 push하게 된다.(ex: /bin/ls –l foo bar)

top(esp)

이후 word size로 align될수 있도록 word alignment를 push 하고 argument들의 stack상의 주소들을 아래와 같이 다시 stack에 push하게 된다.

top(esp)

다음으로 아래와 같이 첫 argument(argv[0][…])의 주소를 담고 있는 stack 공간의 주소를 stack에 push하고, 이어서 argument의 개수, return address 0을 각각 stack에 push하여 주게 된다.

 top(esp)

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

Pintos 상에서의 invalid memory access란 User program이 Null pointer나, Unmapped virtual memory, Kernel’s Virtual Address space와 같이 접근해서는 안 되는memory를 접근하는 pointer를 passing하여 이와 같은 메모리 공간에 접근하는 것을 의미한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

Invalid memory access를 막기 위해서는 먼저 User program이 passing한 pointer의 유효성을 체크하거나, 단순히 user pointer가 PHYS\_BASE 밑을 가리키는지를 체크하게 된다. 이 때 pagedir\_get\_page(), is\_user\_vaddr(), is\_kernel\_vaddr()등의 function을 사용할 수 있고, 만약 user pointer가 invalid하게 되면 이와 같은 경우에 exit code -1과 함께 프로그램을 종료 시켜주어 Invalid memory access를 막을 수 있다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

유저 레벨의 프로세스는 커널 영역에 직접적으로 접근이 불가하다. 이는 메모리나 디스크와 커널이 가지고 있는 자료구조에 대한 접근이 불가능하다는 것을 의미한다. 따라서 유저 레벨의 프로세스가 커널이 할 수 있는 수행 작업을 요구하는 경우에는 System call과 같은 커널과의 communication channel이 필요하다.

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

Halt: halt system call이 들어오게 되면 shutdown\_power\_off()함수에 의해 Pintos를 종료하게 된다.

Exit: exit system call이 들어오게 되면 current user program을 종료하게 되고, 종료 상태인 exit status를 커널로 반환하게 된다.

Exec: exec system call이 들어오게 되면 file 이름과 argument들을 통해 새로운 process를 만들어 수행하게 된다.

Wait: wait system call이 들어오게 되면 자식 프로세스가 끝날 때까지 부모 프로세스가 기다리게 된다.

Read: read system call이 들어오게 되면 STDIN으로부터 입력을 받아 버퍼에 저장 후, 저장한 바이트 수를 반환하게 된다.

Write: write system call이 들어오게 되면 버퍼에 있는 내용을 size만큼 STDOUT으로 출력하게 되고 성공 시 size를 반환하고 실패 시 -1을 반환한다.

Fibonacci: fibonacci system call 이 들어오게 되면 argument로 들어온 수에 대한 피보나치 수를 반환하게 된다.

Max\_of\_four\_int: max\_of\_four\_int system call이 들어오게 되면 argument로 들어온 4개의 정수 중에서 가장 큰 수를 반환하게 된다.

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

유저 레벨에서 시스템 콜 API를 호출하게 되면, system call number와, additional argument들이 caller의 stack에 쌓이게 되고, int $0x30 명령을 사용하여 interrupt를 발생시켜 유저 레벨에서 커널 레벨로 넘어가게 된다. 이후 intr\_handler에서 interrupt 번호를 가지고 indexing 하여 system call의 handler를 호출하게 되고, syscall\_handler()가 control을 받아 intr\_frame의 esp멤버를 통하여 stack에 접근하여 system call number와 argument들을 가지고 call number에 해당하는 루틴을 호출하여 루틴을 수행하게 된다. 이후 return값을 받아 intr\_frame의 eax에 담아 유저레벨로 돌아오게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

|  |  |
| --- | --- |
| 일정 | 내용 |
| 10/3~10/5 | 숙제 강의 자료 및 강의 확인 |
| 10/6~10/7 | 관련 Pintos Manual 확인 |
| 10/8~10/13 | Argument Passing 구현 |
| 10/14~10/15 | User Memory Access 구현 |
| 10/15~10/24 | System Calls 구현 |
| 10/25~10/27 | Additional Implementation 구현 |
| 10/28~10/31 | Document 작성 |

* 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

Argument Passing을 구현하기 위해 src/userprog/process.c의 load함수를 수정할 것이다. load함수의 parameter로 넘어오는 file\_name은 각각의 argument들로 parsing되지 않은 상태이기 때문에 먼저 이를 parsing하여 각각의 argument들로 분리하여 저장하는 요소를 추가하여 줄 것이다. 이후 load함수 내에서 stack에 대한 setup이 끝난 이후에 앞에서 parsing한 각각의 argument들을 esp를 이용하여 위의 II.B의 Argument Passing에서 설명한 내용과 같은 순서로 stack에 push하여주는 코드를 추가할 것이다.

* User Memory Access

User Memory Access를 구현하기 위하여 src/userprog/syscall.c에 pointer에 대하여 체크하는 함수를 추가하고, memory에 접근이 필요할 때마다 이를 사용하여 체크할 수 있도록 하여 invalid한 경우 적절하게 핸들링할 수 있도록 할 것이다. 또한 page fault가 발생했을 때 이러한 경우를 적절하게 핸들링하기 위한 코드를 src/userprog/exception.c의 page\_fault()에 추가할 것이다.

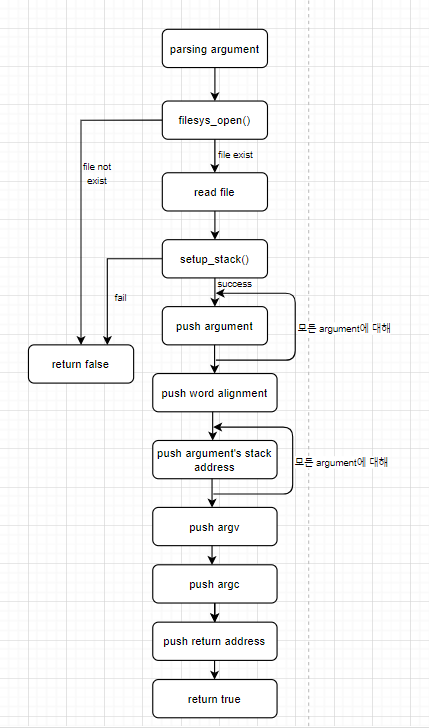
* System Calls

추가 구현 System call의 경우 먼저 src/lib/syscall-nr.h에 System call#을 추가해주고, src/lib/user/syscall.h에 추가 구현 system call에 대한 prototype을 정의해 줄 것이다. max\_of\_four\_int의 경우 인자가 4개이기 때문에 src/lib/user/syscall.c에 인자를 4개 받을 수 있는 syscall4()함수를 작성하여 인자들을 넘길 수 있도록 할 것이다. 이어 모든 System Calls를 구현하기 위하여 src/userprog/syscall.c 안에 각각의 system call들의 작동에 대한 함수를 추가하고 syscall\_handler함수를 수정하여 각각의 system call number에 따라 해당 함수를 호출하여 작동할 수 있도록 하는 코드를 추가할 것이다. 또한 wait의 경우에는 child process의 종료여부와 종료 상태를 알 수 있어야 하기 때문에 각각의 thread들이 자기 자신의 exit status(종료 상태)와 exit flag(종료되었는지의 여부), childList(자식들의 list)를 저장할 수 있도록 threads/thread.h의 thread구조체에 이를 저장할 수 있는 요소들을 추가하여 줄 것이다.

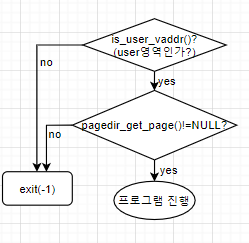
1. **연구 결과**
   1. **Flow Chart**

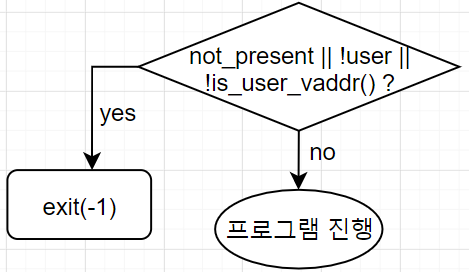
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

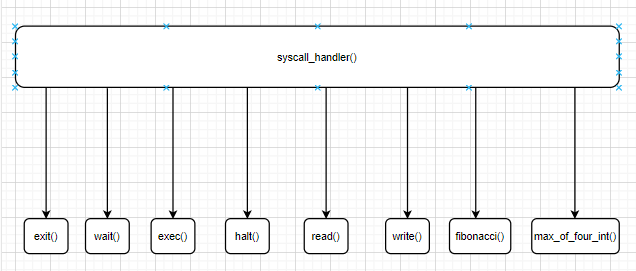


1. User Memory Access

 checkPointer()의 flow chart

 page\_fault()의 flow chart

1. System Calls

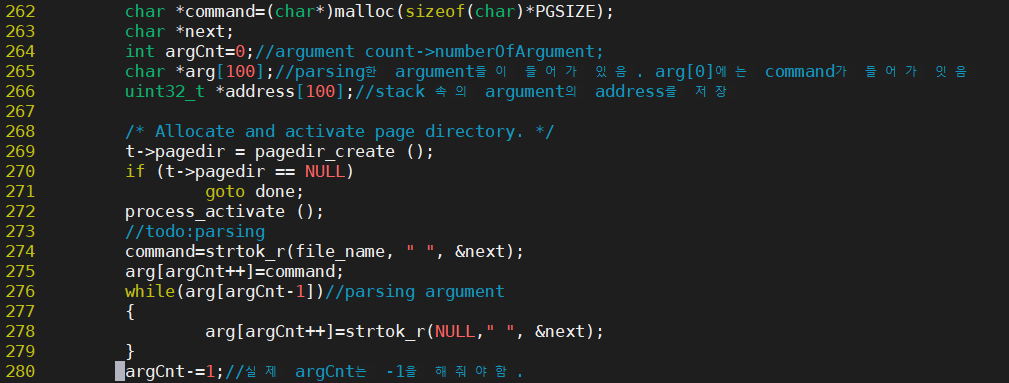


|  |  |
| --- | --- |
| exit() | wait() |
| exec() | halt() |
| read() | write() |
| fibonacci() | max\_of\_four\_int() |

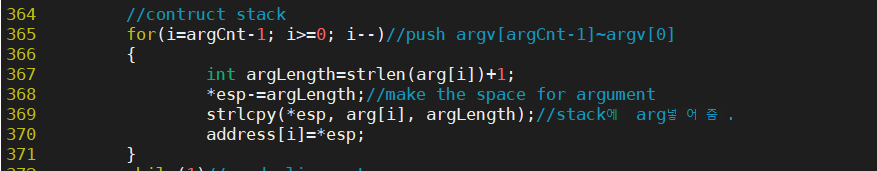
* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

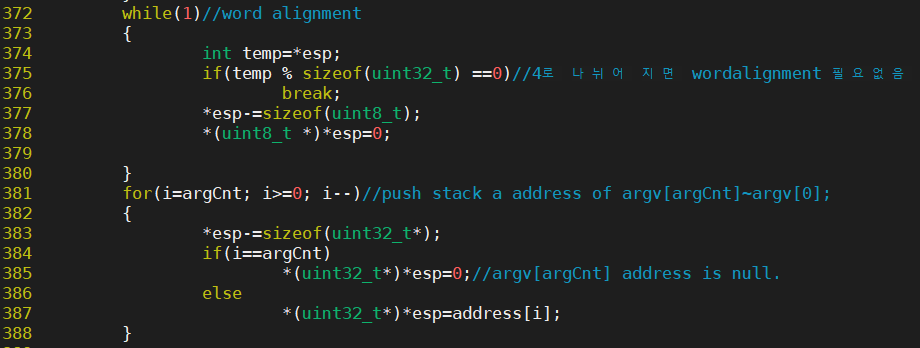
(src/userprog/process.c) load함수의 일부



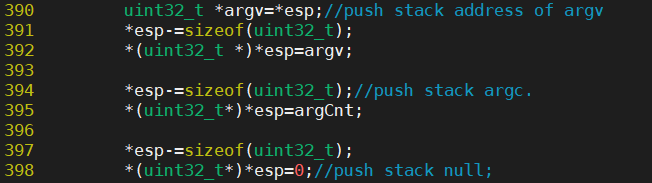
먼저 load함수로 들어온 file\_name을 각각의 argument들로 parsing하기 위해서 위와 같이 문자열을 delimiter단위로 parsing하여주는 strtok\_r()함수를 이용하였고, 각각의 parsing된 argument들을 배열arg의 각 원소들이 포인팅하도록 하였다.



이후 stack에 대한 set\_up이 끝난 이후에 위와 같이 stack 포인터 \*esp를 이용하여 먼저 argument의 길이+1만큼(argument+‘\0’) 공간을 만들고 argument들을 들어온 순서의 역순으로 stack에 복사하여 push하게 된다.



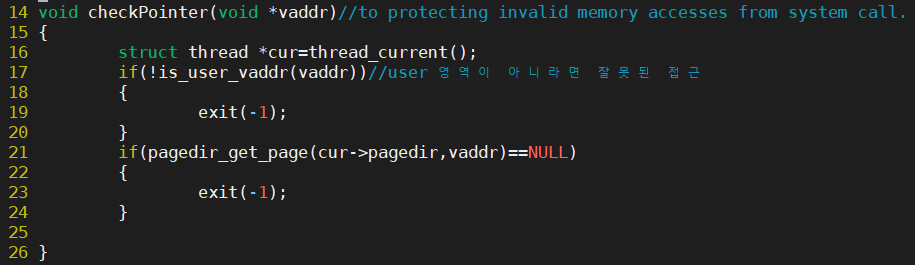
이후 위와 같이 word align이 되어있는지 여부를 확인하였는데 \*esp가 4로 나누어 진다면 이는 word align이 이미 되어있는 것이기 때문에 word align을 해줄 필요가 없고, 4로 나누어 지지 않는다면 word align이 필요하다는 것을 의미하기 때문에 esp를 1씩 감소하면서 word alignment를 넣어주었고, word align 후에 argument들의 stack상의 주소들을 stack에 push하여 주었다.



다음으로 위와 같이 첫 argument(argv[0][…])의 스택 상의 주소를 담고 있는 stack 공간의 주소를 stack에 push하여 주었고, 이어서 argument의 개수를 나타내는 argCnt를, 이어서 return address 0을 각각 stack에 push하여 주어 Argument passing을 구현하였다.

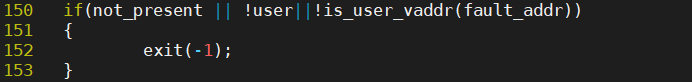
1. User Memory Access

(src/userprog/syscall.c) checkPointer 함수.



잘못된 메모리 접근을 막기 위해서 위와 같이 is\_user\_vaddr() 함수를 이용하여 user pointer가 user영역이 아닌 곳에 접근하였는지를 체크하였고, 또 pagedir\_get\_page() 함수를 이용하여 unmapped virtual memory를 체크하였다. memory에 대한 접근이 필요할 때마다 이 함수를 호출하여 잘못된 접근이면 exit(-1) system call을 통하여 프로세스를 끝내도록 하였다.

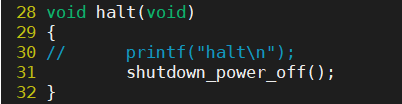
(src/userprog/exception.c) page\_fault()함수의 일부



또한 pointer가 invalid한 memory영역에 접근을 시도하면 page fault가 발생하기 때문에 이와 같은 경우에 exit(-1) system call을 통하여 프로세스를 끝내도록 하였다.

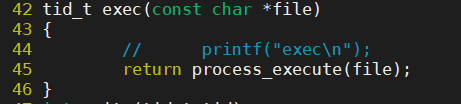
1. System Calls

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

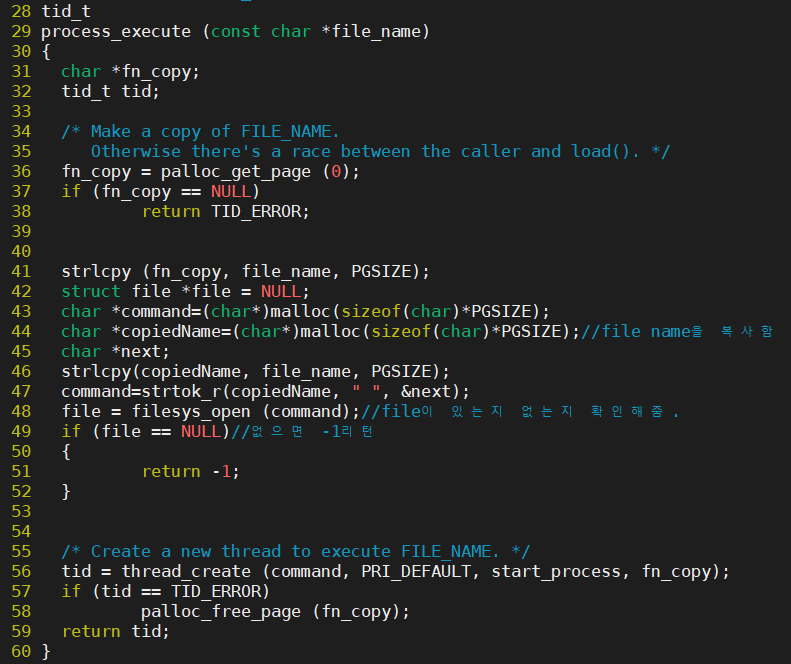


src/userprog/syscall.c에 핀토스를 종료시키는 함수인 shutdown\_power\_off()함수를 이용하여 halt함수를 구현하였고 이 halt함수를 syscall\_handler에서 호출하여 syscall halt가 들어왔을 때 핀토스를 종료시켜줄 수 있도록 하였다.

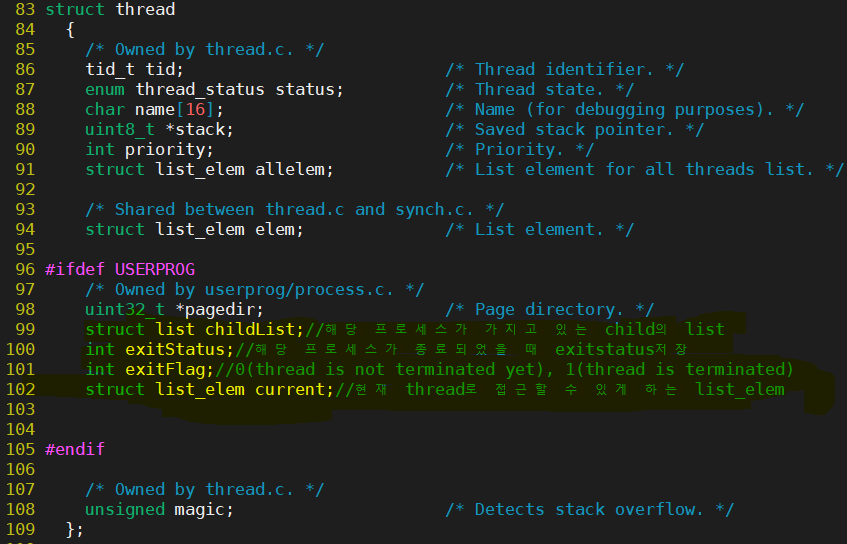
* Exec



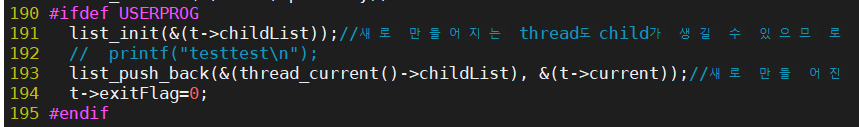
src/userprog/syscall.c에 위와 같이 파일로부터 새로운 프로세스를 실행할 수 있도록 하는 process\_execute()함수를 이용하여 exec()함수를 구현하였고, 이 exec함수를 syscall\_handler에서 호출하여 syscall exec이 들어왔을 때 새로운 프로세스를 실행할 수 있도록 하였다.



src/userprog/process.c의 process\_execute함수에서는 위와 같이 strtok\_r()을 이용하여 실제 파일 이름만을 추출해낸 다음, 실제로 있는 파일인지 없는 파일인지 체크하여 없다면 -1을 반환하도록 하였다(없는 프로그램이면 load하여 실행할 수 없으므로). 이후 파일이 존재한다면 이를 로드 하여 실행하는 thread를 만들게 되고, 해당 thread의 id를 반환하게 된다.

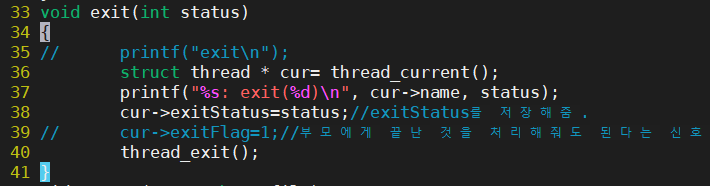


또한 모든 thread는 자식 thread를 가질 수 있으므로 위와 같이 src/threads/thread.h에 정의되어있는 thread 구조체에 list childList를 추가하여 주어 나중에 자식이 생기게 되면 자식thread element를 list에 추가할 수 있도록 하였다. 그 외에도 종료 상태를 저장하는 exitStatus와, 종료여부를 판단하여주는 exitFlag를 추가하여 주었고, list\_entry() 매크로를 통하여 해당 thread에 접근할 수 있도록 하는 list\_elem current를 추가하여 주었다.

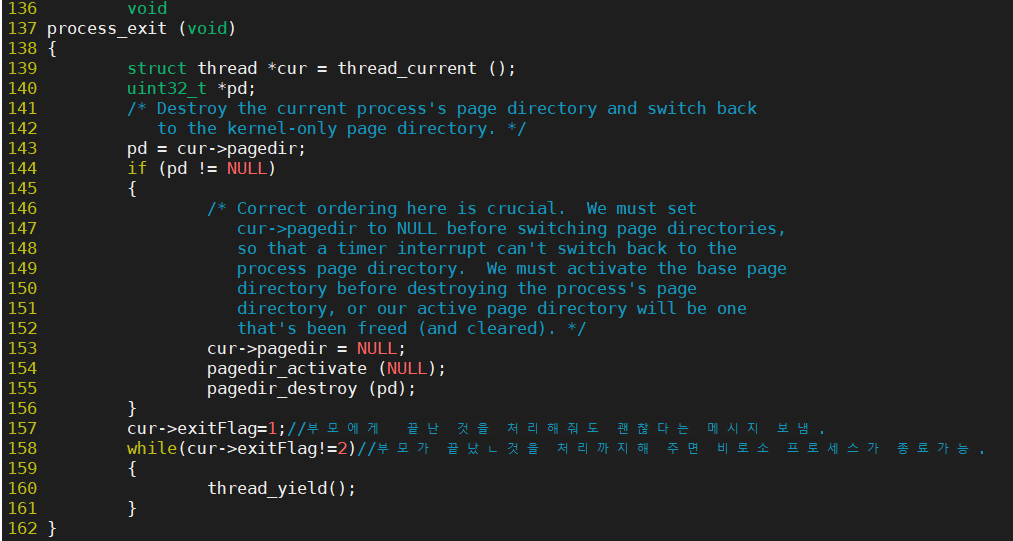


마지막으로 src/userprog/process.c의 process\_execute함수는 src/threads/thread.c의 thread\_create()함수를 통하여 thread를 만들어 새로운 프로그램을 실행하게 되는데, 이렇게 새로 생긴 thread역시 자식을 가질 수 있으므로 thread\_create()함수에 위에서 정의한 childList를 사용할 수 있도록 list\_init()을 하여 주었고, 새로 생긴 thread가 이를 만든 thread의 childList에 새로 연결 될 수 있도록 하였다. 또 새로 생긴 thread의 exitFlag를 0(종료되지 않음)으로 초기화 하여 주었다.

* Exit

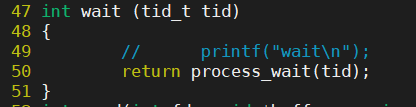


src/userprog/syscall.c에 위와 같이 thread\_current()함수로부터 현재 thread를 받아, 이의 exit에 관한 정보를 출력하도록 하였고, thread\_exit()함수를 통해 그 thread를 종료하는 exit()함수를 구현하였다. 이 exit함수를 syscall\_handler에서 호출하여 syscall exit이 들어왔을 때 현재 프로세스를 종료할 수 있도록 하였다.

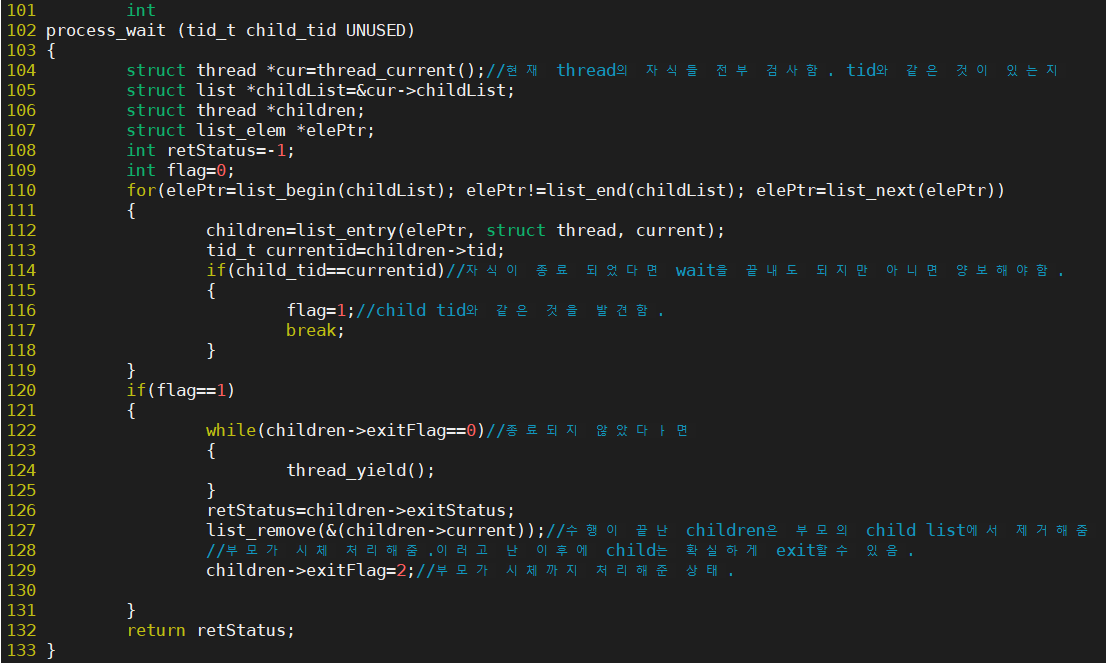


또한 thread\_exit()함수 내에서 src/userprog/process.c의 process\_exit()을 호출하게 되는데, 이 process\_exit()에서 종료될 thread(cur)의 exitFlag를 1로 바꾸어주어 이 프로세스를 wait하고 있는 부모프로세스에게 wait을 끝내도 된다는 신호를 보내주도록 하였고, 부모가 이 thread가 끝나는 것을 알고 이 thread를 childList에서 제거하였다는 flag인 exitFlag=2의 상태가 될 때까지 thread\_yield()를 통해 다른 thread에게 제어를 양보하도록 구현하였다.

* Wait

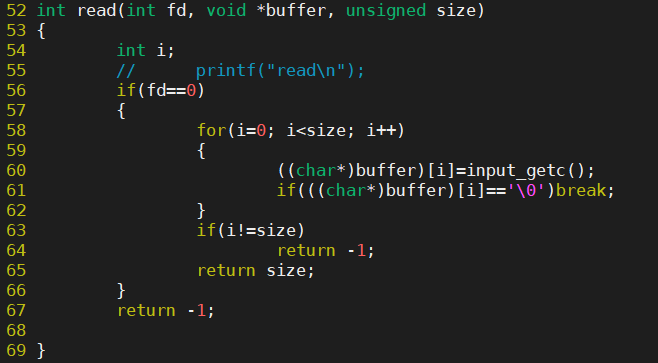


src/userprog/syscall.c에 위와 같이 process\_wait()함수를 이용하여 wait()함수를 구현하였고, 이 wait함수를 syscall\_handler에서 호출하여 syscall wait이 들어왔을 때 자식 프로세스(위의 파라미터 tid는 부모가 기다려야 하는 자식의 id)가 끝날 때까지 부모 프로세스가 기다리도록 하였다.



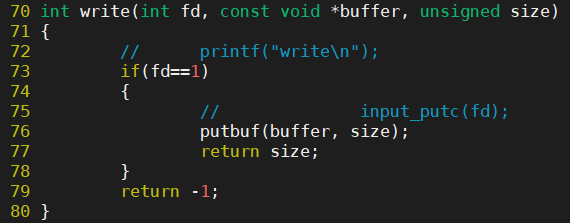
src/userprog/syscall.c의 wait()함수에서 src/userprog/process.c의 process\_wait()을 호출하여 처리하기 때문에 process\_wait()에 실제 wait에 관한 내용을 구현 하였다. process\_wait()에서는 부모프로세스의 childList를 순회하여 현재 부모 프로세스가 기다리고 있는 자식프로세스를 찾도록 하였는데, 만약 해당 자식 프로세스를 부모의 childList에서 찾지 못했다면 곧바로 -1을 반환함으로써 유효하지 않은 id인 경우와 같은 예외 케이스를 처리할 수 있도록 하였다. 그리고 만약 찾았다면 자식 프로세스의 exitFlag가 1이 될 때까지 즉 자식 프로세스가 종료되었다는 신호를 받을 때까지 thread\_yield()함수를 통해 다른 thread에게 제어를 양보하도록 구현하였다. 그리고 종료 되었다는 신호를 받으면 자식 프로세스의 종료상태를 retStatus에 받고, 자식 프로세스를 부모프로세스의 childList에서 제거한 뒤 exitFlag를 2로 바꾸어 부모가 처리하여 주기를 기다리던 자식프로세스가 완전하게 종료될 수 있도록 하였다. 그리고 마지막으로 자식 프로세스의 종료상태(retStatus)를 반환하여 주도록 하였다.

* Read



src/userprog/syscall.c에 read함수를 구현하였다. 이번 과제에서는 stdin에 대해서 고려하고 fd가 0일 때 이러한 stdin을 의미하기 때문에, 이 경우에 input\_getc()를 이용하여 size만큼 buffer로 읽어들일 수 있도록 하였다. 그리고 만약 size만큼 입력을 받기 전에 ‘\0가 들어오게 되면 이는 실패한 것이므로 -1을 반환하도록 하였고, read에 성공한 경우 size를 반환하도록 하였다. 이 read함수를 syscall\_handler에서 호출하여 syscall read가 들어왔을 때 stdin으로 입력을 받을 수 있도록 하였다.

* Write



src/userprog/syscall.c에 write함수를 구현하였다. 이번 과제에서는 stdout에 대해서 고려하고 fd가 1일 때 이러한 stdout을 의미하기 때문에, 이 경우에 putbuf를 이용하여 buffer의 내용을 size만큼 stdout으로 출력하고, size를 반환하여 주도록 하였다. 이 write함수를 syscall\_handler에서 호출하여 syscall write이 들어왔을 때 stdout으로 출력할 수 있도록 하였다.

* Syscall\_handler() 함수(src/userprog/syscall.c)

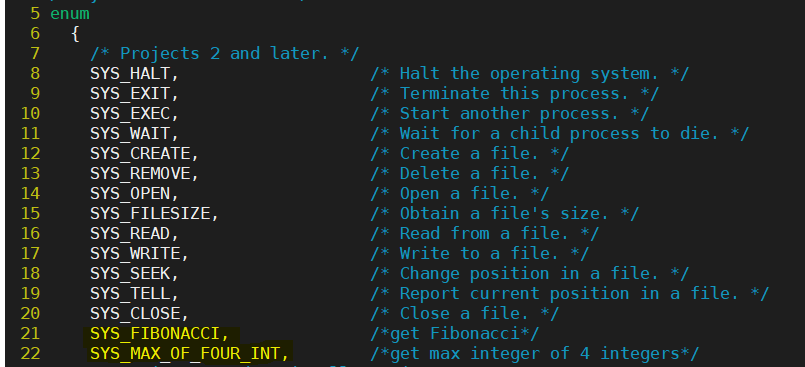




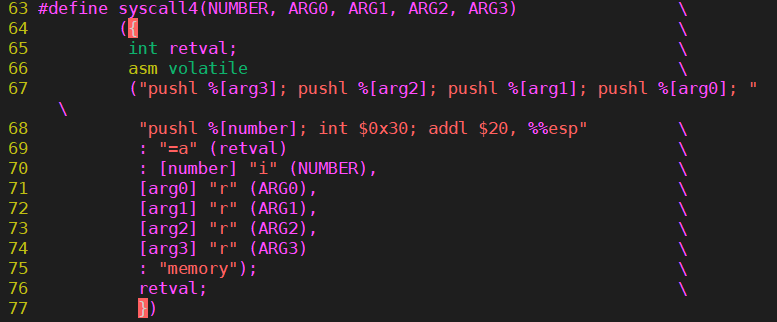
이어서 위와 같이 src/userprog/syscall.c의 syscall\_handler함수에서 f->esp에 system call number가 저장되어 있기 때문에 이를 접근하여 call number 변수에 저장하도록 하였다. 그리고 call Number를 통해 각각의 system call에 따라 먼저 stack에 저장된 인자들을 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자들에 접근하고 이를 변수로 저장하여 위에서 구현한 system call 함수들의 파라미터로 넘겨 각각의 System call에 해당하는 내용을 실행할 수 있도록 하였다. 그리고 return값이 있는 경우 이를 returnValue에 저장한 후 returnValue를 f->eax에 assign 하여 줌으로써 system call의 return값이 전달될 수 있도록 하였다.

1. Additional System calls

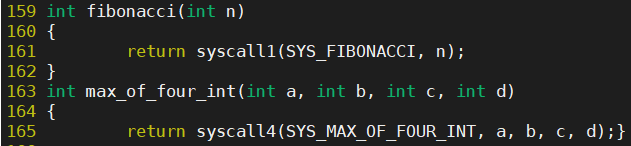
* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**



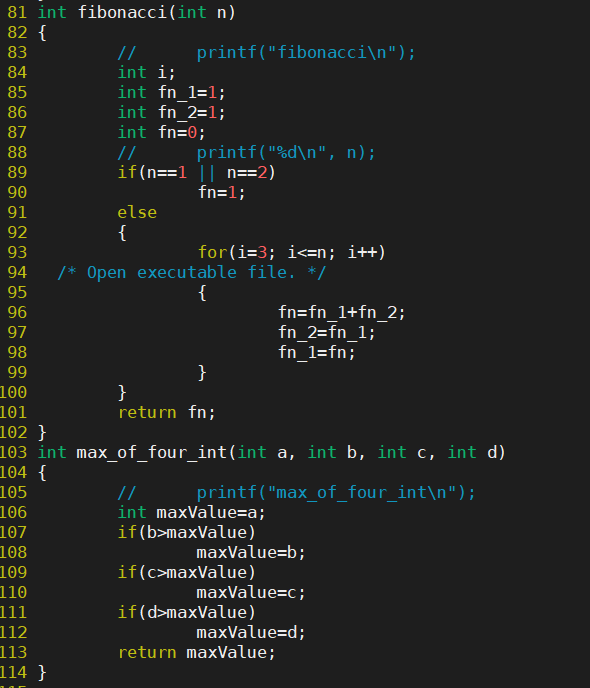
먼저 새로운 system call인 fibonacci와, max\_of\_four\_int에 대한 system call number가 필요하기 때문에 위와 같이 src/lib/syscall-nr.h에 SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INT를 추가하여 주었다.



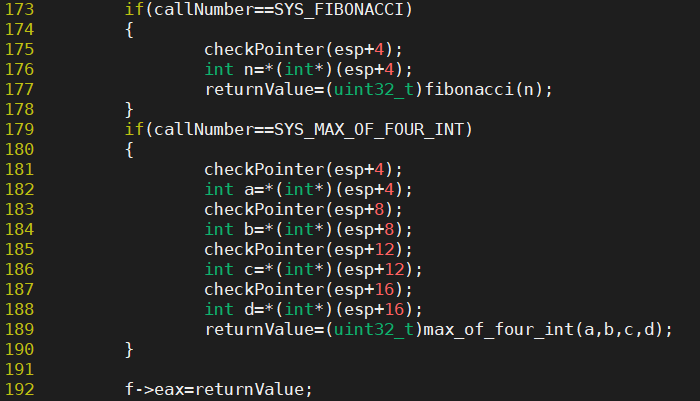
이후 max\_of\_four\_int의 경우 인자가 4개 전달되기 때문에 위와 같이 src/lib/user/syscall.c에 인자를 4개 받아 전달할 수 있는 syscall4()함수를 syscall3()함수를 보고 응용하여 작성하였다.



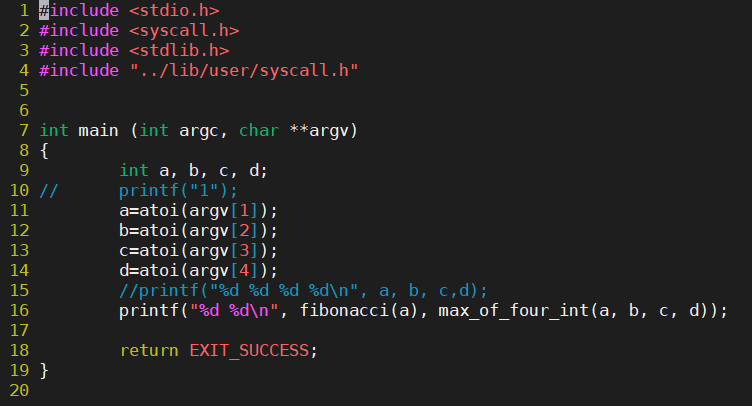
다음으로 위와 같이, src/lib/user/syscall.c에 fibonacci와 max\_of\_four\_int에 대한 system call을 추가하여 주었다.



다음으로 위와 같이 src/userprog/syscall.c에 시스템 콜 fibonacci(인자에 대한 Fibonacci수 반환)와 max\_of\_four\_int(4개의 인자 중 가장 큰 수 반환)의 동작에 대한 함수를 작성하여 주고 이 fibonacci함수와 max\_of\_four\_int 함수를 각각의 경우에 따라 syscall\_handler에서 호출하여 syscall fibonacci 또는 syscall max\_of\_four\_int가 들어왔을 때 각각의 동작을 수행할 수 있도록 하였다.



이어서 위와 같이 src/userprog/syscall.c의 syscall\_handler함수에 syscall number가 fibonacci일 때와 max\_of\_four\_int일 때에 따라 valid한 메모리영역인지를 체크한 후 인자를 stack에서 가져오고, 위에서 작성한 함수들의 파라미터로 넘겨 각각에 해당하는 내용을 실행할 수 있도록 하였다. 그리고 returnValue를 f->eax에 assign 하여 줌으로써 system call의 return값이 전달될 수 있도록 하였다.



(src/examples/additional.c 구현부)

additional.c에서는 먼저 인자들이 문자열로 들어오기 때문에 atoi를 사용하여 이를 정수로 바꾸어 주었고 또 이들을 파라미터로 하여 fibonacci()와 max\_of\_four\_int() 시스템 콜을 호출하여 주었다.

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

