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

담당 교수 : 김영재

조 / 조원 : 20201604 원종윤

개발 기간 : 23.09.18 - 23.10.08

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* 1. Argument passing

argv를 저장한 file\_name을 전달받아, 이를 분리하여 argc와 argv[]에 저장한다. 파일을 오픈하고, stack setup 이후 저장된 값을 pintos stack convention을 바탕으로 stack에 push한다. 올바르게 저장되었는지 hex\_dump를 통해 확인한다.

* 2. User memory access

Pintos stack convention에 위배되는 memory accessing을 제한하고, 적절한 오류 메시지를 출력하도록 한다. vaddr.h에서 제공하는 is\_kernel\_vaddr과 is\_user\_vaddr을 바탕으로 argument로 전달된 address가 kernel 영역을 침범하는지, user 영역을 침범하고 있는지 확인할 수 있다.

* 3. System Calls

기본 pintOS는 system call API들이 구현되어 있고, 각각의 상황에서 적절히 system call number에 따라 system call handler를 호출한다. 그러나, system call handler가 구현되어 있지 않아 호출 이후의 동작을 구현해야 한다. f->esp에 arguments와 system call number가 적혀 있는 것을 바탕으로, switch를 통해 분류하여 Halt, exit, exec, wait 등의 handler 동작을 작성한다.

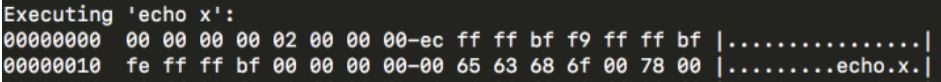
1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

file\_name을 분해하여 argv[]에 “ “단위로 각각 분리하고, 그 개수에 따라 argc를 count한다. 이 때, 다중 blank를 사전처리 하고, argc가 0인 경우 등을 flag를 바탕으로 분류하여 파일 처리를 원활히 한다. 분류 작업을 거치고 나면, argc와 argv[] 배열이 완성된다.

이후 파일을 오픈하고, stack을 setup한 후 argc와 argv[]를 차례로 stack에 삽입한다. 삽입 작업을 완료하면, 다음과 같은 형태의 stack이 완성된다.



1. User Memory Access

vaddr.h의 is\_user\_vaddr, is\_kernel\_vaddr을 바탕으로 safe\_addr 함수를 구현한다. 이를 통해 syscall에서 잘못된 영역의 f->esp를 참조하는 경우, exit(-1)을 호출하여 종료한다. 또한, exception.c에서 fault\_address를 처리하여 user가 아닌 영역 혹은 kernel 영역을 침범하는 경우 exit(-1)을 호출하도록 한다. (추후 B.2에서 서술)

1. System Calls

각각의 syscall에 따른 system handler가 호출되어 switch를 통해 적절히 분류된 후, 그에 맞는 함수를 호출하여 interrupt를 처리한다. 이 때, 2의 일환으로 f->esp와 + f->esp + 4\*n이 적절한 stack 영역을 가리키고 있는지 확인한다. Handler 함수 호출 종료 후, 필요하다면 그 결과를 f->eax에 저장한다.

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

argc에 argument 개수, argv에 argument가 pasring된 후 저장되어 있다. 변수 total\_len을 총 push length(stack 사용 byte라 생각하면 편리하다)를 관리하기 위해 선언한다. 이후 convention에 맞게 마지막 argument부터 stack에 저장한다.

마지막 argv의 내용부터 시작한다. 이 때, total\_len에 그 길이를 기록하고, \*esp를 len만큼 감소시킨다. 이를 통해 stack에 push할 빈 공간을 esp가 가리키도록 한다. 이후 strlcpy를 통해 arguments를 stack 공간에 복사하고, 주소를 가리키는 배열 addr에 해당 esp를 기록한다.

이후 convention에 따라 word-alignment를 준수하고자 기록된 total\_len을 4로 나눈다. 만약 나눠지지 않는다면, \*esp를 1 감소시키고, 그 공간에 0을 저장한다. 이를 나눠질 때까지 반복하면 word-alignment가 성립한다. (2.A.1 사진 참고)

convention에 따라 \*esp를 4 감소시키고, NULL에 해당하는 0을 저장한다.

이후 마지막 argument부터 stack에 저장된 주소를 담고 있는 addr 배열 값을 stack에 저장한다.

다시 \*esp를 4 감소시키고, \*esp + 4 (이전 loop 종료 직후 stack esp, argv[0]저장된 공간)를 stack에 저장한다.

\*esp를 4 감소시키고 argc를 저장한다.

마지막으로, \*esp를 4 감소시키고 return address에 해당하는 0을 저장한다.

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

User program이 stack에 삽입한 포인터가 NULL, mapped되지 않은 virtual memory 혹은 kernel 영역을 가리키고 있는 경우 pintOS kernel panic이 발생한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

Manual에 의하면 다음과 같이 두 가지 방법이 있다.

The first method is to verify the validity of a user-provided pointer, then dereference it. The second method is to check only that a user pointer points below PHYS\_BASE, then dereference it.

본 프로젝트는 첫 번째 방법을 선택하여 vaddr.h의 함수를 바탕으로 구현했다. Is\_user\_vaddr과 is\_kernel\_vaddr을 이용해 해당 stack 영역이 적절한 공간인지 확인하고, 이를 어겼다면 exit(-1)을 호출하도록 하였다. 또한 추가적으로, page\_fault error가 발생한 후 exception.c에서 프로그램 kernel panic이 발생하는 것을 확인하였다. 이는 test에서 의도적으로 발생시킨 오류 상황이므로, 해당 esp를 올바르게 수정하는 것은 불가능하다. 따라서 exception.c에서 해당 fault\_address가 user가 아니거나 kernel을 가리키고 있는지 확인한 후, exit(-1)을 호출하도록 하였다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

System call은 user level에서 kernel level 조작을 할 수 있도록 해주는 API이다. System call을 거치지 않고 kernel service를 조작할 수 없다. 만약 system call 없이 user mode와 kernel mode 사이에서 하드웨어를 조작할 경우, 보안 문제가 발생하고 stack 영역 침범과 같은 다양한 오류가 발생할 가능성이 높아진다.

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

Halt

deadlock과 같은 상황에서 데이터 손실이 발생할 수 있기에 자주 사용하지 않는다. pintOS를 종료시킨다. pintOS 내장 함수 shutdown\_power()를 호출하여 구현한다.

Exec

cmd\_line으로 전달받은 파일 이름에 맞는 새로운 파일을 실행한다. 파일을 실행하기 위해 새로운 thread를 생성한다. 그 결과로 thread의 tid를 전달받고 이를 반환한다.

Wait:

Wait을 위한 child\_pid를 전달받아 process\_wait을 호출한다. 이후 list, list\_elem 등의 자료구조를 활용하여 살아있는 자식인지 확인하고 semaphore를 통한 synchronize에 주의하여 제거 처리한다.

Write:

Stack에 저장된 fd, buf, size를 바탕으로 내장 함수 putbuf를 이용해 출력한다. 본 프로젝트는 STDOUT만 고려한다.

Read:

Stack에 저장된 fd, buf, size를 바탕으로 내장 함수 input\_getc를 이용해 처리한다. 본 프로젝트는 STDIN만 고려한다.

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

user level에서 system call API를 호출하면, argument 개수에 맞는 syscall을 호출하고 argument를 전달한다. 이후 syscall에 따라 user stack에 argument와 system call number를 push한다. 이후 interrupt를 위해 int$ 0x30을 설정하고, 이에 따라 interrupt handler를 실행한다. 이후 system call handler가 호출된다. Handler는 미리 정의된 내용에 따라 f->esp를 통해 case를 분류한 후 처리한 다. 처리 결과를 f->eax에 저장한다. 모두 종료된 후, 다시 돌아와 argument 개수에 맞게 addl $4\*x를 통해 push된 내용을 제거(엄밀히 제거되지는 않음)한 후 syscall을 종료한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

1. Argument passing

09.18 – 09.21

file\_name 분해와 stack constructing을 구현한다.

1. User memory access

09.22

vaddr.h 내장 함수를 이용하여 safe\_address 함수를 구현한다.

1. System call

09.23 – 10.03

halt, exit, write, read를 구현한다.

10.03 – 10.08

wait, exec를 구현한다.

* 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)**

1. **process.c**

**process\_execute()**

함수 내부에서, echo x와 같은 multiple blanks 입력이 존재하는지 확인한다. 또한 argc가 1, 즉 추가 argument가 존재하지 않는지 확인한다. 이후 그 결과에 따라 적절히 file\_name을 first\_file\_name에 복사 및 조작하여 filesys\_open에 전달하도록 수정했다.

**Process\_wait()**

추후 서술할 thread에 추가된 child thread 목록을 저장한 list\_child를 순회하며 전달받은 살아있는 child\_tid가 존재하는지 확인하고, synchroniazation을 위해 semaphore\_child을 down시킨 후, 반환 값에 상태를 저장하고 list에서 child를 제거시킨다. 이후 semaphore\_mem을 up시키고 종료한다. 이외의 경우, -1을 반환한다.

**Process\_exit()**

Synchronization을 위해 process\_wait()에 반대되게 semaphore(child\_lock)을 up시키고 semaphore(mem\_lock)을 down시킨다.

**Load()**

File\_name에 multiple blanks가 발생한 경우, word 사이마다 하나만 남기고 모두 제거한 후 이를 blk\_erased에 저장한다. 이를 통해 multiple blanks가 제거된 file\_name을 얻는다. 이후 argc를 count한다.

이에 따라 적절히 argv와 argc를 동적 할당한 후, argc가 2 이상이라면 strtok\_r 함수 특성인 original string 손상을 방지하고자 blk\_erased를 tmp2에 복사한다. 이를 strtok\_r를 통해 적절히 parsing하여 argv[]에 저장한다.

blk\_erased를 tmp에 복사하여 파일 이름에 해당하는 첫 단어를 first\_file\_name에추출한다.

만약 추가 argument가 존재하지 않는다면 first\_file\_name을 추출하고 이를 argv[0]에 저장한다. 이후 filesys\_open에 file\_name이 아닌 first\_file\_name을 전달한다.

setup stack까지 호출한 후, convention에 따라 마지막 argument부터 차례로 stack에 저장한다. Word\_alignment를 위한 변수 total\_len을 선언하고, 이후 push한 길이를 저장한다. argc-1부터 0까지, argv의 길이에 따라 \*esp를 감소시켜 공간을 만든 후 내용을를 저장한다. Addr에 각 argv가 저장된 공간의 주소 \*esp를 저장한다. 이후 word\_alignment를 위해 total\_len이 4로 나누어 떨어질 때까지 \*esp를 한 칸씩 감소시키고 0을 저장한다. 다음으로 \*esp를 4 감소시키고 NULL을 위한 0을 저장한다. 이후 argc-1부터 차례로 \*esp를 4씩 감소시키며 각 argv가 저장된 주소 addr[]을 저장한다. 다시 \*esp를 4 감소시키고 addr[0]이 저장된 주소를 가리키는 곳의 주소인 \*esp+4를 저장한다. \*esp를 4 감소시키고, argc를 저장하고 argv와 addr을 free시킨 후 마지막으로 \*esp를 4 감소시키고 return address 0를 저장한다.

1. **Exception.c**

user영역이 아니거나 kernel 영역을 가리키는 주소가 발생하면 esp를 16 더한 후 exit(-1)을 호출한다.

1. **Syscall.c (syscall 호출과 함수를 동시에 설명)**

**Safe\_addr()**

전달받은 주소가 vaddr.h 함수를 통해 커널 영역을 가리키거나 유저 영역이 아니라고 판단할 경우 exit(-1)을 호출한다.

**Halt()**

Shut\_down\_power\_off를 호출한다.

**Exit()**

f->esp + 4가 안전한 주소인지 확인한 후, 이를 전달하여 exit을 호출한다.

종료되는 스레드의 이름과 상태를 출력하고, 종료 상태를 저장한 후 스레드를 종료한다.

**Exec()**

f->esp + 4가 안전한 주소인지 확인한 후, 이를 전달하여 exec를 호출한다. Exec는 다시 이를 전달하여 process\_execute를 호출한다. 이후는 앞서 설명한 process\_execute와 같다. 마지막으로, 반환된 값을 f->eax에 저장한다.

**Wait()**

f->esp + 4가 안전한 주소인지 확인한 후, 이를 전달하여 wait을 호출한다.

Wait은 다시 이를 전달하여 process\_wait을 호출한다. 이후는 앞서 설명한 process\_wait과 같다. 마지막으로, 반환된 값을 f->eax에 저장한다.

**Write()**

f->esp + 4, 8, 12가 안전한 주소인지 확인한 후, 이를 전달하여 write을 호출한다. Fd가 STDOUT인지 확인하고 아니라면 -1을 반환한다. 맞는 경우, 내장함수 putbuf를 호출하여 출력하고 size를 반환한다. 이후 f->eax에 반환 값을 저장한다.

**Read()**

f->esp + 4, 8, 12가 안전한 주소인지 확인한 후, 이를 전달하여 read를 호출한다. Fd가 STDIN인지 확인하고 아니라면 -1을 반환한다. 맞는 경우, 내장함수 input\_getc를 호출하여 입력받고 입력 size를 반환한다. 이후 f->eax에 반환 값을 저장한다. 본 프로젝트의 평가 요소 21가지에 반영되지 않는 것으로 확인되어 syscall과 실제로 연결하진 않았다.

1. **Thread.c**

**init\_thread()**

thread 구조체에 새로 선언된 semaphore mutex를 위한 sema\_child과 sema\_mem을 initialize한다. 뿐만 아니라 list\_child를 새로운 list로 등록하고, list에 새로운 thread의 elem\_child을 push\_back한다. 이 때, make에서 오류를 방지하고자 ifdef를 사용한다.

1. **Thread.h**

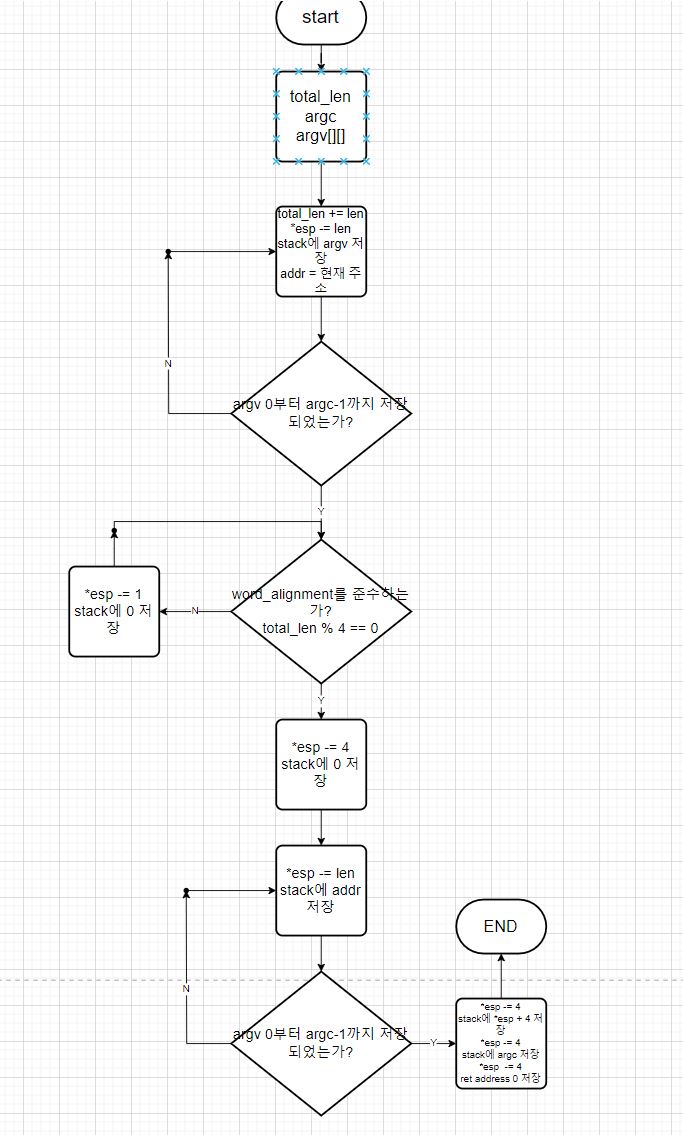
**Struct thread**

Process\_wait에서 child thread인지 확인하기 위해 child list를 새로 선언했다. 또한 list의 element로 사용할 child\_elem을 embed하였다. Semaphore mutex를 위한 sema\_child과 sema\_mem을 선언하고, 종료 정보를 전달하기 위해 exit\_status를 선언했다. 이 때, make에서 오류를 방지하고자 ifdef를 사용한다.

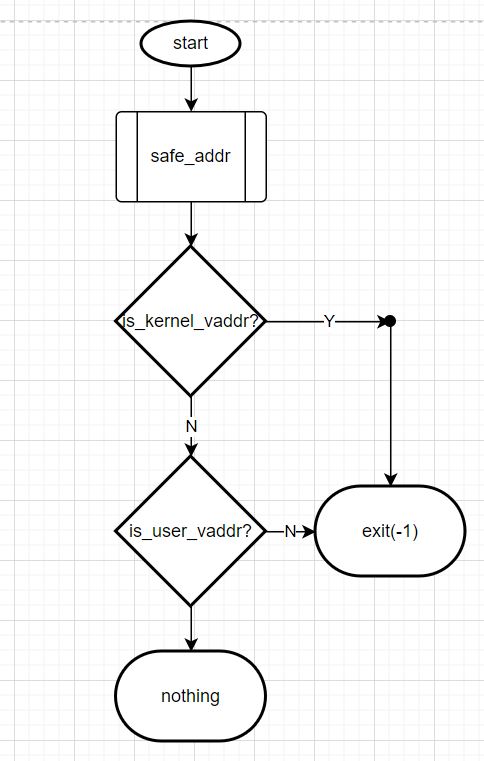
**연구 결과**

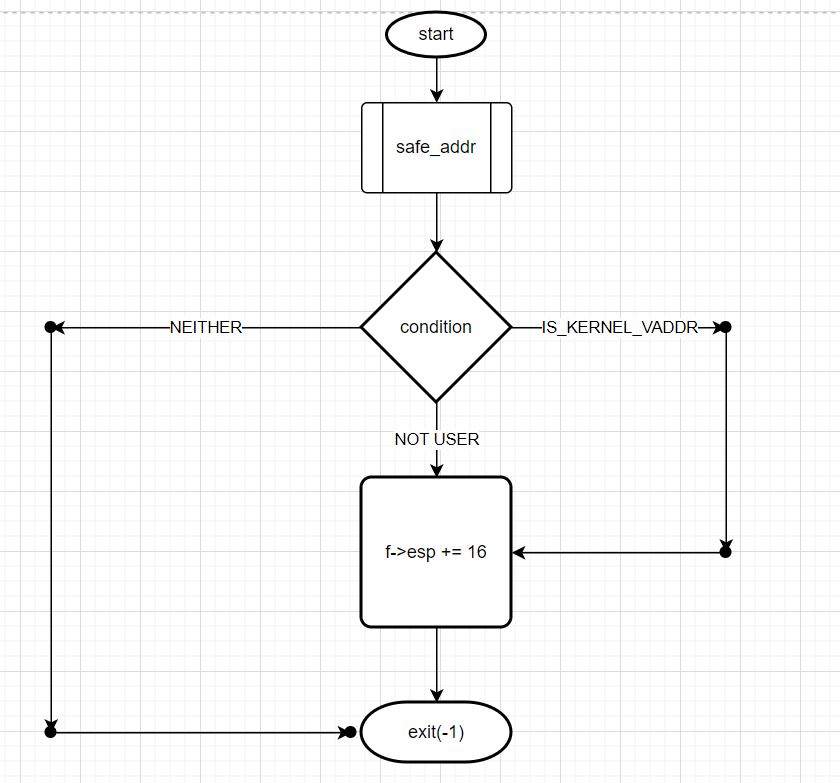
* 1. **Flow Chart**
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

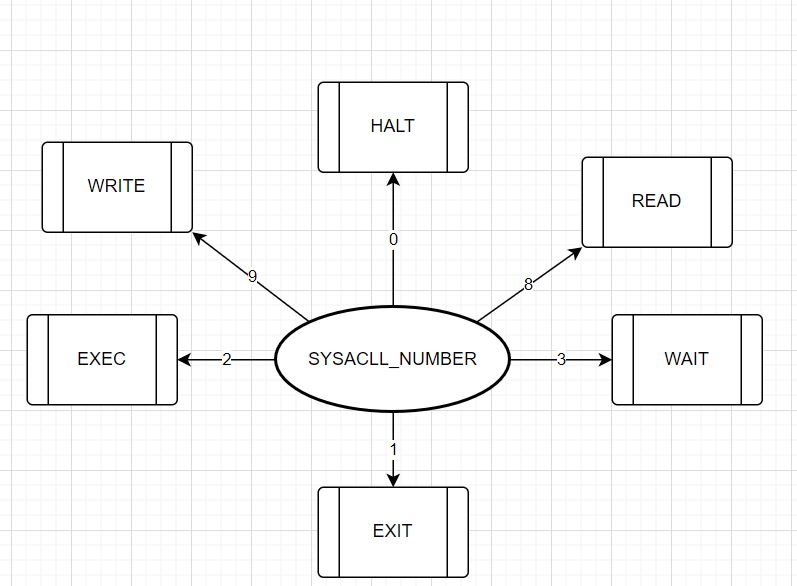


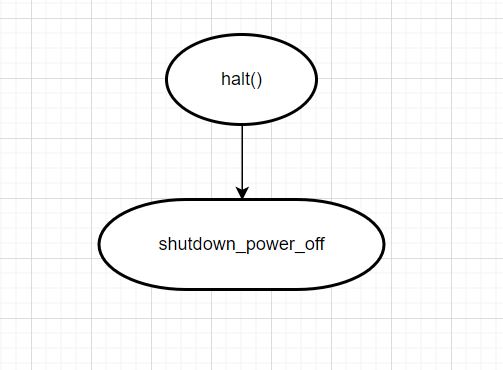
1. User Memory Access

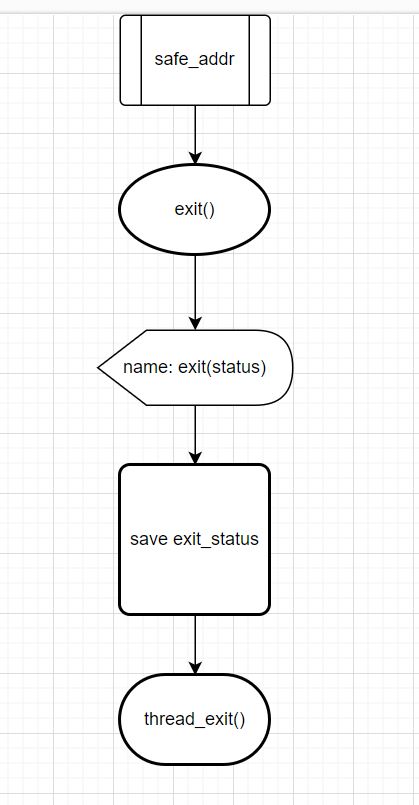


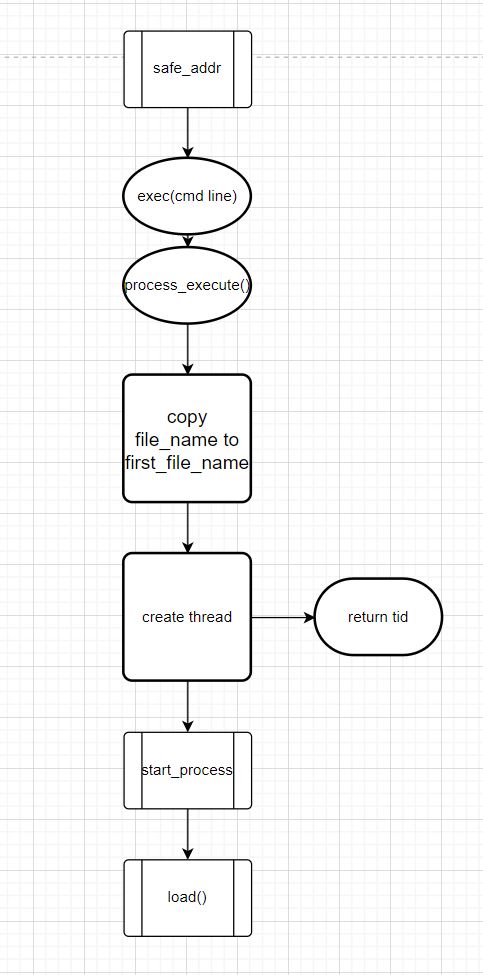


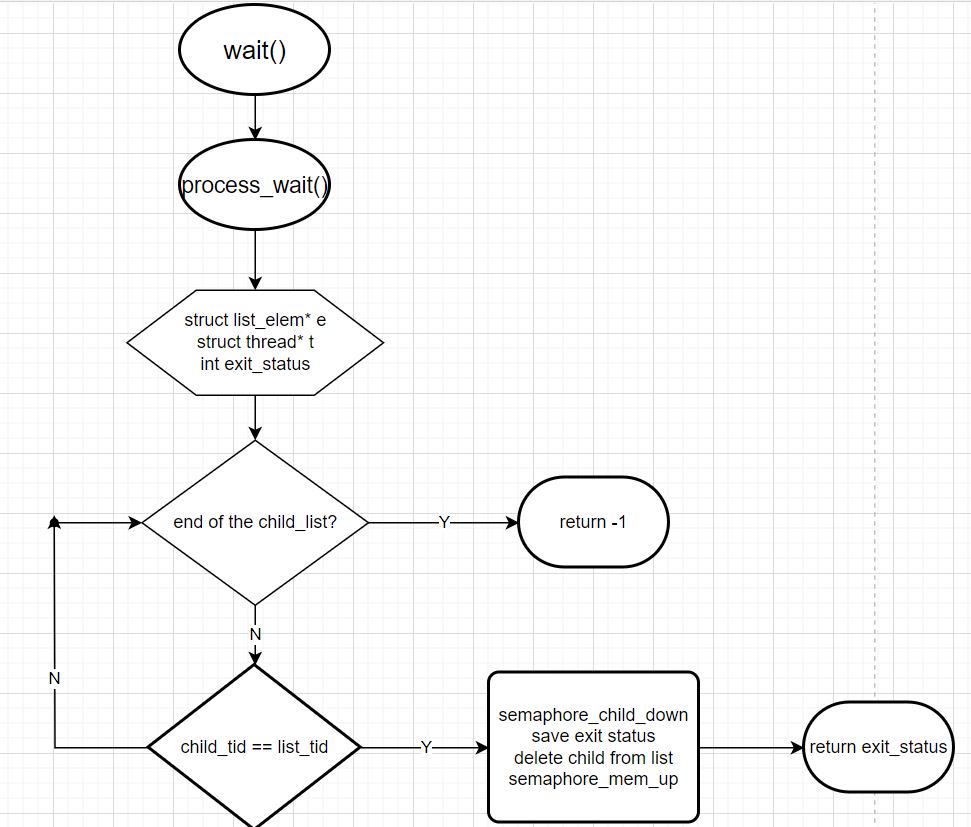
1. System Calls

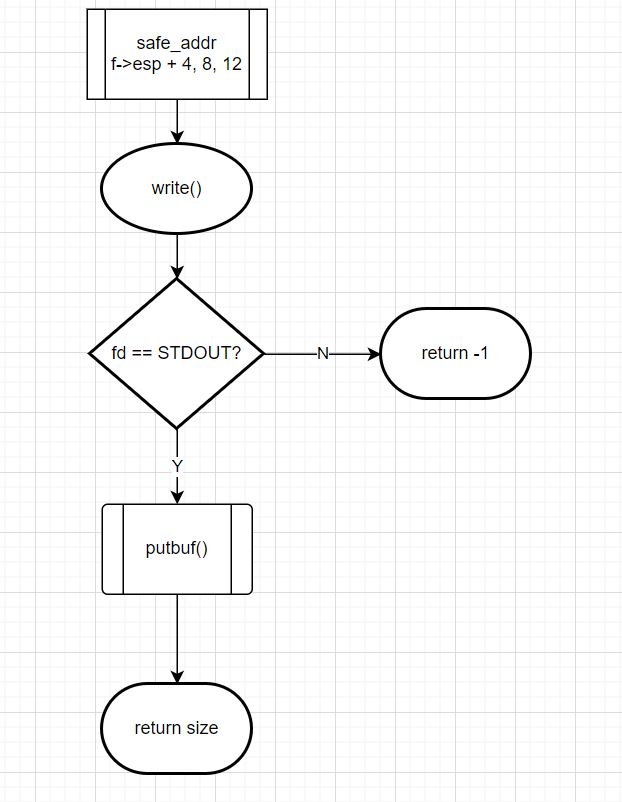












* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

**process\_execute()**

multiple blank check는 본래 유저 구현 함수 blank\_check를 이용해 구현하려 하였으나, data 전달에서 발생하는 overhead와 구현에 어려움을 주는pointer의 사용을 최소화하고자 in-line으로 작성하였다. Flag\_dbl을 0으로 설정하고, multiple blank가 발생하면 1로 설정한다. #110-121을 살펴보면, file\_name을 순회하며 blk가 둘 이상 발생하면 flag를 set하고 break한다.

#123-143은 실행 파일 명을 분리한다. First\_file\_name에 첫 단어를 복사한다. 우선 file\_name을 tmp에 옮겨 담아 tmp를 순회하여 blank를 만나면 복사한다. 만약 flag\_dbl가 발생하면 strtok\_r을, argc가 1임을 의미하는 non\_arg가 발생하면 file\_name을 그대로 복사한다. Thread\_create에 분리 실행 파일명을 전달해야 exit에서 올바른 출력이 수행되기 때문이다.

**Load()**

#344-358 전달받은 file\_name에서 multiple blanks를 제거한 문장을 저장할 blk\_erased를 선언한다. 이후 file\_name을 순회하며 중복 blank를 제거하고 결과를 blk\_erased에 저장한다.

#359-370 Argument 개수 argc, argument 저장 배열 argv, argv 주소 저장 배열 addr, strtok\_r에 사용할 next\_ptr, 임시 문자열 arg를 선언한다. Blk\_reased를 순회하며 argc의 개수를 센다.

#374-399 argv, addr에 argc 개수에 따라 메모리를 동적 할당하고, argc가 1이 아니라면 strtok\_r을 통해 argv를 분해한다. 이후 first\_file\_name에 실행 파일 이름을 분리한다. 1이라면 first\_file\_name에 실행 파일 이름을 복사하고 argv[0]에 파일 이름을 저장한다. 이후 실행 파일 이름을 전달한다.

#483-491 word\_alignment를 위한 변수 total\_len=0을 선언한다. Convention에 따라 argc-1부터 0까지, len에 argv의 길이 + 1 (NULL문자)를 저장하고, total\_len에 더해준 후 \*esp를 len만큼 감소시켜 stack의 적절한 크기의 빈 공간을 가리키게 한다. 이후 strlcpy로 공간에 argv를 복사하고, addr[i]가 저장 공간의 주소를 가리키도록 한다.

#493-497 이때까지 사용한 총 공간의 길이가 4로 나누어 떨어질 때 까지 \*esp를 1씩 감소시키고 0을 저장하는 것을 반복한다.

#499-500 convention에 따라 \*esp에 4를 빼고 NULL을 저장한다.

#503-506 argc-1부터 0까지, \*esp를 4씩 빼고 addr을 저장한다.

#508-509 \*esp를 4 빼고 argv[0]을 저장한 곳의 주소를 담은 주소(\*esp + 4)를 저장한다.

#511-512 \*esp를 4 빼고 argc를 저장한다.

#513-518 argv, addr을 해제하고 \*esp를 4 뺀 후 return address 0을 저장한다.

이슈 1. Hex\_dump의 주소 저장 방식

f9 ff ff bf는 0xbffffff9라고 읽는다.

1. User Memory Access

**Safe\_addr()**

#54-60 vaddr.h의 is\_kernel\_vaddr을 통해 kernel인지 확인하고, 그렇다면 exit(-1)을 호출한다. Is\_user\_vaddr을 통해 user인지 확인하고, 아니라면 exit(-1)을 호출한다.

#72-104 f->esp는 Syscall number를 저장 중이다. argument 개수에 따라 esp + 4\*n을 safe\_addr에 전달하여 안전한지 확인한다.

**Page\_Fault()**

#151-154 user가 아니거나 kernel 영역을 참조한다면 esp에 16을 더하고exit(-1)을 호출한다.

이슈 1. f->esp

f->esp에 대한 조작 없이 exit(-1)을 호출하면, exit(1234215)와 같은 dummy value가 출력된다. Hex\_dump와 debugging을 통해 확인한 결과, esp+16에 exit value가 저장되어 있는 것을 알 수 있으므로 f->esp에 16을 더해준 후 exit을 호출하도록 수정하였다.

1. System Calls

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

**SYS\_HALT**

Halt를 호출한다.

**Halt()**

Devices/shutdown.h에 내장된 Shutdown\_power\_off()를 호출한다.

**SYS\_EXIT**

f->esp + 4가 안전한지 확인하고 exec에 전달하고 그 결과를 eax에 저장한다.

**Exit()**

요구 사항에 따라, thread 이름과 status를 출력하고, exit status에 status를 저장한 다. 이후 thread를 종료한다.

**SYS\_WAIT**

f->esp + 4가 안전한지 확인하고 wait에 전달하고 그 결과를 eax에 저장한다.

**Wait()**

Process\_wait에 pid를 전달하고 결과 값을 반환한다.

**Process\_wait()**

#191-212 pointer e에 list\_begin을 통해 현재 thread의 child\_list의 시작을 저장한다. 이후 list\_end에 다다를 때까지 list\_next를 통해 순회하며, thread pointer t에 list\_entry를 통해 해당 child\_elem을 embed한 thread를 전달받는다. 이후 해당 thread tid가 함수가 전달받은 child\_tid와 동일한지 확인하고, 같다면 child\_lock semaphore는 down시켜 데이터 오염을 차단한 후 exit status에 thread exit status를 저장하고 thread child list에서 해당 thread를 제거한 후 mem\_lock semaphore를 up시킨다. 이후 exit\_status를 반환한다. 만약 child\_tid의 process가 정상적이지 않다면 return -1을 한다.

**SYS\_WRITE**

f->esp + 4, 8, 12가 안전한지 확인한다. 이후 write에 셋 모두 전달하고 결과를 eax에 저장한다.

**Write()**

Fd가 STDOUT인지 확인한 후, 아니라면 -1을 반환한다. 맞다면 내장 함수 putbuf()를 사용해 출력하고, 출력한 size를 반환한다.

**SYS\_READ**

f->esp + 4, 8, 12가 안전한지 확인한다. 이후 read에 셋 모두 전달하고 결과를 eax에 저장한다.

**Read()**

Fd가 STDIN인지 확인한 후, 아니라면 -1을 반환한다. 맞다면 내장 함수 input\_getc()를 사용해 출력하고, 입력한 size를 반환한다.

이슈 1. wait child 탐색

wait에서 전달받은 tid가 자식인지 확인하기 위해 처음에는 all\_list를 사용하고자 하였으나, all\_list를 사용하는 경우 오류가 발생하여 thread 내부에 새로운 child\_list를 embed한 후 이를 탐색하도록 하였다.

이슈 2. Semaphore

본래 ppt에 따라 busy wating을 사용하려 하였으나, 소스 파일을 읽어보니 semaphore API가 잘 정의된 것을 보고 sema\_child과 sema\_mem을 각각 embed하여 semaphore mutex 방식으로 구현하였다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
  1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**