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

담당 교수 : 김 영 재

조 / 조원 : 장 수 길

개발 기간 : 2021/9/10 – 2021/9/29

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

이번 운영체제 프로젝트에서는 주어진 Pintos 운영체제의 기본 코드를 바탕으로 user program을 정상적으로 실행할 수 있도록 코드를 작성하는 것이었다.  
해당하는 요구사항들을 만족하기 위해서 크게 3가지의 주요 개발 목표들이 주어졌는데, 다음과 같다. User program에서 호출하는 system call의 인자들을 stack에 쌓아주는 **Argument Passing,** stack에 쌓은 값들이 유효한 값인지 점검하는 **user memory access,** 그리고 마지막으로 호출할 **system call** 들을 구현하는 것이었다.  
추가적으로 주어진 과제는 **user-defined system** call들인 fibonacci system call과 max\_of\_four\_int system call을 구현하는 것이었다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

초기 상태의 pintos는 명령과 인자(argument)들을 구분하지 못한다.  
Argument passing을 구현하여 user stack에 쌓아주는 것을 통해서 생성되는 프로세스는 자신이 접근해야 할 인자들의 주소값과 실제 값에 접근하는 것이 가능하다.  
이 단계까지 자신의 작업을 검증하는 것은 hex\_dump()함수를 통해 가능하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명  
<사진 1: ‘echo x’ 명령에 대한 hex\_dump 함수 결과>

1. User Memory Access

User memory access를 구현하는 것을 통해 pintos는 유효하지 않은 메모리에 접근하는 포인터 값들을 차단할 수 있다.

1. System Calls

System call을 구현하는 것을 통해 pintos는 주어진 명령어와 인자에 해당하는 system call을 호출하고 수행할 수 있다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명  
    초기 Pintos는 명령어와 인자들을 구별하지 못하기 때문에 먼저 커널 내 스택에 argument들을 쌓아주어야 한다.   
    따라서 먼저 커맨드 라인에 주어진 문자열을 delimiter 단위로 parsing 한 뒤, 구분된 단위를 순차적으로 스택에 쌓아준다.   
    이때 스택에 단순히 무작정 쌓는 것이 아닌 다음과 같은 과정을 통해 스택에 쌓아주게 된다.
    1. 먼저 구분한 문자열 토큰들을 순서대로 스택에 쌓아준다.  
       (단 널문자 ‘\0’를 포함한다)
    2. 다음 4바이트 단위를 맞춰주기 위한 word alignment byte를 추가한다.
    3. Null sentinel을 추가한다.
    4. 각 인자들의 주소값을 추가한다.
    5. 인자의 개수를 쌓아준다.
    6. 반환값 0을 쌓아준다.

위와 같은 과정을 통해 pintos는 프로세스에 대한 인자를 구분할 수 있게 된다.

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

Argument passing을 통해서 스택에 쌓인 값들을 통해서 프로세스는 어떤 system call을 호출하고 어떤 argument들을 첨부할지 결정한다. 하지만 스택에 쌓여있는 값이 유효하지 않은 (즉 null 포인터, 커널 메모리에 접근하거나, 가상 메모리 페이지로 매핑되어있지 않은) 곳을 가리킬 경우 프로세스는 조치를 취할 필요가 있다.

* Invalid memory access를 어떻게 막을 것인지 설명

프로젝트 요구사항 명세서에서는 두가지 방법을 제시한다.  
첫번째 방법으로는 포인터 값이 유효한지 먼저 검증한 후, 참조하는것이고  
두번째 방법으로는 PHYS\_BASE값과 먼저 비교한 뒤 page\_fault의 유무를 확인하여 처리하는것이다.  
*이번 프로젝트에서는* ***첫번째 방법을*** *사용하였습니다.*

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명  
    운영체제는 시스템 내에서 돌아가는 프로그램에 대한 컨트롤이 필요하다. 만약 프로그램들의 실행 내용들에 대해 최소한도의 제약이 없다면 운영체제는 단순한 라이브러리일 뿐이며, 컴퓨터는 다양한 위험에 노출된다.  
    이를 방지하기 위해 운영체제는 user mode와 kernel mode로 나누어져 있으며, 두 모드 간 전환을 통해서 운영체제는 컴퓨터에서 제공하는 기능들을 유저 프로그램에 전달한다. 이 전달과정에서 사용하는 것이 시스템 콜이다.
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)  
    **SYS\_HALT**: Pintos의 작동을 정지시킨다.  
    **SYS\_EXIT**: 현재 실행중인 유저 프로그램을 종료시키고 커널로 돌아온다.  
    **SYS\_EXEC**: 새로운 쓰레드를 만들어서 유저 프로그램을 실행시킨다. 유의해야 할 점은 이 새로운 쓰레드가 종료되기 전 부모 프로세스가 종료될 수도 있다는 점이다.  
    **SYS\_WAIT**: 특정 thread가 종료될때까지 기다린다. 마찬가지로 현재 부모 프로세스가 자식 프로세스의 종료를 기다리지 않는 상태이기 때문에 주의해야 한다.  
    **SYS\_READ**: 모든 기능을 구현하진 않고, 표준 입력에 대해서만 처리한다.  
    **SYS\_WRITE**: SYS\_READ와 유사하게 표준 출력을 처리한다.  
    **SYS\_FIBO**: 인자로 주어지는 값 (n이라고 하자) 에 대해서 n번째 피보나치 수를 구한다.  
    **SYS\_MAX**: 인자로 주어지는 4개의 정수에 대해서 가장 큰 정수를 구한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

프로젝트 시작 시작일 2021/9/10  
Argument Passing 구현 기간 : 2021/9/10 – 2021/18  
System Call 구현 기간 : 2021/9/19 – 2021/9/26  
User memory Access 및 추가 System call 구현 기간 : 2021/9/27 -2021/9/30

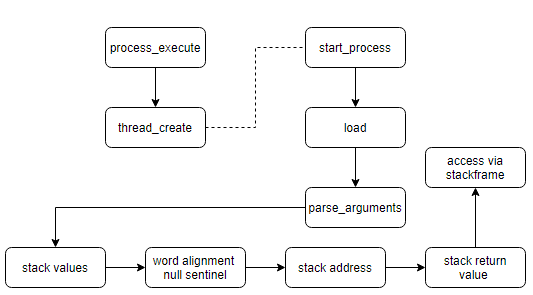
* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

Argument passing을 구현하기 위해 먼저 load 함수 및 process\_execute 함수에서 커맨드 라인에서 명령어만 추출하는 코드를 작성하였다.  
이후 parse\_arguments 함수를 추가하여 스택에 인자들을 쌓는 과정을 구현하였다.  
이후 system call을 구현하기 위해 syscall\_handler에 분기문을 통해 각 system call에 대한 대응을 추가하였다.   
각 system call이 실행될 수 있도록 전용 함수들을 작성하였다.  
User memory access를 구현하기 위해서 check\_valid 함수를 구현하였다.   
마지막으로 system call들 중 exec 와 wait 를 구현하기 위해서 thread 구조체에 몇가지 요소를 추가하였다.   
먼저 메모리를 적재하기 전까지 부모 프로세스를 대기시켜주도록 하는 sema\_load 세마포어, 자식 프로세스가 종료하기 전까지 부모 프로세스를 대기시켜주도록 하는 sema\_exit 세마포어, 부모 프로세스를 알 수 있도록 하는 parent\_thread의 구조체 포인터를 추가하였다. 또한 child 프로세스들을 관리할 수 있도록 pintos에서 제공하는 list 자료구조를 활용하여 child 프로세스들의 리스트를 추가하였다.  
그리고 반환값을 위한 exit\_code, 메모리 적재와 프로세스 종료를 확인할 수 있도록 하는 플래그 exit\_status와 load\_status 두가지를 추가하였다.   
exec 함수 구현에 pid 값으로 특정 자식 쓰레드를 찾을 필요가 있기 때문에 이를 수행해주는 get\_child\_pid 함수를 구현하였고  
wait 함수 구현을 위해 child process가 종료될 경우, child 프로세스가 관리하는 리스트에서 그 프로세스를 삭제시켜주는 기능을 수행하는 remove\_child\_process라는 함수를 추가하였다.

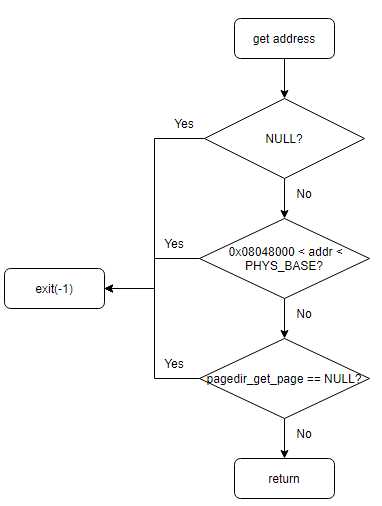
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

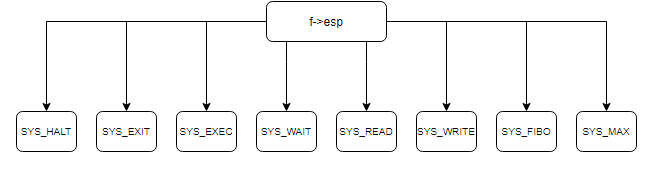
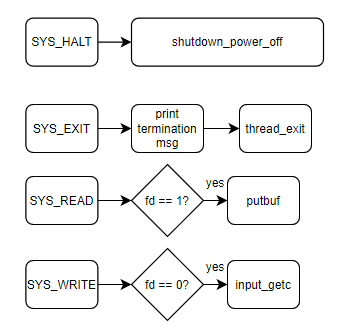
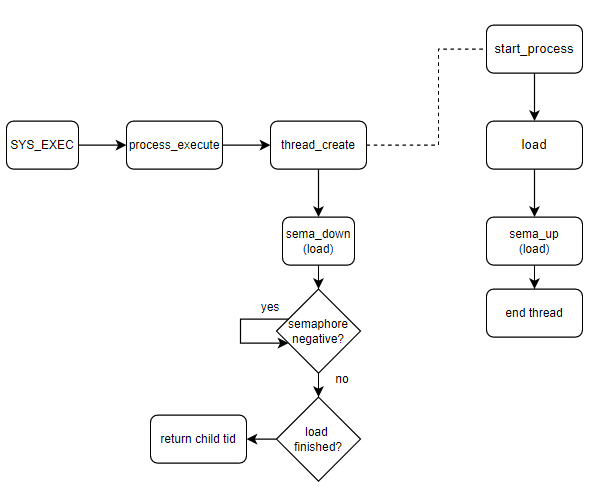
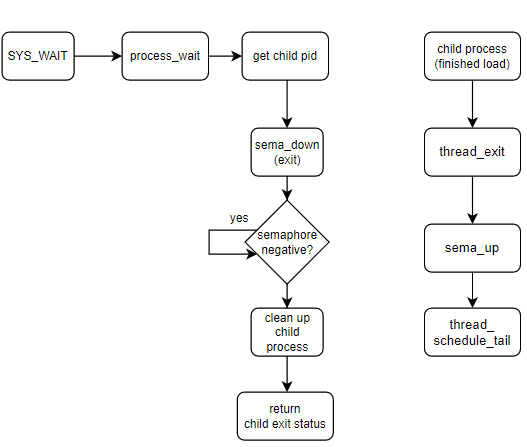
1. Argument Passing

  
<그림2 : Argument Passing Flowchart>

1. User Memory Access

  
<그림3 : User Memory Access Flowchart>

1. System Calls

  
<그림4 : System call Flowchart>  
  
<그림5 : HALT, EXIT, READ, WRITE Flowchart>  
  
<그림6 : EXEC Flowchart>   
<그림7 : WAIT Flowchart>

* 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

먼저 load 함수 및 process\_execute함수에서 요구하는 명령어의 분리는 단순히 첫번째 단어에 한정되어있기 때문에 단순 for loop을 이용하여 구현하였다  
(process.c lines 257-263, 36-40)  
이후 본격적으로 stack에 인자들을 쌓기 위해서 앞서 언급한 새로운 함수 parse\_argument를 구현하였다.   
parse argument함수에서는 strtok\_r 함수를 이용해서 delimiter인 공백을 기준으로 for loop을 이용하여 command line을 끊어준다. 분리해낸 토큰들은 그대로 스택에 memcpy함수를 통해서 쌓아주었다.  
이후 2-B에서 언급한 과정과 마찬가지로 word alignment를 위해 0을 스택에 쌓는 것을 통해 4의 배수를 맞추어주었다.  
그 다음으론 null sentinel을 스택에 추가해주고, 각 토큰들의 주소값을 모두 스택에 쌓아준다. 마지막으로는 반환값인 0을 추가해주었다.   
마찬가지로 memset 및 memcpy함수들을 이용하였다.

1. User Memory Access

User memory access를 위해서는 앞서 언급한 바와 마찬가지로 check\_valid 함수를 사용하여 관련기능들을 구현하였다. 먼저 시스템 콜들을 위한 프레임을 제작하였는데, esp스택의 첫부분에 있는 값을 기준으로 switch문을 사용해 각 시스템 콜로 분기하였다. 이 과정에서 중요했던 것은 각 시스템 콜의 인자의 개수가 각각 다르다는 것이었는데, 시스템 콜 관련 문서를 참고하여 각 인자에 대해 주소값이 유효하지 않은지 check\_valid 함수를 통해 검사해주었다. Check valid 함수는 앞서 언급한 두가지 방법 중 1번 방법을 통해 구현하였다.  
PHYS\_BASE(0xC0000000)와 할당된 스택의 하한선인 0x08048000 주소값 사이인지 검증하였고, 가상 메모리에 매핑이 되어있는지 확인하기 위해서 pagedir\_get\_page함수를 통해 확인해주었다.   
**issue**: 과제를 수행하던 중 테스트 케이스 중 exec-bad-ptr에 대해서 메모리 유효 체크를 하지 않는 문제가 있었다. exec-bad-ptr 테스트의 소스코드를 살펴보니 0x20101234, 즉 유효하지 않는 포인터를 접근하는 것을 확인하였는데, check\_valid 함수에서 걸러주지 못하는 이슈가 있었다. argument들을 효율적으로 접근하기 위해 f->esp를 (int\*)로 형변환하여 4바이트씩 접근하는 과정에서 문제가 있는것으로 해석하여 exec 시스템 콜의 경우 인자와 동일한 (char\*) 형으로 그대로 체크할 수 있도록 수동 체크문을 작성하여 수정하였다.

1. System Calls

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

이번 프로젝트에서 개발한 시스템 콜들을 순서대로 설명하도록 하겠다.  
pintos\lib\user\syscall.h에 작성되어있는 형식 그대로 함수의 뼈대를 구성하였다.   
**SYS\_HALT**: shutdown\_power\_off()를 통해 pintos를 그대로 종료시켰다.  
  
**SYS\_EXIT**: thread\_exit()함수를 통해서 기본적으로 현재 실행중인 쓰레드를 종료시키지만, 정확한 exit code를 반환시키기 위해 thread.h에서 thread 구조체에 추가한 status 값을 인자로 변경시켜준다. 추가적으로 명세서에서 요구한바와 동일한 출력 문구를 생성한다.  
  
**SYS\_WRITE:** file descriptor 값이 표준 출력인 1일 경우 putbuf함수를 통해 값을 전달하고, 전달한 크기만큼 반환한다.  
  
**SYS\_READ:** file descriptor 값이 표준 입력인 0일 경우 인자로 들어온 버퍼에서 size만큼 값을 input\_getc() 함수를 통해 입력받는다.   
  
**SYS\_EXEC:** 기본적으로는 process\_execute 함수를 통해 주어진 명령어를 새로운 쓰레드를 만들어서 수행시킨다. 관련된 synchronization 문제를 해결하기 위해서 세마포어를 사용하였다. Process\_execute함수를 수행시키게 되면 thread\_create 함수를 통해 새로운 쓰레드를 생성하게 되는데, 이때 새롭게 생성된 쓰레드는 start\_process 함수를 통해서 시작된다. 하지만 이경우 새롭게 생성된 쓰레드가 종료하기 전에 부모 프로세스가 종료되는 문제가 발생할 수 있다. 따라서 세마포어를 사용하는데, 초기 세마포어의 값은 0으로 설정한다. 부모 쓰레드는 자식 쓰레드가 종료되기 전까지 대기할 필요가 있기 때문에 process\_execute 함수를 호출한 후 바로 sema\_down을 해준다. 자식 프로세스에서는 로드가 끝난 시점에서 sema\_up을 호출한다.   
이와 같은 설계를 통해 자식 프로세스의 로드가 끝나기 전 sema\_down이 이루어질경우 semaphore의 값은 -1이 되며, semaphore의 정의에 따라서 프로세스는 중지된다.  
실제 주요 구현내역은   
thread.c 의 thread\_create 함수 세마포어 및 플래그 초기화 (line 100 – 108)  
process.c 의 start\_process 함수에서 load 호출 이후 sema\_up (line 79)  
process.c 의 start\_process 함수에서 load 성공시 플래그 변경 (line 87 – 90)  
syscall.c 의 exec 함수에서 process\_execute 이후 sema\_down(&sema\_load) (line 192)  
의 과정으로 요약된다.  
  
**SEMA\_WAIT:** 자식 프로세스들이 종료될때까지 대기하는 시스템 콜이다. 기본적으로는 process.c 에 정의되어있는 process\_wait 함수를 호출한다. 인자로는 tid를 받는데, 종료를 기다리는 대상의 thread id를 의미한다.   
프로세스가 끝나는 것을 기다린다는 것은 생성된 자식 프로세스가 정상적으로 종료했는지 확인하는 것이기 때문에, 역시 세마포어를 사용하여 구현해준다.   
여기서 하나의 의문이 들 수 있는데, 프로그램의 메모리 적재 (로드) 와 프로세스의 종료 세마포어를 구분하는 이유이다. 이유는 자식 프로세스의 종료에 있는데, 여러 개의 프로세스가 자식 프로세스로 존재할 경우 부모 프로세스는 리스트에 등록된 (방금 종료된)자식 프로세스를 삭제해주어야만 한다. 하지만 만약 자식 프로세스가 그대로 종료해버린다면 적재되어있는 메모리는 모두 소실되고 부모 프로세스는 자식 프로세스를 찾을 수 없게 된다. 이 과정에선 sema\_mem 과 sema\_exit을 통해 thread\_exit 와 thread\_schedule\_tail 간의 실행과정을 보조해준다.  
따라서 두개의 세마포어를 단계별로 나누어주게 된다.   
다시 돌아가면, 자식 프로세스가 종료된 이후(load 세마포어 값이 증가해서 부모 프로세스가 다시 가동되는 시점)에는 부모 프로세스가 이 자식 프로세스를 리스트에서 제거해준다 (이때 자식 프로세스는 exit 세마포어에 걸려있어 진행하지 못하고 있다) 부모 프로세스가 자식 프로세스(get\_child\_pid를 사용하여 얻은)를 프로세스 리스트에서 제거(remove\_child\_process) 해준 이후에는 자식 프로세스는 그대로 종료되고, 종료됨과 동시에 exit 세마포어를 증가시켜주는 것을 통해서 부모 프로세스가 진행할 수 있도록 해준다.  
실제 구현과정은   
thread.c 의 thread\_create 함수 세마포어 및 플래그 초기화  
process.c 의 process\_wait 함수의 sema\_down(&sema\_exit) (line 112 - 120)  
thread.c 의 thread\_exit 함수에서 sema\_up(&sema\_exit) (line 312)  
process.c 의 process\_wait 함수에서 해당 child process remove (line 112 - 120)  
로 요약된다.

1. Additional System calls

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

새로운 시스템 콜을 구현하기 위해서는 먼저 sample program을 작성해주어야 했다.   
4개의 정수를 command line argument (편의상 a,b,c,d 라고 하자) 로 입력받아 a번째 피보나치 수와 4개의 정수 중 가장 큰 값을 출력해주는 sample program인 additional.c를 pintos\src\examples에 작성하였다.  
주어진 시스템 콜을 수행하기 위해서는 우선 4개의 인자를 받는 시스템 콜을 어셈블리 레벨에서 정의해주어야 한다. 이를 위해 lib\user\syscall.c에서 다른 시스템 콜들을 참고하여 syscall4라는 프로시져를 정의해주었다.  
4개의 인자를 받는 시스템 콜 프로시져를 정의했다면 대응하는 실제 시스템 콜을 작성할 차례이다. lib\user\syscall.h 에서 int fibonacci 및 int max\_of\_four 함수를 정의하였다.  
이 시스템 콜들이 어떤 방식으로 작동하는지는 src/userprog/syscall.c 에서 구현하였다. 시스템 콜을 분기해주는 switch 문에 SYS\_FIBO와 SYS\_MAX를 추가하여 사용하였다.

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

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명