

Pintos Project 1: User Program (1)

담당 교수 : 박성용

이름 : 박준혁

학번 : 20171643

개발 기간 : 09/18 ~ 10/03

1. 개발 목표

본 User Program (1) Phase는 이번 2022년도 가을학기 운영체제 강의에서 학기 내내 학습하고 개발할 'pintOS 프로젝트'의 대망의 첫 번째 단계이다. 타이틀에서 알 수 있듯, 본 Phase에서의 핵심은 User Program이 pintOS 상에서 정상적으로 돌아갈 수 있도록 OS의 기본적인 요소들을 갖추게 해주는 것이다. 그러한 '기본적 요소'에는 'Argument Passing', 'System Calls Implementation', 'Memory Validation' 등이 포함된다.

각 요소를 구현하기 위해선 단순히 Code-Level Skill만 중요한 것이 아니라, pintOS 자체의 구조, pintOS의 Execution Flow, Virtual Memory Concepts 등을 확실하게 이해하는 것이 핵심이다. 본인 역시 이 점을 명확히 인지하고 있고, 이러한 '기본적인 목표는 운영체제 상에서 User Application Program이 CPU & Memory Resource Hardware와 소통하기 위해서 운영체제가 User Program에게 어떠한 Virtualization과 Execution Flow를 제공하는지를 직접 체험하고 이해하는 데에 있다.'

이러한 목표를 토대로 본인은 실제로 9월 18일부터 금일(10월 3일)까지 약 2주간 성실하게 개발에 집중하였으며, 그 과정에서 체득한 OS의 핵심 Concept, 그리고 Code-Level의 Detail 및 Troubleshooting 과정 등은 이하 항목에서 차례대로 상세히 설명도록 하겠다.

2. 개발 범위 및 내용

A. 개발 범위

1. Argument Passing

제공된 Project Description PDF에서 소개된 내용 그대로, Argument Passing 과정은 크게 Command-Line Parsing 과정과 Stack Set-Up 과정으로 구분할 수 있다. 전자에선 User Program이 입력으로 받아들인 Command-Line에서 Space 문자를 효과적으로 걷어내어 정확히 Token(Argument)만을 뽑아내는 것이 핵심이며, 후자에선 pintOS에서 지정한 Stack Figure에 맞게 Stack의 Default State를 다 차례의 Push 연산으로 구성하는 것이 핵심이다.

본인은 상기한 과정을 성공적으로 구현하였으며, 현재 본인의 pintOS 상에서 Argument Passing은 정상적으로 수행되고 있다. 자세한 Background Concept 및

Code-Level Detail은 후술하겠다.

2. User Memory Access

OS를 구현하는 개발자 입장에서 가장 주의해야 하는 것은 OS의 이용자(대부분 User Program)가 OS에게 옳지 않은 Pointer를 건네는 상황이다. 우리의 pintOS 역시 마찬가지로 개발 시 이러한 점을 주의해야 하며, User Program이 Passing한 Data를 직접적으로 다루는 부분에서 특히나 이는 중요하다.

따라서 본 프로젝트 Phase에선 User Memory Access Check가 필요하며, 조금 더 구체적으로 말하면 User Program이 OS의 Code를 사용하기 위해 접근하는 Interface, System Call 구현 부분에서 이러한 작업이 필요하다. 본인은 이 점을 명확히 인지했고, 이에 따라 User Program이 OS에게 넘긴 Pointer가 User Address Space에서 온 것인지, 또는 NULL이 아닌지를 확인하는 Macro를 마련해 정확한 메모리 체크를 도모하였다. 역시 마찬가지로 자세한 Detail은 후술하겠다.

3. System Calls

2번 항목 설명 간에 언급한 것처럼, System Call이란 (User Program이 원하는) OS의 Functionality를 User Program이 사용할 수 있도록 OS가 API의 형태로 제공하는 Interface이다. 이는 OS의 Limited Direct Execution을 실현하기 위한 Dual Mode Operation Concept의 일환으로 도입된 개념으로, 본 pintOS 프로젝트에서도 역시나 마찬가지로 구현해야 한다.

본 프로젝트 Phase에선 아직 User Program 구현의 모든 과정을 구현하는 것은 아니기에 halt, exit, exec, wait 등의 기초 System Call들 위주로 구현한다. 본인은 pintOS Manual을 중점적으로 확인하면서 이 과정을 구현했으며, 1/2번 항목과 마찬가지로 정상적으로 기능하고 있다.

가장 난해하고 중요했던 부분은 바로 Synchronization이 필요한 wait과 exit, exec System Call의 구현 과정이었으며, 이를 포함해 전체적인 System Call Implementation에 대한 설명은 이어서 진행한다.

B. 개발 내용

- Argument Passing

✓ 커널 내 스택에 argument를 쓰는 과정

User Application Program의 Execution 과정에 필요한 Argument들을 Stack에 Push하는 과정이다. 우선, 가장 먼저 해야 할 일은 Argument Parsing으로, Program 실행 시작 시에 넘겨받은 Command-Line을 Space 문자 기준으로 여러 Argument로 분리하는 작업이 필요하다.

Command-Line이 Argument로 잘 분리되었으면, 해당 Argument들을 기준으로 입력이 정확히 들어왔는지를 판단한다. Argument 개수가 올바른지, 또는 Program File Name이 정확한지 등을 확인한다.

그러한 Checking 과정이 마무리되면 pintOS에선 Process의 Stack 대한 Memory를 할당해준다. 이는 미리 제공되는 setup_stack() 함수를 통해 이뤄진다. Stack Memory 공간 할당이 마무리되면, OS에선 esp Stack Pointer를 통해 Stack의 Top을 추적할 수 있게 된다.

우리가 해야 할 일은 바로 이 esp Pointer를 가지고 pintOS에서 요구하는 Stack Configuration을 진행하는 것이다. pintOS Manual에는 다음과 같은 예시 그림이 제공된다. "/bin/ls -l foo bar"에 대한 Stack Pushing 결과이다.

Address	Name	Data	Type
0xbfffffff8	argv[3] [...]	'bar\0'	char[4]
0xbfffffff4	argv[2] [...]	'foo\0'	char[4]
0xbfffffff0	argv[1] [...]	'-l\0'	char[3]
0xbfffffdfe	argv[0] [...]	'/bin/ls\0'	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbfffffff8	char *
0xbfffffe0	argv[2]	0xbfffffff4	char *
0xbfffffdcc	argv[1]	0xbfffffff0	char *
0xbfffffd8	argv[0]	0xbfffffdfe	char *
0xbfffffd4	argv	0xbfffffd4	char **
0xbfffffd0	argc	4	int
0xbfffffccc	return address	0	void (*) ()

본 프로젝트에선 위의 구성을 그대로 맞춰서 Stack을 구성하면 Argument Passing 구현을 마무리하게 된다. 그 과정은 다음과 같이 요약할 수 있다. 위

의 그림과 함께 이해할 수 있다.

- Argument의 역순(ex. argv[k], argv[k-1], ..., argv[1], argv[0])으로, 각 Argument의 Data Value를 Stack에 1바이트 단위로 Push한다. 즉, Data의 크기에 따라 Push되는 Byte의 개수가 달라지는 것이다.
 - 이때, 각 Data Value의 Byte 길이를 특정 변수에 누적시킨다. **Word Alignment**를 하기 위해!
 - 또한, 각 Argument Data Value가 저장된 Stack 내의 Address를 특정 변수 혹은 Array에 기억시킨다. PDF의 구성을 따르기 위해!
- Argument Data Value Push가 마무리되면 길이 누적 값을 이용해 80x86 Calling Convention의 Word Alignment를 수행한다. Padding을 집어넣는 것이다. 이때, 이 Padding의 Value도 0으로 초기화한다.
- 이어, 앞서 Argument Data Value의 Stack 내 Address를 이용해 해당 Value들을 가리키는 Pointer를 Push한다. 이때, Argument List를 나타내는 argv Array의 마지막 Element(ex. argc(Argument Count)가 4일 경우, argv[4])의 Pointer도 Push하되, 값은 NULL로 설정해준다.
 - 각 Element는 Pointer이기 때문에 Word(4Bytes) Size이다. 이하 Push되는 모든 Data도 전부 Word Size이다.
- 마지막으로 Argument Count 값 Return Address 값을 순서대로 Stack에 Push한다.

결국, 핵심은 esp Pointer를 이용한 Pointer Arithmetic, 그리고 Pointer가 가리키는 Stack Address에 Data Value Size에 알맞게 값을 Setting하기 위한 Type Casting 과정이다. 이 Pointer Arithmetic, Casting 과정을 주의 깊게 수행하면 되는 단계이다.

- User Memory Access

✓ Pintos 상에서의 invalid memory access 개념

앞서 언급한 바 있듯, System Call이란, User Application Program이 OS의 주요 Functionality를 이용하기 위해 OS에게 전달하는 '요청'이자, OS와 소통하

는 Interface이다. 이때, System Call에 User Program의 일부 Data를 전달하는 일도 당연 존재한다. 문제는 바로 여기서 발생한다.

"만약, User Program이 OS(Kernel)에 보낸 Data(Pointer)가 Invalid한 영역(가령, Kernel Code 부분 접근이나 NULL Pointer의 전달, 또는 Mapping되지 않은 Virtual Memory 주소 접근 등)을 접근한다면 어떡할 것인가?"

바로 이러한 상황은 System Call 및 OS Error 처리 시에 가장 주의 깊게 다뤄야 하는 부분이다. 본 pintOS 프로젝트에서도 마찬가지로 이 점을 주의해야 한다. 이를 방지하지 않는다면 OS 및 Device Code에 대한 User Program의 공격이나 오염이 가능해지고, 이는 나아가 System에 대한 중대한 위협이 될 수 있다.

✓ **Invalid memory access를 어떻게 막을 것인가**

다행스럽게도, pintOS에선 이러한 Memory Access Checking 기능을 가진 함수를 다음과 같이 제공한다.

- **bool is_user_vaddr (const void *va);**
- **bool is_kernel_vaddr (const void *va);**

함수 이름에서도 알 수 있듯, is_user_vaddr() 함수는 인자로 넘겨 받은 Address가 User Virtual Memory에 위치한지를, is_kernel_vaddr() 함수는 인자로 넘겨 받은 Address가 Kernel Virtual Memory에 위치한지를 확인한다. 어떻게? Project PDF나 Manual에서 언급되는 것처럼 PHYS_BASE Macro를 통해서 말이다. 이 변수는 Process Virtual Memory Address Space의 3GB 지점을 나타내는 변수로, 넘겨받은 주소의 값이 3GB를 기준으로 위에 있는지, 아래에 있는지 확인하는 것이다.

우리는 이러한 기본 제공 함수들을 이용해 Invalid Memory Access Checking Routine을 마련할 수 있다. 기본적으로 User Program의 Corruption 위험이 존재하는 System Call Implementation 부분에서 이 작업이 필요하며, 나아가 Page Fault와 같은 Error Handling 상황에서도 Error를 야기한 Address가 어느 영역을 참조했는지 등을 체크해야 한다.

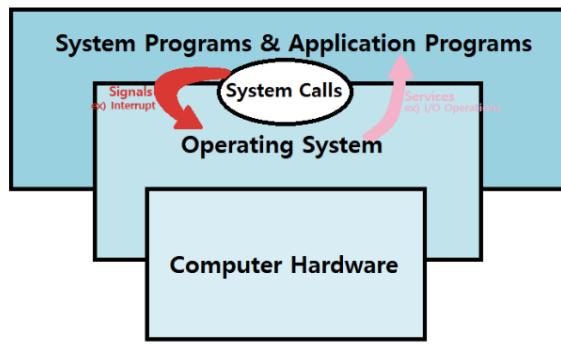
Page Fault에서는 왜 체크해야 할까. 이유는 간단하다. Page Fault는 이미

pintOS 상에서 기본으로 구현되어 있고, Page Fault라 함은 'Attempt to access kernel virtual memory'가 원인이 되기 때문이다. 즉, Page Fault Handler에서 체크 루틴을 마련함으로써 Kernel Memory Access 상황도 처리할 수 있다.

이러한 Invalid Access가 감지된다면, 기본적으로 Process를 **-1 Error Status Set**과 함께 종료시켜주면 된다. 어렵지 않게 구현할 수 있겠다.

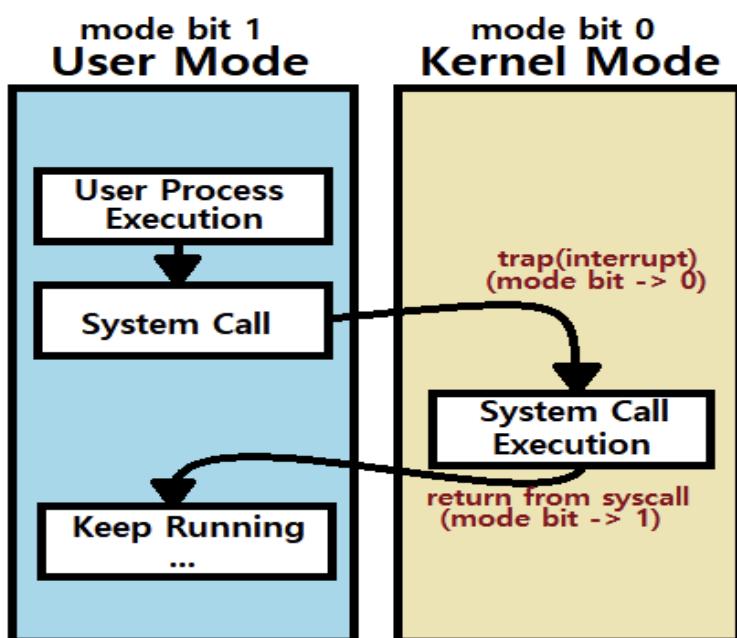
- System Calls

✓ 시스템 콜의 필요성



System Call은 앞서 벌써 두 번이나 언급한 것처럼 User Application이 OS의 특정 기능을 사용하기 위해 요청하는 Interface로, API Library 형태로 User에게 제공된다. 이는 결국 Limited Direct Execution의 일

환으로 도입된 Dual Mode Operation의 구현 결과이며, 이번 프로젝트 Phase에서 가장 중요한 부분을 차지하는 개념이다.



Dual Mode Concept에 따라, OS는 두 개의 Mode를 만들어 Process의 부분과 OS의 부분으로 Virtual Memory를 구분한다. 이를 통해 **OS가 자기 자신, 그리고 다른 System Component를 User Program으로부터 보호할 수 있는** 것이다. SW에 Limit을 가해 Privileged Operation에 쉽게 접근하지 못하도록 하는 것이다. 그러한 Privileged Operation에는 I/O Request, File System Request 등이 포함된다.

✓ 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명

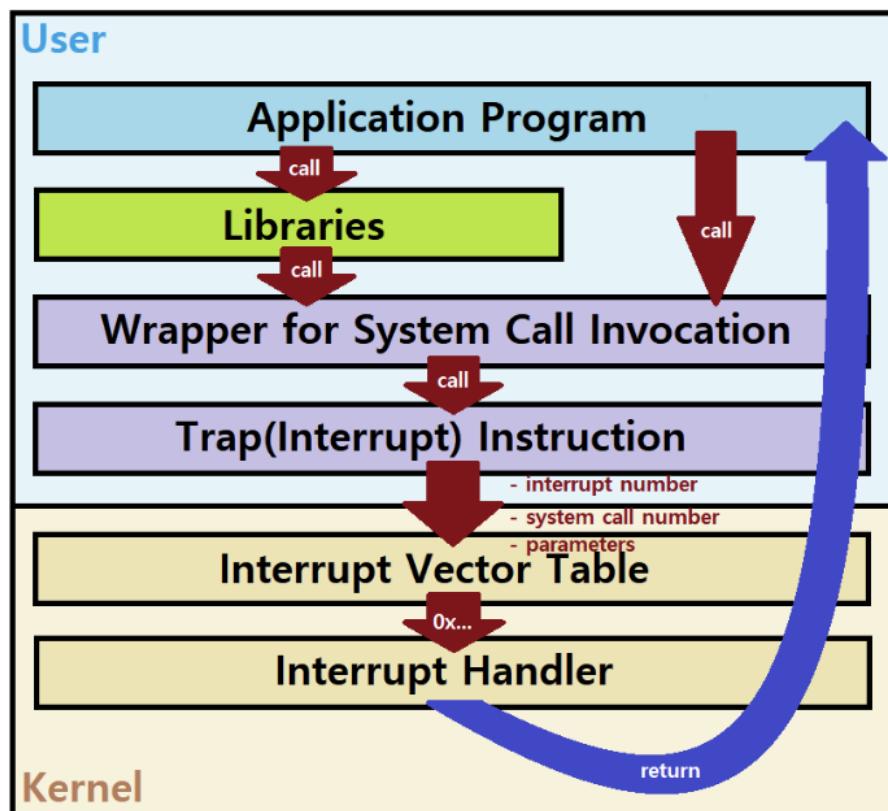
본 프로젝트 Phase에선 앞서 언급한 바와 같이 총 6개의 System Call만 구현하면 된다. 거기에 추가로 2개의 Additional System Call (Fibonacci, Max of four integers)의 구현이 필요하다. 그들은 다음과 같다.

- 1) **halt** : System을 모종의 이유로 Power-Off 하는 System Call로, pintOS 상에선 shutdown_power_off()라는 이름의 Device 함수를 호출해 간단히 구현한다. 일부 정보가 손실될 수 있기 때문에 주의하여 호출해야 한다.
- 2) **exit** : 현재 수행 중인 User Program이자 Caller인 Process를 종료시키는 System Call로, Exit Status를 Kernel에게 알리는 기능이 있다. 후술하겠지만, wait, exec System Call과 밀접한 관계가 있다.
- 3) **exec** : 넘겨 받은 Command-Line을 기준으로 해당 Line의 File Name이 가리키는 Program을 새로운 Process로 생성시키는 System Call이다. Linux의 fork와 exec 기능이 하나로 합쳐진 System Call이라 볼 수 있다.
- 4) **wait** : Parent(Current) Process가 자신의 Child Process가 종료하고 Exit Status를 반환하는 것을 기다리게 하는 System Call이다. Zombie Process 생성을 방지하기 위해 꼭 필요한 과정이며, Synchronization이 중요하다.
- 5) **read** : 인자로 넘겨 받은 File Descriptor가 가리키는 File에서 Size 만큼의 Bytes를 읽고, 그 읽은 내용을 Buffer에 저장하는 System Call로, 본 Phase에선 아직 File System이 구축되지 않았기에 Console에서만 읽는다.
- 6) **write** : 5)와는 반대로 Buffer의 Data를 Descriptor가 가리키는 File에 출력하는 System Call로, 마찬가지로 아직 File System이 구축되지 않았기에 Console(Standard Output)에만 출력한다.

- 7) **fibonacci** : 본 Phase의 Additional System Call로, 넘겨 받은 Integer에 해당하는 순번에 위치한 Fibonacci Number를 생성 및 반환한다. 다양한 구현 방식이 존재할 것이다.
- 8) **max_of_four_int** : 본 Phase의 Additional System Call로, 넘겨 받은 네 개의 Integer 중 가장 Value가 큰 수를 반환한다. 다양한 구현 방식이 존재할 것이다.

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

User Application Program에서 System Call을 호출한 후 Kernel Code 부분에서 System Call을 Handling한 후, 다시 User Level로 돌아가는 과정을 몇 가지 시작 자료를 이용해 소개한다. 아래를 보자.

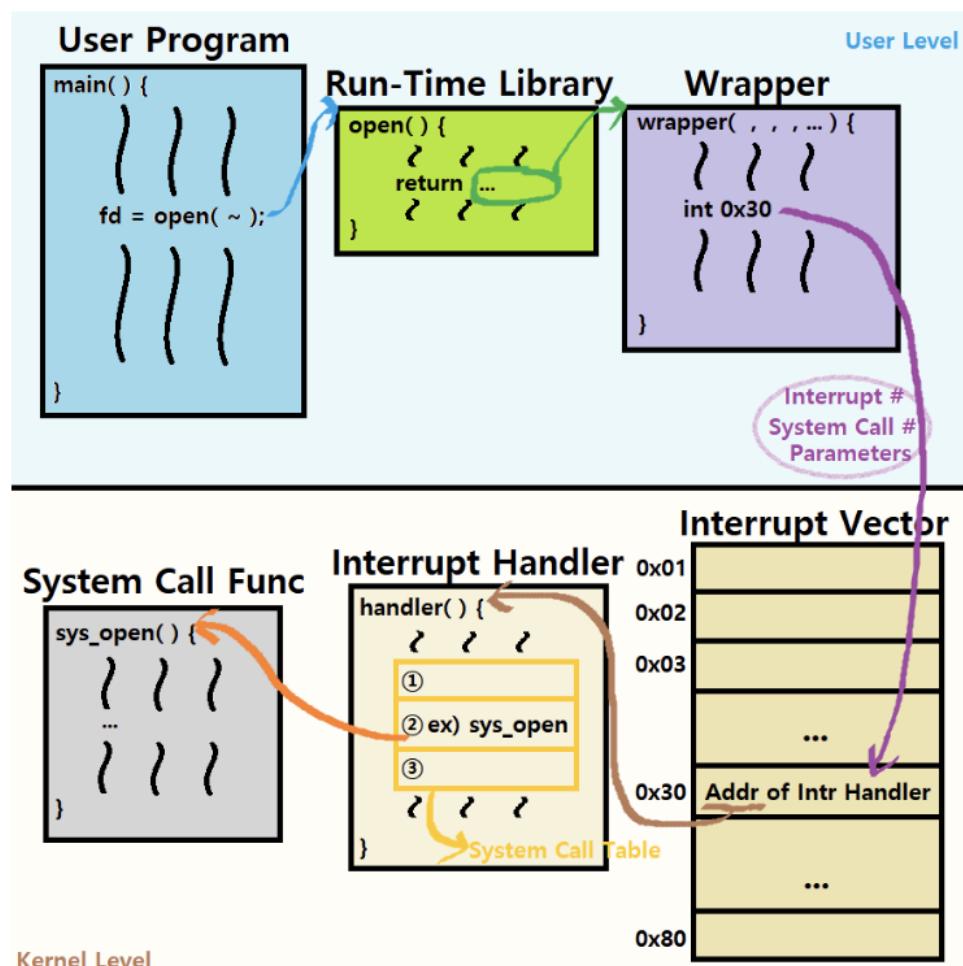


Application Program이 System Call을 호출하면, 사실 그 Call의 곁면에는 **Wrapper**라 불리는 보조 캡데기가 존재한다. Kernel에게 Trap Interrupt Signal을 보내 Trap Interrupt Handler(사실상 System Call Handler)를 동작시키는 역

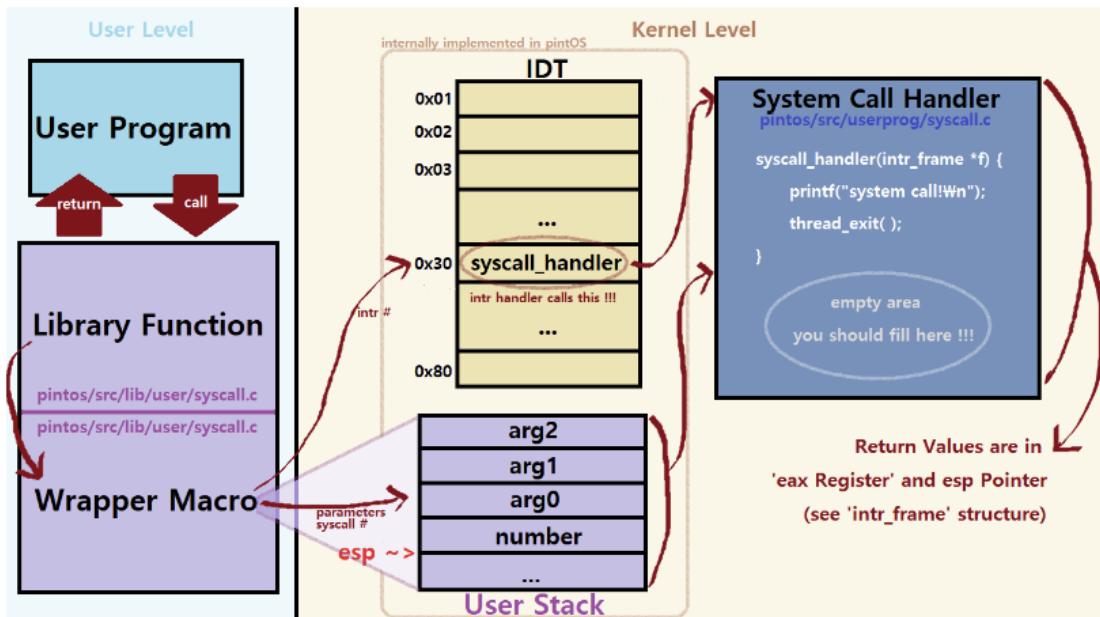
할을 수행한다. 대체로, 이러한 Wrapper에 C Library Function의 형태로 또 하나의 겉 껍데기가 쌓여있는 경우가 많다.

이어, Wrapper가 보낸 Signal인 Interrupt Instruction을 토대로 System Call Handler가 수행되고, 거기서 넘겨 받은 System Call Number를 기준으로 적절한 Functionality를 제공하는 것이다. 즉, Wrapper에서 Interrupt Vector Table로 데이터가 넘어갈 때, 단순히 Interrupt Instruction만 넘어가는 것이 아니라, System Call Number, 그리고 Parameters(Arguments)도 함께 넘어가는 것이다. 이 과정은 실제로 본 Phase에서 구현한다.

System Call Handling이 마무리되면, 쭉 연쇄적으로 반환되어 User Level로 돌아가게 된다. 아래의 그림을 보자. (참고로, 시각자료는 모두 본인이 직접 제작하였다. 강의 자료 또는 인터넷 자료를 참조했다)



위 그림의 'System Call Table'이란 것은, 본 pintOS에선 switch문을 통해 논리적으로 구현된다. 이 역시 마찬가지로 본 Phase에서 구현한다.



위와 같은 형태로도 표현할 수 있겠다. IDT라 함은 Interrupt Descriptor Table, 다른 말로 Interrupt Vector Table로, 이는 pintOS에 내부적으로 이미 구현되어 있다. 각 Interrupt에 대한 Handler 주소가 담겨있다. 과거 System Programming에서 학습한 바와 같이, System Call의 경우 Trap Exception이라 칭하는데, 즉, pintOS 상에선 위의 그림과 같이 IDT의 30번 Slot에 위치한다. 해당 Handler가 호출되어 System Call Handling이 진행되는 것이다.

Handling을 마치면, eax Register에 Return Value를 넣어 User Program에서 System Call 결과를 전달할 수 있다. (관습)

이렇게, System Call이 발생했을 때 OS 내부에서 어떠한 일이 발생하는지를 여러 요소 별로 구분해 알아보았다.

3. 추진 일정 및 개발 방법

A. 추진 일정

크게 보았을 때 본 프로젝트 Phase는 Argument Passing, System Call Handling, Invalid Memory Access Checking, Additional System Call Implementation 등으로 나눌 수 있다. 본인은 프로젝트 안내 PDF와 Manual에 나와있는 Suggested Order of Implementation을 참고하되, System Call 구현을 Memory Check 구현보다 먼저 하였다. 결국 Memory Check는 System Call에서 부분적으로 사용하는 기능에 불과하기에 System Call의 큰 틀을 잡고, 주요 구현을 마친 후에 메모리 확인 과정

을 가미하는 식으로 진행했다. 개발 일정을 요약하면 다음과 같다.

- 1) **Understanding pintOS (09/18-09/22)** : 우선, 기본적으로 본인은 프로젝트 자체, 그리고 pintOS 그 자체에 대한 이해가 가장 중요하다고 보았고, 따라서 초반의 많은 시간을 이 과정에 할애하였다. 배부된 프로젝트1 안내 PDF와 pintOS Manual을 여러 차례 반복적으로 정독했으며, 중간 중간 Execution Flow와 Stack Configuration 등을 그려가며 '구현해야 하는 부분이 어디인지'를 파악하는 데에 집중했다.
- 2) **Argument Passing (09/22-09/24)** : Command-Line Parsing과 Stack Set-Up 과정을 수행했다. 제공된 Stack 구조 예시 그림을 토대로 여러 차례의 디버깅과 hex_dump() 함수 호출, printf()문을 이용한 Naïve한 분석 등을 통해 성공적으로 이 과정을 마무리했다.
- 3) **System Calls & Invalid Memory Checking Implementation (09/24-09/30)** : 약 일주일 정도 System Call 구현에 힘을 썼으며, 사실상 wait과 exit System Calls를 제외한 나머지 부분은 하루 만에 완성하였다. 그 외 약 6일 가량은 wait과 exit의 구현에 집중했다. 아무래도 Synchronization이 필요한 부분이었기에 여러 차례의 Troubleshooting이 있었다.

Invalid Memory Checking의 경우 System Call 구현과 거의 동시에 수행하였으며, 이 부분은 어렵지 않게 해결하였다. 이미 제공된 함수가 있어 'Check할 위치 선정'만 하면 됐기 때문이다.

- 4) **Additional System Calls Implementation (09/30-10/01)** : fibonacci와 max_of_four_int System Call, 그리고 이를 이용한 additional Program의 작성은 상당히 수월하게 수행할 수 있었다. 이후 금일(10/03)까지 코드 정리 및 주석 달기, 보고서 작성 등의 나머지 작업을 수행하였다.

B. 개발 방법

✓ Argument Passing 과정

Argument Passing 과정은 결국 Command-Line Parsing, Stack Set-Up 과정인데, 이들은 모두 **pintos/src/userprog/process.c** 파일 내에서 구현한다. 여러 위치에서 이 개념을 구현할 수 있는데, 본 프로젝트에선 **load** 함수 내에

서의 구현을 권장하였으므로 load 함수에서 구현한다.

◆ **Command-Line Parsing** : load 함수 내에서 File Open 이전에 수행한다.

본인의 경우, strtok 함수를 가져다 쓰지 않고 Iterative Algorithm을 이용한 Parsing Function을 직접 작성해 사용했다. strtok의 경우 과거 System Programming Project 수행 간에 상당히 불안정한 요소(주로 메모리 관리 측면에서. 주관적인 서술임)가 있음을 몸소 경험한 바 있기 때문에 나만의 Parsing 함수를 도입하는 것이 더 좋을 것이라 판단하였다.

Parsing 과정에서, Command-Line으로부터 분리된 Argument들을 argv라는 Character Pointer Type Array에 저장하고, 그 개수를 argc 변수에 저장한다. 이때, Parsing을 마치고 나면, thread_name() 함수의 반환 값에 argv[0](=File Name)을 복사하는 것도 추가하였다. 추후, exit System Call 구현 시의 출력 사항을 구현하기 위함이다.

◆ **Stack Set-Up (Arguments Pushing)** : load 함수 내에서 setup_stack() 함수 호출을 말미암은 Stack Memory 할당 과정이 마무리된 이후에 수행한다. 제공된 Stack Configuration 그림에 맞추어 esp Pointer를 적절히 조정하여 수행할 수 있다. 앞서 언급했듯, Type Casting을 통한 정확한 Value 입력이 중요한 부분이기도 하다.

이때, 본인은 다음과 같은 Macro Function들을 추가하여 Program Code 가독성 향상을 도모하였다. Stack을 아래 방향으로 확장하는 작업과, Stack에 데이터를 삽입하는 작업을 간단하게 표현할 수 있다. 아래를 보자.

```
/* Macros for raising a readability */
#define S_EXPAND(amount) *esp = (uint8_t*)(*esp) - amount
    // S_EXPAND: Expand stack downward, by using subtraction of pointers
#define S_SETVAL(value) *(uint32_t*)(*esp) = value
    // S_SETVAL: Set the value of argument on the address pointed by esp
```

✓ **System Calls Implementation & Invalid Memory Access Checking 과정**

앞선 항목에서 pintOS 상에서의 System Call Flow를 시각 자료를 통해 확인한 바 있다. 그 과정에서 Interrupt Instruction이 IDT의 Handler를 호출하는 과정은 이미 pintOS에서 제공한다고 했다. 따라서, Manual에도 언급되어 있듯이 우리는 **pintos/src/userprog/syscall.c** 또는 **syscall.h**를 수정하기만 하

면 된다. 특히나, syscall.c 파일의 **syscall_handler()** 함수를 수정하는 것이 핵심이다. 이 함수는 이름대로 System Call Handler로서 동작하며, 우리는 그 안에서 switch문과 같은 '논리적 System Call Table'을 구축해 Handler를 완성하면 된다. 본인의 경우, 각 System Call의 Implementation을 Switch문 내에 일일이 하게 되면 프로그램 Code의 가독성이 떨어질 것임을 우려해 switch문의 각 Case에서 Corresponding Function을 호출하도록 구현했다.

◆ System Call Number 및 Parameters(Arguments)를 얻는 방법

pintos/src/lib/user/syscall.c 파일의 **Wrapper** 부분을 Hacking해보면 알 수 있듯, System Call Handler 입장에선 넘겨받은 Interrupt Frame 내부의 esp Pointer 정보를 통해 각종 Parameter를 확인할 수 있다. 이때, 각 Parameter는 Pointer의 형태로 Data를 다루기 때문에 모두 uint32_t, 즉, 4Bytes Word Size이며, 따라서, 이 점을 이용해 Systemic하게 이 데이터들을 뽑아낼 수 있다. (바로 아래서 설명)

◆ switch문의 각 Case에서 수행하는 Routine(including Memory Checking)

이때, 앞서 설명한 바와 같이 System Call에선 Invalid User Memory Access를 체크하는 것이 중요하다. 따라서, **is_user_vaddr()** 함수를 이용한 Memory Check Routine을 마련하는 것이 중요하다. **is_user_vaddr()** 함수가 Virtual Address Mapping 여부와 User Area 판단 등을 수행하므로, 우리는 여기에 OR로서, 넘겨 받은 Pointer가 NULL인지 추가로 확인하기만 하면 된다. 이러한 체크 루틴을 본인은 아래와 같이 **Macro Function**의 형태로 구축하였다. 위에서 설명한, Systemic한 Parameter 추출도 Macro로 구현하였다. 아래를 보자. (**pintos/src/userprog/syscall.h** 파일 내부)

```
#define ARG_ADDR(k) ((uint8_t*)esp + 4*k)
// ARG_ADDR: returns the address of given passed-by-syscall arguments

#define ARG(k, type) *(type*)(ARG_ADDR(k))
// ARG: returns the value of given pointer, by casting of 'type'

/* Macros for readability of codes */
#define CHECK_ROUTINE(vaddr) if (vaddr == NULL || is_user_vaddr(vaddr) == false) exit(-1);
// CHECK_ROUTINE: checks if an argument is in the user address space
//                 with pre-provided 'is_user_vaddr' function. And, check NULL also!
#define USER_ADDR_CHECK(param_num) for(int i=1;i<param_num;i++){CHECK_ROUTINE(ARG_ADDR(i))}
// USER_ADDR_CHECK: checks all the parameters that a system call needs
//                 with consequitively calling 'CHECK_ROUTINE' macro above!
```

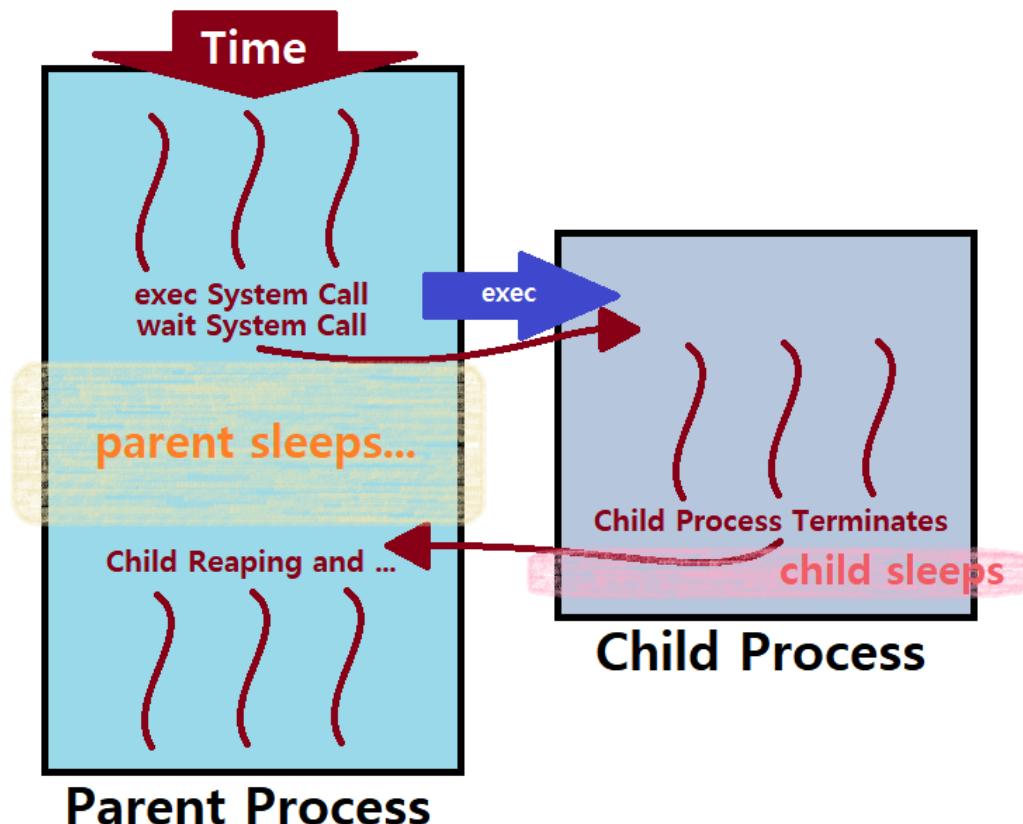
이들을 이용해 각 Case에서 메모리 체크, eax Register에 Return Value 저장, System Call 처리 함수 호출 등의 작업을 수행한다. (후술)

◆ halt, exit, read, write System Calls

이들은 pintOS Manual을 참고하여 어렵지 않게 구현할 수 있다. halt의 경우 devices/shutdown.c에 정의된 shutdown_power_off() 함수를, exit의 경우 threads/thread.c에 정의된 thread_exit() 함수를, read와 write의 경우 PDF에서 명시한 input_getc(), putbuf() 함수를 통해 간단하게 구현할 수 있다. 자세한 구현 디테일은 후술한다.

◆ Synchronization이 필요한 wait, exit System Calls

이 부분이 가장 까다로운 부분이다. 하지만, 구현을 마친 지금에서 보았을 땐 간단하기도 하다. 본인의 경우, wait과 exec의 실질적인 처리 작업은 **pintos/src/userprog/process.c**의 process_wait()과 process_execute() 함수에 위탁했으며, 이 과정에서 **pintos/src/threads/synch.h**에서 제공하는 Semaphore를 이용해 Parent Process와 Child Process의 동기화를 만들어주는 것이 중요하다. 동기화 과정은 아래 Code-Level 설명 항목에서 좀 더 자세히 설명하겠지만, 우선 간략히 소개하자면, 아래와 같다. 세마포어를 이용한 전통적인 동기화 기법을 사용해 구현할 수 있다.



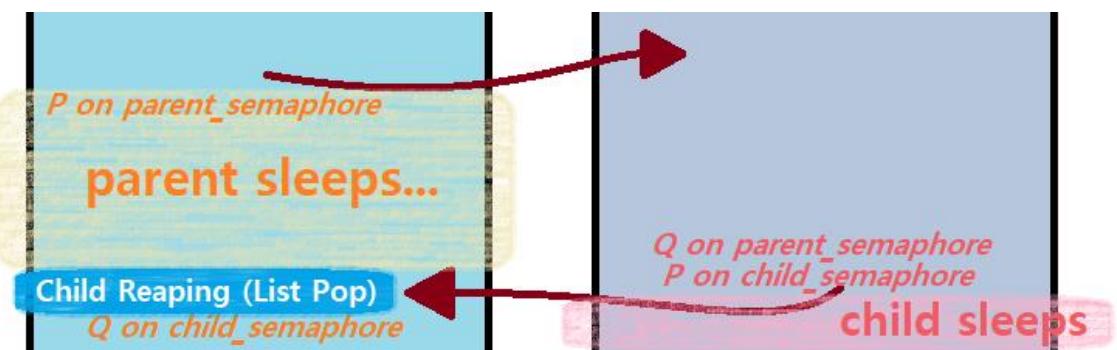
위 그림에서 주목해야 할 부분은 Parent의 Sleep 부분이 아니다. 바로 Child의 Sleep 부분이다. 이 부분의 발견이 꽤나 Tricky했는데, 이유는 다음과 같다.

"Parent(Current) Process가 exec System Call을 통해 Child Process를 생성한다. 이어서, Parent Process에서 wait System Call을 통해 Child Process의 종료를 기다린다(잠을 잔다). 왜? Child Reaping을 위해서 말이다. 이 부분은 Semaphore로 가볍게 수행할 수 있다."

"이때, 중요한 부분은, Child Process가 종료되고 나서이다. Child가 수행을 마치면, Parent를 잠자게 한 Semaphore에 대한 Q Operation을 통해 잠을 깨운다. 그리고 나서, Child는 바로 종료될 것이다. 그런데 이때, Parent에서는 Child Reaping, 즉, Child를 Data Structure(주로 List)에서 제거해야 한다. 이 제거 과정이 과연 Child의 종료보다 먼저 일어날까? 결코 그렇지 않을 것이다. Child가 Reaping보다 먼저 메모리 해제될 경우, 이 Parent-Child Fork-Execution 관계는 분명 문제가 생길 것이다."

"따라서, 우리는 Child Process가 수행을 마친 직후에 Child의 '진짜 종료' 직전에, Parent를 깨우고, 동시에 Child를 잠시 동안 잠자게 해야 한다. Parent에서 효과적인 Child Reaping을 하기 위해서 말이다. Reaping이 마무리되면, 다시 Child를 깨워서 진정하게 Child가 종료되는 것이다."

이 과정을 다시 한 번 묘사하면 아래와 같다. Semaphore 관점에서 이해하자. 이 발견이 어려울 뿐, 상당히 기초적인 Semaphore 적용임을 알아챌 수 있다.



◆ Page Fault Handler에서의 Invalid Memory Checking

User Program이 Kernel Address Space를 접근 시 Page Fault가 발생한다. 이는 이미 pintOS에서 제공되어 있다고 했다. 이때, 우리는 Page

Fault 처리 부분에도 메모리 체크 과정을 추가해, Page Fault가 일어날 시 User Program이 알아서 -1 Exit Status로 죽을 수 있도록 만들어주어야 한다. 이는 아주 간단히 해결할 수 있겠다. **pintos/src/userprog** 디렉토리의 **exception.c**에서 이를 수행할 수 있다.

✓ Additional System Calls Implementation

Additional System Call은 fibonacci와 max_of_four_int로, 프로젝트1 안 내 PDF에 그 요구사항이 적혀 있다. 요구사항대로 간단하게 작성하면 된다. 단, 이때, **pintos/src/lib/syscall-nr.h**에 New System Call Number를 등록하는 것은 주목할만하다.

이어, 이 System Call들을 사용하는 additional.c Program을 작성하면 되겠다. additional Program에선, Command-Line에서 입력 받은 '문자열' 입력을 실제 Number로 바꿔주는 것이 필요하다.

(atoi의 필요성)

Load Function

Allocate Page

Command Line
Parsing

Open File
Success?

No

Read Header
Success?

No

Stack Alloc
Success?

No

Stack Set-Up
(Args Pushing)

Free Page

Thread Exit

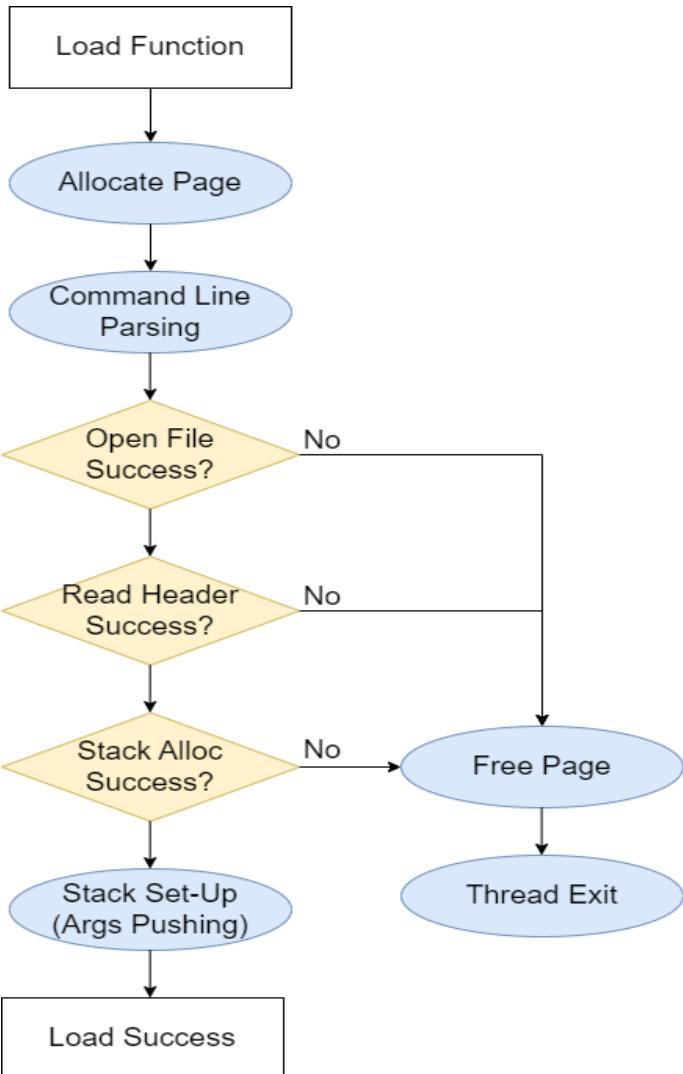
Load Success

4. 연구 결과

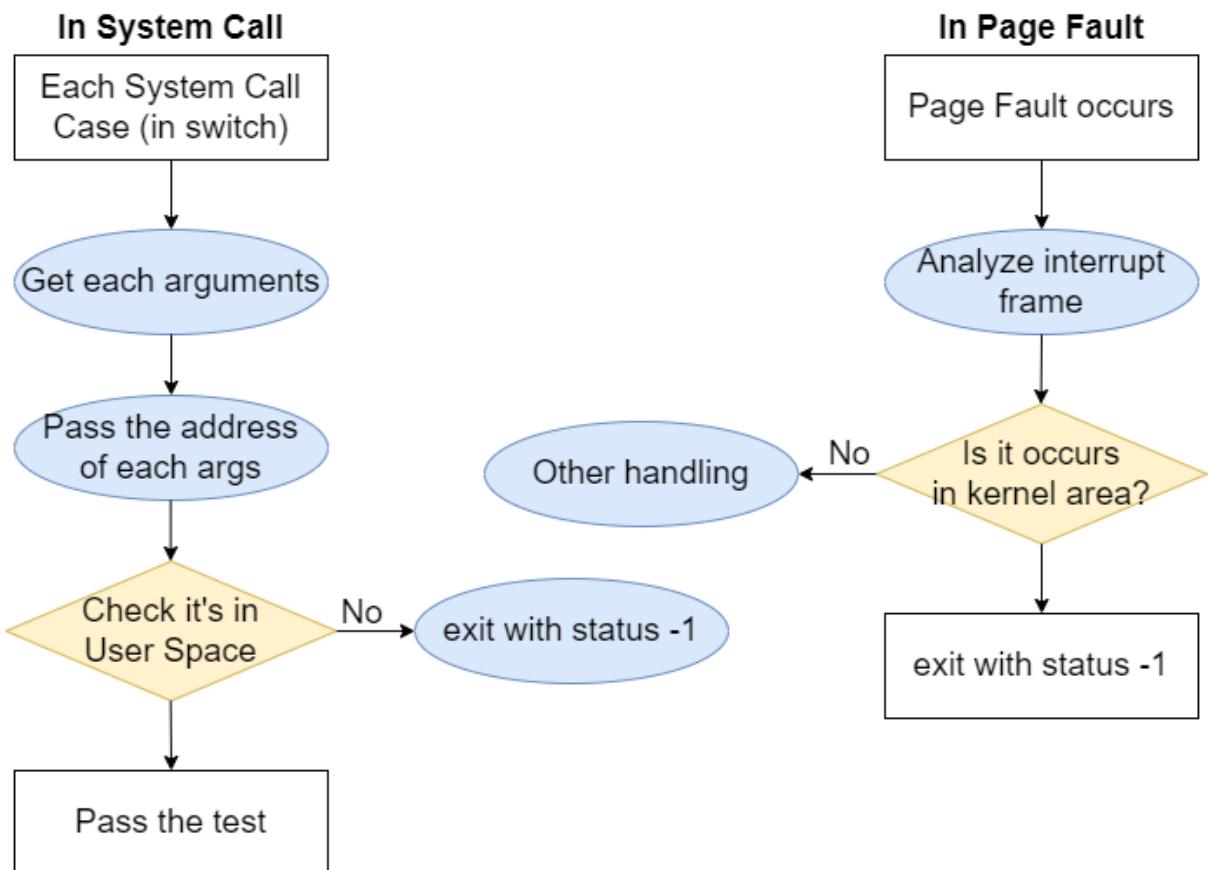
A. Flow Chart

1. Argument Passing

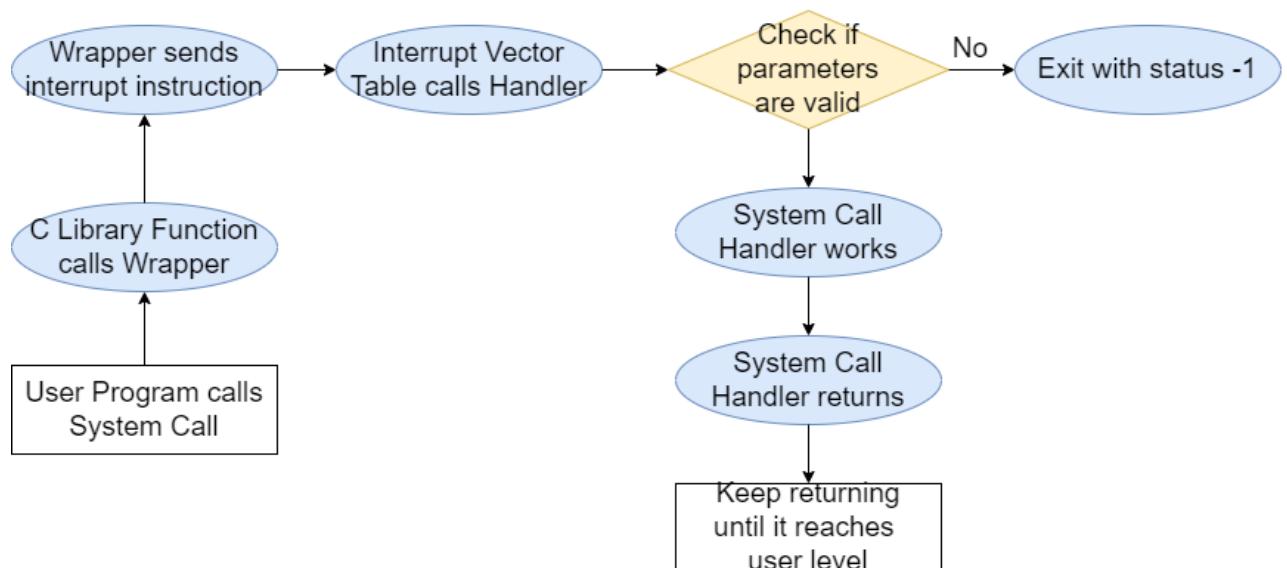
(우측 Flow Chart) ->



2. User Memory Access



3. System Calls



(공간상의 이유로 Flow Chart를 눌혀놓았다)

B. 제작 내용

1. Argument Passing

A. Command-Line Parsing

지속적으로 언급한 바와 같이, Argument Passing은 크게 Command Parsing, 그리고 Stack Set-Up(Arguments Pushing)으로 구분할 수 있다. PDF 안내에 따라, 두 단계를 모두 pintos/src/userprog/process.c 파일의 load 함수에서 구현하였다. 아래를 보자.

```
/* Loads an ELF executable from FILE_NAME into the current thread.
   Stores the executable's entry point into *EIP
   and its initial stack pointer into *ESP.
   Returns true if successful, false otherwise. */
bool
load(const char* file_name, void (**eip) (void), void** esp)
{
    struct thread* t = thread_current();
    struct Elf32_Ehdr ehdr;
    struct file* file = NULL;
    off_t file_ofs;
    bool success = false;
    char *fn_ptr = file_name;                      // pointer to file name
    char *argv[MAX_ARGS];                          // argument list
    int argc, i;                                  // argument count
    size_t total_len, temp_len;                    // variables for stack pushing

    /* Allocate and activate page directory. */
    t->pagedir = pagedir_create();
    if (t->pagedir == NULL)
        goto done;
    process_activate();

    /* Parse the entire command line into argument tokens */
    argc = parse(fn_ptr, argv);                   // parse file name
    strlcpy(thread_name(), argv[0], strlen(argv[0]) + 1); // for 'exit' syscall
```

- ◆ fn_ptr 변수는 'File Name Pointer'의 약자로, load 함수가 넘겨 받은 file_name 문자열을 가리킨다. load 함수 호출 시점에서의 file_name 문자열은 전체 Command-Line을 담고 있다. fn_ptr을 도입한 이유는, const 선언이 이루어진 file_name을 직접적으로 접근해 변형시키기 위함이다. 이어서 바로 소개할 parse 함수의 결과로서, file_name은 Command-Line의 파일명만을 가리킬 것이다.

- ◆ Character type pointer array인 argv는 당연 Argument 문자열들을 담게 될 배열이며, argc는 Argument Count, total_len 및 temp_len은 Stack Set-Up 과정에서 사용될 변수들이다.
- ◆ parse 함수 수행 직후 추출된 File Name을 thread_name() 반환 문자열에 넣어주고 있음도 주목하자. 상술했듯, 이는 exit System Call에서 printf() 문에서 사용하기 위함이다. 참고로, thread_name()은 thread 구조체의 name 문자열을 가리킨다.

```

/* Function that parses input string, and makes an array
   consists of tokens. */
int
parse (char* buf, char** argv)
{
    char* delim;                                // points to space delimiter
    int argc;                                    // number of arguments

    if (buf[strlen(buf) - 1] == '\n')
        buf[strlen(buf) - 1] = ' ';              // replace trailing '\n' with space
    else buf[strlen(buf)] = ' ';

    while (*buf && (*buf == ' '))
        buf++;

    argc = 0;
    while ((delim = strchr(buf, ' ')))          // build the argv list
    {
        argv[argc++] = buf;                      // parse each token
        *delim = '\0';

        buf = delim + 1;
        while (*buf && (*buf == ' '))
            buf++;
    }
    argv[argc] = NULL;

    return argc;
}

```

- ◆ 넘겨 받은 문자열에서 Space Character를 기준으로 Token들을 뽑아내 Argument Array를 구축하는 함수이다. Token 사이에 복수의 Space가 있더라도 문제 없이 처리한다. 최종 반환 값으로는 Argument Count를 반환하며, 이 Code는 과거 System Programming Shell Project에서 사용했던 Code를 약간 변형한 Code이기도 하다. (개인적인 의견으로) strtok보다 이 함수가 더 안정적이라 판단해 이 함수를 사용한다.
- ◆ 결과적으로 Argument들이 문제 없이 Parsing되어 argv Array에 담기게 된다.

B. Stack Set-Up (Arguments Pushing)

앞선 항목에서 소개한 S_EXPAND, S_SETVAL Macro를 기억한 채로 아래의 Code를 보자.

```
/* Push(argc) arguments into stack, by modifying esp pointer */
total_len = 0;
for (i = argc - 1; i >= 0; i--)
{
    temp_len = (strlen(argv[i]) + 1);           // get the length of argument
    total_len += temp_len;                      // accumulate (for alignment)

    S_EXPAND(temp_len);                         // expand stack
    memcpy(*esp, argv[i], temp_len);            // push each arg into stack
    argv[i] = *esp;                            // argv[i] stores slot addr
}

temp_len = 4 - (total_len % 4);                /* Word Alignment for 80x86 */
if (temp_len != 4)
{
    while (temp_len--)
    {
        S_EXPAND(1);
        memset(*esp, 0, 1);                    // push paddings into stack
    }
}

for (i = argc; i >= 0; i--)                  // just follow the stack figure
{                                              //   on the proj pdf and manual
    S_EXPAND(4);
    if (i == argc) S_SETVAL(0);
    else S_SETVAL((uint32_t)argv[i]);
}

S_EXPAND(4); /**/ S_SETVAL((uint32_t)((uint8_t*)(*esp) + 4)); // argv addr
S_EXPAND(4); /**/ S_SETVAL(argc);                                // argc
S_EXPAND(4); /**/ S_SETVAL(0);                                  // ret addr
// hex_dump((uintptr_t)*esp, *esp, PHYS_BASE - (uintptr_t)*esp, true);
```

- ◆ S_EXPAND, S_SETVAL Macro를 사용하였기 때문에 코드 가독성이 상당히 높음을 알 수 있다. (개인적인 의견)
- ◆ 이 Stack Set-Up 과정에서 가장 Tricky했던 부분은 Pointer에 Data를 Size에 맞춰 정확하게 집어넣는 과정이었으며, 이를 위해 다 차례의 Type Casting을 시도했다. esp Pointer가 Double Pointer이기 때문에 Asterisk가 붙은 *esp는 Pointer이고, 해당 Single Pointer를 uint32_t, 즉, 4Bytes의 형 태로 Casting한다. esp는 void Type이기 때문이다.

- ◆ 정확한 메모리 복사를 위해 memcpy 함수를 사용하고 있으며, Word Alignment 시엔 memset을 이용해 1Byte 단위로 일일이 0 Value를 기입하고 있음을 주목하자. 전체적으로 PDF에서 소개한 Stack Figure를 착실하게 표현하고자 노력했음을 강조한다.

2. User Memory Access

User Memory Access를 위해 구축한 Macro Function에 대해 앞서 소개한 바 있다. 해당 Macro를 기억한 채로 아래의 코드를 보자. System Call Handler 내부의 개별 Case Routine 중 하나이다. 이처럼, 매 Case에서 User Program이 넘긴 Data의 Valid Test를 진행함을 주목하자. (pintos/src/userprog/syscall.c)

```
case SYS_READ: /* Read from a file. */
    USER_ADDR_CHECK(3);
    f->eax = read(ARG(1, int), ARG(2, void*), ARG(3, unsigned));
    break;
```

USER_ADDR_CHECK에 넘긴 3이란 숫자는, System Call Handler가 바라보고 있는 esp Pointer로부터 처음 4 Bytes인 System Call Number Data를 제외하고, 그 이후의 총 12바이트, 4바이트 기준 총 3개의 Parameters에 대해, 각 Parameter의 Address를 모두 Validation한다는 것이다. (Macro를 보면 알 수 있음)

한편, pintos/src/userprog/exception.c의 page_fault() 함수 내에도 아래와 같은 Code를 추가해 User Memory Access Checking을 진행한다. 이유는 앞서 충분히 설명했기에 생략한다.

```
/* Check the cause of page fault is from
   accessing the kernel area */
if (is_kernel_vaddr(fault_addr) == true ||
    user == false) exit(-1);
```

3. System Calls

System Call Handler에 대한 설명이 곧 System Calls Implementation에 대한 설명이다. 아래는 pintos/src/userprog/syscall.c에 구현된 Handler의 모습이다.

```

static void
syscall_handler (struct intr_frame *f UNUSED)
{
    uint32_t *esp = f->esp;
    uint32_t sys_num = *esp;

    /* System Call Table: Call a corresponding routine
       to perform system function requested by user program */
    switch (sys_num)^M
    {^M
        case SYS_HALT:           /* Halt the operating system. */
            halt();
            break;

        case SYS_EXIT:          /* Terminate the process. */
            USER_ADDR_CHECK(1); // Check whether it's in user space
            exit(ARG(1, int)); // Use C-Macro for readability
            break;

        case SYS_EXEC:           /* Start another process. */
            USER_ADDR_CHECK(1);
            f->eax = exec(ARG(1, char*));
            break;

        case SYS_WAIT:           /* Wait for a child process to die. */
            USER_ADDR_CHECK(1);
            f->eax = wait(ARG(1, pid_t));
            break;

        case SYS_READ:            /* Read from a file. */
            USER_ADDR_CHECK(3);
            f->eax = read(ARG(1, int), ARG(2, void*), ARG(3, unsigned));
            break;

        case SYS_WRITE:           /* Write to a file. */^M
            USER_ADDR_CHECK(3);^M
            f->eax = write(ARG(1, int), ARG(2, void*), ARG(3, unsigned));^M
            break;^M

        case SYS_FIBONACCI:      /* Calculate the N-th fibonacci number */^M
            USER_ADDR_CHECK(1);^M
            f->eax = fibonacci(ARG(1, int));^M
            break;^M

        case SYS_MAXOFFOUR:      /* Calculate the maximum among four integers */^M
            USER_ADDR_CHECK(4);^M
            f->eax = max_of_four_int(ARG(1, int), ARG(2, int), ARG(3, int), ARG(4, int));^M
            break;^M

        // Only 8 system calls are implemented for now (Project 1 (Sogang OS)) ^M
        default: break;^M
    }
}

```

switch문은 Handler 내부에 논리적으로 구현된 System Call Table이고, 각 Case Routine에 대한 설명은 바로 앞 항목에서 진행했으니 생략한다. 이후 프로젝트 Phase에서 구현할 예정인 나머지 System Call에 대한 항목은 코드 내에 따로 마련해두지 않았다.

A. System Call Routines

syscall.c에 정의한 System Call Routine, 즉, 실질적으로 System Call 기능을 수행하는 함수들에 대해 소개한다. Handler는 이들을 호출해 User Program에서 기능을 제공하는 것이다. 이때, exit, wait, exec의 경우 단순한 외부 함수 호출을 통해, 외부 코드에 그 기능을 위탁했음도 주목하자.

```
/* Halt routine: it terminates pintOS via shutdown func */
void
halt (void)
{
    shutdown_power_off();
}

/* Exit routine: simply stores an exit status of running thread,
   and calls 'thread_exit'. 'thread_exit' will do the main job! */
void
exit (int status)
{
    printf("%s: exit(%d)\n", thread_name(), status);
    thread_current()->exit_status = status; // record the exit status
    thread_exit();
}

/* Exec routine: simply calls 'process_execute'.
   Again, 'process_execute' will do the main 'execution' job! */
tid_t
exec (const char* cmd_line)
{
    return process_execute(cmd_line);
}

/* Wait routine: simply calls 'process_wait' */
int
wait (tid_t pid)
{
    return process_wait(pid);
}

/* Read routine: reads 'size' bytes from the standard input and
   returns the read bytes. It uses 'input_getc' func from 'input.h' */
int
read (int fd, void* buffer, unsigned size)
{
    unsigned byte_cnt = 0, i;
    char c;

    if (fd == 0) /* STDIN_FILENO */
    {
        for (i = 0; (i < size) && ((c = input_getc()) != '\0'); i++)
        {
            *((char*)buffer) = c; // read per character
            buffer = (char*)buffer + 1;
            byte_cnt++;
        }
        *((char*)buffer) = '\0';
    }
    else byte_cnt = -1; // exception

    return byte_cnt;
}
```

```

/* Write routine: writes 'size' bytes from 'buffer' to the standard
   output and returns the written bytes count. */
int ^M
write (int fd, const void* buffer, unsigned size) ^M
{^M
    unsigned byte_cnt = 0;^M
^M
    if (fd == 1) /* STDOUT_FILENO */^M
    {^M
        putbuf(buffer, size);           // writes as many bytes as possible
        byte_cnt += size;^M
    }
    else byte_cnt = 0;

    return byte_cnt;
}

```

이때, read System Call에선 input_getc()라는 함수를 이용해 한 문자 단위로 읽기를 진행하고 있고, write System Call에선 putbuf()라는 함수를 통해 통째로 쓰기를 진행하고 있음에 주목하자. pintos/src/lib/kernel/console.c에 정의된 putbuf() 함수는 아래와 같다.

```

/* Writes the N characters in BUFFER to the console. */
void
putbuf (const char *buffer, size_t n)
{
    acquire_console ();
    while (n-- > 0)
        putchar_have_lock (*buffer++);
    release_console ();
}

```

Buffer를 통으로 비우고 있다. 이렇게 현재 프로젝트 Phase 상에서 구현해야 하는 임시 write System Call에선 내부가 위와 같이 구현된 것이다.

input_getc() 역시 확인해보자. pintos/src/devices/input.c에 정의된 내용이다.

```

/* Retrieves a key from the input buffer.
   If the buffer is empty, waits for a key to be pressed. */
uint8_t
input_getc (void)
{
    enum intr_level old_level;
    uint8_t key;

    old_level = intr_disable ();
    key = intq_getc (&buffer);
    serial_notify ();
    intr_set_level (old_level);

    return key;
}

```

함수 Return Value의 Type이 uint8_t, 즉, 1 Byte Character이다. 따라서 우리는 이 함수가 한 문자 단위로 Keyboard 입력을 받아들이는 함수임을 알 수 있고, 나아가 위와 같이 read를 구현할 수 있는 것이다.

B. exec System Call의 구현

exec System Call은 위와 같이 System Call Routine에서 동일 디렉토리 내의 process.c 파일에 정의된 process_execute()를 호출하기만 한다. 따라서, process.c의 process_execute()를 들여다보자.

```
tid_t
process_execute (const char *file_name)
{
    char* fn_copy;
    tid_t tid;
    char temp_name[MAX_ARGS];           // 128 is enough for length of file name
    int cmd_len = strlen(file_name) + 1, i, idx = 0;

    /* Make a copy of FILE_NAME.
       Otherwise there's a race between the caller and load(). */
    fn_copy = palloc_get_page (0);
    if (fn_copy == NULL)
        return TID_ERROR;
    strlcpy(fn_copy, file_name, PGSIZE);

    /* Check whether there exists a file named 'file_name' in file system
       It handles the execution-with-missing-arguments situation*/
    for (i = 0; file_name[i] == ' '; i++);           // ignore the leading spaces
    for (; i < cmd_len; i++)
    {
        if (file_name[i] == ' ' ||           // take only the name of file
            file_name[i] == '\0') break;
        temp_name[idx++] = file_name[i];
    }
    temp_name[idx] = '\0';

    if (filesys_open(temp_name) == NULL)           // if file does not exist, exit
        return TID_ERROR;                         // with -1, without creating thread

    /* Create a new thread to execute FILE_NAME. */
    tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
    if (tid == TID_ERROR)
        palloc_free_page (fn_copy);

    return tid;
}
```

중간의 문자열 처리 부분을 제외하고는 모두 pintOS에서 미리 제공한 코드이다. 이 함수는 Linux 기준 fork의 역할을 수행한다고 볼 수 있다. thread_create() 함수로 Thread를 생성하고, 해당 Thread가 start_process()를 수행하도록 만들어준다.

start_process() 설명에 앞서, 참고로, process_execute() 중간 부분의 문자열 처리는 Program Execution 시에 '애초에 존재하지 않거나 Open할 수 없는 File Name'을 전달한 경우를 Handling한다. 그러한 경우, -1을 반환해 Program을 죽이는 것이다.

다시 돌아와 설명을 이어간다. start_process() 함수를 보자. 이 함수 역시 pintOS에서 기본적으로 제공한다.

```
/* A thread function that loads a user process and starts it
running. */
static void
start_process (void *file_name_)
{
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;

    /* Initialize interrupt frame and load executable. */
    memset (&if_, 0, sizeof if_);
    if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
    if_.cs = SEL_UCSEG;
    if_.eflags = FLAG_IF | FLAG_MBS;
    success = load (file_name, &if_.eip, &if_.esp);

    /* If load failed, quit. */
    palloc_free_page (file_name);
    if (!success)
        thread_exit ();

    /* Start the user process by simulating a return from an
interrupt, implemented by intr_exit (in
threads/intr-stubs.S). Because intr_exit takes all of its
arguments on the stack in the form of a `struct intr_frame',
we just point the stack pointer (%esp) to our stack frame
and jump to it. */
    asm volatile ("movl %0, %%esp; jmp intr_exit" :: "g" (&if_) : "memory");
    NOT_REACHED ();
}
```

Thread에 File Name이 가리키는 Program을 덮어씌우고 있음을 보자. Linux 기준으로 exec System Call의 역할을 수행하는 것이다. 즉, pintOS의 exec System Call은 'process_execute -> thread_create -> start_process -> load'의 과정을 통해 Linux 기준의 'fork -> exec'을 한 번에 수행하는 것이다. 아무튼, 구현적으로 보았을 때, 본인이 변화시킨 부분은 거의 없다. 이렇게 해서 exec System Call은 간단히 구현할 수 있다.

C. wait & exit System Call의 구현 (Synchronization)

wait System Call과 exit System Call 역시 각각 process.c의 process_wait(), thread.c의 thread_exit()에게 자신의 역할을 위탁한다. pintos/src/threads/thread.c 파일의 thread_exit() 함수에는 다음과 같은 처리가 있기 때문에, 사실상 exit은 process_exit()을 호출하는 것이라 볼 수 있다.

```
void  
thread_exit (void)  
{  
    ASSERT (!intr_context ());  
  
#ifdef USERPROG  
    process_exit ();  
#endif
```

이어, 앞서 시각 자료를 통해 다 차례 설명했던 Synchronization을 적용하자. Semaphore를 사용할 터인데, Semaphore는 각 Thread 별로 작동해야 하므로 thread structure에 존재해야 한다. Parent Semaphore와 Child Semaphore를 두자.

한편, 각 Process는 자신이 생성하는 복수의 Child를 하나하나 기억해야 한다. 그래야 Reaping을 할 수 있기 때문이다. 따라서, Children에 대한 정보를 기억하기 위해 thread structure를 Element Type으로 하는 List도 생성한다.

이러한 Thread 초기 Setting 작업은 threads/thread.h에서 수행한다. 아래와 같이 말이다. struct thread 내부이다. User Level의 Process에게만 해당되는 내용이므로 #ifdef USERPROG를 적고 있음에 주목하자.

그리고 무엇보다도, 이러한 Setting을 토대로 초기 Thread 생성 시 이들을 초기화해야 한다. 이는 threads/thread.c의 init_thread 함수에 추가하면 된다. 각각의 Semaphore를 생성하고, 각 Process에 대한 List도 생성 및 초기화한다. 아래 코드에서 list_push_back()을 주목하자.

```
#ifdef USERPROG
    /* Create a counting semaphore */
    /* for child & memory locking */
    sema_init(&(t->parent_lock), 0);
    sema_init(&(t->child_lock), 0);
    /* Initialize the children list */
    list_init(&(t->child_list));
    list_push_back(&(running_thread()->child_list), &(t->child_elem));
#endif
```

이제 본격적으로 wait System Call의 기능을 확인하자. userprog/process.c의 process_wait 함수이다.

```
int
process_wait (tid_t child_tid UNUSED)
{
    struct list *c_list = &(thread_current()->child_list);
    struct list_elem *iter; struct thread *entry;

    for (iter = list_begin(c_list);           // iterate the list of child processes
         iter != list_end(c_list);
         iter = list_next(iter))
    {
        entry = list_entry(iter, struct thread, child_elem); // for each entry

        if (entry->tid == child_tid)                      // if an entry is the target,
        {
            /* Make current(parent) process go to sleep */
            sema_down(&(entry->parent_lock));

            /* If parent wakes up and child goes to sleep, then
               remove the list entry of current child */
            list_remove(&(entry->child_elem));

            /* Tell the child process to wake up! */
            sema_up(&(entry->child_lock));

            return (entry->exit_status);                  // return the exit status
        }
    }
    return -1;
}
```

Current Process의 Child List를 순회하며, wait System Call의 Target Child 를 찾는다. 찾게 되면, Project 0에서 소개된 list_entry Macro를 통해 List Element를 뽑아내고, 해당 Element(Child Thread)에 대해, Parent Semaphore 에 대한 P Operation으로 Current(Parent) Process를 잠자게 만든다.

잠시 멈추어, exit System Call을 보자. 같은 Code 내의 process_exit() 함수 가 이를 위탁 처리한다. 코드는 아래와 같다. 맨 하단 두 개의 명령을 제외 하곤 pintOS에서 제공된 그대로이다.

```
/* Free the current process's resources. */
void
process_exit (void)
{
    struct thread *cur = thread_current ();
    uint32_t *pd;

    /* Destroy the current process's page directory and switch back
       to the kernel-only page directory. */
    pd = cur->pagedir;
    if (pd != NULL)
    {
        /* Correct ordering here is crucial. We must set
           cur->pagedir to NULL before switching page directories,
           so that a timer interrupt can't switch back to the
           process page directory. We must activate the base page
           directory before destroying the process's page
           directory, or our active page directory will be one
           that's been freed (and cleared). */
        cur->pagedir = NULL;
        pagedir_activate (NULL);
        pagedir_destroy (pd);
    }

    /* When a current(child) process exits, make
       the sleeping parent process wake up! */
    sema_up(&(cur->parent_lock));

    /* And the current(child) process immediately
       and shortly sleeps for the list pop operation */
    sema_down(&(cur->child_lock));
}
```

자, Child Process가 열심히 수행되었고, 자신의 일을 끝마쳐 종료될 때가 되었다고 하자. exit System Call을 호출해 종료하고자 할 것이다. 이때, 쭉 수행되다가 sema_up(&(cur->parent_lock)); 명령, 즉, Parent Semaphore에 대한 Q Operation을 맞이한다. 바로 직후 Child Semaphore에 대한 P

Operation이 수행되어 Child Process가 잠자게 된다.

잠에서 깬 Parent Process는 아래의 코드 부분을 수행한다.

```
/* If parent wakes up and child goes to sleep, then
   remove the list entry of current child */
list_remove(&(entry->child_elem));

/* Tell the child process to wake up!
sema_up(&(entry->child_lock));

return (entry->exit_status);           // return the exit status
```

Child Process에 대한 Data Structure Information을 Reaping한다. PCB를 제거한다고 볼 수도 있겠다. 이 Reaping 작업을 마친 Parent는 다시 Child Semaphore에 대한 Q Operation을 수행해 Child가 이제는 진짜 죽을 수 있도록 해준다.

이것이 바로 **wait & exit System Call**이 동작하는 방식이다.

4. Additional System calls

Additional System Call인 fibonacci와 max_of_four_int는 아래와 같이 간단하게 구현하자. 이 System Call의 구현을 위해 pintos/src/lib/syscall-nr.h에 System Call Number를 등록하는 것이 먼저 선행된다.

```
SYS_CLOSE,                      /* Close a file. */
SYS_FIBONACCI,                  /* Calculate the N-th fibonacci number */
SYS_MAXOFFOUR,                 /* Calculate the maximum among 4 integers */

/* Project 3 and optionally project 4. */
SYS_MMAP,                       /* Map a file into memory. */
```

이어, max_of_four_int System Call의 경우 Parameters가 System Call Number 제외 4개나 되므로 이를 커버할 수 있는 Wrapper도 정의해야 한다. 이 Wrapper는 아래와 같이 pintos/src/lib/user/syscall.c에 정의된다. 동일한 파일에 두 New System Calls에 대한 C Library Function도 정의한다.

```
int
fibonacci(int n)
{
    return syscall11(SYS_FIBONACCI, n);
}

int
max_of_four_int(int a, int b, int c, int d)
{
    return syscall14(SYS_MAXOFFOUR, a, b, c, d);
}
```

```

/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, ARG2,
   and ARG3, and returns the return value as an `int'. */
#define syscall4(NUMBER, ARG0, ARG1, ARG2, ARG3)
    ([
        int retval;
        asm volatile
            ("pushl %[arg3]; pushl %[arg2]; pushl %[arg1]; "
             "pushl %[arg0]; pushl %[number]; int $0x30; "
             "addl $20, %%esp"
            : "=a" (retval)
            : [number] "i" (NUMBER),
              [arg0] "r" (ARG0),
              [arg1] "r" (ARG1),
              [arg2] "r" (ARG2),
              [arg3] "r" (ARG3)
            : "memory");
        retval;
    ])

```

C Library Function, Wrapper 순이다. Wrapper의 경우 int \$0x30;이라는 Interrupt Instruction 이후 다시 esp Pointer를 조정하고 있음에 주목하자.

```

/* Fibonacci routine: returns N-th value of fibonacci sequence.
   It produces sequence with simple iterative algorithm. */
int
fibonacci (int n)
{
    int f = 0, f1 = 1, f2 = 0;
    int i;

    if (n == 0) return 0;
    if (n == 1) return 1;
    for (i = 2; i <= n; i++)
    {
        f = f1 + f2;
        f2 = f1;
        f1 = f;
    }

    return f;
}

/* Max_of_four_int routine: returns the maximum among arbitrary decimals
   It uses simple bubble-sort to figure out the maximum. */
int
max_of_four_int (int a, int b, int c, int d)
{
    int arr[4], temp;
    int i, j;

    arr[0] = a; arr[1] = b; arr[2] = c; arr[3] = d;
    for (i = 0; i < 3; i++)
        for (j = i + 1; j < 4; j++)
        {
            if (arr[i] > arr[j])
            {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

    return arr[3];           // ascending order sorting
}

```

userprog/syscall.c에 정의된 System Call Routine들은 다음과 같다.

Fibonacci System Call의 경우 Iterative Method로 Naive하게 구현하였다.

max_of_four_int System Call의 경우 간단한 Bubble Sort를 적용해 구현하고 있음을 알 수 있다.

두 System Call 모두 여타 다른 System Call과 마찬가지로 정상 작동한다. 이는 아래와 같은 additional.c Program을 통해 증명된다.

```

/*
 * additional.c

 Prints the result of 'fibonacci' system call to the console,
 using arg1 as parameter.
 Prints the result of 'max_of_four_int' system call to the console,
 using arg1-4 as parameters. */

#include <stdio.h>
#include <syscall.h>

static int atoi(const char* str);

int
main(int argc, char* argv[])
{
    if (argc != 5)
    {
        printf("%s: open failed\n", argv[0]);
        exit(-1);
    }

    printf("%d %d\n", fibonacci(atoi(argv[1])), max_of_four_int(
        atoi(argv[1]), atoi(argv[2]), atoi(argv[3]), atoi(argv[4])));

    return 0;
}

/* My atoi function that produces a number indicated by string */
int
atoi(const char* str)
{
    int num = 0, i = 0;

    while (str[i] && (str[i] >= '0' && str[i] <= '9'))
    {
        num = num * 10 + (str[i] - '0');
        i++;
    }

    return num;
}

```

굉장히 간단하게 구현할 수 있다. atoi 함수의 경우, 같은 Program 내에서 자그맣게 구현하여 사용하였다. 참고로, 이러한 additional Program이 pintOS 내에서 User Program으로 취급 받기 위해선, [pintos/src/examples](#) 디렉토리 내부의 Makefile을 수정해야 한다.

C. 시험 및 평가 내용

- fibonacci 및 max_of_four_int 시스템 콜 수행 결과를 캡처하여 첨부.

```
cse20171643@cspro9:~/pintos/src/userprog$ pintos --filesys-size=2 -p ../../examples/additional -a additional -- -f -q run 'additional 10 20 62 40'
Copying ../../examples/additional to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/31aG7d4Dmr.dsk -m 4 -net none -nographic -monitor null
WARNING: Image format was not specified for '/tmp/31aG7d4Dmr.dsk' and probing guessed raw.
        Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.
SeaBIOS (version 1.13.0-1ubuntu1.1)
Booting from Hard Disk...
PpiLoo hhddaa1
1
LLooaaddiinngg.....
Kernel command line: -f -q extract run 'additional 10 20 62 40'
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 381,337,600 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 197 sectors (98 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 104 sectors (52 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'additional' into the file system...
Erasing ustar archive...
Executing 'additional 10 20 62 40':
55 62
additional: exit(0)
Execution of 'additional 10 20 62 40' complete.
Timer: 62 ticks
Thread: 3 idle ticks, 59 kernel ticks, 0 user ticks
hda2 (filesys): 67 reads, 212 writes
hda3 (scratch): 103 reads, 2 writes
Console: 900 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
cse20171643@cspro9:~/pintos/src/userprog$
```

additional Program이 위와 같이 문제 없이 돌아가고 있음을 알 수 있다.

본인의 프로젝트1 User Program (1)의 최종 pintOS의 make check 결과는 아래와 같다.

```
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
```

```
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
```

**이렇게 해서 pintOS Project 1 User Program (1)을 마친다. 이어지는 Project Phase도
열심히 수행하겠다.**