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

담당 교수 : 김영재

조 / 조원 : 만점받으면행복하조 / 박민준(20212020)

개발 기간 : 9/23 ~ 10/2

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* 본 프로젝트의 주요 목표는 Pintos 운영 체제 내에서 사용자 프로그램을 실행하고 관리할 수 있는 기능을 구현하는 것이다.
* 가장 먼저 사용자 프로그램이 실행될 때, 명령행 인자를 효과적으로 parsing하여 해당 인자들을 스택에 올릴 수 있는 기능을 구현한다. 인자의 수와 내용을 정확히 스택에 저장하고, 프로그램에서 이를 참조할 수 있도록 주소를 설정한다. 이를 통해 사용자는 명령행에서 인자를 제공하여 프로그램의 동작을 제어할 수 있게 된다.
* 이 외에도 Pintos의 사용자 프로그램이 시스템 resource를 사용하기 위해 필요한 다양한 시스템 호출과 handler 함수를 구현한다. 대표적으로 exec, exit, wait, read, write 등의 호출을 구현하여 파일 입출력 및 프로세스 관리 기능을 지원한다. 또한 프로젝트 요구 사항에 따라 fibonnaci 함수와 max\_of\_four\_int와 같은 사용자 정의 시스템 호출을 추가하여 프로그램의 기능을 확장한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

- file name에 대한 parsing 작업을 수행한 후 이를 스택에 쌓으며 저장하였다. hex\_dump 함수의 결과를 통해 스택에 정확히 어떤 데이터가 올라갔는지 확인할 수 있으며 그 결과는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

[사진1. “echo x”에 대한 hex\_dump 결과]

- 사진을 보면 “echo x”라는 명령어가 실행될 때, “x”와 “echo”가 순서대로 스택에 저장된 것을 확인할 수 있다. 메모리 dump에서 echo는 65 63 68 6f 00으로 표현되며, 이는 16진수 ASCII code로 “echo”를 나타낸다. 마찬가지로 x는 78 00으로 나타나게 된다. 각 인수의 뒤에는 NULL이 포함되어 있어 문자열의 끝을 표시한다. 또한, kernel memory 영역의 시작 주소인 PHYS\_BASE(0xc0000000)를 기준으로 그 아래 부분은 kernel 영역이며 알 수 없는 값들이 들어 있다.

1. User Memory Access

- 운영 체제는 사용자가 접근할 수 있는 메모리와 접근할 수 없는 메모리를 구분하여 보호할 수 있어야 한다. userprog/syscall.c 파일에서 is\_user\_vaddr() 함수를 통해 사용자가 memory에 access할 때 전달되는 주소가 비정상적인지 아닌지를 확인하는 작업을 추가하였다. 즉, 현재 프로세스가 유효하지 않은 메모리 접근을 시도했다면 프로세스는 강제로 종료된다.

1. System Calls

- system call을 다루기 위한 syscall\_handler() 함수와 halt, exec, exit, wait, read, write 등의 system call 함수들, 그리고 fibonacci, max\_of\_four\_int와 같은 사용자 정의 시스템 함수를 직접 구현하였다. 이러한 시스템 호출들은 사용자 프로그램이 운영 체제와 상호작용할 수 있게 해주는 중요한 interface를 제공한다. 각 호출은 시스템 작동을 위해 필요한 특정한 기능을 수행하며, 성공적으로 실행되었을 때 적절한 값을 반환하여 사용자 프로세스가 다음 동작을 결정할 수 있도록 돕는다.

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

먼저, argc 변수를 사용하여 입력된 명령어에서 인자의 개수를 센다. 각 인자는 공백으로 구분되어 있으며, 이를 기준으로 count한다. 이후 argv 배열을 선언하고, 각 인자를 동적으로 할당하여 저장한다. 이 배열은 각각의 인자가 저장될 메모리 공간을 제공한다. 인자를 복사할 때는 strlcpy 함수를 사용하여 각 인자의 문자열을 argv 배열에 저장한다. 인자를 커널 스택에 쌓기 위해 스택 포인터 \*esp를 조정한다. 인자의 총 길이만큼 스택 포인터를 아래로 이동시키고, 이 위치에 각 인자를 복사한다. 인자를 스택에 복사하는 과정에서는 memcpy 함수를 사용하여 argv의 각 인자를 스택의 현재 위치에 복사한다. 그 후, 인자의 주소를 업데이트하여 나중에 사용할 수 있도록 한다. 스택 포인터는 4byte 정렬이 되도록 조정한다. 이를 위해 스택 포인터를 확인하고, 필요한 경우 추가적인 공간을 확보해 정렬한다. 마지막으로, argv 배열의 끝을 나타내기 위해 NULL 포인터를 스택에 push한다. 이 과정은 프로그램이 인자 리스트의 끝을 인식할 수 있도록 한다. 인자의 주소를 스택에 push하는 단계에서 argv 배열의 각 인자의 주소를 스택에 추가한다. 이 주소들은 이후 프로그램에서 인자를 사용할 때 참조된다. 최종적으로 인자의 개수인 argc와 argv의 주소를 스택에 push하여, 프로그램이 실행될 때 필요한 인자 정보가 모두 스택에 쌓이도록 한다.

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

invalid memory access는 사용자 프로세스가 접근할 수 없는 메모리 영역에 접근하려고 할 때 발생하는 상황을 의미한다. 이는 잘못된 메모리 주소를 참조하거나, 해당 프로세스가 접근할 권한이 없는 메모리 영역에 접근하려고 할 때 발생한다. Pintos는 사용자 프로세스에 대해 별도의 주소 공간을 할당한다. 이 주소 공간은 일반적으로 사용자 프로세스가 접근할 수 있는 User Memory 영역과 그렇지 않은 Kernel Memory 영역으로 나뉜다. 사용자 프로세스는 자신의 주소 공간 내에서만 메모리에 접근할 수 있으며, kernel memory 영역에 접근할 경우 invalid memory access가 발생하게 된다.

* + Invalid memory access를 어떻게 막을 것인지 설명

is\_user\_vaddr(const void \*vaddr) 함수를 통해 주어진 주소 vaddr이 사용자 주소 공간의 범위(프로세스가 접근 가능한 메모리 영역) 내에 있는지를 확인한다. 일반적으로 이 주소는 kernel memory 영역의 시작 주소(PHYS\_BASE)와 비교하여 결정된다. 해당 함수는 주소 접근 시도 전에 호출되어야 하며, 주소가 유효하다면 프로세스는 계속해서 메모리 작업을 수행할 수 있지만 유효하지 않은 주소에 접근하면 -1을 exit하여 프로세스를 종료시킨다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

System call은 user mode와 kernel mode 간의 경계를 허물며, user program이 운영 체제의 핵심 기능에 안전하게 접근할 수 있도록 한다. user mode에서는 프로그램이 hardware와 직접 상호작용할 수 없지만, system call을 통해 kernel mode로 전환하여 필요한 자원(file, memory 등)에 접근할 수 있다. 또한, kernel은 system call을 통해 user program의 요청을 처리하며, 이 과정에서 자원 관리를 수행하여 시스템의 효율성을 극대화한다. 즉, system call은 두 mode 간의 원활한 통신을 보장하고, 안전한 자원 관리를 가능하게 하기 때문에 반드시 필요한 기능이다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
    1. halt(): shutdown\_power\_off() 함수를 호출하여 pintos를 안전하게 종료한다. 해당 system call은 모든 process를 종료하고 전원을 끄는 역할을 한다.
    2. exit(): 현재 실행 중인 user program을 종료하고 종료 상태 코드를 kernel에 반환한다. 종료 상태 코드가 0이면 정상 종료, 0이 아니면 오류를 나타낸다.
    3. exec(): cmd\_line으로 주어진 이름의 process를 새로 생성하여 실행한다. 이때 자식 process의 pid를 반환하여 부모가 자식 process를 추적할 수 있도록 한다.
    4. wait(): 주어진 pid의 자식 process가 종료될 때까지 대기한다. 유효한 자식 process를 확인하고, 종료 시 상태 코드를 가져오는 역할을 하며, 동기화를 통해 process 종료 전에 대기하는 상황을 방지한다.
    5. read(): open된 file descriptor(fd)로부터 data를 읽어 지정된 buffer에 저장한다. 이번 구현에서는 표준 입력(STDIN)에서 읽으며, 실제로 읽은 byte 수를 반환한다.
    6. write(): open된 fd에 data를 출력한다. 이번 구현에서는 표준 출력(STDOUT)으로 출력을 수행하며, 출력된 byte 수를 반환한다.
    7. fibonacci(): 입력받은 정수 n에 해당하는 fibonacci 수를 계산하여 반환한다.
    8. max\_of\_four\_int(): 4개의 정수 값 중에서 가장 큰 값을 계산하여 반환한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

User process가 system call을 호출하면 interrupt가 발생하여 kernel mode로 전환된다. Kernel은 system call handler를 통해 호출된 system call 번호에 따라 적절한 함수를 실행한다. 이 함수는 user process의 요청에 필요한 작업을 수행하며 필요에 따라 process memory, file system 또는 다른 kernel 자원에 접근한다. system call 함수가 필요한 작업을 완료한 후 결과 값을 반환하는데, 이 값은 주로 system call의 성공 여부 또는 요청한 data가 될 수 있다. 이후, kernel은 프로세서 상태를 복원하고 다시 user mode로 돌아가며, user process의 stack pointer와 register를 원래 상태로 되돌린다. 이로써 user process는 system call 호출 직후의 지점으로 되돌아와 정상적으로 실행을 계속한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

[총 개발 기간: 9/23 ~ 10/3]

1. 9/23~24: Argument Passing 구현
2. 9/27~28: User Memory Access 구현
3. 9/29~30: System Call Handler 및 System call 구현
4. 10/1: Addition 구현 및 Test case Debugging
5. 10/2~3: 보고서 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)**

1. Argument Passing

- src/userprog/process.c 파일의 load() 함수에서 인자를 처리하기 위해, 파일 이름을 분석하여 argv 배열에 각 인자를 저장하는 코드를 추가한다. 이를 통해 process\_execute() 함수가 실행될 때 인자들이 stack에 올바르게 push될 수 있도록 한다.

- 각 인자의 주소를 stack에 push하고 마지막에 NULL pointer를 추가하여 인자의 끝을 표시한다. 또한, 인자 주소 배열과 인자의 개수를 stack에 push한다.

- src/userprog/process.c 파일의 process\_execute() 함수에 command name을 parsing하는 코드를 추가한다.

2. User Memory Access

- src/threads/vaddr.h 파일의 is\_user\_vaddr() 주소 검증 함수를 이용하여 src/userprog/syscall.c 파일을 수정한다. syscall.c 파일에 check\_user\_vaddr() 함수를 추가하여 사용자 주소가 유효하지 않으면 -1을 반환하여 프로그램을 종료하도록 설정한다.

- src/userprog/syscall.h 파일에 check\_user\_vaddr() 함수의 prototype을 추가한다.

- src/userprog/exception.c 파일의 page\_fault() 함수 내에 현재 발생한 page fault가 kernel mode에서 발생했는지를 검사하는 코드를 추가하였다. 만약 kernel mode에서 발생했다면, -1을 exit하여 프로그램을 종료시키도록 구현하였다.

3. System calls

- 먼저, src/userprog/syscall.c 파일에 halt(), exit(), exec() 등과 같은 필요한 system call에 대한 handler 함수를 추가하여 각각의 기능을 구현한다.

- wait() system call 구현을 위해 src/threads/thread.h 파일의 thread 구조체를 수정한다.

- 이후 syscall\_handler() 함수에서 system call 번호를 읽어 해당 handler를 호출하도록 수정한다. 해당 함수 내에서 각 system call의 인자를 검증하고, 필요한 경우 return값을 설정한다. 예를 들어 read와 write 함수의 경우 실제로 읽거나 쓴 byte수를 반환한다.

4. Additional

- src/userprog/syscall.h, src/lib/user/syscall.h 파일에 fibonacci와 max\_of\_four\_int system call의 prototype을 추가한다. src/userprog/syscall.c 파일에 fibonacci 함수와 max\_of\_four\_int 함수의 구현을 추가한다. fibonacci 함수는 입력받은 정수 n에 대해 n번째 피보나치 수를 계산하여 반환하고, max\_of\_four\_int 함수는 4개의 정수를 입력받아 가장 큰 값을 반환한다.

- src/examples 폴더에 user program에 해당하는 additional.c 파일을 생성하여 command line 인자를 받아 피보나치 수와 최댓값을 계산하는 logic을 구현한다. 이후 src/examples/Makefile 코드를 수정하고 additional object file을 생성하여 program을 실행한다.

- src/lib/syscall-nr.h 파일에서 system call 번호를 정의한다. SYS\_HALT, SYS\_EXIT 등과 같이 이미 정의된 system call 번호를 제외하고 additional 함수에 해당하는 fibonacci와 max\_of\_four\_int 함수의 system call 번호를 정의한다.

- src/lib/user/syscall.c 파일에 max\_of\_four\_int 함수를 위한 syscall4 매크로를 추가 정의한다. 해당 매크로는 system call을 수행할 때 4개의 인자를 전달하는 기능을 제공한다. fibonacci와 max\_of\_four\_int 함수는 각 함수의 인자 개수에 대응되는 syscall 매크로를 사용하여 인자를 수용하고, 해당 system call을 수행하도록 구현한다.

1. **연구 결과**
   1. **Flow Chart**

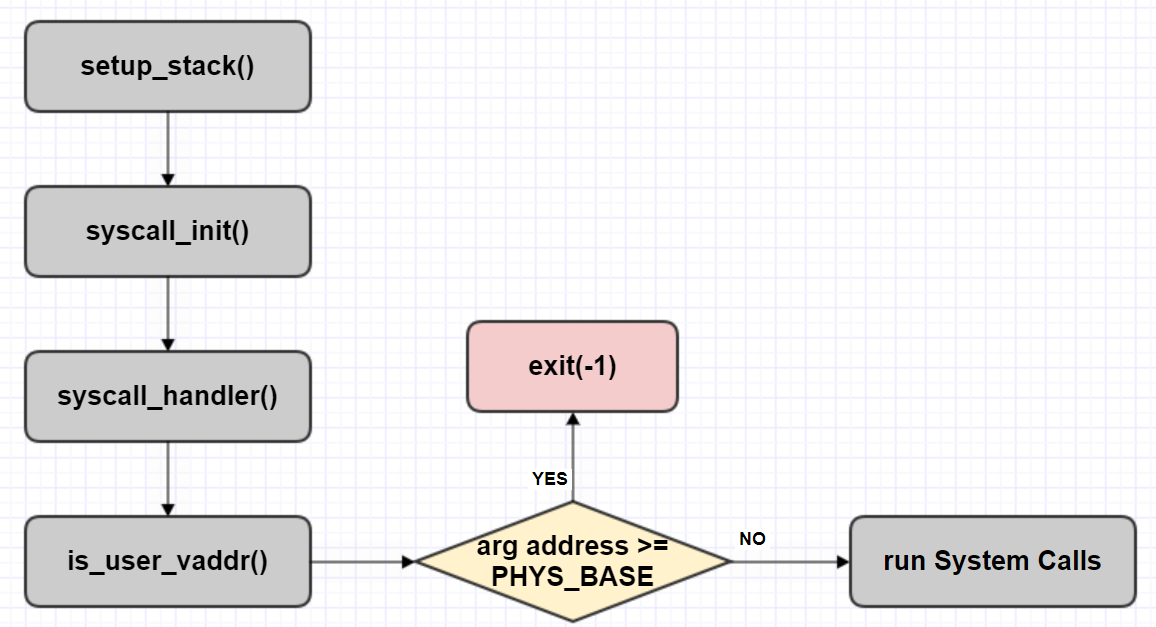
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

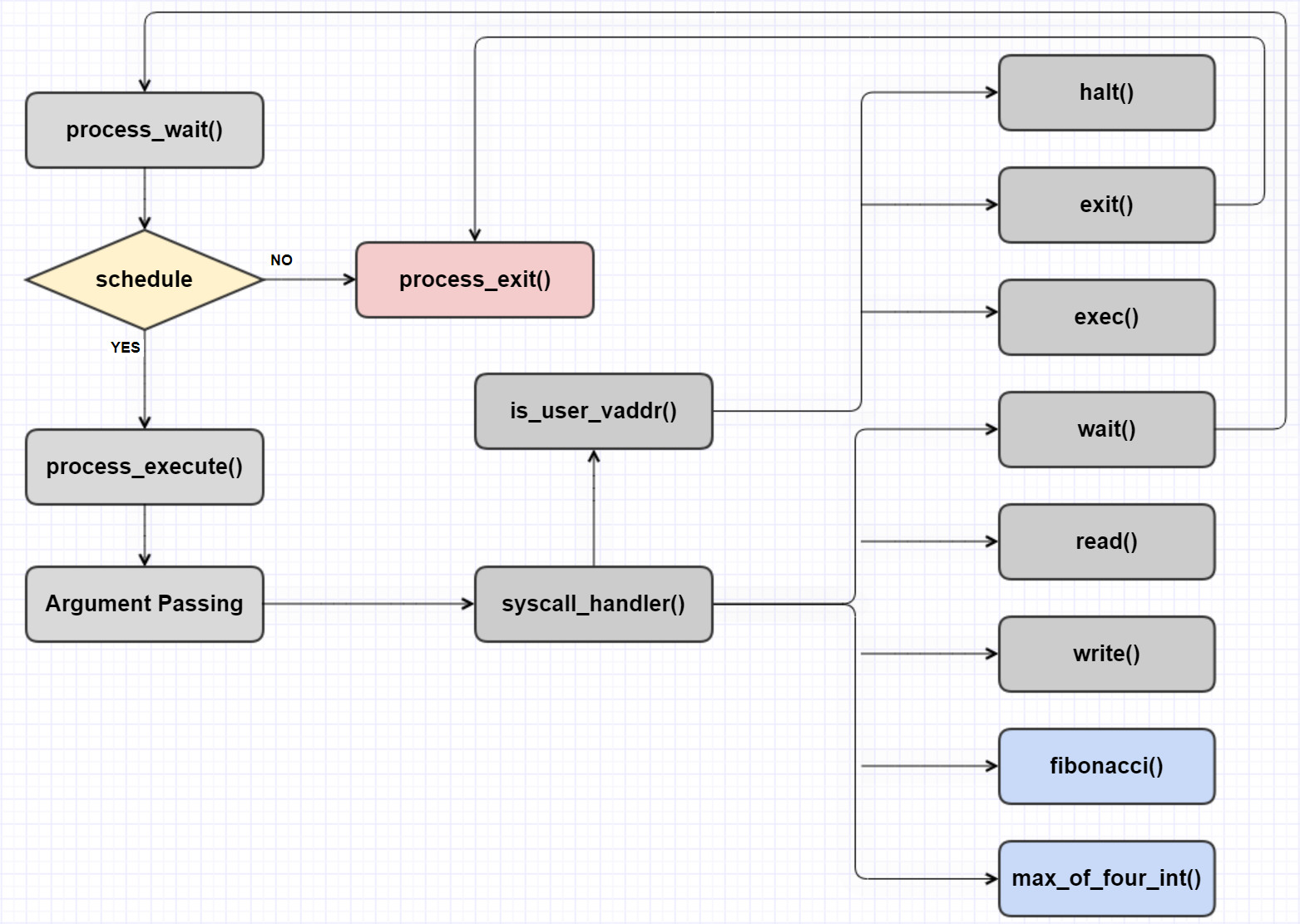
텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

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

[src/userprog/process.c]

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- 먼저, file\_name을 file\_copy 배열에 복사한다. file\_name을 그대로 쓰지 않고 복사하는 이유는 이후 문자열 조작을 위해 원본 문자열을 변경하지 않도록 보호하기 위함이다. strlcpy() 함수는 안전하게 문자열을 복사하며, 최대 크기를 지정하여 버퍼 오버플로우를 방지한다.

- argc는 명령어 인자의 개수를 저장하는 변수이며, argv는 각 인자의 포인터를 저장하는 배열이다. 최대 128개의 인자를 처리할 수 있도록 배열의 크기를 설정한다.

- strtok\_r() 함수를 사용하여 file\_copy 문자열을 공백(“ “) 기준으로 토큰화한다. 첫 번째 호출에서 토큰을 추출하고, 이후 NULL을 전달하여 다음 토큰을 가져온다. 각 토큰은 argv 배열에 저장되며, argc는 증가하여 인자의 개수를 기록한다.

- 해당 과정에서 입력된 문자열에 공백이 포함될 경우, 인자가 정확하게 분리되지 않을 수 있다. 이러한 문제를 해결하기 위해 strtok\_r() 함수를 사용하여 안전하게 문자열을 처리하였다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

1) total\_length는 전체 길이를 저장할 변수이다. for문을 통해 각 인자의 길이를 계산하고, 각 인자에 대해 null terminator를 포함하여 총 길이를 구한다. 이후 stack pointer를 총 길이만큼 감소시켜 스택에서 인자들을 저장할 공간을 확보한다. 각 인자를 stack의 현재 위치에 복사하고, argv 배열에서 해당 인자의 주소를 업데이트한다. current\_position 변수를 통해 각 인자를 복사한 후 다음 인자를 저장할 위치로 이동한다.

2) stack pointer가 4byte 정렬인지 확인하고, 그렇지 않으면 stack pointer를 조정하여 정렬한다. 정렬이 필요한 경우, stack의 새 위치를 0으로 채운다.

3) 인자 목록의 끝을 나타내기 위해 NULL pointer를 push한다. 이는 프로그램이 인자 목록의 끝을 확인할 수 있도록 해준다.

4) 각 인자의 주소를 stack에 push한다. 해당 과정은 오른쪽에서 왼쪽으로 진행되어야 하며, 각 주소는 stack에 저장된다.

5) argv 배열의 시작 주소를 stack에 push한다. 이는 프로그램이 인자 목록의 시작 위치를 참조할 수 있도록 해준다.

6) 인자의 개수를 stack에 push한다. 이는 프로세스가 얼마나 많은 인자를 받았는지 알 수 있도록 해준다.

7) 호출이 종료된 후 복귀할 가짜 주소를 push한다. 이는 시스템 호출이 완료된 후 적절한 위치로 복귀할 수 있도록 해준다.

8) 스택의 내용을 확인하기 위해 hex\_dump 함수를 호출할 수 있다. 이 단계는 debugging 목적으로 사용된다.

- 해당 과정에서 인자의 길이를 계산할 때, null terminator를 포함하지 않아 전체 길이가 잘못 계산되었다. 이는 스택에 필요한 공간이 부족해지고, 프로그램 실행 중에 스택 오버플로우 오류를 발생시키거나 잘못된 메모리 주소에 접근하는 오류를 발생시킬 수 있다. 따라서 인자 길이를 계산할 때 항상 strlen(argv[i] + 1)로 계산하여 null terminator를 포함시키도록 하였다. 또한, stack pointer를 조정할 때도 이 값을 반영하여 정확한 위치에 인자를 저장하도록 구현하였다.

[src/userprog/process.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- src/userprog/process.c 파일의 process\_execute() 함수에 command name을 parsing하는 코드를 추가한다. cmd\_name 배열을 만들어서 file\_name을 복사하고, strchr() 함수를 사용하여 첫 번째 공백 문자를 찾는다. 만약 공백이 발견되면, 해당 위치에 null 문자(‘\0’)를 삽입하여 문자열을 종료한다. 이를 통해 command name만 남긴다.

1. User Memory Access

[src/userprog/syscall.c]

폰트, 텍스트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

- src/userprog/syscall.c 파일에 check\_user\_vaddr() 함수를 추가하고, src/threads/vaddr.h 파일의 is\_user\_vaddr() 함수를 이용하여 주어진 가상 주소 vaddr이 사용자 메모리 영역에 속하는지를 확인한다. 만약 주소가 사용자 메모리 영역에 속하지 않는다면, 즉 커널 메모리 영역에 속한다면 -1을 exit하여 프로세스를 종료한다.

[src/userprog/exception.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- src/userprog/exception.c 파일의 page\_fault() 함수 내에 if 조건문을 추가한다. 해당 조건문은 page fault가 발생한 위치인 f->eip가 커널 메모리 공간에 있는지를 검사한다. 만약 커널 메모리 공간에 있으면, 이는 커널 모드에서 잘못된 주소 접근이 발생했음을 나타낸다. 이 경우, exit(-1)을 호출하여 현재 프로세스를 종료시킨다.

1. System Calls

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

[src/userprog/syscall.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- \*(uint32\_t \*)f->esp를 통해 현재 stack pointer(f->esp)에 저장된 system 호출 번호를 읽는다. 이 번호는 사용자 프로그램에서 호출된 시스템 호출을 식별한다. 이후 switch – case 문에서 각 시스템 호출 번호에 대한 적절한 handler 함수를 호출한다. 예를 들어, SYS\_HALT가 감지되면 halt() 함수가 호출되는 구조이다.

- 각 case에 대해 사용자로부터 입력된 주소가 유효한지 확인하기 위해 check\_user\_vaddr() 함수를 호출한다. 유효하다면 해당하는 handler 함수를 호출하고, 만약 유효하지 않는 주소라면 exit(-1)을 호출하여 프로세스를 종료한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

- halt(): shutdown\_power\_off() 함수를 호출하여 Pintos를 종료한다.

- exit(): 현재 프로세스를 종료하고, 종료 상태를 출력한다. thread\_exit() 함수를 호출하여 프로세스를 종료한다.

- exec(): 인자로 주어진 cmd\_line을 통해 새로운 프로세스를 실행한다. process\_execute() 함수를 호출하여 child process를 생성하고, 해당 프로세스의 id를 반환한다.

- read(): file descriptor(fd)가 0일 경우, 즉 표준 입력(STDIN)으로부터 입력을 읽어 buffer에 저장한다. 반환 값으로 읽은 byte 수를 반환한다.

- write(): fd가 1일 경우, 즉 표준 출력(STDOUT)으로 buffer 내용을 출력한다. 실제로 출력된 byte 수를 반환한다.

- wait(): 주어진 pid를 가진 child process가 종료될 때까지 기다린다. process\_wait() 함수를 호출하여 child process의 종료 상태를 확인한다.

[src/threads/thread.h]

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

\*wait() system call 구현을 위해 src/threads/thread.h 파일의 thread 구조체를 수정한다. struct thread에 int exit\_status, struct list child\_list, struct semaphore child\_lock 필드를 추가하여 child process의 종료 상태를 저장하고 관리한다. int exit\_status는 child process의 종료 상태를 저장하는 변수다. struct list child\_list는 child process를 추적하기 위한 list이다. struct semaphore child\_lock은 child process가 종료될 때까지 parent process가 대기하도록 하는 semaphore다.

[src/userprog/process.c]

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

- wait() 함수를 제대로 작동시키기 위해 process\_wait() 함수를 수정한다. cur 변수는 현재 실행 중인 thread를 나타내며, thread\_current()는 현재 thread의 pointer를 반환한다. child\_thread 변수는 찾은 자식 thread를 저장하기 위한 pointer이다. for문을 통해 현재 thread의 child\_list에서 각 자식 thread를 순회한다. list\_begin()은 자식 thread list의 시작을, list\_end()는 list의 끝을 나타낸다.

- list\_entry()를 사용하여 list 요소에서 child\_thread 구조체를 추출한다. 자식 thread의 tid가 인자로 전달된 child\_tid와 일치하는지를 확인한다.

- sema\_down()을 호출하여 해당 자식 thread가 종료될 때까지 대기한다. 이 semaphore는 자식 thread의 종료가 발생할 때까지 block한다. 자식 thread의 종료 상태를 exit\_status 변수에 저장한다.

- list\_remove()를 통해 list에서 자식 thread를 제거하고, 자식 thread의 종료 상태를 반환한다. 만약 자식 thread list에서 child\_tid와 일치하는 thread를 찾지 못한 경우, -1을 반환하여 자식 process가 존재하지 않음을 알린다.

1. Additional System calls

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

[src/lib/syscall-nr.h]

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- src/lib/syscall-nr.h 파일에 additional 함수에 해당하는 새로운 system call 번호를 추가하여 각 system call을 식별할 수 있도록 하였다.

[src/userprog/syscall.c]

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- src/userprog/syscall.c 파일에 additional 함수 코드를 구현하였다.

[fibonacci 함수]

- int n: 계산할 피보나치 수의 index를 의미한다.

- int a=0, b=1: 피보나치 수열의 첫 두 항을 초기화한다.

- int fib=0: 현재 피보나치 수를 저장할 변수를 초기화한다.

- if(!n) return a; 만약 n=0이면, 피보나치 수의 첫 항인 0을 반환한다.

- for문을 2부터 n까지 반복하여 n번째 index에 해당하는 피보나치 수를 계산한다. for문이 종료된 후, 최종적으로 계산된 피보나치 수를 반환한다.

[max\_of\_four\_int 함수]

- 먼저, a와 b 중 더 큰 값을 x에 저장한다. 이후 c와 d 중 더 큰 값을 y에 저장한다.

- 최종적으로 x와 y중 더 큰 값을 반환한다.

스크린샷, 텍스트, 멀티미디어 소프트웨어, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- src/userprog/syscall.c 파일의 syscall\_handler() 함수에 새로운 system call에 대한 handler를 추가하고, 인자 수를 적절히 처리하여 호출 결과를 반환하도록 코드를 구현하였다.

- 해당 case에 대해서도 check\_user\_vaddr() 함수를 호출하여 사용자로부터 입력된 주소가 유효한지 확인해준다.

[src/lib/user/syscall.c]

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- src/lib/user/syscall.c 파일에 syscall4 매크로를 추가로 정의(syscall3 참고)하여 4개의 인자를 전달할 수 있도록 설계하였다. retval은 system 호출의 반환 값을 저장하기 위한 변수이다.

- push1 %[number]; int $0x30: system 호출 번호를 push하고, interrupt를 발생시킨다. $0x30은 system 호출을 처리하는 interrupt 벡터 번호이다.

- addl $20, %%esp: stack pointer를 조정하여 push된 인자들의 크기만큼 이동한다. 각 인자는 4byte 크기이므로, 총 4개의 인자를 push하면 16byte가 추가된다. system 호출 번호를 push하기 때문에 총 20byte를 stack에서 제거해야 한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- src/lib/user/syscall.c파일에 fibonacci 함수와 max\_of\_four\_int 함수의 system call을 구현한다.

- fibonacci() 함수는 syscall1 매크로를 사용하여 시스템 호출 SYS\_FIBONACCI를 호출한다. 해당 매크로는 n을 인자로 전달하고, 시스템 호출의 반환값을 받아온다.

- max\_of\_four\_int 함수는 위에서 추가로 정의한 syscall4 매크로를 사용하여 시스템 호출 SYS\_MAX\_OF\_FOUR\_INT를 호출한다. 해당 매크로는 총 4개의 인자를 전달한다.

[src/examples/addition.c]

스크린샷, 텍스트, 멀티미디어 소프트웨어, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- src/examples 폴더에 additional.c 파일을 추가하여 사용자 입력 프로그램을 생성한다.

- argc는 프로그램에 전달된 인자의 개수를 나타내고, argc는 인자 문자열 배열을 나타낸다. 인자는 문자열 형태로 제공된다.

- 첫 번째 인자(argv[1])를 정수로 변환(atoi)하여 해당 인자에 대한 피보나치 수를 계산한다.

- 첫 번째 인자부터 네 번째 인자(argv[1]~argv[4])를 정수로 변환(atoi)하여 최댓값을 계산한다.

- 각 함수의 system call을 통해 구한 피보나치 수와 최댓값을 출력한다. 각각의 결과는 정수 형태로 출력된다.

[src/examples/Makefile]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- src/examples/Makefile을 다음과 같이 수정하여 terminal에 make 명령어를 입력했을 때 additional.c 파일에 대한 object file (additional)이 생성되도록 하였다.

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

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

- argument로 10, 20, 62, 40을 입력했을 경우 실행 결과는 다음과 같다,

- fibonacci(10)의 결과값인 55, max\_of\_four\_int(10, 20, 62, 40)의 결과값인 62가 올바르게 출력되는 것을 확인할 수 있다.