Pintos Project 1 : User Program Basic

(설계 프로젝트 수행 결과)

과목 명 : [CSE4070] 운영체제

담당 교수 : 소정민

조 / 조원 : 1조 / 20141552, 20161662

개발 기간 : 2018/10/1 ~ 2018/10/7

프로젝트 제목 : Pintos Project 1 : User Program Basic

제출일 : 2018/10/12

참여 조원 : 이규연, 허남규

I. 개발 목표

- Pintos 운영체제의 기본 코드는 사용자의 프로그램을 메모리에 적재하고 실행시킬 수 있도록 구현되어 있습니다. 하지만 입출력이나 기타 상호 작용을 하지 못하는 매우 제한적인 상태로 작동합니다. 이번 프로젝트는 사용자의 프로그램이 운영체제와 상호작용할 수 있도록 시스템 호출 (system call)을 구현하는 것이 목표입니다.

II. 개발 범위 및 내용

A. 개발 범위

- **Argument Passing**

  - Pintos의 기본 코드의 process_execute() 함수는 입력 받은 명령어를 새로운 프로세스로 넘기는 기능이 완전히 구현되지 않은 상태입니다. 인자로 받은 문자열을 공백을 기준으로 적절하게 파일명과 인자로 나누어 명령어를 수행할 수 있도록 기능을 확장해야 합니다. 예를 들어, process_execute("grep foo bar")이 실행 되면 grep파일에 foo와 bar이라는 인자를 넘겨 실행시키도록 하는 것입니다.

  - Pintos의 소스코드 중 process.c에 construct_stack()이라는 새로운 함수를 만들어 입력 받은 문자열을 공백을 기준으로 끊은 후 스택에 저장합니다.

- **User Memory Access**

  - 시스템 호출 시 커널은 사용자 프로그램으로부터 받은 포인터를 통해 메모리에 접근합니다. 사용자가 넘긴 포인터는 종종 할당되지 않은 메모리에 대한 주소, 커널 메모리 또는 NULL 주소를 가리킬 수 있기 때문에 커널은 이에 대한 검증을 해야 합니다. 이는 커널에 부정적인 영향을 줄 수 있기 때문에 커널을 보호해야 합니다.

  - Pintos 소스코드 중 syscall.c에서 포인터의 커널 영역 침범 여부를 검사하고 exception.c를 적절히 수정하여 페이지 부재를 일으키는 대신 프로세스가 종료되도록 합니다.

- **System Call**

  - 운영체제는 인터럽트를 통해 시스템 호출에 필요한 번호와 인자를 받아서 일을 수행합니다. 현재 Pintos에 구현된 시스템 호출은 현재 프로세스를 종료시키는 것으로 구현되어 있습니다. 이를 토대로 halt(), exit(), exec(), wait(), read(), 그리고 write() 시스템 호출을 구현하는 것이 목표입니다.

  - Pintos 소스코드 중 syscall.c 내의 syscall_handler() 함수에서 적절히 인자를 해당되는 시스템 호출에 넘겨줌으로써 구현합니다.

  - 동기화가 필요한 경우 바쁜 대기 (busy waiting)방식만을 사용하여 구현합니다.

B. 개발 내용

- **Page**

  ■ 페이징 기법은 컴퓨터가 메인 메모리에서 사용하기 위해 2차 기억 장치로부터 데이터를 저장하고 검색하는 메모리 관리 기법입니다. 속도가 느린 2차 기억 장치를 페이지라는 단위로 나눈 후 이를 바탕으로 메모리에 적재하는 방식입니다. 프로세스가 적재되지 않은 페이지에 접근하는 경우는 페이지 부재라고 하여 해당 페이지를 메모리에 적재하는 과정 (현재 Pintos에는 구현되어 있지 않습니다)을 거칩니다.

- **Threads**

  ■ 스레드는 Pintos 운영체제의 프로세스 수행 단위입니다. 스레드 구조체는 해당 프로세스의 메모리 영역인 페이지의 시작 부분에 위치합니다. 스레드의 스택은 페이지의 나머지 끝 부분부터 시작하여 자랍니다.

  ■ 스레드는 thread.c 내의 thread_create()함수를 통해서 생성됩니다. 여기서 페이지와 tid를 할당 받고 스택의 초기화 등이 이루어집니다. 생성이 완료되면 인자로 받은 함수를 수행합니다.

- **Kernel Memory Protection**

  ■ 사용자 프로그램이 인자로 넘기는 포인터의 유효함을 검증하기 위해서 해당 포인터 변수의 값이 사용자 프로그램의 메모리 영역인 PHYS_BASE이하인지 확인한 후 만족하는 경우 데이터에 접근하는 것입니다. 이를 만족한다면 커널 영역을 침범하지 않기 때문에 운영체제를 보호할 수 있습니다. 만약 할당되지 않은 사용자의 가상 메모리에 접근하는 경우 페이지 부재가 발생하지만 이는 page_fault() 함수를 적절히 수정하여 해결할 수 있습니다.

- **Synchronization Strategy**

  ■ Pintos 운영체제의 문제점은 새로운 프로세스가 실행될 경우 동기화가 이루어지지 않기 때문에 의도한 대로 작동하지 않습니다. 이 문제를 해결하기 위하여 프로세스간의 부모-자녀 관계를 정의한 후, 자녀 프로세스가 수행을 완료할 수 있도록 부모 프로세스가 수행을 멈추도록 (기다리도록) 해야 합니다. 또한 특정 프로세스가 종료할 때, 해당 프로세스가 아직 수행 중인 자녀가 있는지 확인한 후, 이들의 종료를 기다린 후 종료하도록 합니다.

III. 추진 일정 및 개발 방법

A. 추진 일정

- 2018.10.1 ~ 2018.10.2: Pintos 코드 분석, 문제 상황 파악, 전략 수립.
- 2018.10.3: Argument Passing 구현.
- 2018.10.4 ~ 2018.10.6: System Call 일부 구현. 동기화 구현.
- 2018.10.7: System Call 구현 및 보고서 작성.

B. 개발 방법

- **Argument Passing**
  - 입력 받은 인자들을 적절히 넘기기는 일을 수행하는 construct_stack()라는 함수를 새로 정의하여 스택에 넘겨야 하는 인자들을 esp를 통해 접근하여 저장합니다.
  - userprog/process.c에 construct_stack()함수를 새로 정의합니다.
- **User Memory Access**
  - 사용자 프로그램은 커널 영역을 참조하는 포인터나 NULL 포인터, 그리고 할당되지 않은 포인터를 시스템 호출에 인자로 넘길 수 있습니다. 이를 위하여 우선 syscall.c의 syscall_handler()함수 내에서 포인터의 커널 영역 침범 여부를 검사합니다. 만약 침범하면, 여기에서 exit(-1)을 호출하여 종료합니다.
  - 커널 영역을 참조하지 않는다고 해서 무조건 유효한 포인터는 아닙니다. 만약 받은 포인터가 할당되지 않은 메모리 영역을 참조하는 경우, exception.c에서 페이지 부재 오류를 피할 수 있도록 exit(-1)을 호출하도록 하였습니다.
  - userprog/syscall.c의 is_valid()함수를 새로 정의합니다.
  - userprog/exception.c의 page_fault()를 수정합니다.
- **System Call**
  - Pintos는 시스템 호출을 enum 자료형에 저장하여 구분합니다. 소스코드 중 syscall.c 내의 syscall_handler()함수에 switch()문을 추가하여 각각의 해당되는 시스템 호출을 실행하도록 구현하였습니다.

■ userprog/syscall.c의 syscall_handler()을 수정합니다.

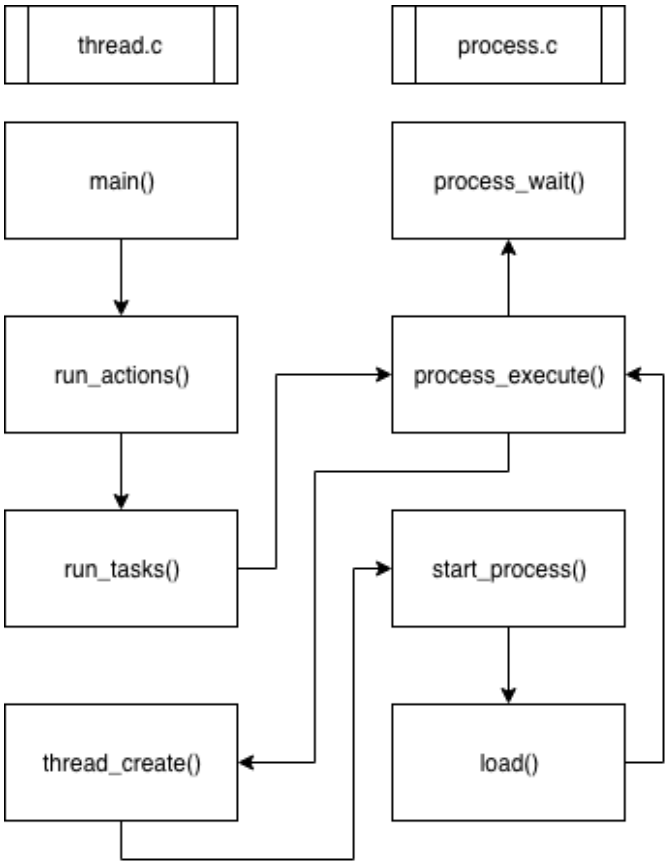
C. 연구원 역할 분담

- 이규연: Argument Passing 및 동기화 기법 구상 및 구현, 보고서 작성.
- 허남규: System Call 구현.

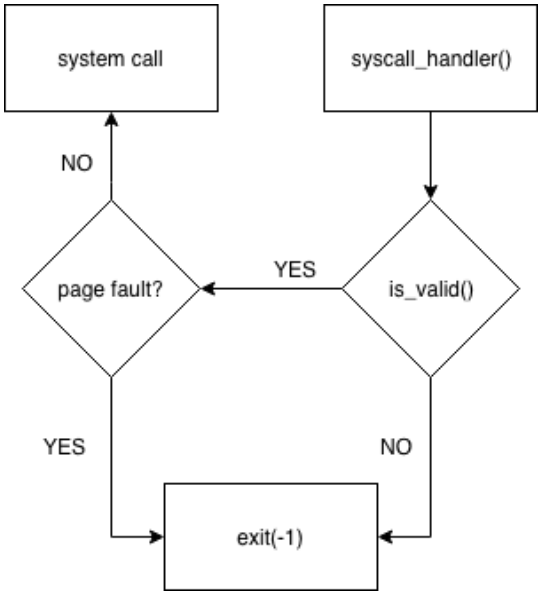
IV. 연구 결과

A. 합성 내용

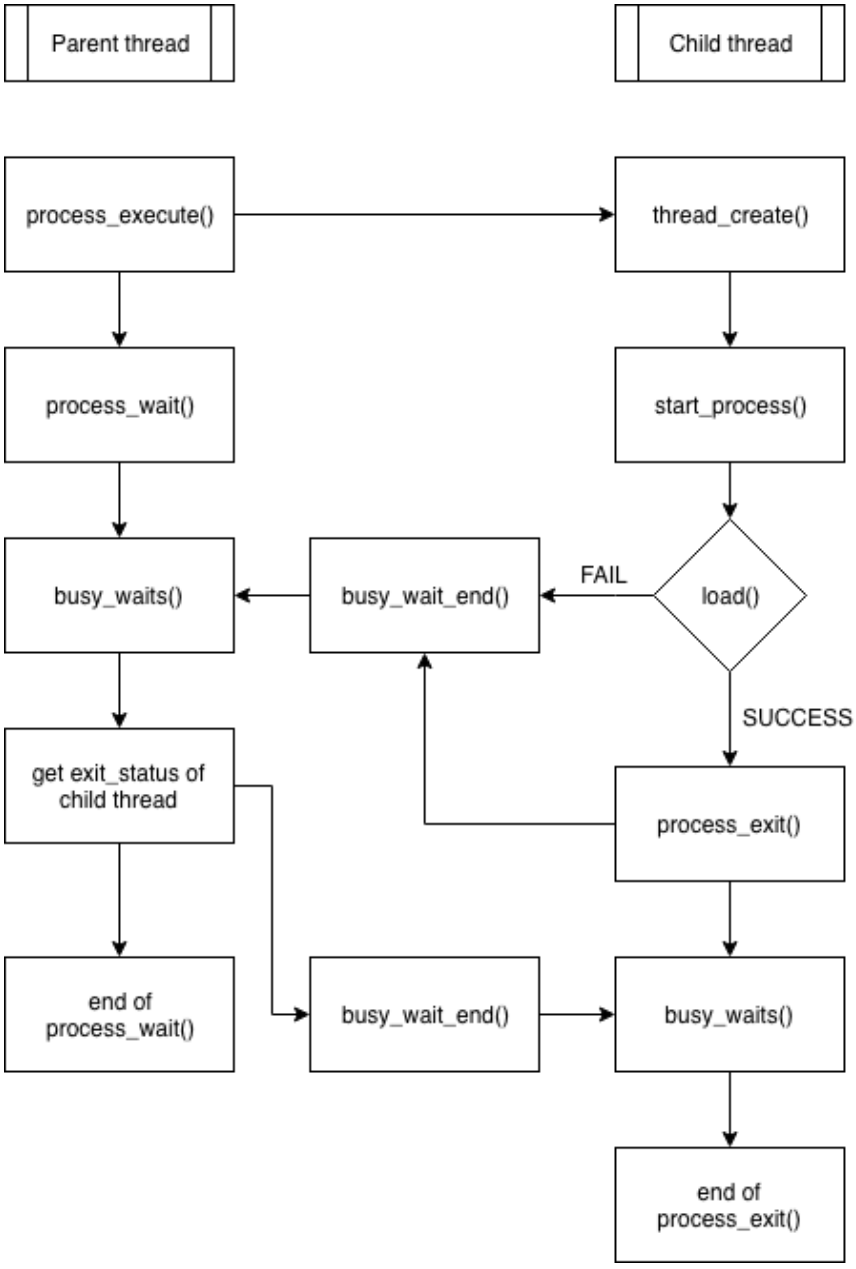
- Pintos의 전반적인 흐름은 다음과 같습니다.



- 시스템 호출과 커널 영역 보호는 다음과 같이 이루어집니다.



- 부모-자녀 프로세스의 동기화 기법은 다음과 같습니다.
  - ■ 이에 대한 자세한 설명은 **B. 제작 내용**에 기술하였습니다.



B. 제작 내용

- **Argument Passing**

  ■ 실제 스택처럼 하나씩 순서대로 삽입하는 것 보다 전체 인자를 버퍼에 저장한 후 '₩0' 문자를 구분자로 삽입한 후 memcpy() 함수를 이용하여 구현하였습니다. 이렇게 하면 메모리에 저장된 방식(엔디언 등)에 대한 추가적인 고민 없이 사용할 수 있습니다.

  ■ 반복문을 돌면서 중간에 중복된 공백이 있는지 확인하여 이를 esp에 더해줍니다. 스택에 저장된 인자들의 주소값은 esp를 업데이트 할 때마다 스택에 저장합니다.

- **User Memory Access**

  ■ 커널 메모리를 보호는 앞서 서술한 방법을 사용하였습니다. syscall_handler()에서 커널 메모리를 참조 여부를 is_user_valid()함수로 판단한 후, 커널을 가리키는 경우 exit(-1)을 호출합니다. is_user_valid()함수를 통과했지만 유효하지 않은 메모리를 참조하는 경우는 페이지 부재 오류가 일어나기 전에 수행을 중지하도록 하였습니다. 이를 구현하기 위해 exception.c의 page_fault() 함수에 조건에 맞게 exit(-1)을 할 수 있도록 작성하였습니다.

- **System Call**

  ■ 시스템 호출은 주어진 함수 원형을 토대로 구현하였습니다. 인터럽트 프레임에 저장된 esp에서부터 명령어와 인자들을 순서대로 받고, 수행 후 반환 값이 있다면 eax에 저장하도록 하였습니다.

- **Synchronization by Using Only Busy Waiting**

  ■ 부모 프로세스가 process_exec()를 이용하여 자녀 프로세스를 생성할 경우 해당 tid를 인자로 process_wait()를 호출하여 자녀 프로세스의 수행 완료를 기다려야합니다. 여기서 주의할 점은 자녀 프로세스의 exit_status를 받아야 합니다. 이를 위해서 exit() 시스템 호출을 하기 직전에 잠시 멈추고 부모 프로세스가 이를 받은 후에 다시 수행되어야 합니다.

  ■ 바쁜 대기 (busy-waiting)는 특정 조건의 충족 여부를 반복적으로 확인하여 해당 조건을 만족하면 다시 정상 수행을 하는 동기화 기법입니다. 바쁜 대기를 구현하기 위해 고려해야할 점은 다음과 같습니다.

- ◆ 스레드의 상태는 항상 수행중이어야 합니다. 만약 동기화를 위하여 thread_block() 등과 같은 스레드의 수행을 멈추는 함수를 사용할 경우, 이는 반복적인 확인이 아닌 다른 객체에 의하여 수동적으로 수행 여부가 결정되기 때문입니다.

- ◆ Pintos의 동기화 API로 제공되는 기능들 중 semaphore와 lock이 있습니다. 소스코드를 분석한 결과, lock은 0 또는 1의 값만 가질 수 있는 semaphore,이며 semaphore은 실행중인 스레드를 thread_block()를 이용하여 상태를 변화시켜 수행을 멈추는 것으로써 동기화를 구현하였습니다. 이들은 이번 프로젝트의 목표인 바쁜 대기를 이용한 구현과는 방향성이 맞지 않기 때문에 사용하지 않기로 하였습니다.

■ 위를 토대로 다음과 같은 방법으로 구현하였습니다.

- ◆ 구조체 thread에 추가한 변수는 다음과 같습니다.

  - bool SJW: false로 초기화 합니다.
  - bool LKY: false로 초기화 합니다.
  - struct thread *parent: 부모 프로세스를 참조합니다.
  - struct list children: 자녀 프로세스를 저장하는 리스트입니다.
  - struct list_elem *siblings: children 리스트를 위한 list_elem입니다.

- ◆ 함수 busy_waits(struct thread*)는 현재 수행 중인 스의 멤버 변수 SJW를 반복적으로 확인하며 true가 되는 경우 반복문을 탈출하도록 하였습니다. 탈출한 다음 SJW를 다시 false로 설정하여 이후에 다시 사용할 수 있도록 합니다. 이를 통해 바쁜 대기를 구현할 수 있습니다. 예외 사항이 발생할 경우를 대비하여 변수 LKY를 통해 해당 스레드가 대기중인지 여부를 표시할 수 있도록 하였습니다. 여기서 주의할 점은 비생산적인 코드를 사용할 경우 컴파일러가 최적화 과정을 거치면서 이를 삭제하기 때문에 boundary()를 사용하여 이를 막아야 합니다.

■ 바쁜 대기 방식으로 구현된 동기화 기법에는 큰 문제점이 있습니다. 존재하는 모든 스레드들이 항상 수행중인 상태로 불필요한 연산을 하기 때문입니다. 여러 스레드가 대기중일 경우, 이는 상당히 많은 CPU의 자원을 낭비합니다. 이는 시스템의 성능에 매우 부정적인 영향을 미치기 때문에 다른 방식으로

개선해야 합니다. 차후 프로젝트에서는 바쁜 대기 방식이 아닌, 스레드의 상태를 변화시켜 수행되지 않도록 하는 방식으로 바꾸어 구현할 계획입니다.

- 동기화가 이루어지는 곳은 process.c의 process_wait() 함수와 process_exit()함수입니다. 이는 현재 실행중인 프로세스가 인자로 받은 pid에 해당하는 자녀 프로세스의 수행을 기다립니다. 만약 인자로 받은 pid의 프로세스가 현재 프로세스의 자녀가 아니거나 이미 process_wait()이 성공적으로 수행된 경우라면 즉시 -1을 반환합니다. 성공적인 process_wait()의 구현을 위하여 process_exit()에서도 대기를 해야 하는데, process_wait()함수에서 부모 프로세스가 자녀 프로세스의 exit_status를 받아야 하기 때문입니다.

- 새로운 프로세스가 생성되면 기존에 수행 중인 프로세스의 자녀로 정의합니다. 동기화 기법이 사용되지 않으면 자녀 프로세스가 생성이 되는 즉시 부모 프로세스가 수행을 이어 나간 후 종료합니다. 이를 그대로 두면 생성된 자녀 프로세스가 수행을 시작하기도 전에 부모 프로세스가 종료하기 때문에 문제가 됩니다. 따라서 자녀 프로세스가 수행을 마치고 부모 프로세스에 exit_status를 반환하여 수행을 종료할 수 있도록 부모 프로세스가 기다려야 합니다.

- 기존의 process_wait()함수와 process_exit()함수를 수정하였습니다. 현재 프로세스에서 인자로 받은 pid에 해당하는 자녀가 있는지 리스트를 탐색합니다. 발견되지 않을 경우 기다림 없이 즉시 -1을 반환합니다. 발견될 경우, 이 부분에서 현재 수행중인 프로세스가 바쁜 대기를 해야 합니다. 이 때 자녀 프로세스가 수행을 마친 후 process_exit()을 끝마치기 직전, 부모 프로세스의 대기를 멈추고 부모 프로세스가 exit_status를 받을 수 있도록 자녀 또한 잠시 대기 합니다. 부모 프로세스는 exit_status를 받는 대로 다시 자녀 프로세스가 process_exit()을 완료할 수 있도록 대기를 멈추고 다시 수행시킵니다. 이렇게 하면 자녀 프로세스는 부모 프로세스에게 필요한 정보를 주고 성공적으로 종료할 수 있습니다.

- 주의할 점은 process_execute()를 사용할 경우 항상 그 반환 값을 인자로 process_wait()을 실행시켜야 한다는 것입니다. 메인 프로세스의 경우 process_wait(process_execute())와 같은 방식으로 되어 있습니다. 사용자가 프로세스에 대한 관리를 철저하게 해야만 고아가 생기지 않고 정상적으로 프로세스들이 수행 및 종료를 합니다.

- **Additional System Calls**

  ■ 추가 구현을 위하여 SYS_FIBONACCI와 SYS_SUM_OF_FOUR_INTEGERS를 syscall-nr.h의 enum 자료형에 추가하였습니다. Pintos는 원래 세개의 인자를 받는 것 까지만 지원하므로 lib/user/syscall.c에 어셈블리 코드를 추가하여 네 개의 인자를 받을 수 있도록 구현하였습니다. 인자 3개를 받는 기존 코드를 참고하여 작성하였습니다.

  ■ 새로이 fibonacci()와 sum_of_four_integers() 함수를 구현하여 해당 어셈블리 코드로 작성된 시스템 호출을 사용하도록 하였습니다.

C. 시험 및 평가 내용

- 다음은 make check의 결과입니다.

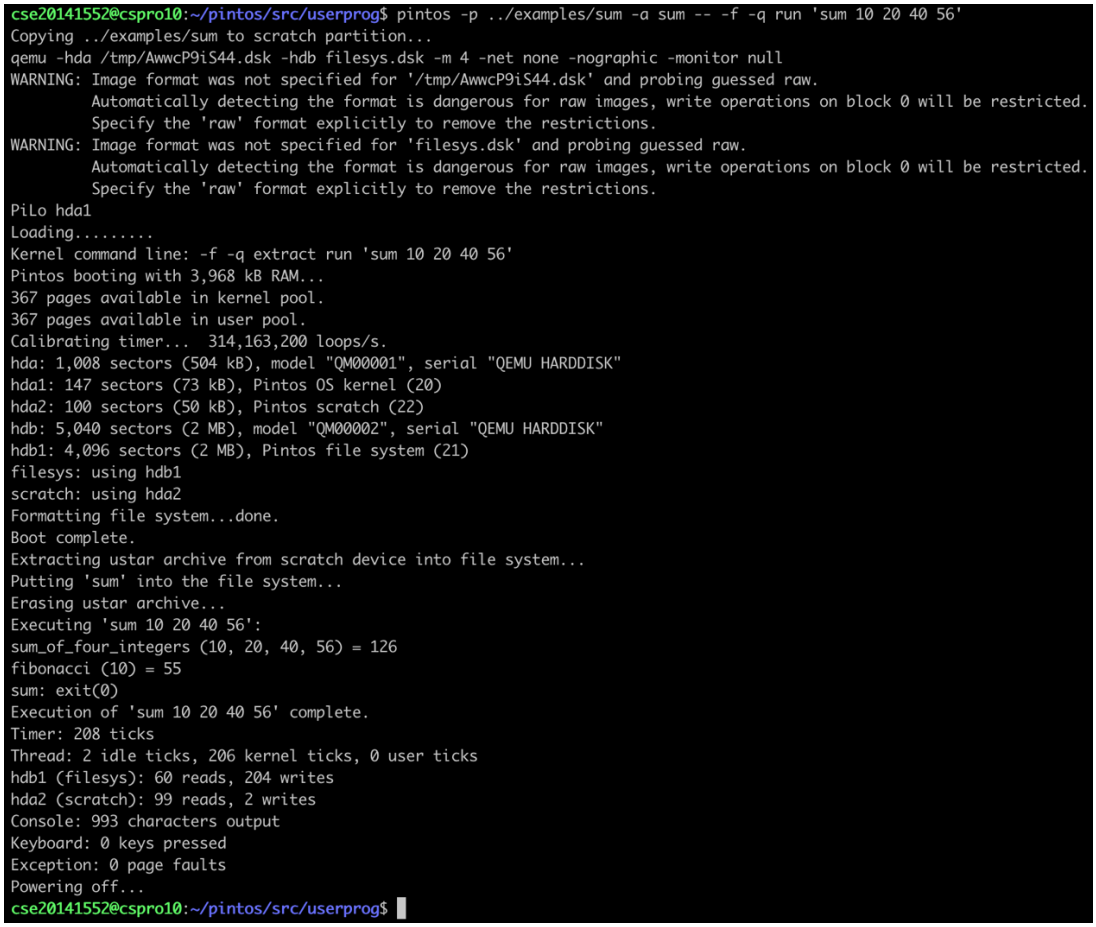
  ■ multi-recurse 테스트 케이스의 경우 시간 초과로 인해 성공하지 못했습니다. 바쁜 대기의 한계점이라고 판단됩니다.

```
634 pass tests/userprog/args-none
635 pass tests/userprog/args-single
636 pass tests/userprog/args-multiple
637 pass tests/userprog/args-many
638 pass tests/userprog/args-dbl-space
639 pass tests/userprog/sc-bad-sp
640 pass tests/userprog/sc-bad-arg
641 pass tests/userprog/sc-boundary
642 pass tests/userprog/sc-boundary-2
643 pass tests/userprog/halt
644 pass tests/userprog/exit
```

```
676 pass tests/userprog/exec-once
677 pass tests/userprog/exec-arg
678 pass tests/userprog/exec-multiple
679 pass tests/userprog/exec-missing
680 pass tests/userprog/exec-bad-ptr
681 pass tests/userprog/wait-simple
682 pass tests/userprog/wait-twice
683 pass tests/userprog/wait-killed
684 pass tests/userprog/wait-bad-pid
685 FAIL tests/userprog/multi-recurse
```

- 다음은 추가 구현 시스템 호출의 실행 결과입니다.

  ■ 시스템 호출에서 결과값을 출력하고 그 값을 반환하도록 하였습니다.



V. 기타

A. 연구 조원 기여도

- 이규연 50%, 허남규 50%

B. 소감

- 운영체제 속에서 실제로 프로세스가 어떻게 작동하는지, 그리고 어떤 동기화 이슈가 있는지 직접 구현해보면서 보다 깊이 있게 배울 수 있는 좋은 기회였습니다. 아쉬운 점이 하나 있다면 Threads 프로젝트를 진행하기 전에 User Program을 진행하여 스레드의 동작 메커니즘에 대하여 이해하는 것이 다소 어려웠습니다.
- 바쁜 대기 (busy waiting)에 의한 구현이 시스템에 오버헤드를 발생시켜 테스트 케이스 중 multi-recurse가 주어진 시간인 60초 내에 수행을 완료하지 못하여 아쉽습니다. 매우 오랜 시간 고민했지만 바쁜 대기를 정확하게 구현하기 위해서는 어쩔 수 없는 문제라고 판단하였습니다.
- 팀원과의 효과적인 협업을 위해서 알아보기 쉽게 프로그램을 작성하는 것과 주석을 꼼꼼하게 작성하는 일이 매우 중요함을 느꼈습니다. 버전 컨트롤 도구인 github를 본격적으로 사용할 수 있는 기회였습니다.