**Pintos Project 1 : User Program Basic**

**(설계 프로젝트 수행 결과)**

**과목 명 : OS**

**담당 교수 : 김영재**

**조 / 조원 : 장지훈, 안시현**

**개발 기간 : 9 / 22 ~ 10 / 19**

**프로젝트 제목 : pintos project 1 user program basic**

**제출일 : 10 / 19 (금)**

**참여 조원 : 안시현 (20141544), 장지훈(20161635)**

1. **개발 목표**

* **이번 프로젝트 시작 전 pintos는 argument passing 및 system call이 구현이 되어 있지 않아서 user program을 실행할 수 없는 상태였다. 또한 User program에서 Kernel address로의 접근을 제한하지 않아서 Kernel 영역에 대한 보호가 제대로 이뤄지지 않고 있었다.**
* **이번 프로젝트의 목표가 최소한의 User Program을 실행하기 위한 기본적인 구조를 갖추는 것이므로 User Program을 실행하기 위해 필요한 부분을 구현한다.**

**이에 따라 기본적인 System call들을 구현하고 구현한 system call이 argument들을 적절히 passing 할 수 있도록 Argument Passing , 또한 각각의 argument들이 kernel 영역에 접근하는 것을 방지해야 하므로 User program이 valid한 address에만 접근하게 하는 함수도 구현해야 한다.**

1. **개발 범위 및 내용**
   1. **개발 범위**

**Argument Passing**

* + **입력 받은 argument를 해석한 후, 80x86 Calling Convention에 맞추어 memory에 배치한다.**
  + **pintos/src/thread/thread.c, pintos/src/userprog/process.c의 load 함수가 수정되어야 한다.**
  + **thread.c의 thread\_create함수에서 filename으로 thread를 만들고 process.c의 load함수에서 stack에 argument를 쌓는다.**

**User Memory Access**

* + **User가 잘못된 Memory access를 할 경우 error를 발생시킨다. 다음과 같은 방식으로 구현한다.**
  + **pintos/src/lib/user/syscall.c에서 user program이 접근하고자 하는 모든 메모리에 대해 유효성이 체크되어야 한다.**
  + **syscall\_handler에서 is\_valid\_addr(User defined) 함수를 통해 valid한 address인지 검사한다.**

**System Call**

* + **다음의 system call을 구현한다 - halt, exit, exec, wait, read, write, pibonacci, sum\_of\_four\_integers**
  + **pintos/src/userprog/syscall.c , pintos/src/lib/user/syscall.c 가 수정되어야 한다.**
  + **system call은 최대 argument가 3개 인데 sum\_of\_four\_integers는 argument가 4개 이므로 syscall4 (NUMBER,ARG0,ARG1,ARG2,ARG3)을 추가한다.**
  + **각각의 system call에 대해 pintos manual을 참조하여 각각의 system call을 구현한다.**
  1. **개발 내용**

**page를 설명하기 전에 우선 가상 메모리에 대하여 설명해야 한다. 가상 메모리란 각 프로그램에 실제 메모리 주소가 아닌 가상의 메모리 주소를 주는 방식을 말한다. SSD나 HHD같은 보조기억장치에 데이터를 저장했다가 필요할때 다시 Main memory로 이동시키는 방법인데, 주기억장치를 하드디스크에 대한 캐시로 설정함으로써 Main memory를 효과적으로 관리 할 수 있다.**

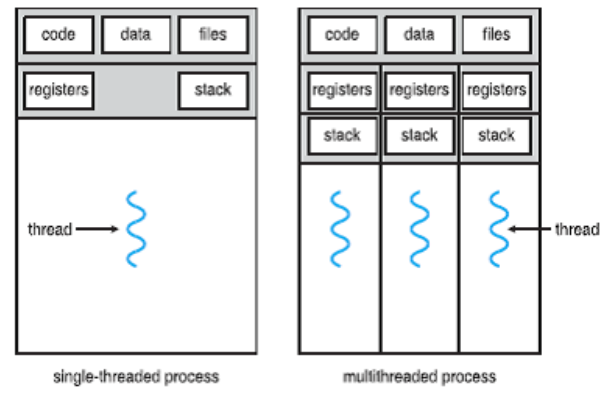
**이를 통해 main memory의 유효 크기를 늘려 process들 사이의 memory 침범이 일어날 여지를 크게 줄일 수 있다.**

**크게 세그먼트 방식과 페이징 방식 두 종류의 방식이 있는데, page는 페이징 방식에서 가상 메모리를 일정된 한 크기로 나눈 블록을 의미한다. 이때 main memory도 page와 같은 크기의 블록으로 나누어 지는데 이것을 프레임이라 한다. 하나의 페이지는 하나의 프레임으로 맵핑되며 페이지 테이블 변환 과정에서 맵핑 데이터를 찾지 못한다면 page-fault가 발생하게 된다.**

**thread는 어떠한 프로그램 내에서, 특히 프로세스 내에서 실행되는 흐름의 단위를 말한다. 우리가 생각하는 프로그램이 실행되기 위해서 하나의 실행흐름으로 처리할 수도 있지만 다수의 실행흐롬으로 처리할 수도 있다.**

**다시 말해서 thread란 프로세스에 존재하는, 프로세스가 실행되는 흐름이다.**

**이때 하나의 process 안에서도 여러개의 thread가 실행 될 수 있으며 이를 multi-threaded process라 하고 하나의 process안에 thread끼리는 stack영역 외의 다른 영역을 공유한다.**

****

* **user process의 memory address는 PHYS\_BASE (0x30000000) 보다 작거나 같아야 한다. 만약 이보다 크면 kernel address를 침범한 것이므로, page fault가 발생하면서 kernel이 panic 상태에 빠진다. 이를 막기 위해서는 user memory를 참조하기 전에 PHYS\_BASE보다 memory address가 작거나 같은지를 확인한 후, 만약 kernel 영역을 침범하고 있다면 해당 thread를 종료하기 위해 exit(-1)을 한다.**

**또한, user program이 syscall handler를 호출할 때 전달받은 argument의 주소와, 그 argument가 주소를 저장하고 있을 때 그 주소에 대해 mapping된 page가 있는지 확인해야 한다. 이를 위해서 pagedir\_get\_page 함수를 사용하였으며, 만약 mapping된 page가 없다면 역시 page fault가 발생하기 전에 exit(-1)을 해야 한다.**

* **Parent process가 child process를 실행시킬 때, parent는 child가 죽기까지 (exit 될 때까지) 기다려야 한다. 만약 parent가 먼저 죽는다면, child process는 orphan process가 되므로 kernel이 비정상적인 상태에 빠진다. 이를 방지하기 위해서는 동기화 기법을 사용해야 하는데, busy waiting 기법을 사용하였다.**

**Busy waiting 기법은 parent process가 child process의 exit 여부를 계속 확인하면서 child가 죽을 때까지 기다리는 것이다. 계속 체크하기 위해서 while문을 사용하였으며, break를 위한 조건문으로는 is\_child\_alive라는 함수를 정의하여 사용하였다.**

**is\_child\_alive(tid\_t tid, int command, int\* \_status)는 process.c에 정의되어 있으며, tid는 exit 여부를 확인하고자 하는 child process의 tid를, command는 함수의 행동을 나타내며, \_status는 child thread가 exit했을 때의 status를 저장한다. 함수 내부에는 static으로 선언된 배열들 (child\_tids[128], is\_child\_waiting[128])이 있으며, 이 배열들은 thread의 생사를 저장하는 역할을 한다. Parameter인 command에 대한 행동은 다음과 같다.**

**1) child가 생성되었을 때 호출된다. thread\_create에서만 호출된다.**

**2) thread가 exit한 후, exit status를 parameter인 \_status에 저장하여 외부로 전달한다.**

**3) parameter로 전달받은 tid에 대해, 이미 wait가 호출된 적이 있는 tid인지 검사한다. child thread는 parent thread를 하나만 가질 수 있고, wait는 따라서 최대 한 번만 호출되어야 한다. 만약 같은 tid에 대해 wait가 2번 이상 호출된다면 오류이므로, thread를 비정상 종료한다. (exit(-1)) process\_wait 함수의 첫 줄에서 호출된다.**

**-1 : child가 exit되었을 때 command = -1로 함수가 호출된다. parameter로 전달받은 tid가 exit되었음을 함수 내부의 static 배열에 저장한다.**

**0 : parameter로 전달받은 tid가 exit되었는지 확인하고, exit되지 않았다면 true를, exit되었다면 false를 반환한다.**

1. **추진 일정 및 개발 방법**
   1. **추진 일정**
2. **project 요구 사항 분석**
3. **pintos manual 분석 후 개발 계획 수립**
4. **프로그램 개발**

* **argument passing 구현**
* **syscall 구현 (wait, write, halt, exit, exec, read 순)**
* **user memory access check 구현**
* **additional syscall 구현 (fibonacci, sum)**

1. **Debugging**
   1. **개발 방법**
2. **argument passing 구현**

* **argument passing을 위해서는 stack에 argument의 내용(word alignment가 고려된), 주소, argument count, return address를 넣어야 한다. 이를 위해서, pintos/src/userprog/process.c의 load 함수에서 setup\_stack() 함수가 수행된 후, esp를 감소시키면서 해당 스택에 argument를 넣는다.**

1. **syscall 구현**

* **wait를 구현하지 않으면 parent process가 child process가 실행되기 전에 혹은 실행 중에 종료되므로, child process의 출력 결과를 볼 수 없다. 그래서 개발 전에 while(1)로 구현하여 출력 결과만 확인할 수 있게 한 후, write, exec, halt, exit, read를 구현한 후, wait를 busy waiting 기법으로 구현한다.**

**busy waiting 기법을 사용하기 위해서 pintos/src/userprog/ process.c에 child process의 생사를 확인하기 위한 함수인 is\_child\_alive 함수를 구현한다. 이 함수를 process.c의 process\_wait 함수에서 while(is\_child\_alive(...))로 계속 체크하여 child가 exit되었다면 loop를 벗어나도록 한다.**

* **write, halt, exit, exec, read의 기능을 수행하기 위해, 각각 putbuf(), shutdown\_power\_off(), thread\_exit(), process\_exec(), input\_getc() 함수를 사용한다. return 값이 있다면 현재 interrupt frame인 f의 eax에 저장한다.**

1. **user memory access check 구현**

* **User가 잘못된 Memory access를 할 경우 error를 발생시켜야 하므로 User define 함수인 Is\_valid\_addr 함수를 정의하여 check할 것이며 함수 내부에서는 미리 정의되어 있는 두가지 함수를 이용한다.**

**하나는 넘어온 주소 자체가 valid한지(User address인지) check 할 수 있는 is\_user\_vaddr 함수이고, 하나는 그 user virtual address에 대해 kernel virtual address를 구해주는 pagedir\_get\_page 함수이다.**

**두가지 함수에서 오류가 발생이 되지 않으면 user영역의 address에 적절히 mapping된 page가 있다는 소리이므로 valid한 address에 접근했다고 판단 할 수 있다.**

1. **additional syscall 구현**

* **sum\_of\_four\_intgeger가 syscall number를 포함한 argument를 총 5개를 받는데 현재 최대 4개까지만 받을수 있도록 되어 있으므로 syscall number를 포함하여 5개까지 넘길 수 있도록 구현하여야 한다.**

**pintos/src/lib/user/syscall.c에 #define syscall4를 추가하여 Number, Arg0 , Arg1, Arg2, Arg3인자를 넘길 수 있게 할 것이다. 그리고 sum\_of\_four\_integer와 pibonacci system call을 추가하여 사용할 user\_program이 두 함수를 system call을 통해 요청 하도록 한다.**

* **또한 pntos/src/userprog/syscall.c에 또한 sum\_of\_four\_integers와 pibonacci 함수를 추가하여 적절히 실행 될 수 있도록 할 것이다.**
  1. **연구원 역할 분담**

1. **argument passing**

* **parse file name : 안시현 개발, 장지훈 검토**
* **construct ESP : 장지훈 개발, 안시현 검토**

1. **user memory access**

* **안시현 개발, 장지훈 검토**

1. **syscall infrastructure**

* **안시현 개발**

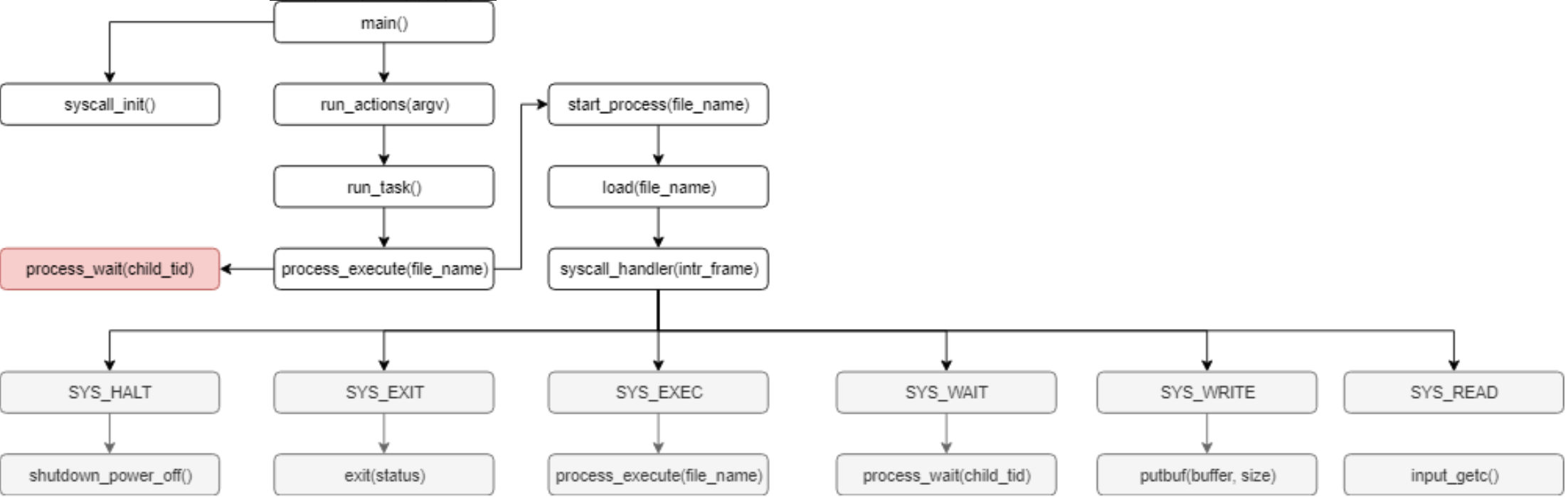
1. **syscall implementation**

* **halt, exit, exec, read, write : 안시현 개발, 장지훈 검토**
* **wait : 장지훈 개발, 안시현 검토**

1. **additional syscall**

* **공동 개발 및 검토**

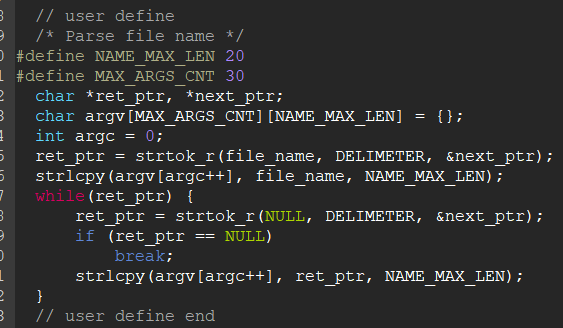
1. **연구 결과**
   1. **합성 내용**

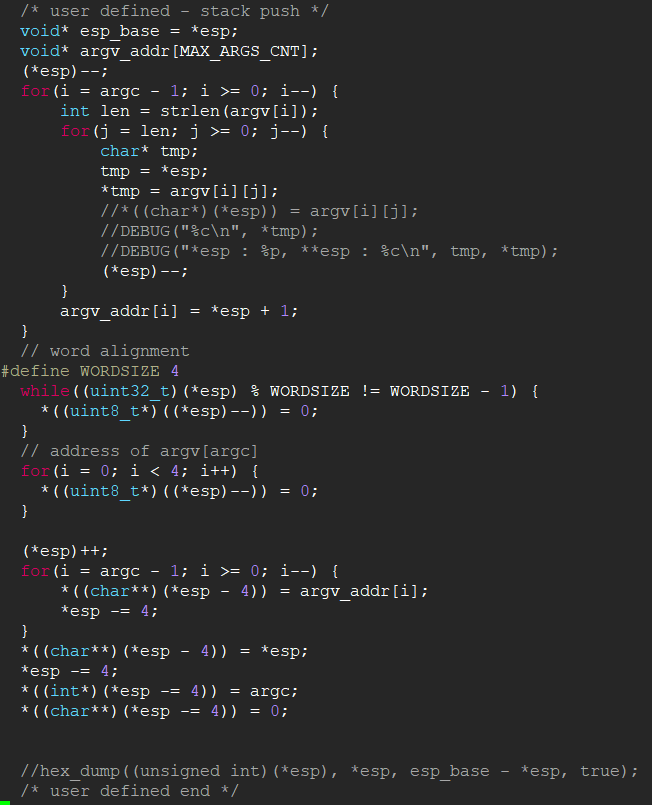
****

* 1. **제작 내용**
* **Argument Passing : 다음과 같은 방식으로 구현한다.**

1. **사용자가 입력한 command를 parsing하여 filename과 각각의 argument들로 분리한다.**
2. **esp를 이용하여 stack에 차례대로 push한다**
3. **push 순서는 차례대로 argv , word alignment , argv배열주소 , argc , return address 이다.**
4. **stack에 저장하기 위해서, 현재 stack의 시작 주소를 저장하고자 하는 자료형의 pointer 형으로 형변환한 후 (예를 들어, argc를 스택에 저장하기 위해서 (void \*)(f->esp)를 (int\*)(f->esp)로 형변환해서 저장하는 방식을 택했다.) 그 주소에 접근하여 실제 자료를 저장했다.**

**구현 코드**

****

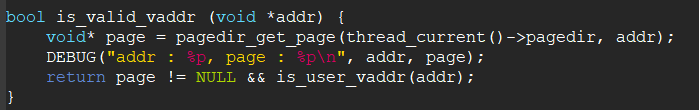
****

* **User Memory Access**

**1) 우선 임의의 주소에 page가 mapping 되어 있는지 check하기 위해서 pagedir\_get\_page함수를 이용 하였다.**

**pagedir\_get\_page 함수는 user virtual address에 대해 kernel virtual address를 구해주는 함수인데 , pagedir -> page table -> physical page로 접근해서 임의의 thread에 page가 mapping되어 있으면 page address를 return해주고 mapping된 page가 없으면 null을 return해 주는 함수이다. 그러므로 함수에 thread에 pagedir 정보도 넘겨 주어야 하는데, 이 부분은 thread.c에 정의되어 있는 thread\_current() ->pagedir을 parameter로 넘겨주어 해결하였다.**

**2) 두번째로 is\_user\_vaddr() 함수를 이용 하였다. 이 함수는 넘어온 주소가 kernel 영역일시 false를 return해서 user가 kernel영역에 접근하는 일을 방지 할 수 있도록 해주는 함수이다.**

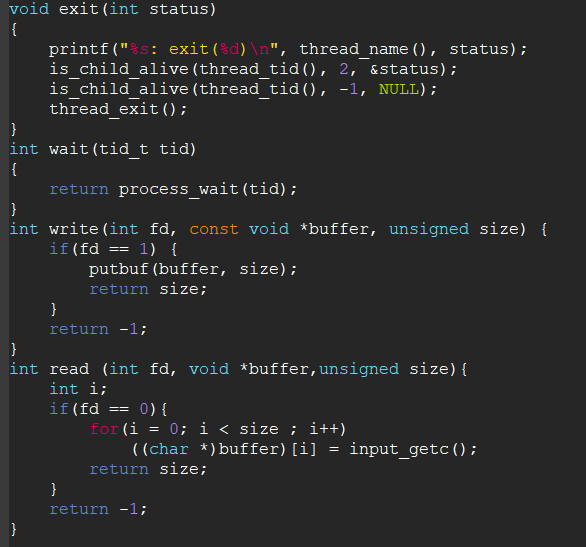
****

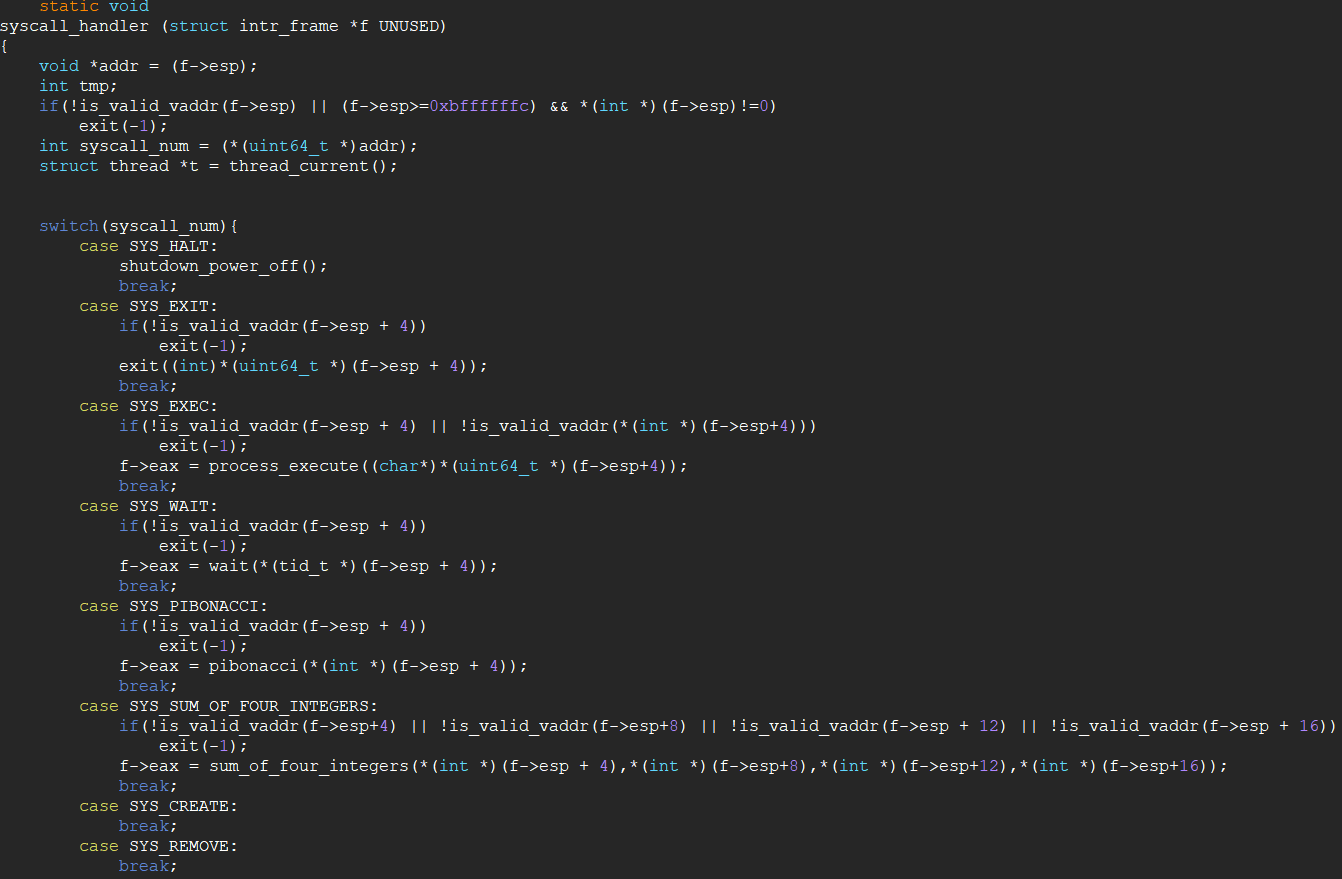
* **System Call (코드 구현은 설명 밑에 있습니다.)**

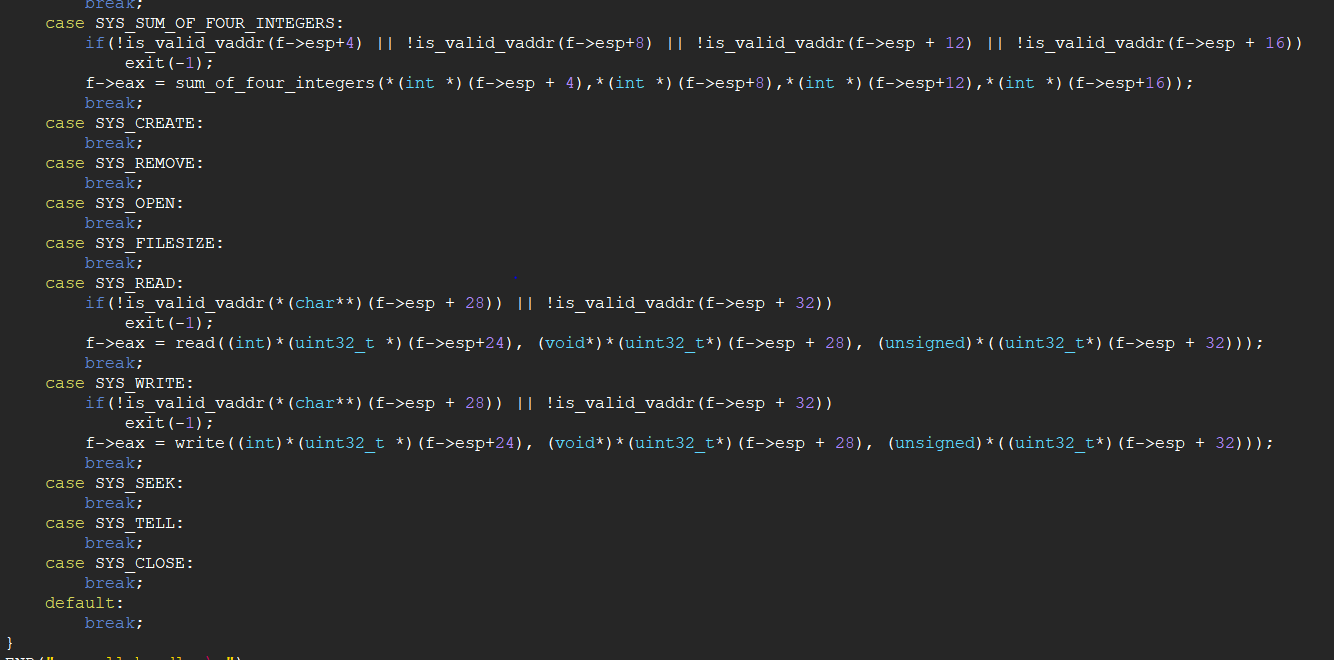
1. **void halt() : Pintos를 종료시킨다. Deadlock situation에 빠질 수 있고, 정보가 손실될 수도 있으므로 사용 빈도가 희박하다. shutdown\_power\_off() 함수를 호출한다.**
2. **void exit (int stauts) : 현재 user program을 종료시키고, status를 kernel에 전달한다. 만약 해당 process의 parent가 기다리고 있다면, status를 wait에 전달한다. 0은 성공을 나타내고, 0이 아닌 값들은 비정상 종료를 나타낸다. thread\_exit() 함수를 사용하였다.**
3. **pid\_t exec (const char \*cmd\_line) : cmd\_line에 주어진 실행 가능한 파일명과 argument들을 통해 user program을 실행시킨다. process\_execute 함수를 사용하였다.**
4. **int wait (pid\_t pid) : 부모 process가 자식 process가 exit할 때까지 기다리기 위한 함수이다. 동기화 기법이 사용되며, busy waiting 기법을 사용하였다. wait 함수가 실행되면 while loop를 돌면서 위에서 설명했던 is\_child\_wait 함수를 통해 input으로 받은 pid에 해당하는 child process가 exit 되었는지 확인하고, 만약 exit되었다면 loop를 벗어나 wait 함수의 실행이 끝나게 된다. 다음은 이를 구현한 코드이다.**

**while(is\_child\_alive(child\_tid, 0, &status)) {}**

1. **int read (int fd, const void \*buffer, unsigned size) : user program에서 stdin으로 input을 입력받을 때 사용되는 함수이다. input\_getc() 함수를 통해 키보드로부터 input을 입력받아 buffer에 저장한다.**
2. **int write (int fd, const void \*buffer, unsigned size) : buffer에 저장된 문자열을 size만큼 stdout으로 출력한다. fd = 1이면 stdout이므로, 이 때만 putbuf로 문자열을 출력한다.**

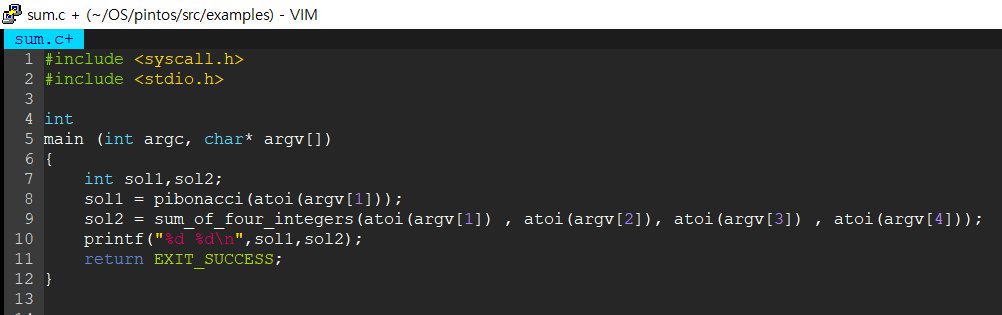
****

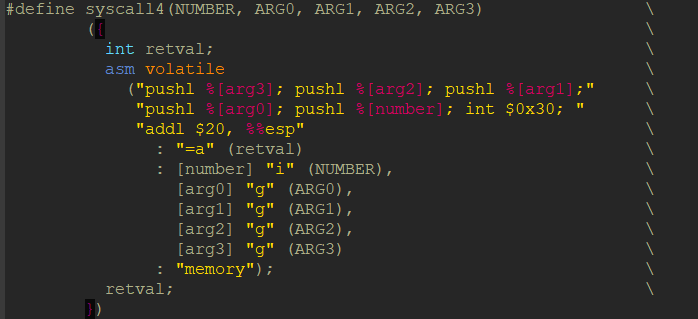
****

****

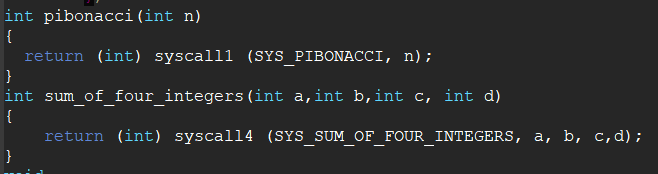
* **Additional System Call : 다음과 같은 순서로 구현한다.**

1. **sum.c 구현**

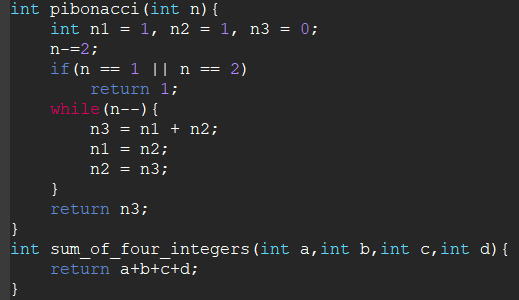
**  
 2) argument 4개 passing하는 syscall을 추가**

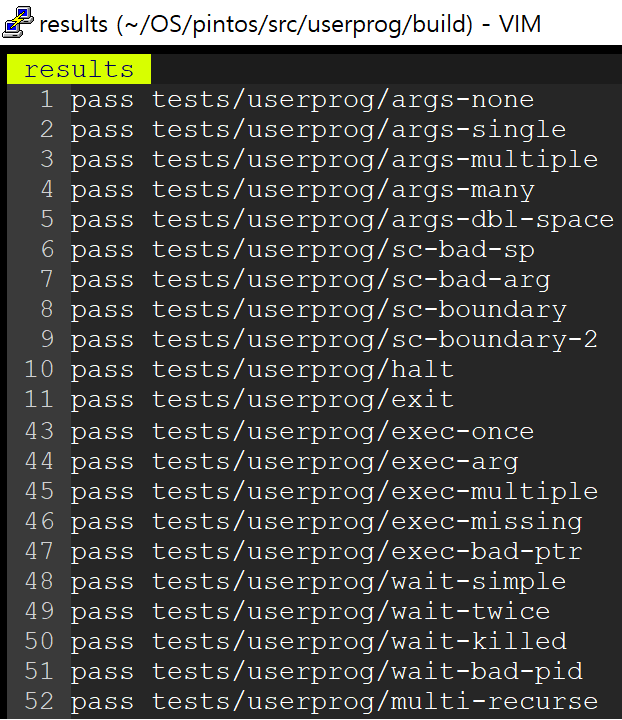
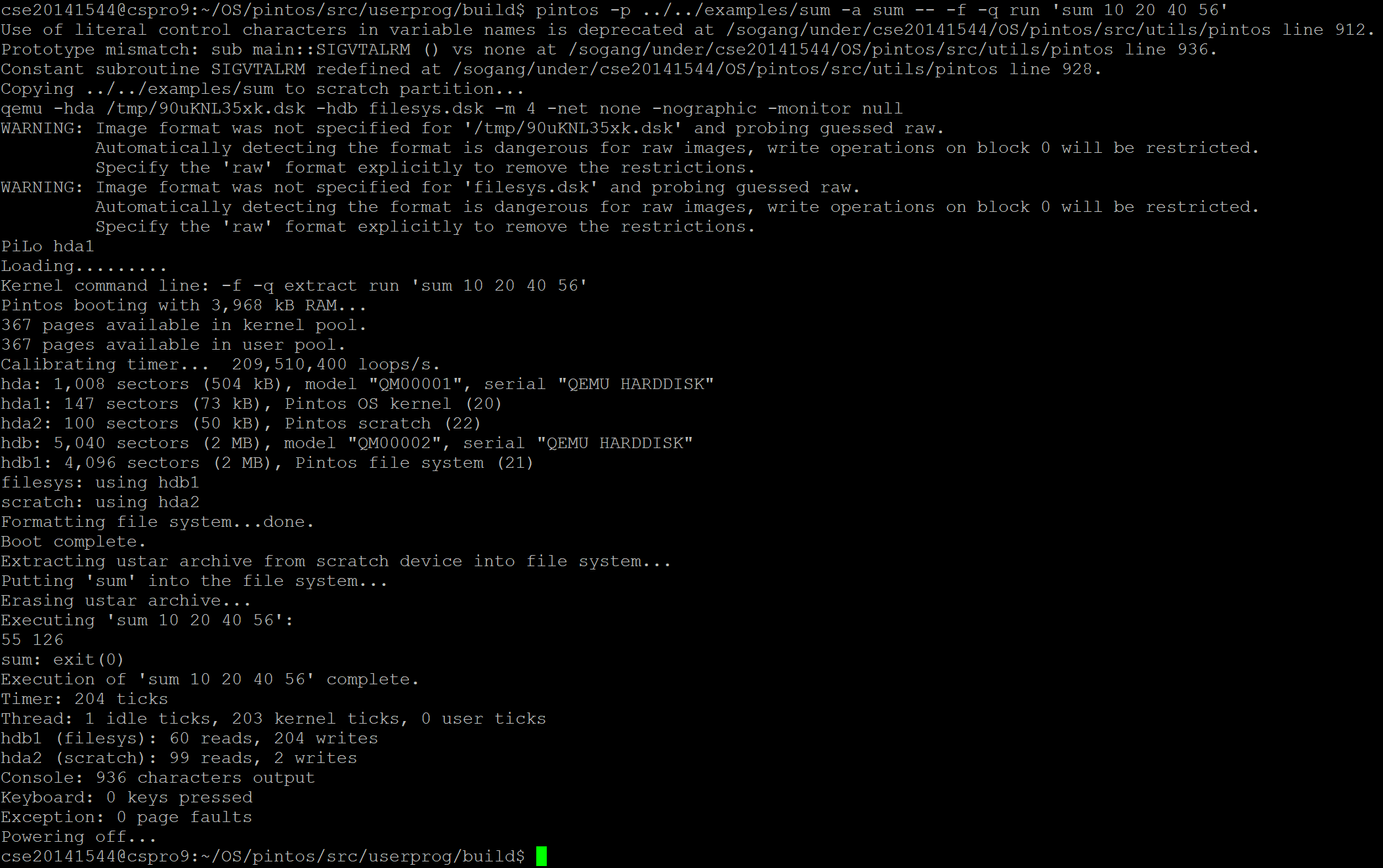
****

**2) src/lib/user/syscall.c 에 pibonacci와 sum\_of\_four\_integers 추가**

****

**3) src/userprog.syscall.c에 pibonacci와 sum\_of\_four\_integers 함수 추가**

****

* 1. **시험 및 평가 내용**
* **argument passing, syscall handler, user memory access 순으로 구현을 진행하였다. 구현이 완성되어 감에 따라 make check로 통과되는 test case의 수가 늘어났으며, 구현을 완성하면서 명세서에 제시된 21개의 test case를 모두 통과하였다.**
* ****
* **pibonacci, sum\_of\_four\_integers의 실행 결과는 다음과 같다.**
* ****

1. **기타**
   1. **연구 조원 기여도**

* **안시현 (50%), 장지훈(50%)**
  1. **소감**
* **장지훈 : 상당히 시간이 많이 소요된 프로젝트였다. 이전에 학교에서 수행했던 프로젝트들과는 달리, 요구사항 분석에만 절반 이상의 시간이 소요될 만큼 주어진 pintos를 이해하기 어려웠다. 하지만 pintos를 직접 구현하면서 이론으로만 접했던 process, thread, page, synchronization 등의 개념이 왜 등장하였고, 왜 필요한지를 설계자 입장에서 느껴볼 수 있었다.**
* **안시현 : 처음 봤을때는 상당히 어렵게 느껴져서 정말 막막했었다. 하지만 pintos manual을 수차례 읽고 나니 많은 부분이 이해가 돼서 잘 끝낼 수 있었다. 아직 다 구현하지는 않았지만 User program이 어떤 과정을 거쳐 호출되고 수행되는지 이해할 수 있게 되었고 막연하게만 알고있었던 개념인 thread나 process에 관해서도 코드를 보고 구현을 하는 과정에서 이해도가 높아 진 것 같아 뿌듯하다.**