**Pintos Project 1 : User Program Basic**

**(설계 프로젝트 수행 결과)**

과목 명 : [CSE4070]운영체제

담당 교수 : 소정민 교수님

조 / 조원 : 41조 20130859 송민규, 20130956 장동욱

개발 기간 : 2018.09.22~10.18

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

**제출일 : 2018.10.19**

**참여 조원 : 20130859 송민규, 20130956 장동욱**

1. **개발 목표**

* **기본적으로 있는 pintos는 필요한 명령을 수행하지 못하고, 바로 꺼지는 상태이다. 명령어를 처리하지 못하고, 기능의 실행에 필요한 부분이 없기 때문이다. 이를 수정하여, 명령어로 들어온 인자를 적절하게 처리해, 실행하고, kernel 모드에서의 동작들이 필요하면, 필요한 동작들을 수행하고, 오류가 있는 접근들을 적절히 처리하는 기능들을 추가할 것이다.**

1. **개발 범위 및 내용**
   1. **개발 범위**

* **Argument Passing**

**명령어로 입력 받은 내용을 시스템이 사용할 수 있도록 만들어야 한다. 넘어온 인자를 스택 구조에 적절히 쌓아주어야 한다. 이를 위해 process.c의 load 함수를 수정한다.**

* **User Memory Access**

**Kernel에 접근할 때 적절한 주소인지, User Mode에 필요한 Memory Access일 때 적절한 주소인지 검사하는 명령들을 각각 exceptions.c와 syscall.c에 추가한다. 추가로 구현할 인자가 4개 짜리 명령을 수행하기 위해서 lib/syscall-nr.h, lib/user/syscall.h, lib/user/syscall.c를 수정한다.**

* **System Call**

**syscall.c에서 필요한 System Call의 종류에 따라 실제 system call을 호출한다. 이에 필요한 인자를 넘겨주고, 각 system call의 동작들을 적절한 곳에서 구현한다. 대부분은 syscall.c에서 구현하며, wait와 exec은 process.c에서 구현한다. 이때 thread의 상태와 결부되는 부분이 많기 때문에 threads/thread.c와 threads/thread.h를 수정해준다.**

* 1. **개발 내용**
* **Page & Thread**

**Pintos가 수행되는 작업은 process, 즉 thread로 관리된다. 각 thread는 스스로가 일정한 메모리공간을 가지고 있는 것처럼 보이지만, 실제로는 page directory에서 physical memory를 연결해 주는 내용을 관리하고 있다. 실제 메모리에선 User Memory는 0x00000000부터 PHYS\_BASE 까지를 점유하고, Kernel Page는 PHYS\_BASE 이후를 점유한다. Kernel Pages에서 thread관리에 필요한 memory가 사용된다.**

* **Valid Address**

**User process는 User 영역으로 써야하는 메모리 공간만 사용해야한다. 이에 접근하기 위해선 thread가 가지고 있는 page\_directory를 거쳐야한다. Kernel memory에 접근하지 않고, NULL 값이 아니며, 실제 메모리 공간에서도 mapping이 되어 있어야 한다. 위 조건에 맞춰 검사하는 함수를 기존 pintos가 제공하는 기능을 이용하여 검사하는 코드를 syscall.c에 작성한다.**

* **Synchronization**

**OS는 여러 threads를 각각 concurrent하게 구동되도록 관리한다. 이때, 부모-자식 threads는 완벽하게 독립되지 못한다. 부모 thread가 종료되기 위해서는 본인이 호출한 자식 thread의 수명이 끝난 것을 확인하여야 한다. 이를 구현하기 위해서 여러 기법들이 있는데, 본 프로젝트에서는 busy-waiting이라는 기법을 사용한다. 부모 thread가 무한 루프를 돌면서 자식 thread의 상태가 dying인지 확인하는 기법이다.**

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **9.22-28 코드, 명세서 탐독 및 자료조사, 9.29-10.12 코드 작성, 10.13-10.18 디버깅 및 보고서 작성**
  1. **개발 방법**
* **Argument Passing**

**userprog/process.c 의 load 함수에서 esp가 설정된 이후부터 esp가 가르키는 곳에 원하는 인자를 push하는 기능을 구현한다. 이때 push하는 것, parsing하는 것은 별도의 함수, 매크로를 작성한다.**

* **User Memory Access**

**userprog/syscall.c의 syscall\_handler에서 각 인자들 주소에 있어 valid 검사를 한다. 이 valid 함수는 syscall.c에 구현한다. 또한 page\_fault 시에 해당 주소가 NULL이거나 kernel space가 아닌경우 -1을 반환하며 종료되도록 userprog/exception.c에서 page\_fault함수를 수정한다.**

* **System Call**

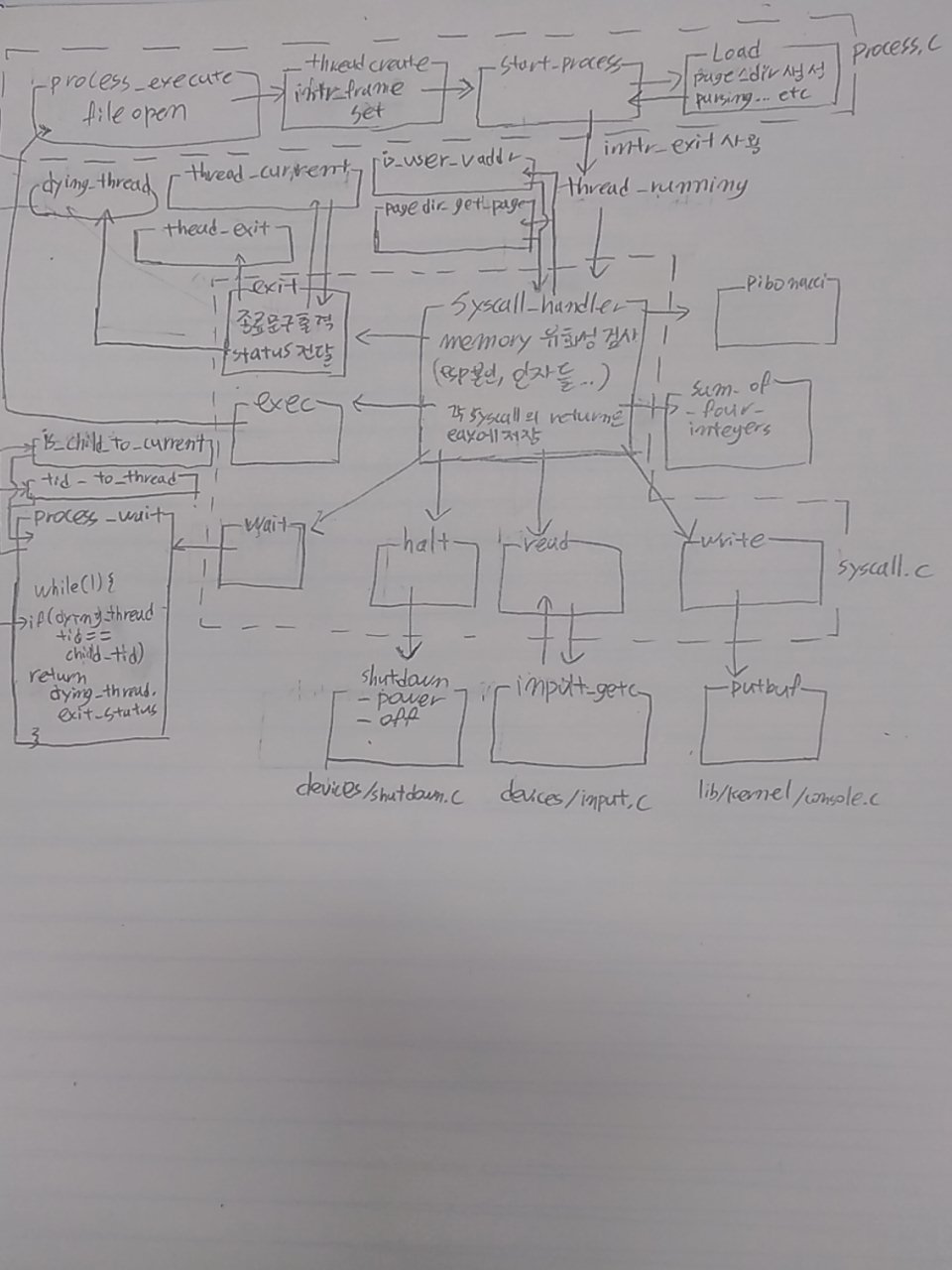
**userprog/syscall.c의 syscall\_handler에서 요청받은 system call에 따라 원하는 함수들을 호출한다. 반환값이 있는 경우는 eax에 저장한다. 각각의 system call에 대응하는 함수도 syscall.c에 별도로 작성해 준다.**

* **Synchronization**

**userprog/process.c 의 process\_wait 함수에 synchronization을 하는 기능을 구현한다. 무한루프를 돌면서 가장 최근에 종료된 thread가 자신이 넘겨받은 정보와 일치하는지 확인하고, 만일 일치하면 종료된 thread의 상태를 반환한다. 종료되는 thread의 정체와 정보, 상태를 알기위해, thread의 구현에 여러가지를 추가해야 한다. 우선, thread.h에 thread 구조체에 자식관계를 정리하는 child\_elem, child\_list를 추가한다. 또한 대기 여부를 확인하는 wait 변수, 그리고 exit\_status 변수를 추가한다. thread.c에 tid를 기준으로 thread를 찾아내는 tid\_to\_thread 함수와, 부모 자식관계를 확인하는 is\_child\_to\_current 함수를 추가한다.**

* 1. **연구원 역할 분담**
* **각자 일정정도 구현하면서 막히는 부분을 같이 풀어보는 식으로 구현한다. 일정정도 진행되면 진행된 것을 한 명의 코드로 취합한 다음, 남은 구현을 마무리하고 결과 검토와 디버깅을 진행한다. 이때 다른 구성원은 보고서 작성을 한다.**

1. **연구 결과**
   1. **합성 내용**

* ****
  1. **제작 내용**
* **Argument Passing**

**명령어로 들어온 인자들을 process.c에 정의된 load 함수까지 넘겨 받는다. 이때 인자들은 const char \* 형태로 넘겨 받는다. 현재 load는 불려질 작업을 관리할 thread가 이미 생성되어 있는 상태이다. Thread는 각자의 빈 메모리 공간을 바라보고 다룰 수 있는데, 이를 실제 물리적인 메모리와 mapping 시키도록 page directory를 생성한다. 그 다음으론 인자들의 문자열을 file\_name\_to\_argv 함수로 parsing해서 별개의 배열에 저장해 놓는다. 그 외에도 여러 load의 필요한 동작을 수행한 후, 인자들을 넣을 stack 자료구조를 setup\_stack 함수로 준비 한다. Esp가 빈 stack의 top을 가리키게 되어 있다. 미리 parsing한 배열을 esp가 가리키는 주소를 기준으로 복사하고, esp의 값을 조정하는 것을 반복한다. esp값을 조정한 후, esp의 값을 별도의 배열에 저장해준다. 이를 완료하면, parsing된 인자가 word\_size 에 맞아 떨어지지 않는지 검사하고, word\_allignment를 해준다. 입력된 전체 길이와 구분자를 합친 길이를 4로 나머지 연산하고, 4와 차이나는 길이만큼 0으로 메모리에 set해준다. 그 후, 구분자를 입력한다. 인자의 개수만큼의 주소를 저장할 만큼의 메모리 공간을 계산하고 앞에서 esp의 주소를 담은 배열을 메모리에 저장한다. 이를 통해 각 인자들에 접근할 수 있는 주소들을 메모리에 담게 된다. 그리고 이 주소들을 가리키는 주소를 다시 메모리에 저장하고, 인자의 개수와 return address를 저장한다. return 주소는 항상 0x00000000으로 한다.**

* **User Memory Access**

**주소가 NULL 이 아니고, User Memory 영역이며, 실제 메모리에 mapping되어 있으면 true를, 그 이외에는 false를 return하는 valid 함수를 만든다. User Memory 영역인지 검사하는 함수는 vaddr.c에 구현되어 있는 is\_user\_vaddr함수를 이용하고, 실제 메모리 영역인지 검사하는 함수는 pagedir.c에 pagedir\_get\_page 함수를 이용한다. syscall\_handler에서 각 syscall 함수를 호출하기 전에 인자들의 개수에 맞춰 해당 함수를 반복해서 검사하는 것으로 접근의 유효성을 확인한다.**

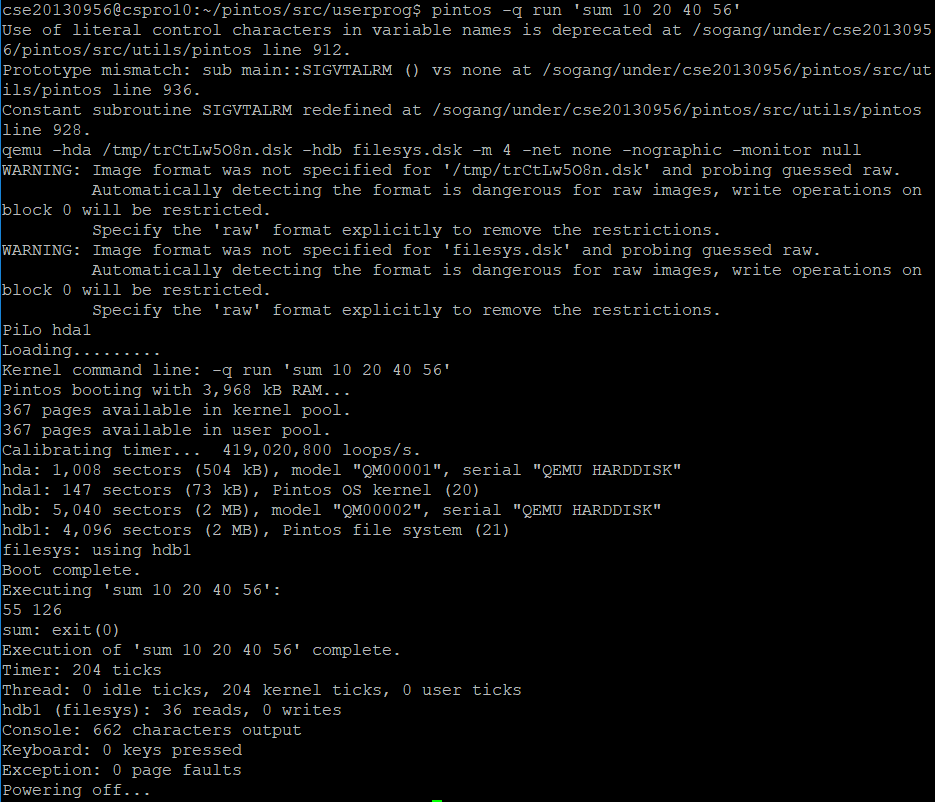
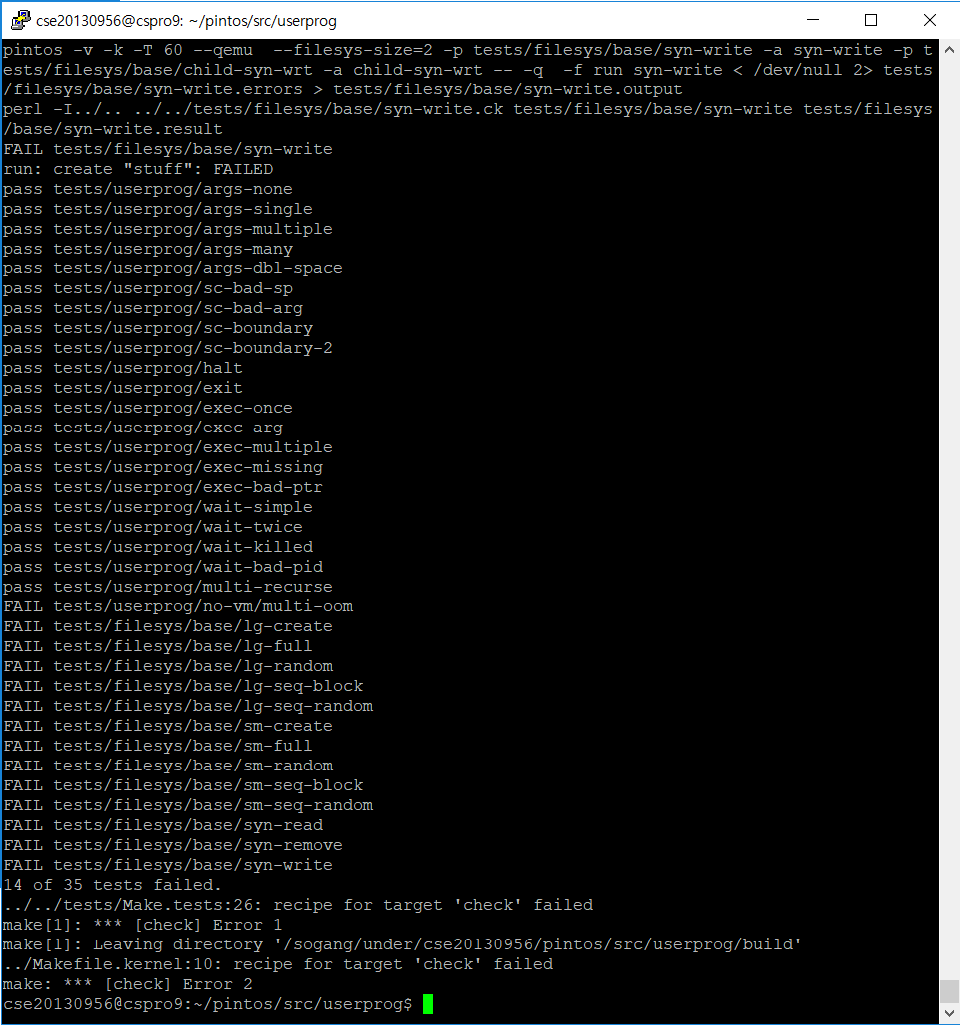
* **System Call**

**halt는 device.c에 구현된 shutdown\_power\_off()를 호출한다. exec은 명령이 유효한 것인지 확인하고, process\_execute를 호출한다. process\_execute의 반환값을 return한다. read는 input\_getc함수를 이용해서 buffer에 값을 입력받도록 한다. write는 putbuf 함수를 이용해서 화면에 출력한다.**

**synchronization의 구현은 exit과 wait에서 이뤄진다. 부모-자식 threads는 완벽하게 독립되지 못한다. 부모 thread가 종료되기 위해서는 본인이 호출한 자식 thread의 수명이 끝난 것을 확인하여야 하기 때문에 exit과 wait에서 이를 수행할 수 있어야 한다. 우선 자식 thread가 exit이 되면 exit 상태 메시지를 출력하고, 본인의 정보를 dying\_thread에 저장한다. thread\_exit이 되면 더이상 해당 정보를 알 수 없기 때문이다. wait의 경우, 기본적으로 process\_wait를 호출한다. process\_wait에서는 기본적으로 3가지를 우선 검사한다. is\_child\_to\_current를 이용해서 부모자식 관계가 아닌 tid가 넘어왔는지 검사한다. 이미 exit된 경우도 이에 해당한다. 그리고 넘겨받은 tid를 의 thread주소를 받아 NULL인지 확인하고, 부모가 이미 기다려 준 경우인지 확인한다. 해당 경우들은 전부 -1을 반환하도록 한다. 그리고 나서 자식thread에게 부모 thread가 기다리는 상태라는 것을 wait변수를 통해 기록하고, 무한 루프를 돌도록 만든다. dying\_thread를 참조해서, 가장 최근에 exit 처리된 thread의 tid가 부모가 넘겨 받은 tid와 같으면 dying\_thread에 남겨진 exit\_status를 반환하도록 만든다.**

* **Additional System Call**

**userprog/syscall.c 에서 pibonacci, sum\_of\_four\_integers를 수행하는 함수를 구현한다. 그러나 이를 kernel이 인식하고 다룰 수 있도록, syscall-nr.h에 적절한 call\_number를 부여한다. 또한 kernel은 인자가 4개인 system call은 다룰 수 없기 때문에, lib/user/syscall.c에 구현된 syscall3을 참조해서, pushl %[args3], addl 20, [arg3] “g” (ARG3)을 추가한 syscall4를 추가한다. 그리고 해당 파일에도 pibonacci, sum\_of\_four\_integers를 구현한다. pibonacci는 syscall1을, sus\_of\_four\_integers는 syscall4를 사용한다.**

* 1. **시험 및 평가 내용**
* 

1. **기타**
   1. **연구 조원 기여도**

* **송민규(50%), 장동욱(50%)**
  1. **소감**
* **협업을 하기 위한 방법이나 tool이 정리되지 않아 업무분담의 불균형이 있었다. 또한 debug가 용이하지 않아 어려움이 많았다. 이번 프로젝트에서 경험한 것을 참조해 다음 프로젝트를 수월하게 진행할 수 있어야겠다.**
* **디버깅 수단이 출력(printf, hexdump), backtrace 등으로 제한되어 시스템의 기제를 파악하고 디버깅하는 데에 어려움이 많았다. 복수의 thread들이 concurrent하게 수행되도록 구현된 특성상, control을 갖고 있지 않은 thread(waiting status)의 수행 내용을 이론적으로 추론해 내야 한다는 점이 어려웠다.**