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

담당 교수 : 김영재

조 / 조원 : 고성빈

개발 기간 : 10/3 ~ 11/1

1. **개발 목표**

* 핀토스에서 유저 프로그램이 종료될 때, Process Name: exit(exit status)를 출력되도록 해준다.
* 80x86 calling convention에 따라 argument passing과 메모리 할당을 진행한다.
* halt, exit, exec, wait와 read, write의 일부 System Call 구현
* 프로세스가 kernel 메모리 영역을 침범하지 않도록 확인

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

* Argument Passing을 구현 시 명령어로 입력한 문장이 각 단어로 parsing되어 전달된다. 이렇게 parsing된 각 단어를 명령어로 인식해 instruction을 수행하거나, 인자로 스택에 저장해주어야 한다.

1. User Memory Access

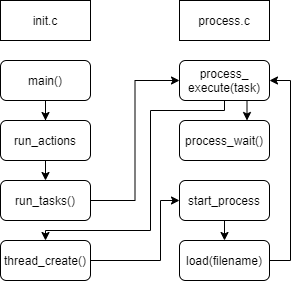
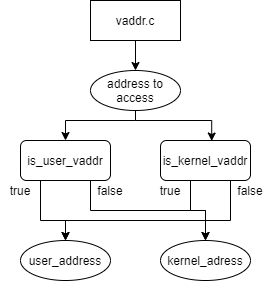
* 프로세스가 수행 중 kernel 메모리 영역에 접근하지 못하도록 막아주어야 한다.

1. System Calls

* 프로젝트에서 요구하는 System call (halt, exit, exec, wait와 read, write의 일부 기능)을 구현하고 non busy waiting을 사용해 system call이 문제없이 사용되도록 한다.
  1. **개발 내용**
* Argument Passing
  + Argument Passing을 통해 받은 문자열을 parsing한 뒤 스택에 저장한다. 스택은 높은 address에서 낮은 address로 쌓이는 구조로, 가장 윗부분에는 입력받은 문자열이 각각 ‘/0’로 구분되어 차례대로 저장되어 있고, 그 다음으로는 구분된 각 문자열의 시작 주소가 저장되어있다. 그리고 parsing된 단어의 갯수와 return address가 저장되어있다. 이를 순서대로 나타내면 낮은 주소값부터 차례로 return address – arg count – address of argv addresses – address of argv[0] ~ argv[n](argv[n] = NULL) – 0 alignment - argv[0]~argv[n-1] data이다. return address부터 argv[n]의 address까지는 모두 1word(4byte)에 저장되고, argv[0] ~ argv[n-1] data는 byte 단위로 저장되며 word 단위를 맞추기 위해 argv[0] data 앞에 0 alignment가 되어야 한다.
* User Memory Access
  + Pintos의 virtual memory 구조는 0~3GB의 user memory 영역과 3~4GB의 kernel memory 영역으로 나뉘어져 있다. 이는 OS 구간을 2 level로 나누어 컴퓨터 내의 resource에 대한 접근 및 명령을 kernel level에 두어 보호하기 위함이다. 이때 user memory에서 작동하는 코드가 system call을 통하지 않고 kernel memory에 접근하게 되면 OS에서 막아주어야 한다. 이를 위해 메모리에 접근할 때마다 메모리의 주소를 확인해 kernel memory(3~4GB 사이)일 경우 접근하지 못하도록 막아주어야 한다.
* System Calls
  + OS는 컴퓨터 내의 resource에 대한 접근 및 명령을 막기 위해 user level과 kernel level 두 단계의 모드를 둔다. 이때 user level에서 작동하는 프로세스가 kernel level의 기능을 필요로 하는 경우 system call을 통해 kernel level의 기능을 요청한다.
  + halt()는 shutdown\_power\_off() 함수를 통해 작동중인 Pintos OS를 종료하는 system call이다.
  + exit()은 현재 실행중인 user program을 종료하고, 그 종료 상태를 kernel로 전달하는 system call이다.
  + exec()는 process\_execute() 함수를 통해 현재 process에서 child process를 생성하는 system call이다.
  + wait()는 child process가 작업을 마치고 종료하는 것을 기다리는 system call이다. child ID가 올바른지 확인하고, child가 종료된 뒤에는 child의 exit status를 반환받는다.
  + write()와 read()는 I/O device로 데이터를 전송하거나 데이터를 전달받는 system call이다. 현재 프로젝트에서는 STDIN과 STDOUT을 통한 입출력만 구현하는 것을 목표로 한다.
  + 유저 레벨에서 시스템 콜 API를 호출할 시 threads/init.c 파일의 run\_actions가 먼저 작동한다. 이후 run\_task 함수가 인자로 전달되어 수행되고, run\_task 함수에서는 process\_execute를 수행하고, execution이 종료될 때까지 wait하게 된다. process\_execute 내부에서는 thread\_create 함수가 호출되고, 함께 인자로 전달된 start\_process가 사용된다. start\_process 내부에서는 load 함수가 호출되어 stack에 parsing된 데이터를 넣어준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 최대한 강의에서 많은 내용을 배우고, 중간고사 기간이 시작되기 전인 10/9~10/11에 1차적인 구현을 시작한다. semaphore의 구현을 제외하고 구현했지만, semaphore가 구현되지 않아 exec와 wait는 통과하지 못했고, user memory address 체크를 할 때, 해당 블록만 user address에 포함되면 된다고 생각해 vaddr + 4를 체크하는 대신 vaddr + 3이 user address인지 체크해주어 대다수의 테스트를 통과하지 못했다.
* 시험이 끝난 뒤 10/30~10/31에 2차 구현을 시작했다. semaphore를 구현했고 에러를 확인해 21개의 테스트케이스를 모두 작동하는 것을 확인했다. 이후 11/1까지 보고서 작성을 마쳤다.
  1. **개발 방법**
* userprog/process.c의 load 함수의 내부에 입력받은 문자열을 parsing한 뒤 stack에 저장해주는 기능을 구현해야 한다. 또한 userprog/syscall.c 내부의 syscall\_handler 함수에서 각 명령어와 parameter를 전달받은 뒤 system call을 구현해주어야 한다. 해당 과정에서 접근하는 주소값이 user\_address인지 확인하는 함수를 사용해야 하는데, 이는 threads/vaddr.h 내부에 is\_user\_vaddr 이라는 함수로 구현되어 있어 이를 활용하면 된다.
* 프로세스의 wait 구현을 위해서 semaphore를 사용해주어야 한다. thread 구조체에 child의 정보를 담은 list와 semaphore child\_lock, memory\_lock을 만들어 주어야 한다. wait시 parent는 child\_lock을 wait하고 이를 통해 child의 기능 종료를 확인한다. 이 시점에서 child는 완전히 종료되지 않고 데이터를 남겨둔 채로 memory\_lock을 기다리고 있다. 이후 parent는 child에서 필요한 정보를 얻어낸 뒤 memory lock을 해제해 child가 메모리를 해제할 수 있도록 해준다.

1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing
3. User Memory Access
4. System Calls

이 때, wait와 exit 사이에서의 synchronization은 다음과 같다.



* 1. **제작 내용**

1. Argument Passing

* 입력받은 명령어 문자열은 instruction과 operand로 자동으로 나누어지지 않기 때문에 프로그램에서 함수를 처리하기 위해서는 입력받은 명령어를 단어 단위로 구분해주어야 한다.
* process\_execute 함수와 load 함수 내부에서 인자로 전달받은 문자열을 parsing해주었다. 이를 위해 strtok\_r 함수를 사용해주었다. 이후 구분된 각 단어를 stack에 저장해주었다.

1. User Memory Access

* vaddr.h와 vaddr.c 내부에 구현된 is\_user\_vaddr 함수와 is\_kernel\_vaddr 함수를 사용해 메모리가 user address에 접근하려고 하는지 kernel address에 접근하려고 하는지 확인해주어야 한다.
* syscall.c 파일에서는 ifuser 함수를 선언해주었다. 확인할 주소를 인자로 입력받아, 해당 주소를 패러미터로 is\_user\_vaddr 함수를 호출해 false가 반환될 경우(kernel address에 접근하려는 경우) exit(-1)으로 프로그램을 종료시켜준다.
* 최초 구현 시 메모리가 속한 블록만 user address에 속하면 된다고 생각해 address + 3을 인자로 넘겨주었지만 에러가 발생하는 것을 확인했고, address + 4를 넘겨주어야 정상 작동하는것을 확인할 수 있었다.

1. System Calls

* halt의 구현에는 shutdown\_power\_off 함수가 필요한데 이는 threads/interrupt.h 내부에 선언되어 있다.
* exit는 종료하는 thread의 이름과 상태를 확인할 thread\_name 함수와 status 변수를 사용해 출력 상태를 출력해주고, 종료할 thread의 exit\_status에 현재 status를 저장해주어야 한다. 이후 thread\_exit을 호출한다. thread\_exit 내부에서는 process\_exit을 호출하고, 프로세스가 제거된 이후에는 할당된 데이터를 지워준다.

process\_exit은 프로세스가 제거되기 전에 현재 thread가 가리키고 있는 pgd를 초기화해준 뒤, signal child\_lock을 통해 waiting하고 있던 parent가 다시 코드를 실행해 child의 exit status를 확인할 수 있게 해준다. parent가 exit\_status를 확인하기 전에 process\_exit가 종료되어 thread\_exit 함수에서 할당된 데이터를 지우게 되면 문제가 발생하므로, signal child\_lock이후에 wait mem\_lock을 통해 parent가 exit\_status 확인을 마칠 때까지 기다리도록 해줘야 한다.

* exec는 process.c에 선언된 process\_execute 함수를을 호출하며 인자로 전체 명령어를 넘겨준다. process\_execute에서는 instruction이 아닌 명령어 전체를 인자로 넘겨받았기 때문에 내부에서 명령어를 parsing해주는 코드를 추가해주었다. 이후 threads/thread.c에 선언된 thread\_create 함수에 parsing된 문자열과 기존 문자열을 parameter로 넘겨준다. 호출된 thread\_create가 해당 명령어를 수행할 thread를 생성해준다.
* wait는 process\_wait 함수를 호출하며 paramenter로 기다릴 process id(pid)를 넘겨준다. process.c에 선언된 process\_wait 함수에서는 현재 실행중인 thread 중에서 자신이 기다리려는 child thread를 탐색한다.

wait child\_lock을 통해 기다리다가 child process가 종료되어 process\_exit에서 signal child\_lock을 해주면 child의 exit\_status를 저장하고 child list에서 해당 child를 제거해준다. 이후 signal mem\_lock을 통해 모든 필요한 데이터를 읽었다는 것을 신호로 보내 child process가 나머지 데이터를 지울 수 있도록 한다.

* read와 write는 IO device로 데이터를 전송하거나 데이터를 전달받는 함수이다. 현재 프로젝트에서는 STDIN과 STDOUT을 통한 입출력만 구현하면 되므로 stdin, stdout 버퍼에서 데이터를 입력받게 된다.

read는 input\_getc 함수를 통해 입력을 받고 \0가 들어올 경우 입력을 종료한다.

write는 putbuff 함수를 통해 파일에 출력해준다.

1. Additional System calls

* 새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술
* 미구현
  1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**