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

담당 교수 :

조 / 조원 :

개발 기간 :

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

simulated pintos disk에 pintos를 위한 user program을 이동 시킨 후 pintos가 해당 프로그램을 실행하도록 한다. 이 때 아래에서 후술할 argument passing, user memory access, system calls 등의 구현이 필요하다. 마지막으로 system calls 에, fibonacci, max-of-four-int도 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

~/pintos/src/userprog 에서 pintos --filesys-size=2 -p ../examples/echo -a echo --f -q run 'echo x' 와 같은 명령을 실행했을 때 바로 고려해야하는 단계이다. esp의 값이 4B aligned에 맞게 줄어들게 되고, echo x 와 연관된 정보들이 4B aligned 된 형태로 들어가게 된다.(stack 형식)

1. User Memory Access

사용자가 커널의 메모리에 접근하는지 확인하는 경우를 방지한다. 커널의 메모리에 접근하면 exit(-1)을 통해 pintos가 종료된다.

1. System Calls

예를 들어 ~/pintos/src/userprog 에서 pintos --filesys-size=2 -p ../examples/echo -a echo --f -q run 'echo x' 와 같은 명령을 실행했을 때 echo는 내부적으로 write()함수를 호출하고, 이는 SYS\_WRITE 라는 인터럽트 시그널을 발생시킨다. 이 시그널이 들어왔을때 호출해야할 적절한 함수를 작성해야한다. 1에서 구현한 stack에 쌓여있는 정보들을 적절하게 가져와서 적절한 함수를 작성하는 것이 가능하다. SYS\_WRITE 뿐만 아니라, SYS\_HALT, SYS\_WAIT, SYS\_WRITE, SYS\_READ, SYS\_FIBO... 등 이번 프로젝트 명세서에서 나타내는 모든 신호에 대해서 구현한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + ~/userprog/process.c 에 construct\_stack() 함수를 통해서 이 부분을 구현하였다. 여기서 strtok\_r() 이라는 함수를 통해 argument들을 분리할 수 있었다. 이제 이렇게 분리된 argument들을 esp의 값을 줄이면서 stack에 저장한다. 모든 arguments를 저장한 후에는 4B aligned를 만족하게 더미 값을 넣는다. 그 후, arguments들이 저장된 주소들을 stack에 저장하고, arguments의 개수, return address를 넣으면 argument passing 단계는 끝이 난다.
* User Memory Access
  + 만약 user process가 user memory 공간을 넘어서 kernel memory 까지 사용하게 된다면, 이는 매우 위험해질 것이다. 또 unmapped virtual memory가 존재할 경우도 데이터를 메모리에 저장할 수 없거나 어디 저장할지 모르는 상황이 벌어질 수도 있기 때문에 매우 위험해질 것이다. 이를 막아야 한다.
  + userprog/pagedir.c 에 unmapped virtual memory를 확인할 수 있는 pagedir\_get\_page() 라는 함수가 존재한다. 이를 통해서 확인 가능하다.

threads/vaddr.h 에는 is\_kernel\_vaddr() 함수가 존재하고, 이를 통해 현재 인자로 넘어간 주소가 kernel 영역인지 아닌지의 여부를 판별 가능하다. 이 두 함수를 통해 invalid memory access를 방지할 수 있다.

* System Calls
  + user는 kernel의 api 기능들을 사용할 수 있어야 한다. system calls는 이러한 kernel의 api를 user가 사용할 수 있게 해준다.

halt : devices/shutdown.h에 정의된 shutdown\_power\_off() 함수를 호출하여 pintos를 끝낸다.

exit : 현재 user program을 끝내고, kernel로 status를 반환한다. 만약 이 프로세스의 부모가 이것을 기다리고 있다면, 이 반환값을 받는다. 0면 성공을, nonzero면 에러를 의미한다.

exec : 인자로 주어진 cmd\_line에 해당하는 실행가능한 파일을 실행시킨다. 그리고 그 새로운 프로세스의 id(pid)를 반환한다.

wait : 인자로 주어진 pid가 끝날 때 까지 기다린다. 이 때, pid의 자식이 끝날때 까지 wait 해야한다.

read : fd를 buffer로 size만큼 읽는다. 이번 프로젝트에서는 fd가 0(stdin)일 때만을 고려한다. 읽은 byte의 수를 반환한다.

write : fd에 buffer로 size만큼 기록한다. 이번 프로젝트에서는 fd가 1(stdout) 일때만을 고려한다. 적은 byte의 수만큼 반환한다.

fibonacci : 1,2번째 피보나치 수가 1일때, 인자로 주어진 n번째 피보나치 수를 반환한다.

max\_of\_four\_int : 인자로 주어진 4개의 수 중, 가장 큰 수를 반환한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

인자 개수에 따라 서로 다른 syscall이 실행된다. 이때 syscall()로 system call number를 넘긴다. 우선 유저모드에서 커널모드로 mode bit를 바꾼다. 그리고 0x30의 시스템 콜 핸들러를 호출한다. 그리고 userprog/syscall.c의 syscall\_handler()가 실행되고, system call number에 따라 그에 해당하는 함수를 esp를 통해 정보를 가져와서 실행한다. 그리고 끝난 후에는 mode bit를 유저모드로 바꾼다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

9/24 : argument passing

9/25 : user memory access

9/26~9/28 : system call handler

9/29~9/30 : system call implementation

10/1 : additional implementation, 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)**

우선 process\_wait를 반복문을 많이 돌림으로서 wait의 기능을 어느 정도 수행할 수 있도록 하였다. userprog/process.c 의 load()함수에서 argument passing을 직접 작성한 construct\_stack() 함수를 통해 수행한다.

그 후 userprog/syscall.c를 수정한다. /lib/syscall-nr.h를 참조하여 system call number을 알아낸 후, syscall\_handler()에서 switch 문을 통해 적절한 system call을 수행하도록 한다. 이 때 system call의 인자들은 f->esp의 정보를 통해 알아낼 수 있다. 그리고 반환값은 f->eax에 저장을 한다. 이 때, threads/vaddr.h 의 is\_kernel\_vaddr() 함수를 통해 만약 유저가 kernel 메모리 영역을 침범하려고 하면 exit(-1)을 호출하도록 예외처리를 해준다.

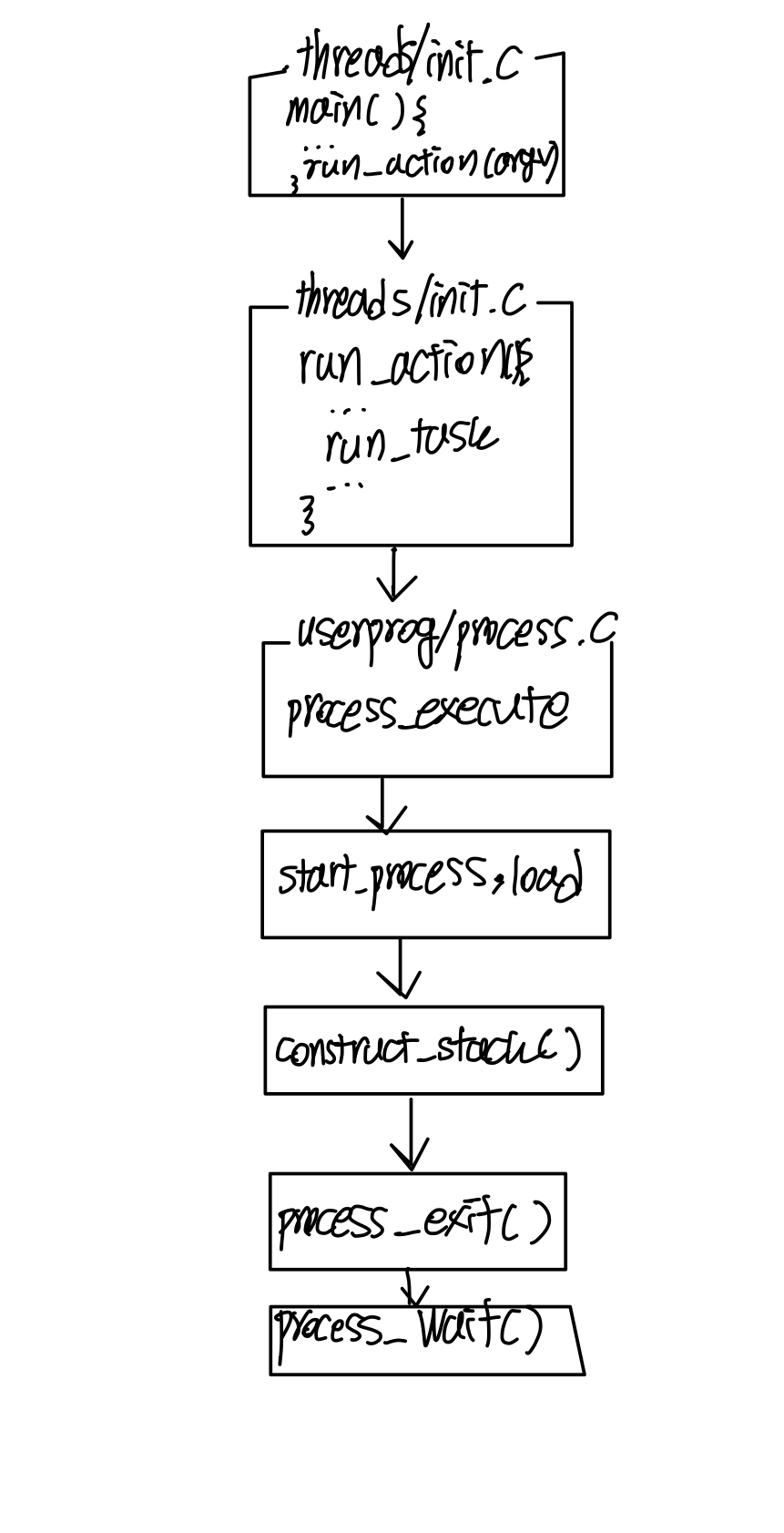
우선 마지막의 thread\_exit()을 지운다. process\_wait를 반복문을 많이 돌림으로서 wait의 기능을 어느 정도 수행할 수 있도록 명세서를 따라 하였는데, 이를 수정해야 한다. thread.h의 thread 구조체에 userprog에서 이용하는 구조체와 변수들을 선언한다. 예로, wait는 child가 실행중이라면 process\_exit(child) 가 될 때 까지 멈춰야 한다. 이를 위한 child\_semaphore, memory\_semaphore을 선언한다. process.c의 process\_wait(), process\_exit() 에서 이 세마포어를 적절히 설정하여 작동하도록 한다.

마지막으로 additional implementation(fibonacci, max-of-four-int)을 구현한다. lib/user의 syscall.c에서 4개의 인자를 받는 syscall4를 새로 만들고, system call number에 해당 수를 추가한다. syscall\_handler()에도 해당 system call number의 경우를 추가하고, /example에도 이를 테스트 하기 위한 파일을 작성한다.

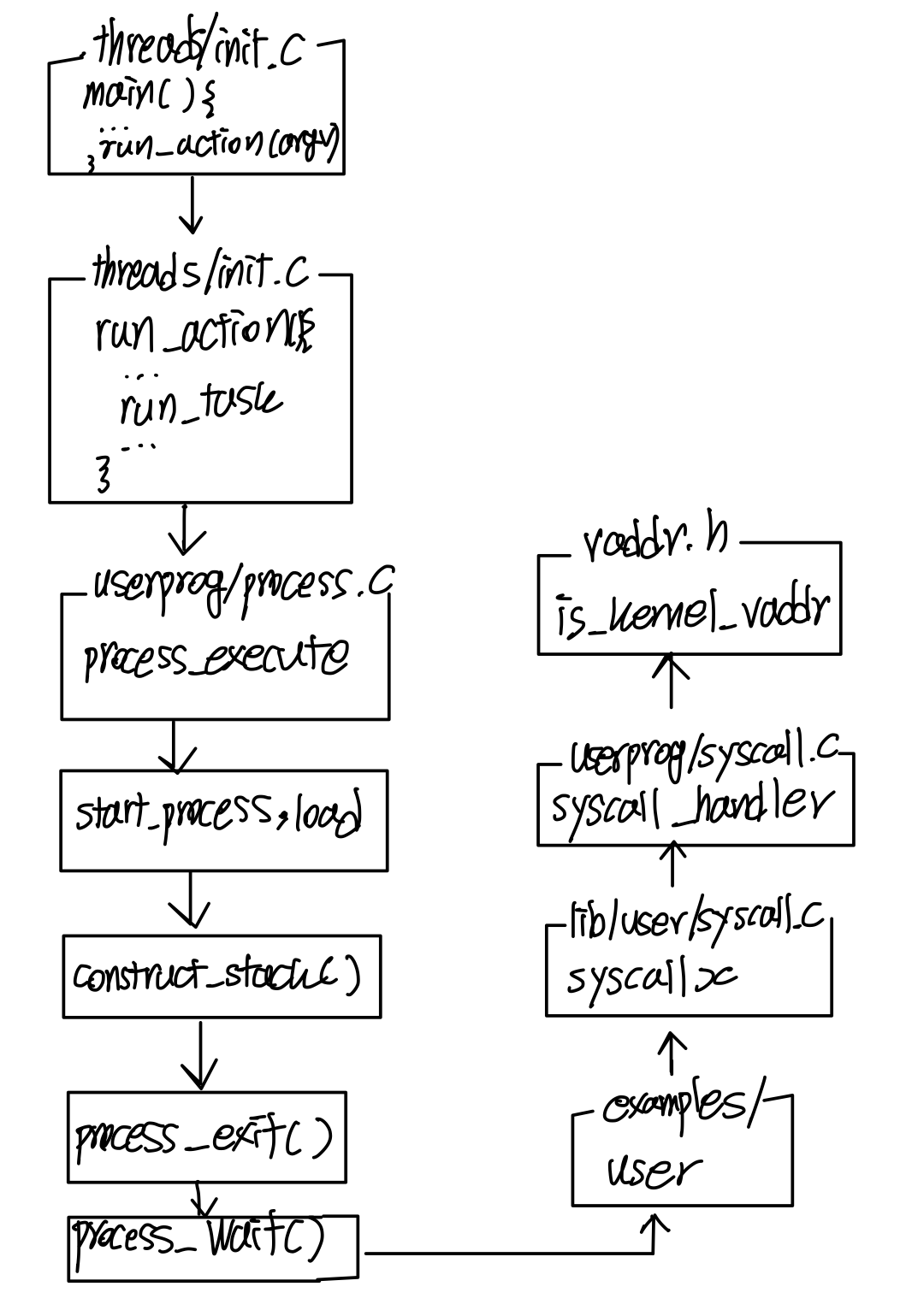
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

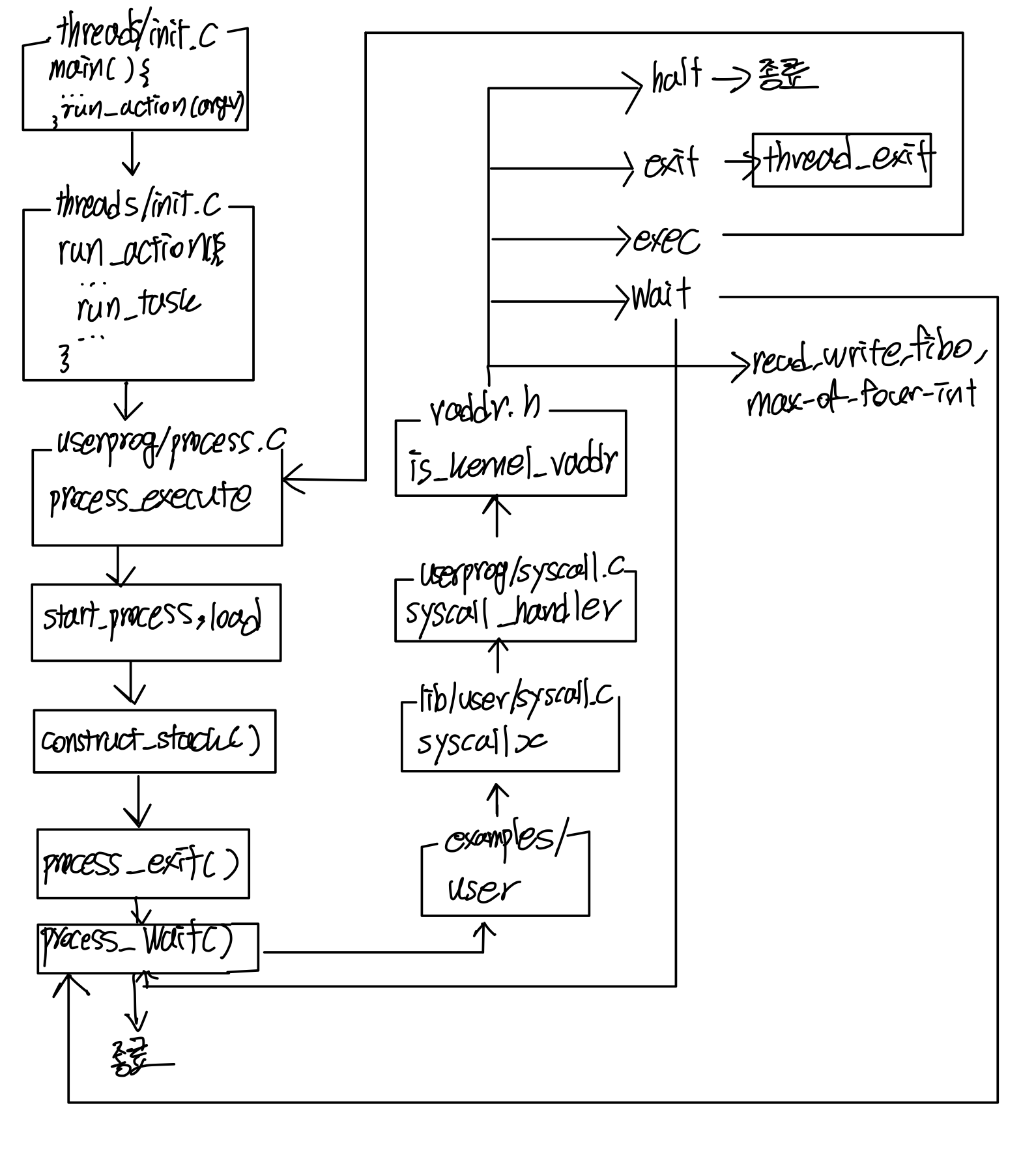
1. Argument Passing



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

userprog/process.c load() 에서, 만약 'echo x' 라면 echo command를 파싱해서 filesys\_open으로 executable file을 연다. executable file을 나타내는 변수는 command로 지정하였다. 그 후, construct\_stack 함수로 넘어간다.

우선 construct\_stack 함수에서 strtok\_r 함수를 통해 argc를 구한다. 그후 똑같은 과정을 통해 argv를 구한다. 이는 stack에 인자가 반대순서대로 들어가기 때문이다. 그 후 4b aligned을 만족시키도록 맞춘다. 이는 total\_byte 변수에 모든 바이트를 저장함으로서 4의 배수인지 아닌지 확인하여 쉽게 가능하다. 그리고 argv[argc-1] ~ argv[0] 까지의 주소를 삽입하고, argv의 주소를 삽입한다. 마지막으로 argc, return address를 삽입하면 construct\_stack 함수는 종료된다. 이로서 argument passing은 완료된다.

1. User Memory Access

일단 unmapped virtual memory의 경우는 skeleton code에 구현이 되어있었다. filesys\_open(command)로 파일을 열었을 때, null인 경우의 예외처리를 userprog/process.c 의 process\_execute() 에 구현한다. 이제 커널 메모리 영역을 침범하는 경우를 고려하면 된다.

is\_kernel\_vaddr() 함수를 이용하여, 이 주소가 커널 메모리인지 검사가 가능하다. userprog/syscall.c에서 만약 kernel 메모리를 침범하는 경우는 이 함수를 이용해 exit(-1)을 호출하도록 하였다. 또한 userprog/exception.c 의 page\_fault() 함수가 존재한다. 이는 page fault handler로서, page fault가 발생하더라도 현재 메모리가 커널 영역인 경우 exit(-1)을 호출하기 위해 위와 비슷하게 그에 맞는 코드를 작성한다.

1. System Calls

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

userprog/syscall.c에서 작성해야 하는 함수는

exit, write, halt, exec, wait, read가 있다. 우선 halt, read, write, exec은 위에서 서술한 대로 예외없이 간단히 작성이 가능하다.

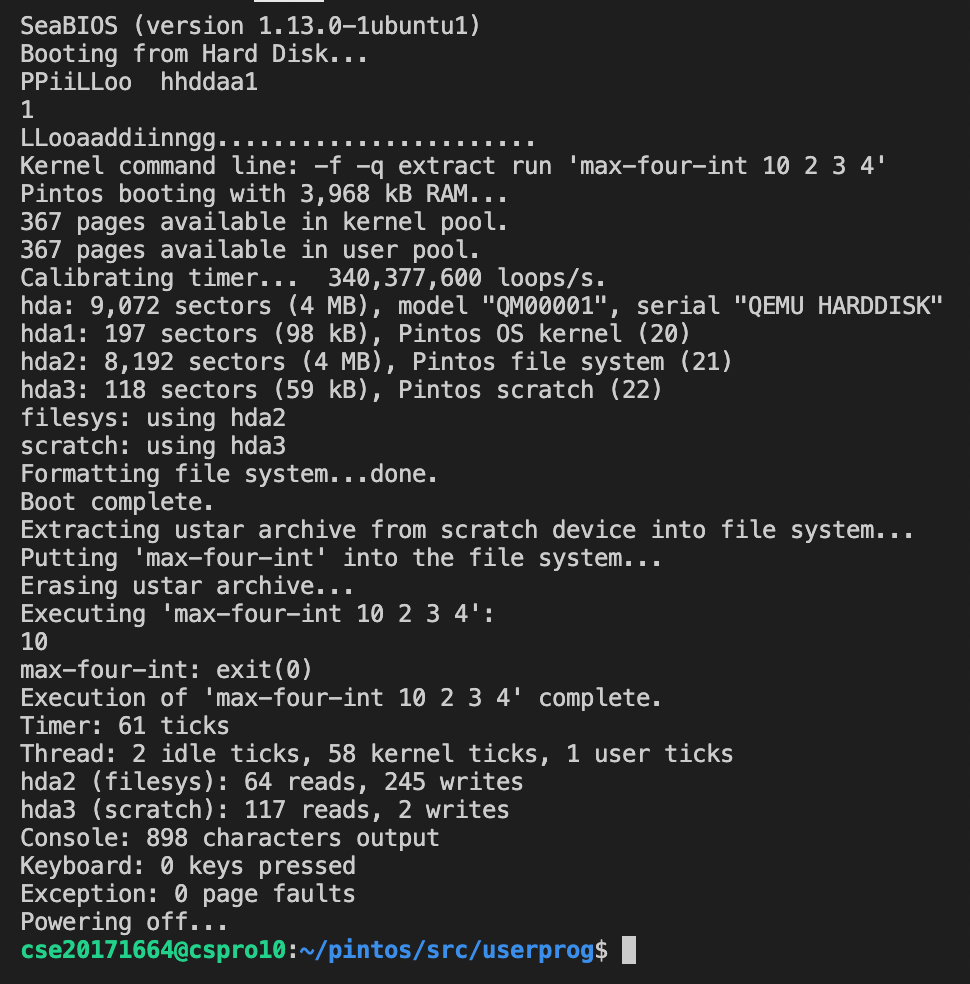
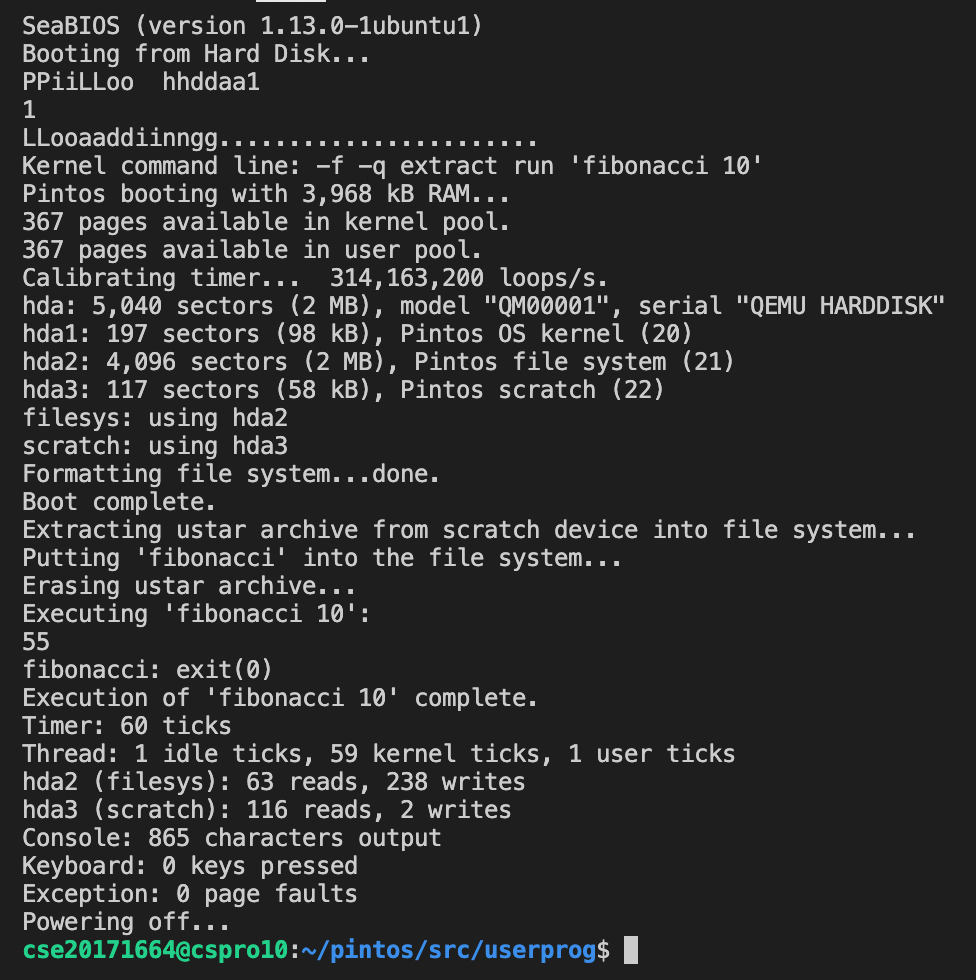
우선 wait()를 작성하여야 한다. 이는 process\_wait() 함수를 완성하여야 한다. 부모는 항상 자식의 pid를 알고 있어야 한다. 자식이 끝날때 까지 wait 해야하기 때문이다. 이를 위해 pintos 내장 라이브러리 list 자료구조를 활용한다. 이 list는 list\_entry()를 통해 list\_elem 바깥쪽 데이터에 접근 가능하다. 반복문을 통해 모든 child thread에 접근하면서 sema\_down()을 호출하여 child\_sema가 실행중이라면 process\_exit()에서 sema\_up()을 할때까지 sleep 시킨다. 그리고 메모리를 지워도 된다는 의미의 memory\_semaphore는 이와 반대의 과정을 거치면 된다.

void exit(int status) 는 process\_exit()를 호출하기 전에, 현재 thread(thread\_current()를 통해 알 수 있다.)의 상태를 갱신 시켜 주어야 한다. process\_exit()는 wait와 세마포어가 상호작용을 잘 할 수 있도록 해야 한다. 현재 스레드가 종료된다면, 현재 스레드를 루트로 하는 서브트리가 종료되었다고 판단한다.

1. Additional System calls

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

우선 example 폴더에 fibonacci.c, max-four-int.c 파일을 작성하였다. 그리고 syscall.c에 피보나치를 구하는 함수, 네개의 정수 중 최대의 값을 반환하는 함수를 구현하여 SYS\_FIBO, SYS\_MAX4INT의 시그널이 왔을때 이 함수를 호출하도록 하였다.

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