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

**(설계 프로젝트 수행 결과)**

과목 명 : [CSE4070] 운영체제

담당 교수 : 박성용 교수님

조 / 조원 : 11조, 20151547, 20151561

개발 기간 : 2019/10/6 - 2019/11/3

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

**제출일 :2019년 11월 3일**

**참여 조원 : 박재상, 신용하**

1. **개발 목표**

* 이 프로젝트를 진행하기 전 pintos의 상태는 system call, system call handler, argument passing, user stack이 **구현되어 있지 않아서** ‘echo x’와 같은 명령어를 실행해도 그 **결과를 확인할 수 없다**.
* 이번 프로젝트에서는 크게 **argument passing, user memory access, system calls**의 기능을 구현하고, 다 구현되면 추가적으로 **fibonacci(), sum\_of\_four\_int()**라는 **system call**을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **Argument Passing**

User program은 argument를 여러 개 가질 수 있는데, 프로젝트 전의 pintos는 이를 나눠서 전달하지 못하고, 통째로 전달한다. (‘echo x’ 는 echo 와 x로 나누어져야 하는데, echo x가 전체 전달되어 사용자가 ‘echo x’의 실행 결과를 확인할 수 없다.) 목표를 위해 userprog/process.c 내부를 수정할 것이다.

* **User Memory Access**

처음 pintos는 kernel 영역에 접근하면 종료되지 않고 kernel을 오염시킬 가능성이 있다. 이를 막기 위해 userprog/syscall.c의 system call에서 넘어오는 pointer들이 user 영역인지 확인하고, userprog/exception.c의 page\_fault() 내부에서 user 영역인지 확인할 것이다.

* **System Calls**

system call은 초기 아무것도 구현이 되어있지 않고, syscall-nr.h 에 적절한 system call number가 제공되어 있다. 이를 이용하여 userprog/syscall.c에서 적절한 system call handling을 하고, 각각의 system call 함수를 만들어 호출한다. 이번 프로젝트에서는 halt, exit, exec, wait, read, write의 여섯가지 기능을 구현한다. wait에서는 특히 synchronization 기법이 필요한데, 이는 userprog/process.c의 process\_wait(), process\_exit()을 수정하고 threads/thread.h, thread/thread.c 의 struct thread와 init\_thread()를 수정한다.

* 1. **개발 내용**

**1. Page, process의 개념**

Page란 실제 메모리는 삽입과 삭제가 빈번하여 항상 연속적인 순서로 존재하기 힘들다. 이를 해결하기 위해 페이지 테이블을 구성하여 메모리를 페이지라는 단위로 나누어 논리 주소를 물리 주소로 변환하여 사용하는 형태이다. 메모리 주소의 앞쪽 bit들은 page 번호로, 뒤쪽 bit들은 페이지 내의 offset으로 사용한다.

Process란 프로그램의 인스턴스로 운영 체제에서 가장 기본적인 실행 단위이다. Page의 virtual memory 개념을 통해 각 프로세스는 메모리를 차지하고, 일정 상태 주기를 가진다. Pintos는 하나의 process가 하나의 thread에 대응되므로, process와 thread가 같은 의미로 사용된다.

**2. User process가 Kernel memory에 접근하는 것을 막기 위한 방법**

**1) User-provided pointer**

이 방법은 현재 pointer가 user 영역에 있는지 is\_user\_vaddr() 함수를 이용해서 확인할 수 있다. userprog/syscall.c에서 system call로 들어오는 모든 pointer에 대해 user 영역인지 확인하고, kernel 영역이라면 exit(-1)을 통해 종료시킨다.

**2) userprog/exception.c의 page\_fault() 수정**

이 방법은 기존 page\_fault() 함수 내부를 고치는 것이다. 이 안에 user라는bool타입 변수가 있는데, 이것이 true라면 user access, false라면 kernel access이다. 이를 통해, user access가 아니라면, exit(-1)을 통해 종료시킨다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **2019/10/6 - 2019/11/3**
  1. **개발 방법**

Argument Passing의 구현을 위해, user stack에 argument들을 parsing 하여 하나씩 저장해주어야 한다. userprog/process.c에서 load로 넘어온 file\_name에는 전체 argument가 저장되어있는데, 이를 적절하게 parsing한 뒤 esp라는 stack pointer를 이용해 parsing 된 argument들을 하나씩 쌓아줄 것이다. load() 와 process\_execute()에서는 전체 argument가 저장된 file\_name을 parsing하여 실제의 파일 이름을 넘겨줄 것이다.

User memory access의 구현을 위해 개발 내용의 2.를 사용한다.

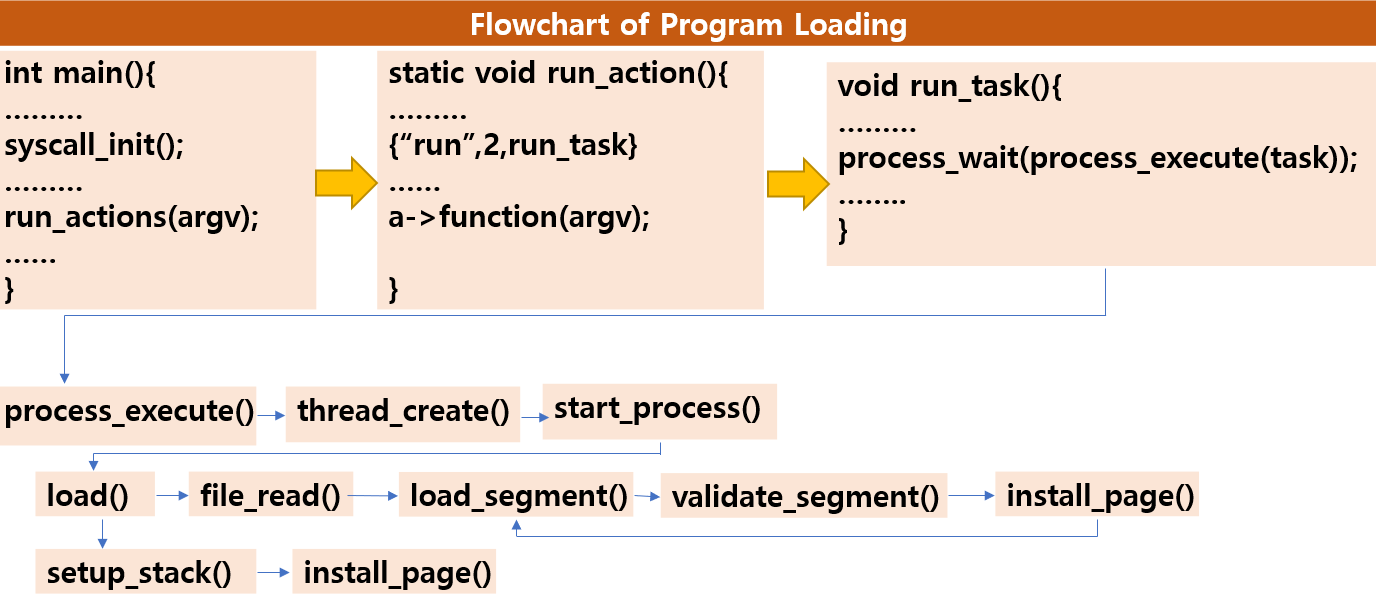
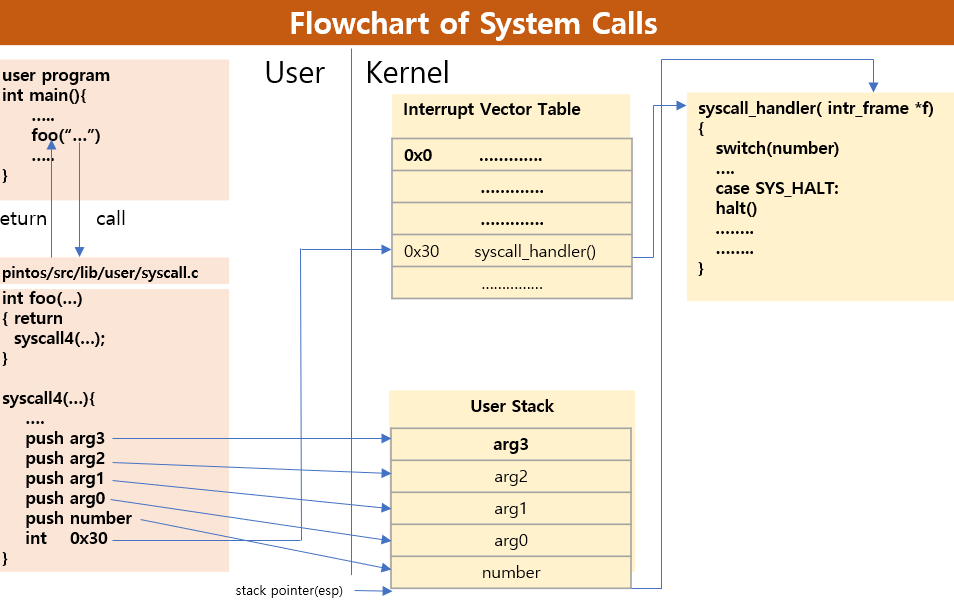
**System Calls의 구현을 위한 과정**

1. 프로그램을 실행하면 내부에서 적절한 system call number를 가지고 있는 struct intr\_frame \* 변수를 system call handler 함수가 인자로 받는데(userprog/syscall.c), 제공된 system call number 를 switch 문으로 각각의 기능을 수행하는 system call 로 나눈다.
2. 각각의 system call마다 요구하는 argument의 수가 다른데, 우리가 직접 이 argument의 개수와 순서를 찾아 넘겨줘야 한다. 이는 esp에 순서대로 저장되어 있다. 여기서 각 argument들의 주소가 user 범위인지, buffer 와 같은 pointer도 user 범위인지 체크해 주어야 한다.
3. System Call을 수행해야 한다. 이번 프로젝트에서는 halt, exit, exec, wait, read, write라는 여섯 가지 기능을 구현한다.(각각의 함수를 만든다.) read, write는 파일에 접근해야 하는데, 이번 프로젝트에서는 STDIN, STDOUT에 대해서만 구현한다.
4. synchronization 기법은 threads/thread.h, thread/thread.c의 struct thread와 init\_thread()를 수정하여 semaphore와 child\_list를 구현한다. 이를 통해 프로세스를 관리한다.우선 userprog/process.c의 process\_exit()에서 종료되는 child\_list\_elem를 free 시킨다. 그리고 process\_wait()과 process\_exit()에서는 두 개의 semaphore를 사용하여 부모가 모든 자식이 죽기 전까지 기다리도록 하는데, 이는 process\_wait()에 구현되어있다.한 자식thread가 끝나면 부모 thread가 해당 자식의 exit\_status를 받아야 한다. 이를 구현하기 위해 두 개의 semaphore를 사용하였다.한 semaphore는 자식이 끝날 때까지 부모가 먼저 끝나지 못하게 하고,다른 semaphore는 process가 종료되어도 부모가 나머지 모든 자식의 exit\_status를 받을 수 있도록 종료되지 않고 기다리게 하는데 사용한다.
   1. **연구원 역할 분담**

**신용하: Argument Passing, System Call Handler, System Call Implementation**

**박재상: Argument Passing, User Memory Access, Additional Implementation**

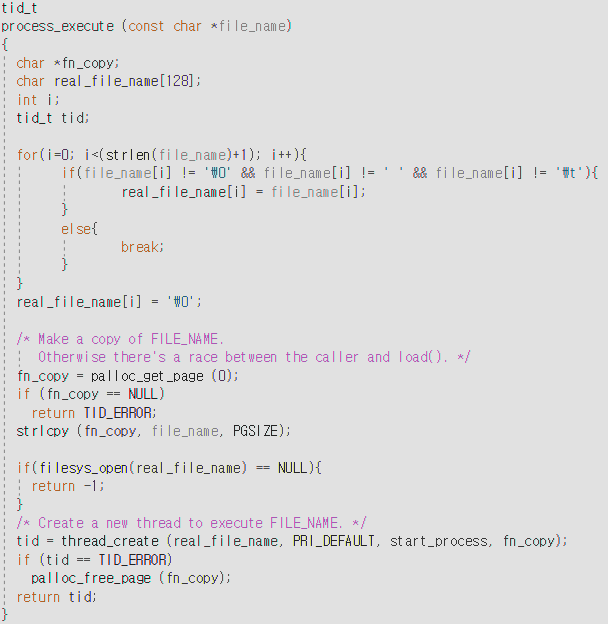
1. **연구 결과**
   1. **합성 내용**

* ****
* ****
  1. **제작 내용**
* **B-1 Argument Passing**

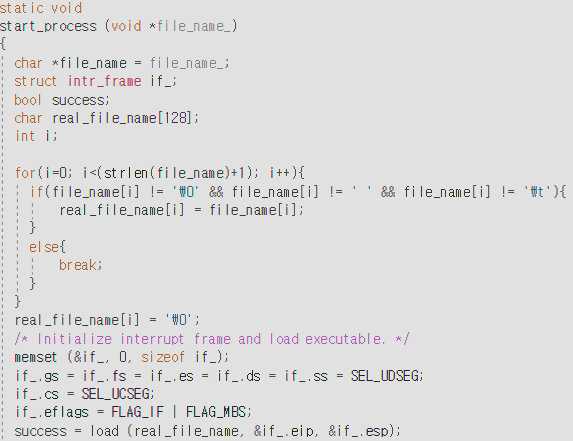
Argument Passing을 위해서 process.c의 수정을 진행했다.

Argument 전체를 file\_name으로 받아오는 process\_execute()에서 먼저 수정을 진행했다.

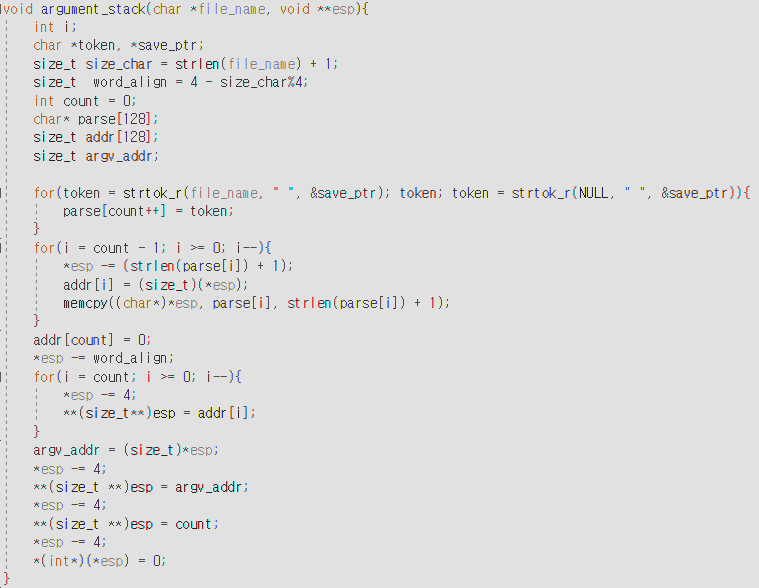
file\_name에서 실제 파일 이름은 첫 단어 이기 때문에, token화를 진행하여 filesys\_open()을 실행시켰다.



그리고 나서 start\_process()에서 load()함수를 부르는데, 이 때도 start\_process()로 넘어오는 file\_name은 전체 argument이기 때문에 token화를 진행한 뒤, 실제 파일이름에 해당하는 값을 load()함수로 넘겨주었다.



그리고 argument\_stack()이라는 새로운 함수를 생성하여 stack에 file\_name을 parsing하여 push했다.

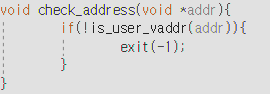


* **B-2 User Memory Access**

kernel영역에 침범하는 것을 막기 위해 userprog/syscall.c의 system call에서 넘어오는 pointer들이 user 영역인지 확인하는 과정을 진행 해야 했다.

이를 위해 추가적인 함수를 구현하였다.

user\_vaddr인지 확인하고, 만약 user\_vaddr이 아니라면 강제로 process를 종료시킨다.

****

* **B-3 System Calls**

**B-3-1.** **프로그램을 실행하면 내부에서 적절한 system call number를 가지고 있는 struct intr\_frame \* 변수를 system call handler 함수가 인자로 받는데(userprog/syscall.c), 제공된 system call number 를 switch 문으로 각각의 기능을 수행하는 system call 로 나눈다.**

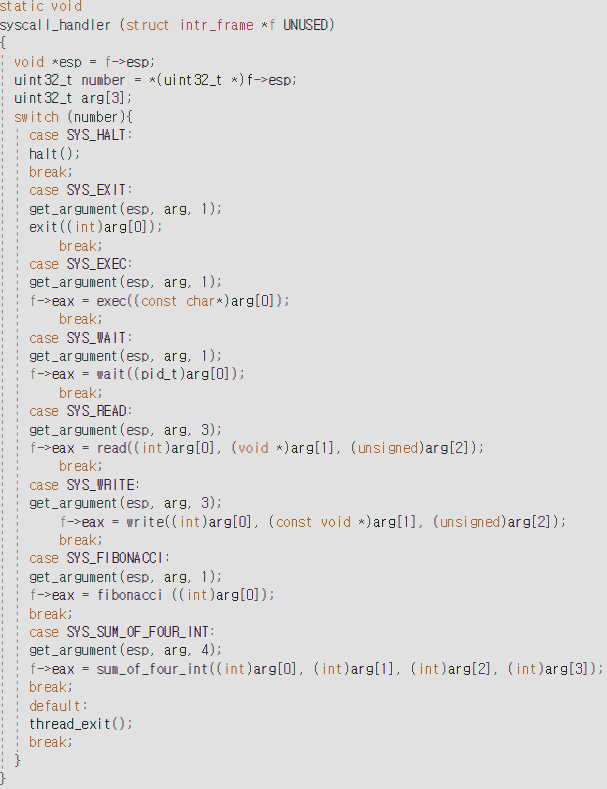
* user가 syscall을 호출했을 때, 어떤 종류의 syscall을 호출했는지(parameter개수, syscall이름)를 확인 해주기 위해서 syscall\_handler()를 적절히 수정해야했다.

우선 parameter로 들어온 intr\_frame \*f를 이용하였다. 위의 flowchart of system calls에서 볼 수 있듯, f->esp에서 systemcall의 종류인 number를 알아낼 수 있다.

switch문을 이 number를 통해서 진행하면서 적절한 syscall을 호출한다.

**B-3-2**. **각각의 system call마다 요구하는 argument의 수가 다른데, 우리가 직접 이 argument의 개수와 순서를 찾아 넘겨줘야 한다. 이는 esp에 순서대로 저장되어 있다. 여기서 각 argument들의 주소가 user 범위인지, buffer 와 같은 pointer도 user 범위인지 체크해 주어야 한다.**

매 case마다, intr\_frame \*f를 통해 필요한 parameter들을 뽑아서 syscall 함수로 넘겨주는 것을 볼 수 있다.

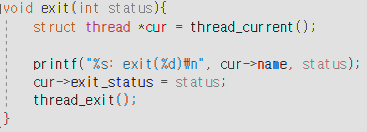
****

**B-3-3** **System Call을 수행해야 한다. 이번 프로젝트에서는 halt, exit, exec, wait, read, write라는 여섯 가지 기능을 구현한다.(각각의 함수를 만든다.) read, write는 파일에 접근해야 하는데, 이번 프로젝트에서는 STDIN, STDOUT에 대해서만 구현한다.**

**1.halt****()**

****

**2. exit()**

****

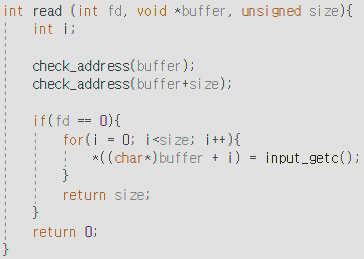
**3. exec()**

****

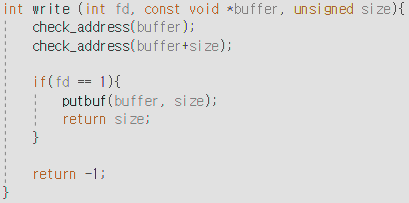
**4. wait()**

****

**5. read()**

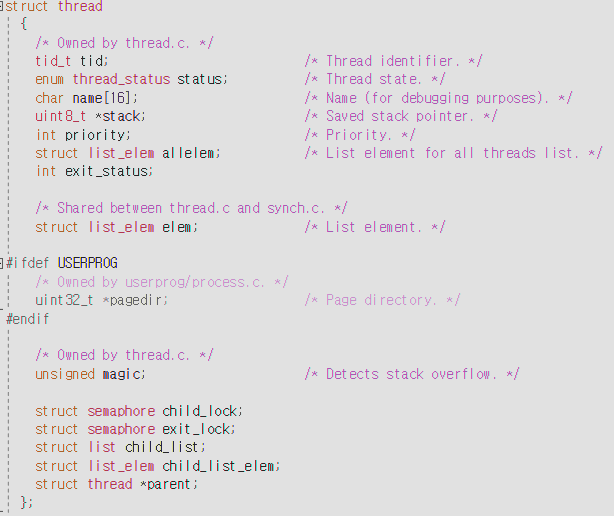
****

**6. write()**

****

**B-3-4 synchronization 기법은 threads/thread.h, thread/thread.c의 struct thread와 init\_thread()를 수정하여 semaphore와 child\_list를 구현한다. 이를 통해 프로세스를 관리한다. 우선 userprog/process.c의 process\_exit()에서 종료되는 child\_list\_elem를 free 시킨다. 그리고 process\_wait()과 process\_exit()에서는 두 개의 semaphore를 사용하여 부모가 모든 자식이 죽기 전까지 기다리도록 하는데, 이는 process\_wait()에 구현되어있다. 한 자식thread가 끝나면 부모 thread가 해당 자식의 exit\_status를 받아야 한다. 이를 구현하기 위해 두 개의 semaphore를 사용하였다. 한 semaphore는 자식이 끝날 때까지 부모가 먼저 끝나지 못하게 하고,다른 semaphore는 process가 종료되어도 부모가 나머지 모든 자식의 exit\_status를 받을 수 있도록 종료되지 않고 기다리게 하는데 사용한다.**

init\_thread()를 수정하기 위해, struct thread에 추가적인 member들을 넣었다.

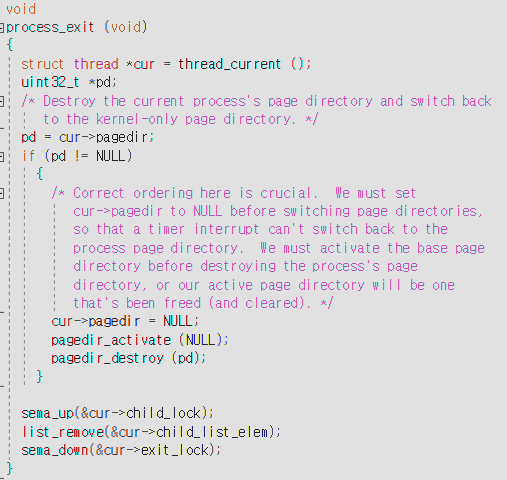


child가 죽을 때 까지 기다리기 위한 child\_lock, process가 끝나도 child가 죽을 때 까지 기다리기 위한 exit\_block, child\_process를 관리하기위한 child\_list와 child\_list\_elem 그리고 parent 를 추가하였다.

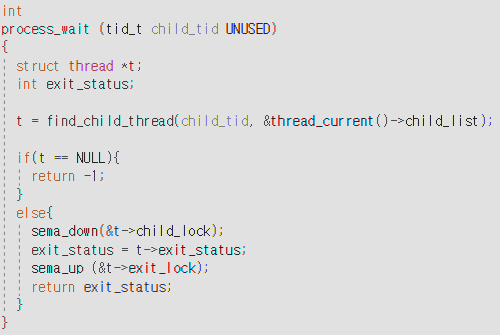
새로운 struct thread를 이용하여 수정된 init\_thread()

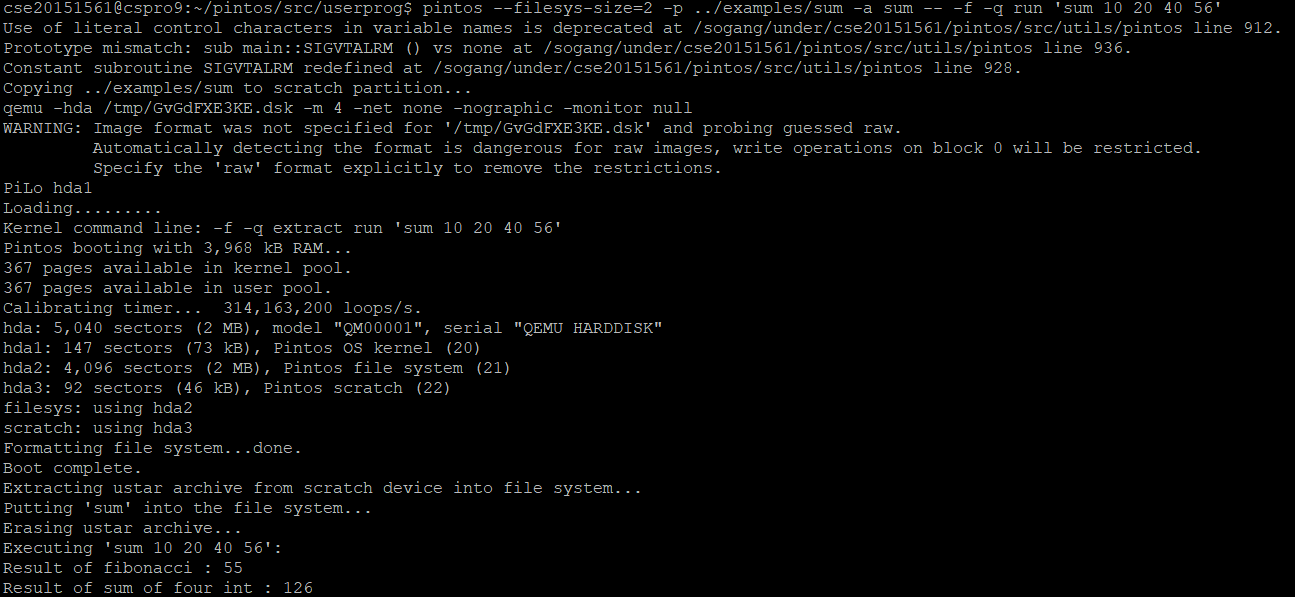


종료되는 child\_list\_elem을 free시키기 위해 process.c의 process\_exit를 수정하였다.



자식 thread가 죽기 전까지 부모 thread가 기다릴 수 있도록 process\_wait()또한 수정했다.



* 1. **시험 및 평가 내용**
* ****

1. **기타**
   1. **연구 조원 기여도**

* **신용하(50%) 박재상(50%)**
  1. **소감**
* **신용하: Pintos에 기본적으로 제공되어 있는 함수들을 하나씩 살펴보고, 이를 활용해 기존의 것을 고쳐 쓰는 것을 처음 해봐서 어려웠다. 특히, Synchronization 기법은 OS수업 때 처음 듣고 사용해 보는 개념이라 더욱 어려웠던 것 같다. 문제를 해결하면서 기본적인 Multi-process의 개념을 알 수 있었다.**
* **박재상: 처음으로 Group Project를 진행해보았다.**

**운영체제 수업을 들으면서, 개념적인 이해는 교수님의 설명을 따라가다보면 금방 할 수 있었다.. 그러나 막상 구현을 하려고 하니 상당 부분 어려움이 있었다. 특히 전체적인 흐름을 파악하는 것이 가장 어려웠다. pintos에서 상당 부분이 구현 되어있지 않아서 중간중간 흐름을 놓치는 경우가 자주 있었다. 그 때마다 google 검색과 pintos manual을 참고하면서 project를 이어 나갔다. 또한 어려운 data structure나 function을 같은 조원과 토의하면서 진행하였는데, 그 과정에서 토의자체만으로도 많은 문제를 해결해줄 수 있다는 것을 알게 되었다.**