---- TEAM ----

**>> Team name.**

**>> Fill in the names, email addresses and contributions of your team members.**

Jae Hyun Lim <limszang2@gmail.com> (50%)

SeongHeon Kim<roupkk@gmail.com> (50%)

contribution1 + contribution2 = 100

**>> Specify how many tokens your team will use.**

4

---- PRELIMINARIES ----

**>> If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.**

**>> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.**

* [www.scs.](http://www.scs.stanford.edu/11wi-cs140/notes/s2-print.pdf)**stanford**.edu/11wi-cs140/notes/s2-print.pdf :‎ 다른 대학 강의 자료 참조
* <http://sp0ngee.tistory.com/51> : strtok 검색

<https://www.ida.liu.se/~TDDI04/2009/labs/lab1.shtml>: pintOS에 대해 잘 설명

<http://www.doc88.com/p-600927355262.html> : 숭실대 pinOS 강의자료

ARGUMENT PASSING

===========

---- DATA STRUCTURES ----

**>> A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.**

* threads/thread.h
  + struct list parent\_child\_list : parent-child 관계를 저장하는 struct들의 리스트. 즉 한 스레드에서 만든 모든 자식 프로세스 파일들을 저장한 것이다.
  + inf fd; inside struct thread : 각 스레드마다 자신이 가진 자식 프로세스들의 key 값 같은 것이다. 즉 자신이 가진 자식 프로세스를 구분하는 데에 이용된다.
* userprog/process.c
  + struct parent\_child : 생성된 parent-child 프로세스들의 정보를 담은 struct. 부모 스레드, 부모 tid, 부모의 wait 여부, 자식 tid, 자식 status 등이 저장되어있다.

---- ALGORITHMS ----

**>> A2: Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order? How do you avoid overflowing the stack page?**

우선 process\_execute()에서 주어지는 file\_name은 arg까지 포함한 하나의 string이므로, parsing을 할 필요가 있다. 이 값은 start\_process()로 전달이 되어 추후의 parsing은 그곳에서 시행하고, 이곳에서는 자식 스레드를 시행하므로 자식과 현재 스레드(부모)의 정보를 struct parent\_child에 저장한다.

start\_process()에서는 먼저 file\_name에 있는 argument의 개수(argc)를 구한다. 이는 strtok\_r를 활용하여 명령어와 argument를 분리한다(분리된 명령어는 file\_name에 저장하여 load()로 넘긴다). argc의 값을 구한 후, 모든 것들을 user stack에 저장하기 위해 if\_.esp와 더불어 하나의 기준점이 될 esp\_start를 설정한다. if\_.esp는 모든 argument와 명령어의 길이 만큼 크기를 확보(-를 한다)하도록하고 esp\_start는 if\_.esp로부터 argc 만큼 크기를 확보한다. 이후 start\_process()가 처음 받은 file\_name에서 strtok\_r로 token을 분리하면서 if\_.esp에는 argument의 내용물을, esp\_start에는 argument address들을 저장한다. 그 다음 argv address, argc, return address를 esp\_start로부터 4씩 크기를 확보하여 저장한다. 이를 통해 pdf에 나온대로의 순서로 stack에 저장이 가능하다.

오버플로잉을 방지하기 위해 미리 전 argument의 개수, 길이를 구하고 그에 맞게 stack 포인터를 조절하여 저장할 위치를 지정하였다.

---- RATIONALE ----

**>> A3: Why does Pintos implement strtok\_r() but not strtok()?**

strtok\_r는 포인터를 사용하는 반면, strtok는 정적 버퍼를 사용한다. Strtok 내부에서는 호출마다 다음 token을 반납하기 위해 저장을 해놓는데, 이때문에 여러개의 thread가 strtok을 호출하면 string이 변질될 우려가 있으므로 thread-safe하지 못하다. strtok\_r은 내부적으로 저장하는 대신 save\_ptr이라는 포인터를 통해 처리하기 때문에 thread-safe하다고 할 수 있다.

**>> A4: In Pintos, the kernel separates commands into a executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.**

* 커널의 부담을 줄여주는 이점이 있다.
* 커널에게 악영향을 끼칠 수 있는 입력을 차단해주는 이점이 있다.  
  만일 한꺼번에 받아서 처리한다면, buffer overflow와 같은 방법으로   
  직접적으로 system call을 호출할 수 있다.

SYSTEM CALLS

===================

---- DATA STRUCTURES ----

**>> B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.**

**>> B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?**

File descriptor는 single process마다 unique하다.

---- ALGORITHMS ----

**>> B3: Describe your code for reading and writing user data from the kernel.**

syscall이 호출되면 esp의 값을 조사해서 user vaddr 내부인지 확인한다. 이는 PHYS\_BASE와 0x08084000사이에 있어야한다. 값이 적절하다면 syscall number가 맞는지 확인한다. Syscall number에 대해 각각 타입을 결정하는데, argument수에 따라서 알맞은 type이 정해진다. Read와 write의 경우 syscall3이기 때문에 esp를 따라 file descriptor와 buffer, size를 받아올 수 있다. 이때 주소값이 유효한지 파악하기 위해 다시 user vaddr을 불러서 확인하고 내부의 값은 barrier\_buf를 실행하여 적절한 값인지 판별한다. 그런 다음 read와 write를 호출한다.

**>> B4: Suppose a system call causes a full page (4,096 bytes) of data to be copied from user space into the kernel. What is the least and the greatest possible number of inspections of the page table (e.g. calls to pagedir\_get\_page()) that might result? What about for a system call that only copies 2 bytes of data? Is there room for improvement in these numbers, and how much?**

**>> B5: Briefly describe your implementation of the "wait" system call and how it interacts with process termination.**

process\_wait()를 부르는 방식으로 코딩하였다. Struct parent\_child를 이용하여 list를 만들어서 관리하였다. child process를 받아서 null이면 끝났다고 생각하고 return을 해줬고, 아니라면 child를 기다려야 하므로 semaphore를 이용하여 child가 parent에 의해 기다려지고 있다고 알렸다. parent는 block되고 child 가 끝나면 다시 semaphore를 이용하여 parent를 깨운다. parent는 깨어나면서 list를 정리하고 parent는 child의 exit status를 받아서 return한다.

**>> B6: Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a "write" system call requires reading the system call number from the user stack, then each of the call's three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues. Give an example.**

userprogram의 pointer는 반드시 kernel\_address함수를 거치게 디자인 하였다. 이곳에서 NULL pointer나 kernel을 가르키는 포인터와, 매핑되지 않은 곳을 가르키는 포인터는 malicious user라고 판단하여 exit하였다. exit하면서 열려있는 file descriptor를 정리하고 list가 있다면 list elem을 list에서 제거하였다. 또한 child를 고아로 만들었다. 제대로 구현되지 않았지만 좀 더 완벽하게 한다면 multi-oom도 통과할 수 있을 것으로 보인다.

---- SYNCHRONIZATION ----

**>> B7: The "exec" system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls "exec"?**

semaphore를 사용하여 thread\_create()한 후에 sema down을 해주면 parent가 child를 기다리게 되고 loading이 끝나면 success의 결과를 parent가 return을 받고 깨어나게 된다.

**>> B8: Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any special cases?**

semaphore를 이용하여 관리했다. P가 C전에 기다릴때는 P는 C가 끝날때까지 semaphore를 기다리며 C가 끝나면서 P가 깨어난다. P가 C다음에 기다릴때는 P는 sema down을 누르면서 그냥 메모리를 반환하고 return한다. Single threaded이기때문에 semaphore를 사용하면 손쉽게 메모리를 반환을 확신할 수 있다. P가 C전에 기다리지 않을때는 C를 고아로 만들었다. P가 C이후에 기다리지 않을때는 위와 같이 메모리를 반환하고 return하게된다.

---- RATIONALE ----

**>> B9: Why did you choose to implement access to user memory from the kernel in the way that you did?**

Struct intr\_frame \*f; 의 멤버 f->esp를 가져와서 stack을 4byte씩 pop한다. 그 다음 알맞은 syscall을 찾고 barrier를 통해 syscall에 들어갈 argument를 validation한다.

숭실대학교 강의자료에 맞춰서 디자인 하였다.

**>> B10: What advantages or disadvantages can you see to your design for file descriptors?**

File descriptor를 process마다 고유라고 생각했기 때문에 열려있는 file에 접근하기가 편하다. 때문에 process가 exit(-1)을 해도 file descriptor를 kernel이 대신 닫아줄 수 있다. File descriptor가 OS 전체에 unique하다면 duplication이 있지 않을것이고 이것은 file descriptor만 가지고도 모든 thread를 뒤지지 않아도 된다.

**>> B11: The default tid\_t to pid\_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?**

no

SURVEY QUESTIONS

================

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

**>> In your opinion, was this assignment or any one of the two problems in it, too easy or too hard? Did it take too long or too little time?**

**>> Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?**

**>> Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?**

**>> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?**

**>> Any other comments?**