---- TEAM ----

**>> Team name.**

42

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

Ina Ryu<ryu\_ina@kaist.ac.kr> (50)

Yeonsoo Kim <kysoo55@kaist.ac.kr> (50)

contribution1 + contribution2 = 100

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

---- 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.**

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.**

**//process.c//**

<process\_execute>

char \*t\_name; -> used in parsing thread name

char \*save\_ptr; -> used in parsing thread name

<start\_process>

char \*\*argv; -> dynamic memory to store each argument’s address

int argc = 0; -> number of arguments

---- 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?**

At first, we parse only file name to insert into thread\_create() at process\_execute().

And then passing full arguments to start\_process() as argument \*f\_name.

start\_process함수 에서는 우선 strtok\_r()을 이용해 executable file name을 잘라 load에 넣어 업로드 한다.

success 이후, 파일 이름을 떼어 낸 나머지를 가지고 argument parsing을 한다.

if(interrupt frame)에 esp가 저장되어 있는데, 우선 stack의 제일 위에부터 parsing한 argument를 넣어준다. 이 때 argv에 각 argument를 저장한 address를 넣어준다. 이 때 stack의 맨 위에서부터 저장하므로 “grep foo bar char” 의 경우

stack top - argv[1] “foo”

argv[2] “bar”

argv[3] “char”

…. 순으로 저장되게 된다. 그러나 각각의 argument 를 가리키는 주소값은 처음 나오는 arguement의 주소를 밑에 저장되게 해야하기 때문에 마지막 argv의 주소값부터 stack 에 넣어줘야 한다. 그리고 주소값을 저장하기 전에 argv[length of argv] 에 0을 넣어줘서 거기까지가 argment들의 주소라는 것을 알려준다.

stack의 top 부터 아래로 넣어주기 때문에 PHYS\_BASE 를 넘어 kernel로 overflow하지 않는다.

---- RATIONALE ----

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

strtok() 함수는 buffer에 있는 데이터를 직접 변형한다. 그러나 strtok\_r() 은 분리하려는 string을 직접 변형하지 않고 그만큼을 따로 떼어준다. 그래서 메모리 공간이 손상될 여지가 없어서 안전성을 위해 strtok\_r을 사용한다.

**>> 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.**

executable name과 argument 를 자르는데 여분의 커널 메모리가 필요하지 않다. - > 많은 exec를 할경우 제대로 free해주지 않으면 leak이 일어날 수 있음

또한 parse 하기 위해 넘겨준 memory 가 너무 많은 경우, 혹은 넣어준 string에 문제가 있을 경우, kernel에서 처리하다보면 kernel 이 죽을 수 있기 때문에 shell 에서 잘라서 위험성을 파악하는 것이 도움이 될 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.**

<process.h>

/\* struct process 의 file\_list에 넣어주는 struct (process가 열어준 file list. fd를 통해 file을 찾기위해 만듦) \*/

struct fd\_file

{

int fd;

struct file\* file;

struct list\_elem elem;

};

/\* struct process의 childeren\_pids에 저장해 주기 위한 struct

load success한 child만 저장 (wait-exit) 에 필요 \*/

struct childpid\_elem

{

tid\_t childpid;

struct list\_elem elem;

};

/\* load success 한 process를 process\_list에 넣어주기 위한 struct \*/

struct process

{

/\* 이 process의 pid \*/

tid\_t pid;

/\* 이 process의 parent 의 pid \*/

tid\_t parent\_pid;

/\* 이 process의 children의 pid \*/

struct list children\_pids;

/\* 이 process가 exec을 call 했으면 child가 load 완료할때까지 기다려 주기 위해 \*/

struct semaphore sema\_pexec;

/\* 이 process가 wait을 call 했으면 child가 exit 할때까지 기다리기 위해 \*/

struct semaphore sema\_pwait;

/\* process\_execute()에서 start\_process를 호출 한 후 child가 load success 했는지를 받아오기 위해\*/

bool load\_success;

/\* 이 process가 exit 했는지 \*/

bool is\_dead;

/\* exit 했다면 exit\_status 가 뭐였는지 \*/

int exit\_status;

/\* process가 open 한 file 개수 (fd를 부여하기 위해) \*/

int fd\_cnt;

/\* 이 process의 exec file를 process가 exit 할 때 free 해주기 위해서 \*/

struct file \* exec\_file;

/\* 이 process가 open 한 file\_list \*/

struct list file\_list;

/\* process list 에 넣어주기 위해 \*/

struct list\_elem elem;

};

<process.c>

/\* 정상적으로 exec된 process의 list -> 전체적인 process의 context를 관리해주기 위해서 \*/

struct list process\_list;

<syscall.c>

/\* filesystem은 critical section이라 한번에 한 process만 접근하게 해야한다. 그러기 위해 사용 \*/

struct lock filesys\_lock;

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

struct fd\_file

{

int fd;

struct file\* file;

struct list\_elem elem;

};

위와 같이 fd와 file을 엮어준 struct를 만들어, 그것을 각각의 process의 file\_list에 저장해주었다. file descriptor는 single process에서만 unique 하다.

---- ALGORITHMS ----

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

user data를 stack에 저장하고 system call을 하면, 그 주소를 intr\_frame을 통해서 kernel에 전달한다. kernel은 intr\_frame의 esp를 통해 올바른 stack 메모리에서 user data를 읽어올 수 있다. kernel이 user에게 전달할때도 마찬가지로, kernel이 stack에 저장을 해주면, user가 그 stack에서 값을 읽어온다.

**>> 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?**

CASE 1: Full page

page의 시작 address가 user space의 시작과 같다면 넘치거나 오버하지 않기 때문에 least number는 1이다. 그러나 page가 1바이트씩 따로따로 끊겨져서 allocate되어 있다면 각 byte씩 검사해야 하므로 greatest number는 4096 이다.

CASE 2 : 2 bytes of data

이 역시 least number는 1이고, greatest number는 1byte씩 검사해야 하는 경우에 최대인데, 바로 2이다.

Yes. 지금 32bit라서 메모리의 기본적인 단위가 1B(32bit)인데, 64bit 단위에서는 2B기준이므로 2바이트씩 떼서 검사할 수 있다

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

기본적으로 process\_wait()이 호출되면, parent는 sema\_down을 해서 child process가 exit을 완료하고 sema\_up()을 해줄때까지 기다린다.

(+child process가 이미 exit한 경우에는 sema\_down을 하지않으며, child 역시 parent가 sema down을 안했으면, sema\_up을 실행하지 않는다.)

child는 exit이 되면서 자신의 exit\_status를 parent에게 전달한다.

**>> 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.**

우리는 이 address가 user의 mapped memory 인지 확인하는 is\_valid\_usradrr() 라는 함수를 만들어, pointer value가 사용될 때마다 is\_valid\_usraddr()를 호출해서 확인해주었다.

또한 error가 생겨 exit 하는 모든 부분 앞에 사용한 resource들을 free해 주었다.

또한, syscall 이 아닌 user 에서 바로 page fault가 일어나는 경우에도, process\_exit()을 call해서 사용한 resource들을 free해 주게 하였다.

---- 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"?**

thread\_create()으로 thread을 준비 해 놓은 후

sema\_down(&curr\_p->sema\_pexec) 와 같이 현재 process의 sema\_pexec을 다운 시킨다. 그러면 스케줄링이 일어나고, ready queue에 있는 스레드가 실행되며 start\_process를 실행한다. 이 때 load의 success 정보를 저장하고, sema\_up(&parent\_p -> sema\_pexec) 과 같이 parent의 sema\_pexec를 up 해서 parent가 로드가 완료될 때 까지 기다릴 수 있게 한다.

앞에서 말했듯, parent에 의해 생성된 child는 load 이후 success 정보를 저장한다. parent는 sema\_up된 이후, child의 load\_success가 true/false인지 보고 그를 통해 return값을 결정 할 수 있다.

**>> 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?**

P calls wait(C) before C exits

* process\_wait에 들어온 child가 아직 exit을 하지 않았다면, curr process의 sema\_pwait를 down 시킨다. 나중에 child가 exit할 때 다음과 같은 프로세스를 진행한다.
* syscall(exit) 이 불리면 자신의 exit status를 proces\_list에 저장되어있는 자신의 process struct에 저장한다. parent가 sema\_pwait down 해 있으니 parrent의 sema\_pwait을 sema\_up 을 해준다.
* 따라서 P는 C가 exit할 때 까지 기다릴 수 있다.

P calls wait(C) after C exits

* 만약 그 child가 이미 exit을 했다면 (process\_exit을 할때 모든 process는 항상 자신의 exit\_status 를 procee\_list 안의 자신의 process struct에 저장한다. 또한 exit 내에서 process의 is\_dead 를 true로 바꿔준다.) child process struct에 저장되어있는 exit\_status를 return 한다.

그리고 위의 경우, C는 죽은 process이므로 그 process정보를 free해준다.

P terminates without waiting, before C exits

* P가 exit 해버리면 child도 자신의 exit status를 P에 넘겨줄 필요가 없다. C 가 exit할 때 P 가 죽어있으면 exit 안에서 자신의 process 메모리도 스스로 free한다.

P terminates without waiting, after C exits

* 이 경우, P가 exit할 때 child 중에서 죽어있는 것을 다 free 하고 간다.

P wait for invalid child pid

* 넣어준 pid가 P의 child pid 중에 없으면 return -1 을 한다.

---- RATIONALE ----

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

우리는 MMU를 사용하지 않는 방법을 선택했는데, 그 이유는 이 방법이 더 직관적이며, pintos 같은 작은 운영체제에서는 굳이 MMU를 이용하여 효율을 높였을 때의 이득이 크지 않다고 생각했다.

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

fd의 경우 0,1을 제외한 2부터 1씩 증가하며 부여해 주었다.(int fd\_cnt 로 process에 저장) file을 오픈 할때마다 새로운 fd를 붙여 (file, fd) set이 프로세스마다 unique할 수있게 했다. 또한 그 정보를 process의 file\_list에 넣어 fd를 통해 file을 찾을 수 있는 함수를 만들어 일종의 fd- file 의 정보를 담은 table을 만들었다.

그러나 이 경우 close해 지워진 fd를 다시는 쓸 수 없기 때문에 그 점은 disadvantage로 생각된다.

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

pid\_t를 tid\_t로 사용하였다.

(pintos에서는 process는 무조건 단일 thread이기 때문에 그냥 써줘도 상관없다)

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?**