+----------------------------------------------+

| CS 330 |

| PROJECT 2: USER PROGRAMS |

| DESIGN DOCUMENT |

+----------------------------------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

김건호(Kunho Kim) <kaist984@kaist.ac.kr>

김철환(Cheolhwan Kim) <maxwell98@kaist.ac.kr>

---- 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.

ARGUMENT PASSING을 위해 구조체나 전역 변수를 변경해준 것은 없습니다.

---- 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?

process\_excute()에서 인자로 받은 file\_name을 복사하기 위해, char \*fn\_copy를 선언해줍니다. 그리고 argv[0]를 복사할 char command[256]을 선언해줍니다. 이후 fn\_copy를 할당해주고, file\_name을 fn\_copy에 복사해줍니다. 이후

void get\_command (char \*command, char \*file\_name);

라는 함수에서 command에 file\_name을 복사해주고, 이를 순회하면서 arg를 구분해주는 ‘ ‘(space character)를 찾아줍니다. 찾은 지점의 공백을 NULL로 바꿔주어 command에는 가장 첫번째 arg만 남도록 합니다. 이후 thread\_create(command, PRI\_DEFAULT, start\_process, fn\_copy)에 1번째 인자인 name으로 command를 넘겨주고, 4번째 인자로 fn\_copy를 넘겨줍니다. 이는 해당 thread가 cpu를 점유하게 되고 start\_process가 시작될 때 start\_process의 인자로 들어가게 됩니다.

start\_process가 시작되면 char \*command를 새로 선언해주고

int seperate\_fn (char\* fn\_copy, char\*\* fn\_token);

라는 함수에서 strtok\_r()을 통해 fn\_copy에 있는 ‘ ‘(space character)를 모두 NULL로 바꿔줍니다. 그리고 tokenize된 args의 개수를 return해줍니다. 그리고 tokenize된 첫번째 arg가 저장된 포인터를 char \*\*argv에 받아 load()이후 메모리 적재에 성공했다면,

void push\_args\_to\_stack (char\*\* argv, int argc, void \*\*esp);

함수를 실행시켜줍니다. 이 함수에서는 가장 마지막 arg가 저장된 argv에서부터 -1씩 해주어 해당 arg를 esp에 복사해줍니다.(가장 마지막 arg를 처음 넣어줍니다.) 이후 word align을 위해 넣은 arg의 길이가 4로 나누어 떨어지지 않는다면 그 나머지 만큼 esp의 주소에서 빼줍니다. 그 다음은 esp에 NULL(0)을 넣어주고, argv의 가장 마지막 arg의 주소값부터 처음 arg의 주소값까지 넣어줍니다. 이후, argv 자체의 주소값을 esp에 넣어주고, argc(인자의 개수)도 esp에 넣어주고, 마지막으로 return address인 0을 esp에 넣어줍니다. 이런 과정을 통해 argument가 stack에 저장되게 됩니다. push\_args\_to\_stack()이 실행될 때, argv는 strtok\_r()이후 가장 마지막 arg의 포인터를 가리키고 있고, 이 arg부터 역순으로 esp에 넣어주기 때문에 argv[]의 원소들은 올바른 순서로 저장되게 됩니다.

---- RATIONALE ----

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

strtok()는 정적 버퍼를 사용하는 반면에 strtok\_r()는 char \* 포인터로 할당된 유저에 대한 포인터를 사용합니다. 그러므로 strtok()는 thread safe하지 않고, strtok\_r()는 thread safe합니다. thread safe 함수는 동시 액세스로부터 shared resources를 보호해줍니다. 이러한 이유로 strtok()를 사용하지 않고 strtok\_r()를 사용합니다.

>> 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.

1. shell command separation(Unix approach)은 새 실행 파일이 시작할 때 다른 프로그램의 권한을 이어받습니다. 이는 사용자에게 액세스 권한이 있는 command와 file만 사용할 수 있도록 제한시켜주어 보안을 단순화시켜줍니다.

2. shell command separation(Unix approach)은 보안 또는 환경을 분석하거나 검증할 필요가 없으며 보안 문제가 될 수 있는 가능성도 제거하여 kernel을 단순화시킵니다.

3. shell command separation(Unix approach)은 매번 커널을 바꾸지 않고도 다양한 용도로 많은 shell interpreter를 가질 수 있게 해주며, 이는 사용자가 다른 shell을 사용할 수 있게 해줌으로써 더 많은 유연성을 창출합니다.

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.

in threads/thread.h

struct thread

{

...

#ifdef USERPROG

/\* These variables are made for project2 : user process. \*/

int exit\_status; /\* Status which thread has been terminated, 0 = terminated

successfully, 1 = not terminated yet, -1 = terminated by kernel. \*/

struct list child\_list; /\* List of child. Push children when they create. Remove all

elements when parent process exit.\*/

struct list\_elem child\_elem; /\* Element of child. \*/

struct semaphore wait\_sema; /\* Sema\_up when child process exit. Sema\_down when

parent's wait() occur. \*/

struct semaphore exit\_sema; /\* Sema\_up after parent remove his child in child\_list.

Sema\_down before child process exit.\*/

struct semaphore load\_sema; /\* Sema\_up after child load memory no matter what he

successes. Sema\_down right after child is created in process\_execute(). \*/

bool load\_success; /\* True if load success, False if not. \*/

struct file \*\*fd\_table; /\* Table of file descriptor. index 0,1 has standard value,

More than 2, user's. \*/

...

#endif

...

}

>> B2: Describe how file descriptors are associated with open files.

>> Are file descriptors unique within the entire OS or just within a

>> single process?

각 프로세스마다 fd\_table을 보유하고 있습니다. 하지만 index 0, 1을 kernel의 영역으로 두어 침범하지 않게끔 조정합니다. 0은 stdout, 1은 stdin을 의미합니다. file을 open하게 되면 이것은 user의 file descriptor이므로 fd\_table의 standard 0, 1을 제외한 index 2부터 순회하면서 해당 index가 비었는지 확인합니다. 이후 비어있는 가장 앞 칸에 open된 file의 정보를 저장해줍니다.

---- ALGORITHMS ----

in userprog/syscall.c

...

int

open (const char \*file)

{

if (file == NULL)

return -1;

lock\_acquire(&file\_lock);

/\* Change file type to struct file \*. \*/

struct file \*result = filesys\_open(file);

if (result == NULL)

{

lock\_release(&file\_lock);

return -1;

}

struct file \*\*table = thread\_current()->fd\_table;

/\* Check user's empty file descriptor. \*/

for (int i = 2;i < 128; i++)

{

if (table[i] == NULL)

{

/\* if current thread uses some files, then other can't access these files. \*/

if (strcmp(thread\_current()->name, file) == 0)

{

file\_deny\_write(result);

}

table[i] = result;

lock\_release(&file\_lock);

return i;

}

}

lock\_release(&file\_lock);

return -1;

}

...

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

>> kernel.

system call interrupt가 발동되어 read write에 필요한 fd 번호, buffer, size를 받아온 후 system call handler를 통해 read, write 가 실행됩니다. 두 함수 모두 parameter의 주소가 user 영역이 아니라면 프로세스를 즉각 종료시킵니다. 만약 file descriptor가 커널의 영역이 아닌 유저 영역이면 직접 참조(fd\_table[fd])를 통해 file을 가져오고, 그것이 NULL일 경우 -1을 리턴해줍니다. 또한, race condition을 막고 synchronization을 위해 각 함수가 시작할때 미리 init된 file\_lock을 acquire하고 함수의 각 조건문 끝에 있는 return 앞에 lock release를 실행시킵니다.

read()는 fd가 1이라면 stdout이므로 -1을 리턴합니다. fd가 0이라면 stdin이므로 file.c의 input\_getc() 함수를 활용하여 parameter로 들어온 buffer를 size만큼 읽어준 후 그 크기를 리턴해줍니다. 만약 fd가 0, 1아니라면 유저가 사용하는 fd이므로, file.c의 file\_read(fd\_table[fd], buffer, size)함수를 이용하여 유저 fd의 read bytes를 리턴해줍니다.

write()는 fd가 0이라면 stdin이므로 -1을 리턴합니다. fd가 1이라면 stdout이므로 file.c의 putbuf(buffer, size)를 활용하여 size만큼의 정보를 write합니다. 만약 fd가 0, 1이 아니라면 유저가 사용하는 fd이므로, file.c의 file\_write(fd\_table[fd], buffer, size)를 이용하여 유저 fd의 written bytes를 리턴해줍니다.

>> 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?

pagedir\_get\_page()를 살펴보면 정상 동작 시 pte\_get\_page (\*pte) + pg\_ofs (uaddr)를 return해 주는 것을 알 수 있습니다. 따라서 load 시 load\_segment(), set\_up\_stack()이 발동하면서 pagedir\_get\_page()의 리턴값만큼의 영역이 할당되고 (4096-return value)가 검색가능한 page table의 최댓값입니다. 이미 pagedir이 가득차있다면, 복사할 수 없으므로 최솟값은 0이 될 것입니다.

통일성을 부여해서 stack에 2byte씩 들어가더라도 4byte 값이 들어가는 것과 마찬가지의 과정을 진행하도록 코드를 짰습니다. 따라서 2byte가 들어가더라도 위에서 답변한 최댓값과 동일한 값이 보장됩니다.

>> B5: Briefly describe your implementation of the "wait" system call

>> and how it interacts with process termination.

child의 start\_process() 수행 완료를 기다리는 함수입니다. parameter로 들어온 tid를 이용해 current thread가 보유하고 있는 자식중에 같은 child가 있는지 검사해준 후 없으면 return -1(강제종료) 있다면 주소값을 저장해줍니다. 이후 찾아낸 child의 수행이 완료되었는지 확인하기 위해 child의 wait sema를 down시켜줍니다. 만약 child의 역할이 아직 끝나지 않았다면 parent는 여기서 block 되었다가 child가 wait sema를 up시켜줄 때 다시 실행될 것입니다. 반대로 sema down에 성공했다면 child의 exit\_status를 미리 저장해둔 후 child list에서 해당 child를 제거해준 후 실행을 마치고 죽음을 기다리고 있는 (exit\_sema를 먼저 down하여 block되어있는 child) child를 위해 exit sema를 up 시켜줍니다. 마지막으로 저장한 child의 exit\_status를 return하고 함수를 종료합니다.

여기서 고려해야 할 점은 만약 child 가 이렇게 의도한 과정대로 종료절차를 밟지 않고 wait sema를 up 시켜주기 전에 어떠한 이유로 강제종료되었다면 parent는 parent의 child list에 있는 강제종료된 child를 child\_list 에서 없애주지 못한 채 return -1(강제종료의 의미) 될 것이고, 이것은 메모리의 누수로 이어질 수 있다는 것입니다. 따라서 exit()하면서 다시금 child와의 관계를 정리해주어야 합니다. process\_exit()에서 current thread(parent) 의 child list를 순회하며 강제종료된 child가 있다면 (child->exit\_status = -1) 해당 child의 wait\_sema를 올려준 뒤 해당 child에 대한 wait를 시행합니다. 이때 시행된 process\_wait() 에서 해당 child와의 관계를 정리해줍니다. 하지만, 해당과정을 process\_wait()를 통해 진행하지 않고 process\_exit() 시에 parent의 child\_list를 순회하며 전부 제거해주는 것으로 이 복잡한 과정을 단순화시켰습니다.

>> 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.

저희는 안전한 처리를 염두해두고 코드를 작성했습니다. syscall\_handler()에서 각 system call case에 맞게 나눠진 항목에서 syscall의 핵심적을 동작을 하는 함수를 실행하기 전에 parameter에 들어갈 요소들을 check\_esp()함수를 통해 user address인지를 확인하여 user address가 아니라면 exit(-1)을 진행하였습니다. 그리고 read()에서는 시작 시에 인자로 받는 buffer 또한 user address인지를 확인해주었습니다.

system call의 동작 자체는 간단해서 system call에서 임시로 할당된 변수들이 free되지 않는 현상은 발생하기 힘들었고, 발생하더라도 금방 디버깅할 수 있었습니다.

(parent를 P, child를 C라 하겠습니다.)

하지만 C의 push\_args\_to\_stack()을 위해 P의 process\_execute()에서 file\_name을 복사해준 fn\_copy는 달랐습니다. P가 실행중에 만들어졌지만 C에서 이용하기때문에 fn\_copy를 free를 어디서 시켜줘야하는지에 대한 문제가 발생했습니다. 처음에는 이용을 마친 C의 start\_process()의 push\_args\_to\_stack() 직후에 실행해주려 했으나 이 방식으로 args-many test case를 통과할 수 없었습니다. 따라서 기존에 구현해 놓았던 load\_sema를 다시 한번 활용하게 되었습니다. 기존의 역할은 그대로 두고 새로운 역할을 부여했는데, 바로 fn\_copy를 전부 활용해서 더 이상 쓰지 않는다고 알리는 것입니다. P가 load\_sema를 얻으면(load\_sema down) 이제 fn\_copy는 더이상 쓸모가 없다는 의미이므로 이때 palloc\_free\_page(fn\_copy)를 진행합니다. 이것으로 fn\_copy의 메모리 누수문제가 해결되었습니다.

---- 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"?

저희는 load\_sema를 추가하여 이 문제를 해결했습니다. 문제상황과 같이 child process가 load를 마쳤음에도 그 결과가 실패라면 parent 역시 강제종료 시켜야하는데 먼저 child의 로드 성공여부와 관련없이 load가 끝났음을 parent가 알아야 합니다. 이를 load\_sema를 통해서 해결했습니다. child는 성공여부와 상관없이 load를 마치는대로 load의 성공여부를 child의 load\_success에 저장후 load\_sema 를 up시켜줍니다. parent의 execute중 만들어진 child 에 대해 기본적인 검사를 마쳐준 후(tid == tid\_error, tid 기반으로 child\_list를 순회, 검색했으나 해당 child를 찾지못함) 바로 load\_sema를 down해주고, 여기서 block 된다면 아직 child가 load를 수행하지 못한 것으로 해석할 수 있습니다. sema down의 다음 instruction으로 넘어간다면 바로 해당 child의 load\_success여부를 불러와 실패 시 return -1, 성공 시 해당 child의 tid를 return해줍니다.

>> 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?

처음에는 wait\_sema만 이용하여 P의 exit를 진행하려고 했습니다. 하지만 주어진 문제의 상황처럼 P의 wait를 마치고 C에 대해서 처리(B5에 기술된 과정입니다.)해주려고 할 때 이미 C가 exit되어 있었다면 해당 과정을 진행할 수 없었습니다. 이것을 해결하기 위해 C가 죽는 것을 P가 허락해줘야 죽을 수 있게끔 새로운 exit\_sema를 추가했습니다. 간단히 설명해서 C : “내 할 일 다했어(wait\_sema up) 집가도 돼?(exit\_sema down)” 라고 요청하면 P : “확인해볼게 기다려봐 (wait\_sema down) 어 잘했네 이제 가봐 (exit\_sema up)” 의 과정을 거치는 것입니다. 이 방법과 B5의 P exit시 강제종료된 모든 C의 처리로 메모리 누수 없이 user process의 진행과정을 마칠 수 있었습니다.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the

>> kernel in the way that you did?

user 가 kernel의 영역에 접근을 시도(system call)한 후 그 결과를 돌려받는 가장 간단하고 확실한 방법은 바로 user가 원하는 값을 return 해주는 것이었습니다. 따라서 저희는 syscall\_handler()의 각 case에 해당하는 모든 함수들에 대해(void type 제외) file.c의 함수를 return하여 유저가 원하는 정보를 즉각적으로 받아 갈 수 있도록 구현하였습니다.

>> B10: What advantages or disadvantages can you see to your design

>> for file descriptors?

kernel 영역과 user 영역을 확실히 구분지어 놓음으로써 kernel 영역이 침범당해 critical한 에러가 나는것을 방지하였습니다. 이 방식은 안전성을 보장할 수 있지만 user가 kernel의 영역에 접근하려 할때마다 system call의 관리를 받아야 하는 과정을 거치기 때문에 속도의 측면에서는 단점이라고 할 수 있습니다.

>> B11: The default tid\_t to pid\_t mapping is the identity mapping.

>> If you changed it, what advantages are there to your approach?

tid는 실제로 커널(스레드)에서 스케줄링 가능한 객체의 식별자인 반면 pid는 메모리와 file descriptors(프로세스)를 공유하는 스케줄링 가능한 객체 그룹의 식별자입니다. pid와 tid를 구분하면 큰 관리(프로그램의 시작/종료)에서는 프로세스 덩어리로 구분할 수 있고, 실행에서는 쓰레드 단위로 구분할 수 있습니다. 이는 관리(프로세스)와 효율(쓰레드)의 장점을 모두 가질 수 있습니다. 하지만 single thread인 pintos에서는 이러한 이점을 살릴 수는 없을 것 같습니다.

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 three problems

>> in it, too easy or too hard? Did it take too long or too little time?

project1에 비해 pintos의 구조를 어느 정도 아는 상태에서 시작해서 쉬울 줄 알았으나, file disk를 setup하는 과정(그 중에서도 file disk를 format하는 과정)에서 project1의 test case에서는 고려하지 않아도 되었던 sema\_up()의 thread\_yield() 부분의 문제를 발견했습니다. project2를 미리 구현하기 위해 추석연휴부터 pintos를 시작했음에도, 이를 발견하고 해결하는데만 1주일이 소요되었습니다. 이후에는 비교적 빠르게 원하는 부분을 구현했으나 test case의 multi-oom을 해결하기 위해 4일 정도를 소요한 것 같습니다. 할당해준 메모리에 대해 메모리 누수에 대한 처리, file descriptor를 여러 프로세스가 접근할 때 synchronization이 되지 않는 것에 대한 처리, 메모리에 load 되지 않았을 경우의 처리 등이 처음에는 구현되어 있지 않았고, multi-oom test case를 pass하기 위해 하나하나 짚어가며 추가적으로 구현했습니다.

이번 project2에서도 project1때처럼 평일에는 매일 12시부터 3~4시까지, 주말과 연휴에는 거의 하루종일 zoom 화상회의를 통해 직접 얘기를 주고받으며 아이디어 design과 코딩을 했습니다. 결과적으로 총 소요시간을 따지면 디버깅을 잘 하지 못해서 project1때보다 많은 시간을 소요한 것 같습니다.

추가적으로 이 document의 문제들이 까다로워서 이것을 작성하는 것만 해도 하루정도의 시간이 소요되었습니다.

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

project2를 통해 os에서 process가 어떻게 작동하는지, child가 끝날 때까지 parent가 어떻게 대기하는지, 사용중인 file에 동시 접근을 막기 위해 어떤 방식을 사용해야 하는지 등을 새롭게 알게 되었습니다. 또한 syscall의 작동 방식을 추상적으로만 알고 있었는데 이를 구체화 시킬 수 있었습니다.

>> 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?

ppt와 공식 문서의 설명 만으로는 multi-oom test case를 통과하기 위한 설명이 부족할 수도 있다고 생각했습니다. 다음 ppt에는 메모리 누수 문제, 고아 프로세스 처리에 대한 문제, load 실패시의 문제 등을 언급해주신다면 처음 큰 그림을 그릴 때 많은 도움이 될 것 같습니다.

>> Do you have any suggestions for the TAs to more effectively assist

>> students, either for future quarters or the remaining projects?

없습니다.

>> Any other comments?

없습니다.