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

| CS5600 |

| PROJECT 2: USER PROGRAMS |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

Guanghao Ding <ding.g@husky.neu.edu>

Wu Jiang <wujiang@ccs.neu.edu>

---- 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.

None

---- 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?

How to implement argument parsing?

----------------------------------

The most important part was to setup the stack. We did it inside setup\_stack ()

after page is installed, when the stack has been initialized.

Process\_execute provides file\_name, including command and arguments

string. First, we separated the first token and the rest, which are command and

arguments. We use command as the new thread's name, and pass down the arguments

string to start\_process(), load() and setup\_stack(). We think it’s implementable

since we can always get the command name from thread->name when needed, like

when load the ELF executable.

When setting up the stack, we memcpy the argument string and then the command

name which is actually the thread name in our case. Then add alignment, scan the

string backward to get each token and push its address into the page underneath

the alignment to generate argv[], finally argv, argc and return address.

Way of arranging for the elements of argv[] to be in the right order.

--------------------------------------------------------------------

We scan through the argument string backwards, so that the first token we get is

the last argument, the last token we get is the first argument. We can just keep

decreasing esp pointer to setup the argv[] elements.

How to avoid overflowing the stack page?

----------------------------------------

The thing is we decided not to check the esp pointer until it fails. Our

implementation didn’t pre-count how much space do we need, just go through

everything, make the change, like add another argv element, when necessary. But

this leaves us two way to deal with overflowing, one is checking esp’s validity

every time before use it, the other one is letting it fails, and we handle it in

the page fault exception, which is exit(-1) the running thread whenever the

address is invalid. We chose the latter approach since the first approach seems

have too much burden and it make sense to terminate the process if it provides

too much arguments.

---- RATIONALE ----

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

The only difference between strtok\_r() and strtok() is that the save\_ptr

(placeholder) in strtok\_r() is provided by the caller. In pintos, the kernel

separates commands into command line (executable name) and arguments. So we need

to put the address of the arguments somewhere we can reach later.

>> 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) Shortening the time inside kernel

2) Robust checking. Checking whether the executable is there before passing it

to kernel to avoid kernel fail. Checking whether the arguments are over the

limit.

3) Once it can separate the commands, it can do advanced pre-processing, acting

more like an interpreter not only an interface. Like passing more than 1 set

of command line at a time, i.e. cd; mkdir tmp; touch test; and pipe.

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 thread.h

/\* used to indicate the child’s status, owned by wait-syscall \*/

struct child\_status

{

tid\_t child\_id;

bool is\_exit\_called;

bool has\_been\_waited;

int child\_exit\_status;

struct list\_elem elem\_child\_status;

};

struct thread

{

...

#ifdef USERPROG

...

/\* direct parent thread id \*/

tid\_t parent\_id;

/\* signal to indicate the child’s executable-loading status:

\* - 0: has not been loaded

\* - -1: load failed

\* - 1: load success \*/

int child\_load\_status;

/\* monitor used to wait the child, owned by wait-syscall and the waiting for

\* child to load executable \*/

struct lock lock\_child;

struct condition cond\_child;

/\* list of children, which should be a list of child\_status struct. Owned by

\* wait-syscall

\*/

struct list children;

/\* file struct represents the executable of the current thread, introduced

\* to deny the running executable and re-enable the write after thread

\* exits.

\*/

struct file \*exec\_file;

#endif

...

}

== File System ==

/\* In syscall.c \*/

/\* file descriptor \*/

struct file\_descriptor

{

/\* the unique file descriptor number returns to user process. \*/

int fd\_num;

/\* the owner thread’s thread id of the open file \*/

tid\_t owner;

/\* file that is opened \*/

struct file \*file\_struct;

struct list\_elem elem;

};

/\* a list of open files, represents all the files open by the user process

\*through syscalls.

\*/

struct list open\_files;

/\* the lock used by syscalls involving file system to ensure only one thread at

\* a time is accessing file system

\*/

struct lock fs\_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?

描述文件描述符如何与打开的文件关联。文件描述符在整个操作系统中还是在单个进程中是唯一的？

1. 在我们的代码实现中，文件描述符和通过系统调用打开的每个文件之间是一对一映射的。
2. 文件描述符是在单个进程中唯一的，不会被子进程继承。我们用struct file\* files\_map[OPEN\_MAX]来记录进程打开的所有文件。

---- ALGORITHMS ----

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

>> kernel.

描述用于从内核读取和写入用户数据的代码。

**读取数据：**

首先，令res=-1，获取filesys\_lock（文件系统锁）。然后判断文件描述符fd的值。

如果fd为0，那么使用input\_getc()从键盘读入至多size个字节，每读一个字节就将其存到buffer中，如果失败了就释放filesys\_lock并调用syscall\_exit函数退出（syscall\_exit函数传参为错误码）。读入完成后返回实际读取的字节数。

如果fd不为0，那么先检查文件描述符是否合法（fd>=0 && fd<128）。合法的话就从当前进程的文件描述符表中找到该文件描述符对应的文件。对应文件存在则读取内容，并设置res为实际读取的字节数。

最后释放filesys\_lock，并且返回res。

**写入数据：**

首先，令res=-1，获取filesys\_lock（文件系统锁）。然后判断文件描述符fd的值。

如果fd为1，调用putbuf()将buffer中的size字节一次性写入控制台，设置res=size。

如果fd不为1，那么先检查fd是否合法（fd>=0 && fd<128）。合法的话就从当前进程的文件描述符表中找到该文件描述符对应的文件，对应文件存在则将buffer的内容写入文件，设置res为实际写入的字节数。

最后释放filesys\_lock，返回res。

>> 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?

假设系统调用导致整页（4,096字节）的数据从用户空间复制到内核。可能导致的对页表的最少检查次数和最大检查次数（例如，对pagedir\_get\_page()的调用）？对于仅复制2个字节的数据的系统调用该怎么办？这些数字是否还有改进的余地，还有多少？

**对于整页数据：**

最小检查次数是1。如果第一次检查能够从地址中得知页头，那么其实不需要再继续检查了，它可以包含一页数据。

如果不连续，最大检查次数是4096，因为需要检查每个地址来确保有效访问。如果连续，最大检查次数是2，也就是如果我们获得了一个不是页头的内核虚拟地址，

就需要检查整页数据的开始指针和结束指针，检查是否映射。

**对于仅复制2个字节：**

最小数字是1。理由同上，如果返回到页面末尾有2个以上字节空间的内核虚拟地址，可以得知一定在此页面中，就无需再次检查。

最大数字则是2。无论连续或者不连续，当返回的内核虚拟地址距离页面末尾只有1个字节时，必须要检查另一个字节的位置。

**改进**：

目前尚未看出有改进的余地。

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

>> and how it interacts with process termination.

We implement wait-syscall in term of process\_wait.

We define a new struct child\_status to represent child’s exit status. And a list

of child\_status is added into parent’s thread struct, representing all children

the parent owns. We also introduce a parent\_id inside child’s struct, to ensure

child can find parent and set it’s status if parent still exists.

A child\_status is created and added to list whenever a child is created, then

parent will wait(cond\_wait) if child has not already exited, child is

responsible to set it’s return status and wake up parent.

We have a monitor in parent’s struct to avoid race condition. Before checking or

setting the status, Parent and Child both should acquire the monitor first.

If parent is signaled or sees the child has exited (checking using the function

we wrote thread\_get\_by\_id ), it will start to check the status.

If child calls exit-syscall to exit, a boolean signal that indicate exit-syscall

is called and the child’s exit status will be set into the corresponding

child\_status struct in parent’s children list.

If child is terminated by kernel, the boolean signal mentioned above is remain

as false, which will be seen by parent, and understood child is terminated by

kernel.

If parent terminates early, the list and all the structs in it will be free,

then the child will find out the parent already exited and give up setting the

status, continue to execute.

>> 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.

First, avoiding bad user memory access is done by checking before validating, by

checking we mean using the function is\_valid\_ptr we wrote to check whetehr it’s

NULL, whether it’s a valid user address and whether it’s been mapped in the

process’s page directory. Taking “write” system call as an example, the esp

pointer and the three arguments pointer will be checked first, if anything is

invalid, terminate the process. Then after enter into write function, the buffer

beginning pointer and the buffer ending pointer(buffer + size - 1) will be

checked before being used.

Second when error still happens, we handle it in page\_fault exception. We check

whether the fault\_addr is valid pointer, also using is\_valid\_ptr we provide. If

it’s invalid, terminate the process. Taking the bad-jump2-test( \*(int

\*)0xC0000000 = 42; ) as an example, it’s trying to write an invalid address,

there is no way we could prevent this case happen, so, when inside page\_fault

exception handler, we find out 0xC0000000 is not a valid address by calling

is\_valid\_ptr, so we call set the process return status as -1, and terminate the

process.

---- 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"?

Our design is to have the child\_load\_status recorded in the parent’s

thread. Child is responsible to set child\_load\_status. Child can get the parent

thread through a new field parent\_id and the function we provide

(thread\_get\_by\_id) to get the parent thread’s access.

The reason we choose this design is that the child thread can exit anytime due

to some odd reason. So, if we save it in the child process, there is no way to

retrieve the status if it exits before the parent checking on it. A thread can

only wait a thread to load at a time, so use only one variable inside the parent

thread is enough.

We also introduce a monitor. When child’s load success/failure, the child will

get the parent thread by it’s id, acquire the monitor to set the the value

inside parent’s thread, signal the parent. When child is exit accidentally, it

will set up the value to fail either. Before the parent create the child thread,

the parent will set up child\_load\_status to 0, which is initial value means

nothing happens so far. After thread is created, the parent acquire the monitor

to wait until child\_load\_status is not 0.

>> 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?

We use a child\_status struct to represents each child process’s status, and a

list of child\_status inside parent’s struct to represent all the children that

the process has. And use a monitor to prevent race condition.

Child is responsible to set it’s status in parent’s thread struct. When parent

exits, the list inside it will be free.

So, in the cases above:

\* P calls wait(C) before C exits

P will acquire the monitor and wait until it exits by checking the child

thread’s existence through a function (thread\_get\_by\_id) we wrote, which checks

all-thread-list. Then parent retrieves the child’s exit status.

\* P calls wait(C) after C exits

P will acquire the monitor and found out C already exits and check it’s exit

status directly.

\* P terminates without waiting before C exits

The list inside P will be free, the lock will be released, since no one will

wait a signal except parent, condition don’t need to be signaled. When C tries

to set it’s status and find out parent has exited, it will ignore it and

continue to execute.

\* P terminates after C exits

The same thing happen to P, which is free all the resources P has.

---- RATIONALE ----

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

>> kernel in the way that you did?

We did by validating it before using it. At the point we implemented it, we

didn’t really understand the second approach which deals with the page fault

inside exception and how the putuser() and getuser() can be used. We are just

not confident enough to choose the second approach.

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

>> for file descriptors?

Advantages:

1) Thread-struct’s space is minimized

2) Kernel is aware of all the open files, which gains more flexibility to

>> manipulate the opened files.

Disadvantages:

1) Consumes kernel space, user program may open lots of files to crash the

kernel.

2) The inherits of open files opened by a parent require extra effort to be

implement.

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

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

We didn’t change it. We think it’s reasonable and implementable.

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?

>> 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?