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

| PROJECT 2: USER PROGRAMS |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

蔡璟辉 10132510143@ecnu.cn

陈逸轩 10132510144@ecnu.cn

朱云韬 10132510145@ecnu.cn

张辰 10132510133@ecnu.cn

---- PRELIMINARIES ----

The code of Project2 has been uploaded to:

<https://github.com/10132510143cai/OS15x133_143_144_145>

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.

struct thread

{

/\* Owned by thread.c. \*/

tid\_t tid; /\* Thread identifier. \*/

enum thread\_status status; /\* Thread state. \*/

char name[16]; /\* Name (for debugging purposes). \*/

uint8\_t \*stack; /\* Saved stack pointer. \*/

int priority; /\* Priority. \*/

struct list\_elem allelem; /\* List element for all threads list. \*/

/\* Shared between thread.c and synch.c. \*/

struct list\_elem elem; /\* List element. \*/

/\* My Implementation \*/

struct alarm alrm; /\* alarm object \*/

int base\_priority; /\* priority before donate, if nobody donates, then it should be same as priority \*/

struct list locks; /\* the list of locks that it holds \*/

bool donated; /\* whether the thread has been donated priority \*/

struct lock \*blocked; /\* by which lock this thread is blocked \*/

int nice; /\* nice value of a thread \*/

int recent\_cpu; /\* recent cpu usage \*/

/\* == My Implementation \*/

#ifdef USERPROG

/\* Owned by userprog/process.c. \*/

uint32\_t \*pagedir; /\* Page directory. \*/

/\* My Implementation \*/

struct semaphore wait; /\* semaphore for process\_wait \*/

int ret\_status; /\* return status \*/

struct list files; /\* all opened files \*/

struct file \*self; /\* the image file on the disk \*/

struct thread \*parent; /\* parent process \*/

/\* == My Implementation \*/

#endif

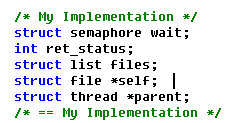
/\* Owned by thread.c. \*/

unsigned magic; /\* Detects stack overflow. \*/

};

In the struct thread

We add



The wait is the semaphore for process\_wait.

Ret\_status is the return value.

Files is to store all opened files.

Self is point to the image on the disk.

Parent is point to the parent process.

---- 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?

First, we know the program execution sequence.

①the program start from main in init.c

②process\_wait (process\_execute (task)) call process\_execute

③in process\_execute call

thread\_create(fn,PRI\_DEFAULT,start\_process,fn\_copy),and wait until the thread have started

④ thread\_create will put start\_process in stack then call it you can see it from the picture

We implemented argument parsing in a function called parse\_args(). This function is called by setup\_stack(). parse\_args() begins by allocating a memory space of 2 char pointers for the argv using malloc(). We then use strtok\_r() to iterate through the command line. When a token is extracted from the command line, the stack pointer will be adjusted to accomodate a new char pointer. The pointer will be placed into argv and the token will be copied to that location using memcpy().

Since there can be any number of arguments passed, we set the default size of argv to 2. The size will doubled accordingly using realloc().

Next we push the arguments in argv in reverse order so the file-name will be at the top of the stack.

Lastly, we push the start of argv onto the stack, followed by the argc the fake "return address". The function call ends with the argv being freed by the free() function call.

---- RATIONALE ----

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

In C, strtok() keeps track of where the last token was located within the function so that it may continue parsing in subsequent calls. In contrast, strtok\_r() requires a save\_ptr object to passed in order to keep track of the last token found.

Using strtok() in PintOS would likely result in a race condition if multiple threads were to invoke the function.

In a word, strtok() is unsafe when used in multi-thread. The strtok()define a global variable char \* strtok; when multi-thread use this function at the same time ,the global variable is uncertain.

While in the strtok\_r(),it use a variable char \*save\_ptr, so it is safe.

>>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.

Advantages:

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.

Syscall.c:

struct fd\_elem

{

int fd;

struct file \*file;

struct list\_elem elem;

struct list\_elem thread\_elem;

};

typedef int pid\_t;

static struct list file\_list;

static struct lock file\_lock;

All the above is added that syscall.c.

The file\_lock is the lock used by syscalls involving file system to ensure only one thread at

a time is accessing file system.

The file\_list is a list of open files, represents all the files open by the user process

through syscalls.

The int fd is the unique file descriptor number returns to user process.

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

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

>> single process?

In our implementation, file descriptors has a one-to-one mapping to each file opened through syscall. The file descriptor is unique within the entire OS. We decided to maintain a list(struct list open\_files) inside kernel, since every file access will go through kernel.

---- ALGORITHMS ----

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

>> kernel.

Read:

First, check if the buffer and buffer + size are both valid pointers, if not, exit(-1). Acquire the fs\_lock (file system lock). After the current thread becomes the lock holder, check if fd is in the two special cases: STDOUT\_FILENO and STDIN\_FILENO. If it is STDOUT\_FILENO, then it is standard output, so release the lock and return -1. If fd is STDIN\_FILENO, then retrieve keys from standard input. After that, release the lock and return 0. Otherwise, find the open file according to fd number from the open\_files list. Then use file\_read in filesys to read the file, get status. Release the lock and return the status.

Write:

Similar with read system call, first we need to make sure the given buffer pointer is valid. Acquire the fs\_lock. When the given fd is STDIN\_FILENO, then release the lock and return -1. When fd is STDOUT\_FILENO, then use putbuf to print the content of buffer to the console. Other than these two cases, find the open file through fd number. Use file\_write to write buffer to the file and get the status. Release the lock and return the status.

>> 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?

For a full page of data:

The least number is 1. If the first inspection(pagedir\_get\_page) get a page head back, which can be tell from the address, we don’t actually need to inspect any more, it can contain one page of data.

The greatest number might be 4096 if it’s not contiguous, in that case we have to check every address to ensure a valid access. When it’s contiguous, the greatest number would be 2, if we get a kernel virtual address that is not a page head, we surely want to check the start pointer and the end pointer of the full page data, see if it’s mapped.

For 2 bytes of data:

The least number will be 1. Like above, if we get back a kernel virtual address that has more than 2 bytes space to the end of page, we know it’s in this page, another inspection is not necessary.

The greatest number will also be 2. If it’s not contiguous or if it’s contiguous but we get back a kernel virtual address that only 1 byte far from the end of page, we have to inspect where the other byte is located.

Improvements:

NONE.

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

>> and how it interacts with process termination.

Wait () works similarly to process\_wait(). It begins by searching for the process denoted by pid in the child\_process list. If the process isnt in the list, then it most likely already terminated, so we just return -1. If pids parent is already waiting on it, then well also return -1. If these two conditions pass, then we set the childs wait variable to TRUE. If the process hasnt yet terminated, we wait on the child\_processs exit semaphore. Once the semaphore is raised when the child exits, then the parent obtain the exit status and remove the child process from the child\_process list.

>> 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.

Before syscall is executed, we perform a test for the pointers that will be passed in as arguments. Depending in the system call, we will check pointers using either check\_valid\_buffer() or check\_valid\_string(). If these functions pass we`ll retrieve the page from emory using user\_to\_kernel\_ptr(). If the page does not exeists--equeals NULL-- user\_to\_kernel\_ptr() will esit with an error. If a process is terminated with an error, process\_exit() will be invoked. In process\_exit() we make sure any resources allocated by the process is correctly deallocated. This included any open files, pages in memory, locks, and semaphores.

---- 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 its 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?

\* 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?

To do a memory lookup in order to correct allocation

>> 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.

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

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

NONE.