**华东师范大学软件学院实验报告（设计文档）**

|  |  |  |
| --- | --- | --- |
| **实验课程**：操作系统 | **年级**：2013级 | **实验成绩**： |
| **实验名称**：USER PROGRAMS | **姓名**：周仪 | **组员：** |
| **实验编号**： | **学号**：10132510114 | **实验日期**： |
| **指导教师**：陈闻杰 | **组号**： | **实验时间**：2015-06-15 |

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

| CSE 451 |

| PROJECT 2: USER PROGRAMS |

| DESIGN DOCUMENT |

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

---- GROUP ----

10132510113 – 胡诗璐<10132510113@ecnu.cn>

10132510114 – 周 仪<10132510114@ecnu.cn>

10132510115 – 董思远<10132510115@ecnu.cn>

10132510116 – 茅凯博<10132510116@ecnu.cn>

---- 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.

/\* struct arguments is used to pass 1, 2, or 3 argumentss to the system

calls that require them \*/

struct arguments {

void \*args[SUPPORTED\_ARGS];

};

/\* struct arguments is used to pass command line args to new user processes \*/

struct arguments {

int num\_args;

char \*args[MAX\_ARGS];

};

---- 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?

We start from main in init.c. Then process\_wait(process\_execute(task)) call process\_execute.And in process\_execute call thread\_create,and wait until the thread have started. Then thread\_create will put start\_process in stack then call it.

In start\_process, we tokenize the given file\_name (which contains the arg on

spaces) and fill an arguments struct up with the args and number of args. We

pass that args struct to load, and load places all the arguments directly

below PHYS\_BASE in the appropriate order:

for (i = args->num\_args - 1; i >= 0; i--)

{

int total\_length = strlen(args->args[i]) + 1;

char \*dest = top - total\_length;

memcpy((void \*) dest, (void \*) args->args[i], total\_length);

stack\_pointers[i] = (void \*) dest;

top = dest;

} // where top is initialized to PHYS\_BASE

Then we finished setting up the stack by following the layout described in

section 3.5.1, Program Startup Details, in the assignment document: PHYS\_BASE,

args in reverse order, word-align, 4-byte zeroed out, pointers to args in

reverse order, pointer to pointer to first arg, num\_args, return (null).

We avoid overflowing the stack by performing a check on the total size of the

args being passed. If it would overflow the stack page size, we exit.

---- RATIONALE ----

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

strtok\_r is more thread safe.The reason is strtok() is unsafe when used in multi-thread.the strtok() define a global variable.It is reentrant, to avoid the case where another thread gains control and also calls strtok, which would change the savepointer.When the original thread regains control, it would pick up where the other thread's strtok left off. With strtok\_r, we provide the saveptr, so we avoid that problem.

>> 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.

The Unix approach is faster,because it don't have to switch to the kernel mode to separate the command.

It seems cleaner to separate the executable name from the arguments before

passing it off to the kernel, since they represent different things. It

shouldn't be the kernel's job to parse that, there's no reason it couldn't be

done by a user program.

While we want to extend the function or change the function of the separate command, with Unix approach,it will more convenient and with less code to change.

Also, perhaps some validation of the input could be done by the shell more

safely than by the kernel. If someone entered a very large amount of text,

perhaps it would cause the kernel a problem if the kernel tried to parse it,

whereas if the shell takes care of it, worst case is the shell crashes.

SYSTEM CALLS

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

---- FROM USERSPACE TO THE KERNEL ----

>> B0: When syscall\_handler() is called, where is the system call number

>> stored? Where are the arguments to the system call, if any, stored?

>> Give the backtrace of the stack during syscall\_handler() (be sure

>> to resolve addresses both in user space and in kernel space!) and

>> give your answer in relation to this backtrace. Explain why the

>> syscall number and arguments appear at this place in the stack.

We store the system call number at the esp of the interrupt frame. And then store the pointers to the args above the esp on the stack.

Here is a backtrace of the stack for a write system call (for open-bad-ptr):

Oxbfffff48: 21

Oxbfffff44: 0x804be20

Oxbfffff40: 1

Oxbfffff3c: 12

Oxbfffff3c is the esp, and above that is stored the file descriptor for write,

above that is stored the pointer to the buffer containing the characters to be

written, and above that is the size of the content in the buffer. No more

arguments are stored above that. They are in these places because they are

pushed onto the stack in that order.

---- 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.

struct child\_thread\_info {

bool dead;

tid\_t tid;

int status;

struct list\_elem waiting\_list\_elem;

};

/\* Added to struct thread:

struct list fd\_list: list of file descriptor structs belonging to this

thread

struct thread \*parent\_thread: pointer to parent thread

struct lock forking\_child\_lock: lock that makes a child wait as it's being

forked until the parent can finish creating it

struct condition forking\_child\_cond: condition variable for

forking\_child\_lock

bool ready: a boolean to track whether the child is fully created

struct intr\_frame i\_f: an interrupt frame to store the state of the parent

that the child uses to resume execution after being created,

struct list child\_list: list of child\_info\_structs to track state of

children

struct condition waiting\_for\_child: used to signal a parent process that is

waiting on a chilwhile that child is still alive

struct lock waiting\_child\_lock: used to lock a parent process that is

waiting on a child while that child is still alive

\*/

struct thread {

...

#ifdef USERPROG

...

struct list fd\_list;

struct thread \*parent\_thread;

struct lock forking\_child\_lock;

struct condition forking\_child\_cond;

bool ready;

struct intr\_frame i\_f;

#endif

// For managing child processes in wait and fork

struct list child\_list;

struct condition waiting\_for\_child;

struct lock waiting\_child\_lock;

...

}

/\* struct fd stores the file descriptor int, a pointer to a file (null if

piped), a pointer to a buffer (null if not piped), a list\_elem \*/

struct fd {

int fd;

struct file \*f;

struct list\_elem fd\_elem;

struct fd\_buffer \*buf;

};

/\* struct fd\_buffer is a buffer that piped fds read/write to/from \*/

struct fd\_buffer {

int ref\_count;

char fd\_buf[FD\_BUF\_SIZE];

int first;

int last;

};

// UNUSED

/\* struct fork\_args is unused \*/

struct fork\_args {

uint32\_t \*pagedir;

uint32\_t \*parent\_stack;

};

/\* struct populate\_child\_args is used to pass information to a child

process that is being forked into existence \*/

struct populate\_child\_args {

uint32\_t \*pagedir;

struct intr\_frame \*i\_f;

tid\_t child\_tid;

};

>> 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 descriptors are typical just within a single process. Every process tracks

a list of its file descriptors ,such as list of struct fd, stored in struct thread,

as well as its next available fd number. Our fd struct is what associates the file descriptor numbers with the corresponding file .

---- ALGORITHMS ----

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

>> kernel.

Before we do the access. we test all pointers and accesses.By ensuring that all pointers we encounter are valid right off the bat, we can access memory directly without worry.

>> 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?

If pagedir\_get\_page() is not used for pointer validation, and the data is distributed in byte-sized segments across 4096 pages, the largest number of times be 4096. And memcpy would be passed pointers returned by pagedir\_get\_page().

If pagedir\_get\_page() is not used for pointer validation, and all the data is stored on a single page, the least number of times could be 1. And memcpy would be passed the pointer returned by pagedir\_get\_page().

The least number of calls would be 1, and the max would be 8192 (2\*4096) without the use of pagedir\_get\_page().

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

>> and how it interacts with process termination.

The given tid corresponding to a child process by iterating

through the list of child\_info\_structs stored by the current thread should be ensured.If the status is Thread\_dying, or t->parent has already waited the child\_tid, then return -1. Nevertheless, if t->status is not the default status, then return the status. t->parent=thread\_current() means that the parent has waited the child. Sema\_down(&t->t\_sema) means that wait until the child thread has died.When the child thread has sema\_up(&t->t\_sema),then printf the termination messages.

>> 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.

Every time we get a pointer we would check that the returned value is not null. This ensures that this pointer references a valid address that is mapped the page directory. We also make sure every pointer that we access is a virtual address, using the is\_user\_vaddr function. This tests to make sure that the pointer is less than

PHYS\_BASE, and therefore not a kernel pointer.

When a pointer argument is used in the system call, we validate that pointer is correct. Each of these types of arguments are passed with size, so we can check

the pointer to the end of the buffer or char \* argument and ensure that

pointer is also valid.

For Example:

A system call in write().

The lock needs to be take care of cautiously. Before starting, I would check whether the buffer is valid and whether the buffer is big enough (!is\_user\_vaddr(buffer) || !is\_user\_vaddr(buffer+length)). And at the end of the function,the lock needs to be released.

---- 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"?

In order to maintain the information of the current thread without starting a child process, we use the function. However, if an executable fails when loading with our implementation, then that thread fails.

If we were to ensure that a failing exec call returns -1 to the calling

process, the following would be a possible design:

First we make a 'test call' to load and check if that test call successfully loads with an interrupt handler. If it does load, then we could continue with a call to start\_process. If it fails to load, then ahead of time we would need to return -1.

>> 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?

After updating C，P，if wait(C) is called before C exits, the parent waits until C enters process\_exit, at which time C updates it's status in P's list. Then it removes the child\_thread\_info struct corresponding to C from its list and returns C's exit status.

while (!child->dead) {

cond\_wait(&cur\_thread->waiting\_for\_child, &cur\_thread->waiting\_child\_lock);

}

If wait(C) is called after the child exits at the first time, P will check its

list of child\_thread\_info structs. In the condition that C is dead, P would remove

the child\_thread\_info struct corresponding to C and return C's exit status. If wait(C) is called after the child exits at second time, P will return -1.

If P terminates after C exits, C's parent pointer is nulled out and P's list of child\_thread\_info structs is freed, along with all of its other malloc'd data.

If P terminates before C exits, just the freeing of P's data.

If P have already waited C, it would return -1, which is a special case.

---- RATIONALE ----

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

>> kernel in the way that you did?

Because it is convenient, less of coding and high rate of success.

Through much modification, current method is more efficient compared to some other methods.

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

>> for file descriptors?

Advantages:

The structure can store the necessary information, and be used in more essentially than the same way.

There is no limit on the number of open file descriptors (until we run out of memory).

Disadvantages:

There exist many duplicate file descriptor structs.

Accessing a file descriptor is O(n), where n is the number of file descriptors

for the current thread (have to iterate through the entire fd list). Could be O

(1) if they were stored in an array.

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

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

Our current method is also the finally method which get through much modification.

The advantage is the ability to store extra information, then you could determine your parent just by looking at your pid.