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

| PROJECT 2: USER PROGRAMS |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

汪润然 10132510147@ecnu.edu.cn

吴旭东 10132510148@ecnu.edu.cn

杨捷 10132510149@ecnu.edu.cn

---- PRELIMINARIES ----

The source code of Project2 is available at https://github.com/Bladestorm94/OS15x147\_148\_149

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.

We did not rely on new struct, global of static variables here to implement

argument passing. However, we do design separate functions for the task,

they are in process.c:

/\* push 4 bytes of data on top of stack at \*p\_stack, adding safety check

to ensure the push does not overflow stack page \*/

static bool push\_4byte (char\*\* p\_stack, void\* val, void\*\* esp);

/\* separates the program file name from command line \*/

static void get\_prog\_file\_name (const char\* cmd\_line, char\* prog\_file\_name);

/\* parse cmd\_line, separate program file name and following arguments

and push to stack exactly as illustrated in pintos document 3.5.1. \*/

static bool argument\_passing (const char \*cmd\_line, void \*\*esp);

---- 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?

To implement argument parsing, we first scan in reverse order (from right

to left) to find each argument within the input command line using pointer

char \*curr, and for each argument encountered, we push it on top of user stack

beginning from PHYS\_BASE - 1; after successfully pushing all the arguments, we

insert word-alignment segement for better accessing speed; after that, we scan

the stack for the address of each argument, and push it on stack. The elements

of argv[] is ensured to be in right order by push the arguments into stack in

reverse order.

We also implement method to avoid overflowing the stack page. Firstly, when we

push any argument to the the stack, we ensure that the argument length is not

longer than available space; secondly, we add a utility function push\_4byte to

push argv addresses, argv, argc and fake return address to the top of user

stack, and by performing boundary check to make sure each push does not

overflow the stack page.

---- RATIONALE ----

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

strtok() typically uses a static pointer to store the states (the position in

the string). The static pointer is subjected to potential race conditions and

is thus not thread-safe.

strtok\_r() takes a third arguments to determine the place within the string

to go on searching tokens. The space to store the states is offered by the

caller, and thus works in a multi-threaded environment in Pintos.

>> 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 first advantage of Unix approach is that it is much safer and simpler to

use shell-based parsing operations. This way shell could help check any

unsafe command line before they arrive at kernel directly, and thus reduce

the complexity of kernel operations.

In Unix-like systems, the separation between executable name and arguments

enables more complex semantics such as redirection, pipelining in shell

operations. With Pintos' approach, we would have to add a lot of code in

kernel to achieve similar goals, and this violates the laying design principle

commonly used in computer system.

Moreover, combined with PATH environment variable, Unix-like shell gives more

flexibility in looking for executable files. Revolving pathname and looking

for files tend to be expensive, and it is best left to external programs such

as shell to finish the task as Unix-like systems did. With Pintos' approach,

the kernel would have to undertake the task to look for a file during the

initialization of a process.

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.

struct thread

{

/\* ... \*/

struct thread \*parent\_thread; /\* Record parent thread \*/

bool is\_kernel; /\* True if current thread is kernel

thread. It's used for exit message \*/

#ifdef USERPROG

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

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

struct file\_info \*array\_files[128]; /\* Array of open files \*/

struct file \*executable; /\* Record current process's executable\*/

struct list child\_list; /\* Record thread's children \*/

struct process\_info \*process\_info; /\* Process metadata \*/

#endif

/\* ... \*/

};

/\* Structure to record relevant file information \*/

struct file\_info

{

unsigned pos; /\* Position within file \*/

struct file\* p\_file; /\* Pointer to actual file structure \*/

};

/\* Global lock on function call to filesys.h and file.h \*/

struct lock glb\_lock\_filesys;

/\* Metadata for process, which could be retrieved by parent process even

after the process exits. \*/

struct process\_info

{

struct semaphore sema\_load; /\* Sema to ensure load order between child process

and parent process \*/

bool child\_load\_success; /\* Indicate success of loading

executable file for child process\*/

struct semaphore sema\_wait; /\* Sema to ensure wait order between child process

and parent process \*/

bool already\_waited; /\* Whether the process has already been

waited by its parent \*/

bool parent\_alive; /\* Whether the parent process is

alive\*/

bool is\_alive; /\* Whether the process is alive \*/

int exit\_status; /\* Record exit status \*/

int pid; /\* Record the pid \*/

struct list\_elem elem; /\* Element in child\_list of its parent

thread \*/

};

>> 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, each process maintains its own list of open files in

an array of length 128. While the first 2 elements are intentionally left

NULL for STDIN\_FILENO and STDOUT\_FILENO, each subsequent array index is used

as a file descriptor corresponding to the open file stored in this array

element.

When opening a file we search in the array to find a vacancy, store the

pointer to file\_info structure there, and return the index as file descriptor.

And when closing a file we can go directly according to file descriptor

(index in array) to find the file\_info structure and close it. This way,

each process has an independent set of file descriptors, and the file

descriptors are not inherited by child processes. In each process, current

opened files has a unique id to distinguish from each other, which achieves

the requirements of the project.

In our implementation, file descriptors are unique just within a single

process. Different processes may have the same file descriptors.

---- ALGORITHMS ----

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

>> kernel.

Pintos uses virtual address for accessing of physical memory. In our design,

we take advantage of this fact and implement checking the validation of

user level memory before the system call actually performs execution. Given

any read or write file system call, first we check if the file descriptor is

valid, in the range of 2 to 127 or STDIN\_FILENO/STDOUT\_FILENO for

read()/write() respectively. For read(), if the file descriptor is

STDIN\_FILENO, we call input\_getc() to read from stdin; while for write(),

we call putbuf() when the STDOUT\_FILENO is encountered.

For the next step, we check whether the address provided by user is valid.

For every page in the range of memory that can be reached from user supplied

address (e.g. from buffer to buffer + size in read() and write()), we first

verify a user virtual address is below PHYS\_BASE, and then we verify that the

particular page is mapped. In doing so, we ensure the system calls all the

conditions required to perform the data transfer tasks. If there is any other

situation that will cause unexpected termination of the process, the page\_fault

handler will take in position.

Finally we deal with the typical case where actual file I/O is requested. The

list of open files recorded in thread structure is queried to find the file

structure according to file descriptor. Then this file structure is used to

call file system methods file\_read\_at() or file\_write\_at() to finish the task.

Of course, all these file system calls are protected by a global lock to avoid

race conditions.

>> 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?

In our current implementation approach, if a system call causes a full page of

data to be copied, the greatest possible number of inspection could be 2, and

the least number could be 1(if we are lucky enought), depending on whether the

data lies on two different pages. We use pagedir\_get\_page check each page once.

So when the data spans two pages, we will check twice, and if the data is

within one page, we only check once.

When copying 2 bytes of data, the situation in is similar. The number of

inspections depends largely on how many pages the data actually spans. But

ofcourse, the probability of checking two pages in this case is far lower than

the above case.

A different approach is not to check the address for read and write, but let

the operations continue until there is a pagefault occuring. Then we can

implement handler to deal with the user pagefault situation properly. In this

way, if the address is valid, then 0 inspection is needed. For the reading

of 2 bytes of data, which would happen much more frequently, significant amount

of time will be saved. And if an invalid span is attempted, the system would

write data to the valid address until the invalid address appears. It is clear

that reading and writing data are much faster than calls to the time-consuming

pagedir\_get\_page for inspections.

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

>> and how it interacts with process termination.

To implement "wait" system call, we add struct process\_info to record the

metadata of the process, such as a process's pid, whether the process is

already waited by its parent process, whether a process's parent is alive,

whether the process is alive, and the exit status of the process.

Whenever a "wait" system call is called, the function operates as explained

below:

First, current thread would search among his child list to find if there

is one process of his child matches the given pid. If there is one, the child

process's metadata is retrieved.

Then we will examine if the parent process has already waited for the child,

by checking the already\_waited member. If the parent has already waited

once for the child, then it immediately returns -1 as required by the

assignment. If the process has not been waited for this child before, then

we set the already\_waited information to true, preventing further waits in

the future.

Next, before actually waiting for the child, we first check whether the child

is still alive by checking is\_alive in process metadata, as it is perfectly

normal for a wait system call to be executed when the child is already dead.

This is important since an already dead child would not have the chance to

wake up its parent process, which would lead to the parent to wait forever.

Note that even though the child may already be dead, its child\_info is not

freed until both of the parent and child is dead, so we are still able to

get the dead child's exit\_status at this time. If the child is not already

dead, then this is the time to put the parent to wait.

We implement this by sema\_down the semaphore sema\_wait child process's

metadata process\_info. By doing so, we ensured that each sema\_up of the

children process is paired with only its parent's sema\_down, or without

its parent's sema\_down in case the parent not waiting for this child

process.

>> 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.

There are two shields in our program making sure that the system call

successfully deal with fail. The first one is the function checkvaddr

as mentioned in above questions. To avoid obscuring the primary function

of code by error handling, we ensured a proper work environment for each

of the calls before it actually starts to work. The error handling function is

summarized together into one function to eliminate the possible causes that

may induce fail. In this layer of shield, we focus on prevention before the

fail actually occurs, rather than to handle all the errors after it actually

happens. If we detect any insufficiency for the process to execute

successfully, we will not start execution but kill the process with

thread\_exit. Another shield is the page\_fault handler when a fail indeed

occurs. This will rarely happen in our design, but if it indeed happens, it

will also follow the thread\_exit routine.

To ensure that all the temporarily allocated resources would be freed, we

implement all the free functions in process\_exit, which is the only way for

a termination of a user process. To prevent memory leakage, we search through

each kind of sources a process may be related with, including its opened files,

dynamic allocated structs for wait, and the page directories. To free all these

different kinds of sources, we also implement functions like free\_info, to

check whether the source is actually needed to be freed. In this method, we

could ensure that every source used before would be searched if it is ever

needed when the process exits.

For instance, whenever a memory accessing error happens in system call read,

(the error could either be detected by the checkvaddr address we imposed or a

page fault), the termination would go through thread\_exit and then process\_exit

for an user process. In either way of the above situations, we would first free

all the terminated process's file arrays to ensure that all the file it opened

is now freed. Second, we would free the executing file of the process, and make

sure the file is now allowed to be written. Next, we will search through the

structures we allocated for wait function. As we require both child and the

parent are dead before freeing their resources, the termination of process

does not necessarily induce the free of this struct. Neverthless,

we examined if the struct satisfies the condition to free, to prevent memory

leak. At the end of exit, we free the page directory.

---- 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"?

Basically we add two member variables to struct thread to ensure this.

One is a bool variable child\_load\_success indicating the load()

success/failure. child\_load\_success is updated every time start\_process()

calls load(), and the parent process calling exec() can read it to get the

information. The other is semaphore sema\_load initialized with value 0, and

by calling sema\_down() in the parent process before reading the

bool variable and calling sema\_up() in the child process after load()

completes, we can ensure the parent process returns only after the child

process has finished load().

>> 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?

In our implementation, for every process, we create a struct process\_info to

record the metadata information of the process, so even when the current

process exits, its parent process is still able to retrieve its exit\_status,

pid, etc. Each thread has a pointer pointing to its metadata, and has a

member to record the struct representing its parent thread.

Let's denote parent process for P and child process for C for simplicity.

For synchronization, we use a sema\_wait semaphore, a member of C's metadata

"process\_info", to ensure the synchronization. When P calls wait syscall, it

examines its child\_list for the one that matches the pid provided, then it

examines whether the process has already been waited (indicated by

already\_wait, a member of C's metadata), or C has already terminated (indicated

by is\_alive, a member of C's metadata struct). If former case is true, the

wait function returns -1 immediately, and for the latter case, the wait

function returns C 's exit status (indicated by exit\_status, a member of C's

metadata). If neither case is true, current process has to wait for C, so it

sema\_down the sema\_wait semaphore, a member of C's metadata, and wait for its

child to wake it up. When C finishes executing and just before exit, it

sema\_up the sema\_wait semaphore in its metadata struct, no matter P actually

waits for it or not, because the C can only be waited by its unique parent

process.

The wait semaphore cannot be a member of P, because if this is the case, the

semaphore is shared between all its children processes. Since there is no way

for the child process at the time of exit to know if its parent is going to

wait for it or not (even P does not know, because you can't predit the

future), it may falsely sema\_up or not sema\_up. For example, if C sema\_up and

P doesn't actually wait for it, the sema-value is falsely increased by one,

causing P's next wait(sema\_down) useless; on the other hand, if C does not

sema\_up and C actually waits for it some time in the future, then P will be

blocked and perhaps never wake up again.

The free of resources is tricky too. The process C's metadata struct should

not be freed, because it should be preserved for inspection by its parent

process(P). Then we have to ask its parent to free its resources when P exits,

but by then, some of P's children processes may be alive and the metadata can

not be freed. This causes a dilemma. In order to avoid memory leakage, our

solution is as follows. Record the status of P and C in C's metadata

(parent\_alive and is\_alive) respectively. When P exits, it examines its

child\_list, and free all its dead children processes, and inform all the

other living children that their parent is not alive; then P examines whether

its parent is alive, if not, it frees its own metadata, because metadata is

exclusively accessed by its parent, and when parent process exits, there is no

need to preserve the data. In this way, a process's metadata struct is freed

either by itself or its parent, depending who exits first.

Particularly, if P calls wait before C exits, then P will sema\_down semaphore

sema\_wait, and goes to sleep; when C finishes executing, C sema\_up sema\_wait

and do not free its metadata, because its parent is still alive (parent\_alive

== true); then P wakes up, it continues executing, and at the time of exit,

it frees C's metadata, because by then C is a dead child process.

If P calls wait after C exits, then before C exits, it will sema\_up and do

not free its metadata; the P then calls wait and sema\_down, since the

semaphore is sema\_up by C once, so sema\_down does not put P to sleep, so P

will continue and at the time of exit, free C's metadata. Note in this case

and above case, whether or not to free P's metadata is depending on the

status of P's parent.

If P terminates without waiting, before C exits, then when P exits, it

informs C that it is dead; when C exits, it sema\_up(though no use here), and

free its own metadata, because P is dead.

If P terminates without waiting, after C exits, then when C exits, it

sema\_up, and when P exits, it frees C's metadata, as expected.

---- RATIONALE ----

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

>> kernel in the way that you did?

To access user memory from the kernel, we write a function checkvaddr. First

check whether the virtual address is larger than PHYS\_BASE, if it is, the

checkvaddr function will return false. If the virtual address is below

PHYS\_BASE, we continue to use the pagedir\_get\_page to check if the virtual

address has ever been mapped. If the system call requires a bulk of memory

like the case in read and write, we check every page's mapping.

We choose this way for its unity and simplicity. While it is true that using

page\_fault may result in faster speed, our function in checking virtual

address is summarized all in a function. Whenever there is a need to use the

virtual address, we could use this function.

Another advantage of this implementation is to prevent fault from actually

happening. In each system call, we check the virtual address provided by the

user, if the address is not valid, the process would be terminated rather

than allowing it to execute for a while. This may reduce the risk of

potential resource waste.

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

>> for file descriptors?

In our implementation, each process maintains its own list of open files in

an array of length 128. While the first 2 elements are intentionally left

NULL for STDIN\_FILENO and STDOUT\_FILENO, each subsequent array index is used

as a file descriptor corresponding to the open file stored in this array

element. When opening a file we search in the array to find a vacancy, store

the pointer to file\_info structure there, and return the index as file

descriptor. And when closing a file we can go directly according to file

descriptor (index in array) to find the file\_info structure and close it.

The main advantages of our design for file descriptors lie in the overall

speed performance. Our implementation gives O(n) time complexity for

allocating file descriptors when opening files, while O(1) time complexity

for all subsequent query operations in any syscall involving file descriptors.

The disadvantages of our design could be that the number of open files

allowed for each process is limited, or that when the number of open files is

small, it wastes memory space. We have considered these short-comings and

understood we can solve them by using dynamic structures such as list. It is

our choice to favor speed that we choose the design with array and file

descriptor as index in the 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?

We stick to the default implementation in mapping tid\_t to pid\_t.

If tid\_t is not mapped with pid\_t, that will support a process running

multiple threads, which is a feature not supported by Pinots. So we did not

change this part.

SURVEY QUESTIONS

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

None.