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

| CS 318 |

| PROJECT 2: USER PROGRAMS |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

---- PRELIMINARIES ----

>> <https://www.cs.jhu.edu/~huang/cs318/fall21/project/project2.html>

>><https://www.cs.jhu.edu/~huang/cs318/fall20/project/pintos_3.html#SEC25>

>> <https://www.youtube.com/watch?v=OE79vNZp1KI> - Intro to OS - Project-2 Discussion

>> <https://www.youtube.com/watch?v=RbsE0EQ9_dY> - EE415: Intro. to Operating System | [Week03] Pintos Project2-1 Background

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.

---- 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?

* In the **process\_execute** function, we tokenized the ***file\_name*** using ***strtok\_r*** to get command and arguments.
* Then, we call **thread\_create**, passing the ***file\_name*** as an argument and the name of that command.
* **start\_process** will then begin to run. We use ***strtok\_r*** in that function to split the parameters and add them to an array.
* The argument stack will then be created by iteratively looping through the argument array mentioned earlier in reverse order.

---- RATIONALE ----

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

* Because **strtok\_r()** is thread-safe and **strtok()** is not. Multiple threads may initialize **strtok()** in a multithreaded environment, which could result in unexpected behavior. Also, **strtok\_r()** uses an additional parameter save\_ptr.

>> 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.

* In contrast to the kernel, argument validation may be done on the shell itself. which may be more secure.
* Advanced pre-processing can be done on the shell itself.
* Before passing the parameters to the kernel, the Unix approach allows them to be checked for bad arguments. This prevents kernel failure.
* The Unix approach is time saving.

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**

**#ifdef USERPROG**

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

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

**struct list list\_of\_pcbs;**

**struct control\_block\_for\_process\* thread\_pcbs;**

**struct list list\_of\_files;**

**#endif**

**In process.h**

* Structure for keeping the file descriptor per process

//The description of files are kept in this structure

struct description\_of\_files

{

int ID;

struct list\_elem elem;

struct file\* file;

};

* Structure for keeping information about process\_control\_block. This is important while waiting for the child process.

//contol\_block of the processes are kept here

struct control\_block\_for\_process

{

proID\_t proID;

int32\_t exitcode;

struct semaphore sema\_init;

struct semaphore sema\_wait;

struct thread\* t\_parent;

const char\* command\_line;

struct list\_elem elem;

bool thread\_waiting;

bool thread\_exited;

};

**In syscall.c**

struct lock file\_sys\_lock; //Lock used for file\_descriptor

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

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

>> single process?

* The open files which are associated with a file descriptor (file\_desc) are only unique to a single process. The process struct contains file\_descriptor which is associated with an open file. Thus, file\_descriptor is unique only within the process.

---- ALGORITHMS ----

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

>> kernel.

**In Reading**

1. Check whether the given pointers are valid. If they are not, exit with code -1.
2. Acquire the lock (file system lock).
3. If fd (file\_descriptor) equals 0, then release the lock and retrieve inputs from standard input.
4. If fd equals 1 or 2, release the lock and return -1.
5. Loop through the file\_descriptor list until the relevant file to fd is found.
6. If such a file exists, then release lock, call file\_read function and return the output bytes.
7. If such a file does not exist, release lock and return -1.

**In Writing**

1. Check whether the given pointers are valid. If they are not, exit with code -1.
2. If fd (file\_descriptor) equals 1, use putbuf to print the contents in the buffer to the console, return -size.
3. Then, loop through the file\_descriptor list to find the relevant file\_descriptor of fd.
4. If such file\_descriptor is not found, return -1.
5. If such found, then acquire the lock (file system lock).
6. Then, use file\_write to write contents of the buffer to the file.
7. Release the lock and return the number of bytes written to the file.

>> 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?

* Least = 1 | Greatest = 2
* For 2 bytes,
  + Least = 1 | Most = 2
  + We can check if the address is less than PHYS\_BASE and not NULL

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

>> and how it interacts with process termination.

When wait is called,

1. Get the pcb\_list associated with the current thread.
2. Then, iterate over pcb\_list and check whether a child process exists such that process\_pid == child\_tid.
3. If such a child process doesn't exist OR that child process is already waiting OR already exited, return -1.
4. Else, the child process's waiting is set to true and the current process will wait until the child process exits.
5. Once the child process is exited, it will be removed from the pcb\_list and exit\_code will be returned.

>> 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.

* In this implementation, we check the pointers before using system calls. To check these pointers while gaining access to user program memory, we created the method **isuser**. If an invalid pointer is found, the process will terminate before making any system calls. Also return an error code. When such an error is found, we use **thread\_exit**, which calls **process\_exit()** in the **process.c** file and frees the resources allocated.

---- 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"?

* By using **semaphores**. We use **sema\_down** when entering the new executable thread. The parent wait until the child process is created. If the thread get loaded successfully it will return the thread id of the child. If for any reason child process does not get loaded properly parent process can know it using the load status field. After successful creation, the parent waits till the process executes and its lock gets released.

>> 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**
  + If another process is already waiting on C, process\_wait will return -1 and P will not wait on C.
  + Otherwise, P will wait on C. This prevents race conditions.
* **P calls wait(C) after C exits**
  + Since C is already exited, process\_wait will return -1 and P will not wait on C. This prevents race conditions.
* **P terminates without waiting, before C exits**
  + If P terminates without waiting before C exits, then the parent\_thread of process C will be equal to null.
  + This helps to avoid race conditions. And also process\_exit handles freeing resources.
* **P terminates without waiting, after C exits**
  + Nothing special would happen.

---- RATIONALE ----

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

>> kernel in the way that you did?

* First approach was easier to understand and implement than the second approach. Even though it was quicker it was not easy to implement.

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

>> for file descriptors?

* **Advantages**
  + **file\_desc** (file\_descriptors) are unique to each open file per process. So this helps in
  + avoiding race conditions.
* **Disadvantages**
  + This way, a single process can only open a single file. So it is unable to open a lot of files.
  + This is a disadvantage in a large process which requires opening multiple files.

>> 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

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?