+-------------------------+
| OS |
| PROJECT 2: USER PROGRAMS |
| DESIGN DOCUMENT |
+-------------------------+

---- GROUP ----

Tobias Orth <tobias.orth@gmx.net>
Mirjam Neu-Weigand <mirjam.neuweigand@gmail.com>

---- 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.

None.

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:
bool is_user_thread - states if thread is a user thread
file * executable - file link to the threads code on disk
thread * parent - parent of the thread
list children - child list
list file_descriptors - list of open files
uint fd_next_id - next fd id

struct child - child element saved in the children list, including all information about the child process

struct file_descriptor_elem - file descriptor element for the fd list, includes all information

necessary for file handling

---- 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 implemented argument parsing by first counting the number of arguments.
Then we allocate an array of appropriate size where we put pointers to
the parsed strings into it. This ensures that the first value in the array
equals the first element of argv[]. After that, we try to load the file
which is given by argv[0]. If this is successful, we put the arguments
in reversed order on the stack. Then we push the required sentinels on the
stack. Since we stored the addresses of the arguments when we put them on
the stack, we can now easily put the addresses in reversed order on the stack.
Finally the remaining values and addresses are pushed on the stack and
we are done.

If an overflow occurs, a page fault exits the current thread. We limit the
space accessible with our functions.

---- RATIONALE ----

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

The reason is that strtok() does not support multi-threading, whereas
strtok_r() supports multithreading by storing information about the next
token in the third argument (save_ptr). This makes it possible to reenter
the string several times.

>> 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) It is easier to use different shell interfaces.
2) It is an additional protection of the kernel if the shell interprets
the commands and invokes system calls. If the user has direct access
to the kernel there could possibly be a chance to do some malicious
manipulations.

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.

Added to syscall.c:

static struct file* syscall_get_file(int file_descriptor);

A filesystem_lock is used to protect the filesystem against unsynchronized
access.
static struct lock filesystem_lock; /* mutex semaphore for filesystem */

/* Members for implementing exit() */
struct thread *cur_thread = thread_current(); /* get current thread */
struct child *list_element = process_get_child(cur_thread->tid); /* get child element of current thread */
struct thread *parent_thread = list_element->parent; /* get parent thread */

/* Members for implementing open() */
struct file *file = filesys_open(file_name); /* open file */
struct list* file_descriptors = &(thread_current()->file_descriptors); /* fetch file descriptor list */
struct file_descriptor_elem *file_descriptor =
(struct file_descriptor_elem *) malloc(sizeof(struct file_descriptor_elem)); /* create new file descriptor for file file */

Added to process.c:

struct thread* process_get_thread(tid_t tid);
struct child* process_get_child(tid_t child_tid);

*/ Member for implementing process_wait() */
struct child* child = process_get_child(child_tid); /* find thread with id child_tid */

*/ Member for implementing process_get_child() */
struct list* children = &(thread_current()->children); /* list of child threads */

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

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

Each process keeps track of the files he has opened by managing a file-descriptor
list. This is a datastructure which is unique within a single process.

---- ALGORITHMS ----

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

To implement reading and writing user data correctly, we chose to implement
the first solution how it was described in the assignment. In the method
syscall_get_kernel_address() we check the validity of a pointer that we got
from a user process. We check whether the pointer points to unmapped or non-user
memory or is a nullpointer. Finally we used the preimplemented method pagedir_get_page()
to lookup the user virtual address.

>> 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?

At least 2 calls - first and last address of the page.
Maximum: for every word.

>> B5: Briefly describe your implementation of the "wait" system call
>> and how it interacts with process termination.

We implemented the wait system call bei calling process_wait() in process.c.
We used a semaphore to indicate that a process is waiting on the child process.
It is increased when the child process exits, whereas the waiting parent process
waits on the semaphore until being woken up and then decreasing it. By that we
ensure proper synchronization between parent and child processes.

>> 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.

Let us consider the "write" system call as an example. We first implemented
a method "handle_write()" which calls the method syscall_get_kernel_address()
(via syscall_get_argument()), which checks the validity of the user address
and will then return the three also checked argument pointers/values. If any
of those fail, we exit the program by calling thread_exit(), which includes
the printout you wished, and releasing system locks and resources.

Next, we try to acquire the lock that protects the file system before we
allocate memory. If the file descriptor is the console, we put the bytes that
should be written into a buffer. If it should be written to a file, we look up
the matching file descriptor. If no matching file descriptor could be found,
cancel the write call.

---- 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"?

We implemented a sempaphore waiting. Until the thread started has not
finished its loading step, the caller is not able to continue. If
the load is not succesfull, the caller is informed.

>> 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?

This is ensured by synchronizing both threads with a semaphore. P waits
on the semaphore until C exits. This is the moment where P is being
signaled to wake up again. If C has already exited, the semaphore is
free and P can call down() and go on. The resources of C are freed

in the method process_exit(), which is called by C in every case.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?

We chose to first check the validity of a user pointer before
dereferencing it because this makes it easier to handle the release of
locks and allocated memory afterwards. As already pointed out in the
assignment, you need not to worry about how to return from a page fault.
So we chose the easier and a bit slower technique.

>> B10: What advantages or disadvantages can you see to your design
>> for file descriptors?

We designed file descriptors as property of a thread. It uses this struct
to manage all open files. An advantage of this design is that the kernel
could also access the file descriptors list, e.g. if the process has to
be killed due to an error. The kernel could then free those resources.

>> 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 did not 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?