Project User Programs Report

Group 52

Name	Autograder Login	Email
Matthew Thomas	student290	calmechestudent@berkeley.edu
Hyeong Yoon	student397	simony1124@berkeley.edu
Evelyn Vo	student128	ettvo@berkeley.edu
Ananya Bahugudumbi	student310	ananyab@berkeley.edu

Changes

Argument Passing

Our first plan for argument passing only involved checking the validity of arguments in the start_process function. However, we decided to do this in the syscall handler; exec was just one of several syscalls whose arguments we had to check, so we decided that checking arguments before each syscall would be a more comprehensive approach. We included functions to make sure that the arguments that were passed to syscalls were valid. Namely, our check_valid_ptr function made sure that the buffers that certain syscalls received were in valid memory.

Also, we initially tried to use a lot of inline assembly in the function process_execute. We though that we would have to directly modify the registers before creating a new process. We realized this was incorrect, and we changed our implementation to just use functions such as memcpy and memset to push arguments onto the stack. This approach wound up being much easier.

Process Control Syscalls

Initially, we had the idea of creating a hash map of child processes that would live in the parent process' PCB. However, it was suggested to us by our TA that this would be a lot of unnecessary work. Instead, we used a Pintos List to keep track of the data that needed to be shared between a parent process and a child process. We added a “children” Pintos List to the PCB. We then malloced and initialized a shared_data list element in start_process, and added this element to the children list.

In the wait syscall, we used the pid field of each shared_data element in the children list to determine if a pid was that of a child process (which would also be used to determine if a pid corresponded to any process at all). We also used the ref_count field of shared_data to determine whether or not the process had exited: if the ref_count was less than or equal to 1, that meant only the parent was referencing the shared_data element (and thus the child had exited). We deleted the temporary semaphore in process_wait and used our own semaphore which we put in shared_data as well.

For the exec syscall, we used a global semaphore to make sure that the process calling exec did not continue until the child process had finished loading. This was a requirement of exec that we overlooked in our initial plans. This satisfied the synchronization requirement for exec, and after adding this a lot of our exec tests started passing.

File Operation Syscalls

Initially, we planned to have all the code handling each syscall inside of the respective if-else in syscall_handler. We moved all the code for each into helper functions to make the code cleaner. We also planned to have the FD table structs inside another file, but found that it caused a number of tests to fail (ex: read-normal looks the same but seems to have additional empty space characters) and the autograder to fail to build due to linkage differences. We also created a lot of helper functions for the FD table (i.e. find, add, remove) that were not specified in the design doc.

We removed the pid_t pid member from our FD table struct because of Shreya’s feedback that it was unnecessary. We had access to the process name and other information when running the syscall handler with thread_current(). We also were going to implement the global file description table before we learned during design review that it was already implemented for us. We also got rid of the idea of using the FD table to store the reference count. Instead, we moved that to the shared data struct.

We made some of the function implementations more complex as well. For example, the read syscall helper now has code to read from the keyboard input. Additionally we added code to the write syscall that checks for write permissions before continuing.

Floating Point Operations

Initially we only created a buffer for the FPU state in the interrupt frame. We added a similar buffer for the switch threads frame. This way we could save the state during context switches and thread switches. In our initial plan, we did not change the offsets that were used in switch_threads, which caused quite a few errors. We changed the offsets to match the new switch_threads frame.

We also forgot to initialize the FPU at the beginning of Pintos by calling fninit. We added this to our start function in start.S.

Additionally, we initially did not take into account saving the FPU for the first processes/threads of Pinots - these processes dont utilize the interrupt/switch_threads logic that we had implemented, so they were causing certain tests to fail. We changed start_process and thread_create so that they saved the state of the FPU for the calling process/thread. Specifically, we created a 108 byte character array in both functions. We then used fnsave to save the state of the FPU into this array. Next, we used fninit to initialize the FPU state, and fnsave -ed it into the interrupt frame (in start_process) and the switch_threads_frame (in thread_create). Lastly, we frstor -ed the FPU state from the character array.

Reflection

Everyone worked on the design document and contributed to planning.

Simon completely implemented argpassing. He figured out the logic for tokenizing the input string and adding those tokens to the stack. He also figured out the math for adding the necessary padding to make sure the data on the stack was 16-byte aligned. Simon also worked on all of the process syscalls and debugged the file syscalls. He came up with the idea of using semaphores implement synchronization in the exe syscall and in the file syscalls.

Evelyn figured out the technical and administrative sides of our project. She figured out how to get VSCode live to work, which was a crucial part of our collaboration. She also debugged quite a bit of issues with Git that we ran into. She helped divide the tasks into smaller chunks that we could individually execute. Evelyn also implemented and debugged file syscalls. She understood the most about how to use the filesystem API and wrote the logic behind manipulating files and file descriptors. She helped Ananya create tests.

Matthew worked on the exec and wait syscalls. He contributed to several drafts of helper functions that would allow a parent process to access data about a child process. He also figured out how a parent process could get the exit code of a child process. Lastly, Matthew implemented the FPU logic. He modified the definitions of the interrupt_frame, switch_threads_frame, intr_entry/exit and switch_threads so that the FPU state could be saved/restored. One area that Matthew could improve on is contributing to the planning phase of the project. A lot of the implementations he wrote were derived in the moment, which made the development process slower than it would have been had he planned ahead.

Ananya helped determine how to implement the FPU section. Through her research and time in office hours, she realized that the FPU would need to be saved and initialized during process/thread creation - not just during context switching and thread switching. Ananya also helped with the logic and debugging for file syscalls. One area that Ananya could improve on is understanding the greater scope of the project and more of the areas than the ones that she contributed to. She had trouble switching between different sections of the project because of less understanding on other sections. She worked on creating tests.