Lab 2 Design Doc: Multiprocessing

Overview

The goal of this lab is to add process management to osv, so that it could support multiprocessing by allowing sharing resources between processes while respecting process isolation.

Major Parts

System calls: fork, wait and exit

fork copies the parent process shares the open files and returns twice: once in the parent process (child's PID) and once in the child process (0).

wait makes fork code deterministic by allowing a parent process to be blocked until a child process exits. This system call can wait for a specific child process or until any child process exits.

exit terminates the running process, cleans up kernel resources from the process, and sets the exit status, which should be accessible to the parent process in wait.

Synchronization between parent and child process

We need to protect the integrity and functionality of synchronization between parent and child processes. For instance, we cannot clean up resources used by the child process when the parent process exits. We will implement synchronization based on three cases: 1. Parent waits before child exits; 2. Parent waits after child exits; 3. Parent exits without waiting for child.

In-depth Analysis and Implementation

Bookkeeping

We will add proc_status to struct proc, this defaults to STATUS_ALIVE.

We will add a List of child_pid in struct proc, initially this is empty. When we fork, we append the process id of the children to this list. Child_pid is cleaned up when the process exits.

fork (kernel/proc.c/proc_fork)

- A new process needs to be created through kernel/proc.c:proc_init.
- The child process should have its own address space.
  - Parent must copy its memory to the child via kernel/mm/vm.c:as_copy_as.
- Handle file descriptors.
  - All the opened files must be duplicated in the new process (call fs_reopen_file).
- Create a new thread to run the process and add a new process to the process table (example in proc_spawn).
- Duplicate the current thread (parent process)'s trapframe in the new thread (child process) (assign one struct to another, results in a copy).
- Set up trapframe to return 0 in the child via kernel/trap.c:tf_set_return, while returning the child's pid in the parent.

wait

syscall.c/sys_wait

- Validate user program input.
  - Check if the parent process does not have a child with the specified pid return.
    - If pid is not in the parent's child_pid list return ERR_CHILD.
  - Validate pointer wstatus.
    - If not valid return ERR_FAULT.
  - Check ptable to see if the parent already waited on the child with the pid.
    - If pid in ptable and status is set to something other than STATUS_ALIVE, return ERR_CHILD.
- Call proc_wait(pid, wstatus).

proc.c/proc_wait

- If pid is ANY_CHILD (-1), wait for the first child process to exit. (1)
  - Wait for signal that any child has exited (conditional variable).
  - Once received signal, it will unblock the parent's process.
- If pid is a specific child process, wait for a signal for the specified child to exit. When the child is done, child_done will be true and the parent process will continue.
- If wstatus is not null, store the exit status there.
  - Search the ptable for the process struct with the specified pid, store it's proc_status in wstatus.

exit

syscall.c/sys_exit

- Call proc_exit.

proc.c/proc_exit

- ❖ Communicate exit status back to the parent process.
  - ➢ Save the exit status of the child to the proc_status field of struct proc with specified pid.
  - ➢ The parent process can then look up the corresponding exit status of the child in the ptable.
- ❖ Releases kernel process resources.
  - ➢ Cleans up the address space (as_destroy), kernel thread, process struct used by the exited process by calling proc_free, considering the three synchronization cases.
    - ■ Close all open files in open_files by calling fs_close_file.
- ❖ Send a signal to the parent process that it has ended/exited.

Synchronization between parent and child process

1. Parent waits before child exits (child process status is STATUS_ALIVE).
   - ➢ wait blocks until the child being waited on exits.
     - ■ Wait for signal that the child has exited (conditional variable). Continue execution.
     - ■ Remove child pid from child_pid list.
   - ➢ The parent will have access to the child's exit status and will clean up all the kernel resources used by the child.
2. Parent waits after child exits (child process status is not STATUS_ALIVE).
   - ➢ The parent will have access to the child's exit status and will clean up all the kernel resources used by the child.
   - ➢ As the child already exited, the parent does not need to be blocked.
3. Parent exits without waiting for child (no call to wait - children with STATUS_ALIVE).
   - ➢ Child's data structures need to be reclaimed when the child exits and parent resources are deallocated.
     - ■ The existing parent process could hand off its children to the initial process.
       - Merge the child_pid of the parent process with the child_pid of the initial process.
     - ■ Dealloc resources used by the parent process.
       - Clean up child_pid, address space (as_destroy), kernel thread, process struct used by the exited process by calling proc_free.
   - ➢ Child will be in ptable with exit status different from STATUS_ALIVE.

Risk Analysis

Unanswered Questions

- How to correctly clean up/reclaim resources?
    - Do we have the right order to achieve this?
- Conditional variables: how exactly to use them?
    - How do we know the signal we received is the correct one?

Staging of Work

First, we will implement fork. Then, exit and wait. Finally, we will add error checking.

Time Estimation

- fork (4-5 hours)
- wait (2-3 hours)
- exit (2-3 hours)
- Edge cases and error handling (3-4 hours)