Lab 2 design document

David Grant (dagr), Allen Tran (alltran)

Overview

Synchronization issues

We must first patch up any remaining synchronization issues in our file and pipe implementations that will be an issue when our O/S is running multiple concurrent processes.

File issues

The `open/close/read/write/dup/sys_pipe` syscall implementations do not need added synchronization because xk processes are single-threaded, so a thread executing a syscall will not be preempted by another thread needing to manipulate the *same* process' open file descriptor table.

We DO need a top-level global file description table so that at least the offsets can be shared between parent/child processes, and for `dup`. (This is how Linux does it.) **Done.**

Pipe issues

Pipes exist across processes and therefore concurrent access to them must be synchronized. We're using a combination of pipe-level spinlocks and condition variables to achieve this.

`fork`

fork() is the standard one from UNIX. A process calls fork() and it is entirely duplicated. The two processes now resume executing the same code, just having returned from fork(). The return value from fork() is either zero, indicating that you're the newly-spawned child process, or the process ID of the child process (and you're the parent.) The parent can then do something with the child PID. (Such as wait() on it.)

`wait/exit`

`exec`

`Exec` loads an executable program and its caller-supplied arguments into the current virtual address space and begins executing its `main()` function. The previously executing code in the

virtual address space is *replaced* with the code from the specified executable. A common use case is to call `fork` immediately followed by `exec`, but `fork`ing first is not required.

Implementation Details

Pipe synchronization

To support multiple readers and multiple writers for pipes, we need to introduce a new `spinlock` on every allocated `struct pipe` that gives a reader or a writer exclusive access to the pipe. We also introduce a condition variable `CV_PIPE_DATA_AVAILABLE` to signal blocked readers. The pipe read and write implementations are roughly as follows:

```
pipe_read()
{
  acquire(&pipe->lock);

  if (pipe not readable) {
    release(&pipe->lock);
    return -1;
  }

  while (nothing to read) {
    if (pipe->numwriters == 0) {
      // empty pipe & no writers.
      release(&pipe->lock);
      return -1;
    }
    sleep(&CV_PIPE_DATA_AVAILABLE, &pipe->lock);
  }

  // … [pipe reading here]…
  release(&pipe->lock);
  return bytes read;
}
```

```
pipe_write()
{
  acquire(&pipe->lock);

  if (pipe not writable) {
    release(&pipe->lock);
    return -1;
  }

  // … [pipe writing here] …
  wakeup(&CV_PIPE_DATA_AVAILABLE);
  release(&pipe->lock);
  return bytes written;
}
```

The condition variable is used for when a reader attempts to read an empty pipe. The reader blocks, and a writer will signal when some data has been placed in the pipe. When a pipe's write-end is closed, we also trigger a `wakeup(&CV_PIPE_DATA_AVAILABLE)`, causing any blocked readers to wake up and notice that the pipe is now not only empty, but free of writers. When this happens, the reader can abandon the read.

The lock over the entire read and write routines ensure that reads and writes happen atomically, and that, for example, the payloads of two different writers do not become interleaved.

Fork

- A new entry in the process table must be created via `allocproc`
    - If `allocproc` fails, fork fails with a -1 return code.
- New process' `parent` should point to parent proc.
- All the memory region descriptor must be duplicated in the new process entry in ptable. This entails copying verbatim the contents of `proc->mem_regions`, since it's just offset and size information.
- User memory must be duplicated via `copyuvm.`
    - If this fails, we clean up by setting the child proc status back to `UNUSED`, deallocate its kernel stack and return -1.
- Trapframe must be duplicated in the new process
    - Trapframe is allocated as part of the new kernel stack in allocproc. Copy its contents verbatim.
- All the opened files must be duplicated in the new process
    - When we duplicate an OFT entry:
        - Copy each `struct open_file` record wholesale.
        - If it's a file, increment the inode structure's refcount via `idup()`.
        - If it's a pipe, increment the appropriate pipe refcounts based on its read/write mode.
            - Pipe lock needed since the pipe is shared with the parent process/potentially others.
- Set the state of the new process to be `RUNNABLE.`
- We need to set the return value PID to 0 when in the child process, and the child PID when in the parent process.
    - Do this by overwriting `childProc->tf->rax` = 0.
- When we return to user code, we want execution to resume in the instruction *following* the usermode `fork` call.
    - (This already works correctly without any trouble.)

Open questions about fork:

- -

Exit

- User calls exit()
- User Stub traps into Kernel stub
- Kernel stub calls kernel implementation of exit(void)
- Acquire ptable lock
    - Modify open file table, state, parent fields
- Clean up this process' open file table
    - Close all files and pipes via syscall close
- Change state to zombie
- Wake up parent or init process
    - First determine if current parent is still running, else assign init to be parent
- Reschedule this thread
    - Tell scheduler to put some other ready process on processor
    - Then because we set state to zombie, this thread will not run again
    - It waits for its parent to eventually run and reap this process
- Release ptable lock

Wait

- User calls wait()
- User Stub traps into Kernel stub
- Kernel stub calls kernel implementation of wait(void)
- Acquire ptable lock
    - Search proc[] atleast once, modify kstack, state
- If this process has no children process' then return an error
- Wait for a zombie child process to be found
    - While (has not found zombie child) { sleep(chan, ptable lock); }
- Clean up zombie child's kernel stack
    - Set it to null after freeing, don't want pointer to a mem region to float around
- Change zombie child's state to unused
- Release ptable lock
- Return zombie child's pid

Exec

(See clarification in Piazza post: https://piazza.com/class/j7yfhsfulwt5wi?cid=107)

- User calls exec(file, argv)
- Syscall mechanisms happen, and kernel executes the `sys_exec` stub
- `sys_exec` stub retrieves arguments, calls kernel's `exec()` (kernel/exec.c)
- Kernel's `exec(char* path, char** argv)`:
    - Use `load_program_from_disk` to overwrite current user's virt addr space.
    - Calculate the usermode argc and argv
        - argv = [program, arg1, arg2, ...]
        - argc = length of argv
    - Copy argc and argv into the usermode stack. (??? possibly w/ copyout() ???)
    - Set %rdi and %rsi to point to argc and argv, respectively, by manipulating trapframe.
    - Call the newly loaded `main()`. (`*rip` pointer returned by `load_program_from_disk`.)
- `exec` returns -1 on error, and on success it does not return at all. (Because the usermode code that *called* exec is completely replaced by the loaded executable.)

Risk Analysis

…

Open questions

- Do we need to acquire lock for the whole duration of fork()?
- Can we do better for wait, holding references to the child nodes, so we don't have to do a linear search every time.