

System calls to implement:

`pid_t fork(void);`

Create a copy of the current process. Return 0 for the child process, and the PID of the child process for the parent.

`pid_t waitpid(pid_t pid, int *status, int options);`

Waits for the process specified by *pid*. Status stores an encoding of the exit status and the exit code. Returns the PID of the process on success, or -1 on error (it can also return 0 if the option WNOHANG is specified and process PID has not exited).


`pid_t getpid(void);`

Returns the PID of the current process.

`void _exit(int exitcode);`

Causes the current process to exit. The exit code is reported to the parent process via the `waitpid` call.

Fork

- 
- Create process structure for child process
 - Create and copy address space (and data) from parent to child
 - Attach the newly created address space to the child process structure
 - Assign PID to child process and create the parent/child relationship
 - Create thread for child process (need a safe way to pass the trapframe to the child thread).
 - Child thread needs to put the trapframe onto the stack and modify it so that it returns the current value (and executes the next instruction)
 - Call *mips_usermode* in the child to go back to userspace

Fork

- Create process structure for child process
 - Use *proc_create_runprogram(...)* to create the process structure
 - Sets up VFS and console (don't worry about these fields for now)
 - Don't forget to check for errors
 - E.g., what happens if *proc_create_runprogram* returns NULL?

Fork

- Create and copy address space
 - Child process must be identical to the parent process
 - *as_copy()* creates a new address spaces, and copies the pages from the old address space to the new one
 - Address space is not associated with the new process yet
 - Look at *curproc_setas* to figure out how to give a process an address space
 - Remember to handle any error conditions correctly (what if *as_copy* returns an error?)

Fork

- Assign PID to child process and create the parent/child relationship
 - How you do this is completely up to you.
 - PIDs should be unique (no two processes should have the same PID)
 - PIDs should be reusable.
 - When can we reuse a PID? This is partially dependent on your implementation. We will talk more about this when we look at `_exit` and `waitpid`.
 - Remember that you need to provide mutual exclusion for any global structure!

Fork

- Create thread for child process
 - Use *thread_fork()* to create a new thread
 - Need to pass trapframe to the child thread
 - Why didn't we pass the trapframe earlier when we created the process structure?
 - Can we just pass the trapframe pointer directly from the parent to the child?
 - Yes (but requires synchronization). Consider other approaches.

Fork

- Child thread needs to put the trapframe onto its stack and modify it so that it returns the current value
 - How? (think local variables)
- What must be changed in the trapframe before going back to userspace? (don't forget the program counter)
 - Why don't we need to manually modify the trapframe for the parent process?

Waitpid

- How do we use waitpid? From widefork.c

```
void
dowait(int childpid, int childnum)
{
    int rval;
    if (waitpid(childpid, &rval, 0) < 0) {
        warnx("waitpid 1");
        return;
    }
    if (WIFEXITED(rval)) {
        if ((WEXITSTATUS(rval)) == childnum) {
            putchar('a'+childnum-1);
            putchar('\n');
        }
    }
    else {
        putchar('x');
        putchar('\n');
    }
}
```

Check if process exited by
calling `_exit()`

Get the exit code

Waitpid

- Only the parent can call waitpid on its children
- If waitpid is called before the child process exits, then the parent must wait/block
 - What should it block on? Semaphore? Condition variable in a process table? Condition variable specific to the parent process? To the child process?
- If waitpid is called after the child process has exited, then the parent should immediately get the exit status and exit code.
 - Cannot lose this information! Must be saved somewhere.
- PID cleanup should not rely on waitpid
 - Parent process is not guaranteed to call waitpid when it exits
- How do we handle the case where the parent exits before its children?
 - Can we free the parent process structure before its children exit? Do the children know how many child processes there are for its parent?
- Does your PID reuse mechanism rely on an exit order?

Waitpid

- Encoding exit status and exit code
 - Exit code comes from `_exit()`
 - In this assignment, exit status should always be `__WEXITED`.
 - Look at `kern/include/kern/wait.h` for helper macros
 - Specifically `_MKWAIT_EXIT`

Waitpid

- Fun extras!
 - Implement the WNOHANG option
 - If child is still alive, return ECHILD instead of blocking
 - Add support for WAIT_ANY (-1 PID)
 - Block until the first child exits. Return that child's PID in the return value.

Getpid

- Returns the PID of the current process
- Simplest system call to implement. May consider implementing this first.
 - Need to implement PID assignment.
- Need to perform process assignment even without/before any fork calls.
 - The first user process might call getpid before creating any children. getpid needs to return a valid PID for this process.

`_exit`

- Causes the current process to exit
 - Don't worry about having multiple threads in a process.
 - We don't expose `thread_fork` as a system call. Therefore, user processes will only have one thread.
- Exit code is passed to the parent process