Assignment 2a Guide

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.

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 a new process structure for the child process.
- Create and copy the address space (and data) from the parent to the child.
- Attach the newly created address space to the child process structure.
- Assign a PID to the child process and create the parent/child relationship.
- Create a thread for child process. The OS needs a safe way to pass the trapframe to the child thread.
- The 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 user space.

Fork

- Create a process structure for child process.
  - You will have to add new field(s) to the process structure, e.g. pid.
  - Use *proc_create_runprogram*(…) to create the process structure.
    - It 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?
  - That said, you can see the tests we run in the assignment marking report and all of them are successful tests.

Fork

- Create and copy the address space.
  - Child process must be identical to the parent process.
  - *as_copy*() creates a new address space and copies the pages from the old address space to the new one.
  - Again, check for errors and if so, return an error code.
  - 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.
    - Do not call it. It changes *your* address space and you want to change the *child's* address space.
  - Remember to handle any error conditions correctly. E.g. what if *as_copy*() returns an error?

Fork

- Assign a PID to child process and create the parent/child relationship.
  - How you do this is completely up to you.
  - PIDs should be unique (i.e. no two processes should have the same PID) and the PID > 0.
  - PIDs do not need to be reusable (for CS350).
    - Can assume there will be a max of 64 PIDs needed and just have a global counter that gets incremented.
    - *proc_bootstrap*() would be a good place to initialize that counter.
    - Remember that you need to provide mutual exclusion for any global structure!
    - In real life, PIDs *do need* to be reusable.

Fork

- Create a thread for the child process.
  - Use *thread_fork*() to create a new thread.
  - The second parameter of *thread_fork*() is the process the thread is going to be attached to. In this case, set it to the child's process.
  - *thread_fork*() also needs a function (for the thread to run) as a parameter.
  - Consider using *enter_forked_process*() but you must change its signature to what *thread_fork*() expects i.e.
    (void *, unsigned long)
  - Need to pass a trap frame (pointer) to the child thread.
  - That will be the first parameter of *enter_forked_process*()

Fork

- The child thread needs to put the trap frame onto its stack and modify it so that it returns the correct value.
  - How? (think local variables)
  - Copy the parent's trap frame to a local variable in the child's address space, e.g.

```
enter_forked_process(void *tf_p, …) {
  struct trapframe tf_c = *(…some cast…) tf_p;
```

  - Can we just pass the trap frame pointer directly from the parent to the child?
    - Yes (but requires synchronization), i.e. parent my run (and return) before child. Consider other approaches.
    - E.g. copy the parent's trap frame to the OS heap, then copy from OS heap to child.

Fork

- What must be changed in the trap frame before going back to user space? In the child's trapframe, you must set

```
tf_c.v0 = 0;          // return code for child
tf_c.tf_a3 = 0;       // no error
tf_c.??? += 4;        // increment pc
mips_usermode ( … ) // return to usermode
```

  - Look at other examples in syscall.c to determine how to increment the pc.
  - Why don’t we need to manually modify the trap frame for the parent process?
- Note: There are some …'s and ???'s in some places in the slides. That means you need to fill in the details.

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.
- It can call it at most once for that child.
- 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.
- Must consider 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?

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
    - This macro combines the argument of _exit() and the exit status into a single integer value.

Getpid

- Returns the PID of the current process
- Simplest system call to implement.
  - 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 (if the parent is alive).
  - Can keep process structure around (in the Zombie state).
  - Can store the child exit status in a kernel data structure.

There are three cases where a process can be fully deleted.

1. If its parent has already exited.
2. If you exit and all of your children are dead, you can full delete your children.
3. If your parent has already called *waitpid*().