## 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.

int execv(const char *program, char **args);

Replaces currently executing program with a newly loaded program image. Process id remains unchanged. Path of the program is passed in as *program*. Arguments to the program (*args*) is an array of NULL terminated strings. The array is terminated by a NULL pointer. In the new user program, argv[argc] should == NULL.

# Runprogram

- execv is very similar to runprogram (kern/syscall/runprogram.c)
- Runprogram is used to load and execute the first program from the *menu*

1. Opens the program file using *vfs_open(progname, …)*

2. Creates a new address space (*as_create*), switches the process to that address space (*curproc_setas*) and then activates it (*as_activate*).

3. Using the opened program file, load the program image using *load_elf*

4. Define the user stack using *as_define_stack*

5. Call *enter_new_process* with no parameters, the stack pointer (determined by *as_define_stack*) *and* entry point for the executable (determined by load_elf)

# execv

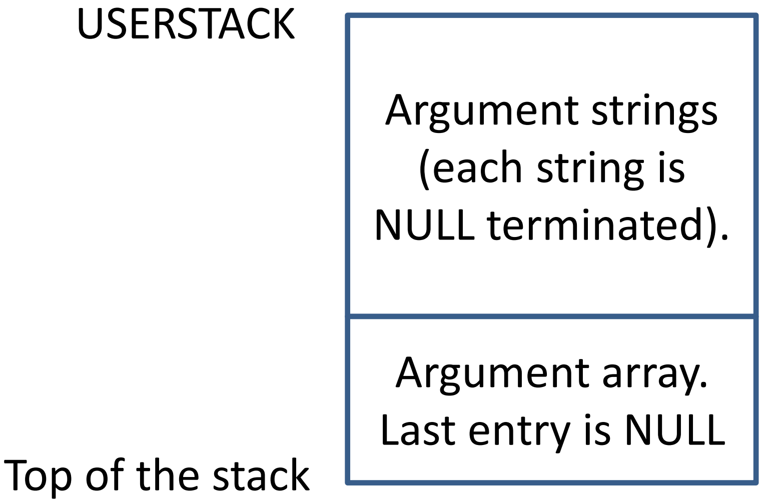
- Count the number of arguments and copy them into the kernel
- Copy the program path into the kernel
- Open the program file using vfs_open(prog_name, …)
- Create new address space, set process to the new address space, and activate it
- Using the opened program file, load the program image using *load_elf*
- Need to copy the arguments into the new address space. Consider copying the arguments (both the array and the strings) onto the user stack as part of *as_define_stack.*
- Delete old address space
- Call *enter_new_process* with address to the arguments on the stack, the stack pointer (from *as_define_stack*), and the program entry point (from *vfs_open*)

# Argument passing

- When copying from/to userspace
  - Use *copyin/copyout* for fixed size variables (integers, arrays, etc.)
  - Use *copyinstr*/*copyoutstr* when copying NULL terminated strings
- Useful defines/macros:
  - USERSTACK (base address of the stack)
  - ROUNDUP (useful for memory alignment)
- Common mistakes:
  - Remember that *strlen* does not count the NULL terminator. Make sure to include space for the NULL terminator.
  - User pointers should be of the type userptr_t
    - E.g, the interface for sys_execv should be int sys_execv(userptr_t progname, userptr_t args)
  - Make sure to pass a pointer to the top of the stack to enter_new_process.

# Alignment

- When storing items on the stack, pad each item such that they are 8-byte aligned
  - E.g., args_size = ROUNDUP(args_size, 8);
- Strings don’t have to be 4 or 8-byte aligned. However, pointers to strings need to be 4-byte aligned.



USERSTACK

Argument strings (each string is NULL terminated).

Argument array. Last entry is NULL

Top of the stack