Assignment 2b 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 (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.

Get these working before starting A2b.

They are needed to test A2b.

int execv(const char **program*, char ***args*);
Replace the currently executing program with a newly loaded program image. Process structure (e.g. pid and parent) remains unchanged. Path of the program is passed in as *program*. Program arguments, *args*, is an array of NULL terminated strings. The array is terminated by a NULL pointer, i.e. in the new user program, argv[argc] == 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 using *as_create*,
     switches the process to that address space (*curproc_setas*),
     and then mark the current TLB entries as invalid (*as_activate*).
  3. Using the opened program file, load the program image into the address space's code and data segments using *load_elf*.
  4. Creates the user stack using *as_define_stack*.
  5. Calls *enter_new_process* (currently without *program* args), 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 from user space into the kernel.
- Open the program file using *vfs_open*(*prog_name*, …).
- Create a new address space, set process to the new address space, and clear the TLB.
- Using the opened program file, load the program image using *load_elf.*

Copy from runprogram

- Copy the arguments from the user space into the new address space.
  - Consider copying the arguments (both the array and the strings) onto the user stack as part of a new modified *as_define_stack*.
- Delete the old address space (if none of the previous steps failed).
- Call *enter_new_process* with
  - the address to the arguments on the stack,
  - the stack pointer (from *as_define_stack*),
  - and the program entry point (from *vfs_open*).

Argument passing

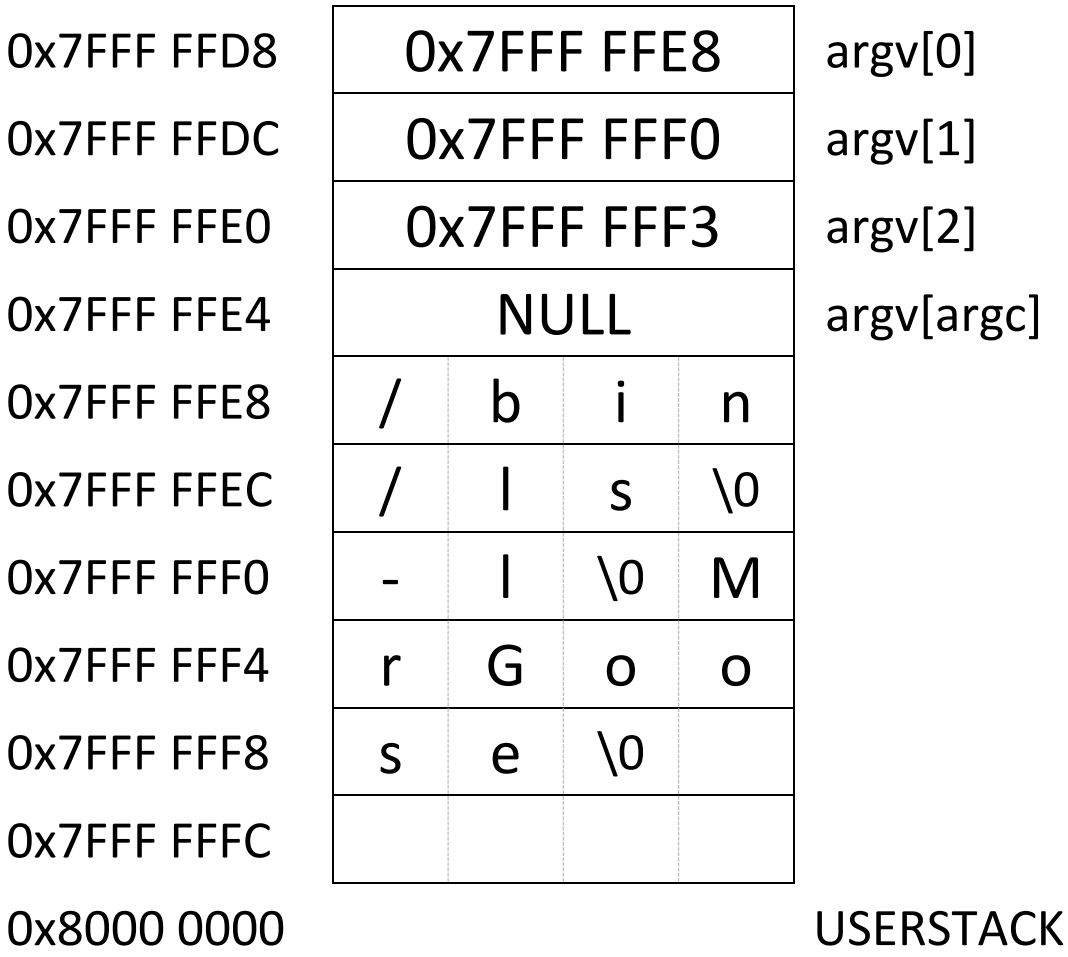
- Caution: this part trips up many students.
- When copying from/to user-space do not assume the pointer is valid.
  - Use special functions that check the validity of the pointers.
  - Use *copyin*/*copyout* for fixed size variables (integers, arrays, etc.)
  - Use *copyinstr*/*copyoutstr* when copying NULL terminated strings.
- Recall the parameters of execv(const char **program*, char ***args*)
  - First copy the program name into kernel space.
  - Next copy in each of the char pointer values (pointer to an arg) until you get a NULL pointer.
  - Then copy in each arg string.
- Don't for get to allocate space for the pointer array and the strings.
- Can assume a total length ≤ 128 bytes.

Alignment

- When storing items on the stack (i.e. copying out to the new addr space) you must worry about alignment.
  - chars: no need to worry about the alignment of chars
  - ints and pointers: must be 4-byte aligned i.e. their addresses must be divisible by 4 (as in CS241)
  - the stack pointer: must be 8-byte aligned (its address must be divisible by 8) just in case a double is pushed onto the stack next
- Useful defines/macros:
  - USERSTACK (base/starting address of the stack) in vm.h
  - ROUNDUP (useful for memory alignment) in lib.h
  - E.g., args_size = ROUNDUP(args_size, 8);
- First push on the args (i.e. the strings) onto the stack and keep track of the address of each string (e.g. 0x7FFF FFE8, 0x7FFF FFF0, 0x7FFF FFF3).
- Next put a NULL terminate array of pointers to the strings.

Alignment

e.g. /bin/ls -l MrGoose



Argument passing

- No need to allocate space for the stack (it has already been allocated). Just copy everything there.
- Note: USERSTACK is a predefined constant 0x8000 0000. You can write up to and including 0x7FFF FFFF but do not write at 0x8000 0000.
- Common mistakes:
  - Remember that *strlen* does not count the NULL terminator. Make sure to include room 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)
  - Pass a pointer to the top of the stack (e.g. 0x7FFF FFD8 in the example above) to *enter_new_process*.

Suggested Steps

1. Copy the body of *runprogram* into *execv* (i.e. use this as your starting point) and compile.
2. Copy the program name into the kernel (and *kprintf* it to the screen to ensure you've done it correctly).
3. Make the modifications to *execv* to actually load that program.

Once this is working, implement argument passing.

4. First, just count the number of args (and *kprintf* it to the screen to ensure you've done it correctly).
5. Next copy the arg strings into the kernel (and *kprintf* them to the screen to ensure you've done it correctly).
6. Finally add the args to the stack.

A2b Grading

- The assignment is worth 50 marks.
- A2a Re-Testing
  - 18 marks are for re-testing A2a functions (so get that working first if you haven't already).
- A2b Testing
  - hogparty and sty (worth 18 marks) do not involve passing args
  - argtesttest, argtest tests, and add (worth 14 marks) involve passing args
- The A2b tests use fork, waitpid and _exit so you must get those working.
- Mercy marks are available (involving code inspection) for the A2b portion of the assignment.