Sys_execv

Overview:
Use run program as a base and then add argument passing. First the arguments are checked to see if they are coming from userspace by using the copyin function from copyinout.c on each of the char pointers. This will make sure the arguments are coming from a valid user address. As the arguments are checked their size is also calculated and added along with the size of a char pointer and one extra byte for '\0' terminator to make sure ARG_MAX is not exceeded. Padding is also included in the size check. Most of the code is the exact same as runprogram's except for the copying in and copying out of arguments. Next the strings need to be copied into the kernel using the copyinstr funciton. The strings are copied into a char array (string) each being terminated by '/0'. At first an array of strings (char **) was used and created dynamically but this ran into problems on testbin/bigexec where large numbers of arguments were passed for some reason (issue contiguous memory allocation?). Once the user address is created the strings are copied out from the buffer that was created in kernel heap to the newly created user addresss space. An array of userptr_t is used to keep track of the start of each string. Once the strings are copied out the pointers or locations to the strings need to be copied out via copyout to the appropriate spot in the user stack. For each char pointer the stack pointer is decremented by the size of a char pointer and then the memory address for a string is copied to that location. Finally the process should warp to user mode with the stack pointer pointing to the start of the argv array.

To limit the number of execv calls a global semaphore with a starting value of 3 is used. The actual value of the number of execv calls that may occur concurrently with ARGS_MAX and the system memory was not taken into consideration. However, more than 3 could most likely run concurrently with the amount of memory in the configuration. This value could be calculated in the future using available memory and ARGS_MAX

Pros/cons

Right now the execv syscall is using an array of userptr_t pointers to keep track of the starting addresses of each string. This uses up a lot of memory and these addresses could always be calculated by using the size of the strings in the argument buffer created in the kernel and the stack pointer location. Currently with this large array of userptr_t the last test of testbin/bigexec fails probably due to allocating a lot of memory. I attempted this approach of calculating the addresses when needed rather than holding them in memory to try to pass the last test of testbin/bigexec but was running into various memory reference errors with limited time left to debug. I was having some issues converting the string location in the stack to a memory address as it was giving memory reference errors with this approach. So to pass test #10 of bigexec and allow for many arguments I would need to get rid of my argv array holding locations to the strings and instead just calculate them when needed.

Also, at first I was using char ** to hold all the strings in kernel but then switched to putting all the args in one large char array instead as noted above.

Runprogram argument passing

I took the same approach as sys execv when it came to argument passing. The arguments were copied into the kernel and then copied to the user stack along with its pointers. For some reason the string values in memory were randomly changed into "??" but was fixed by starting the function with interrupts turned off. I was also not able to access the passed in args after the user stack was defined so I had to copy the args in kernel before I could copy them out. After

the user stack was defined the passed in arg strings came back as “??”. The copying out is done the same way as execv except it is using the previous solution of have a char ** to hold the passed in args in the kernel. There is no limit to the number of runprogram calls but it could share the same semaphore as execv. The argument size limit is still checked though.

Changes to previous sys calls

Changed waitpid to return 0 as the exit code if a child process already exited. This seems to have fixed the issue in testbin/forktest where different exit codes were returned? (Haven’t seen the issue since this change). This issue possibly occured because the passed in exitcode was null to begin with and was never changed after waitpid was called and when the child already exited?