Lab 2 – Multiple Processes

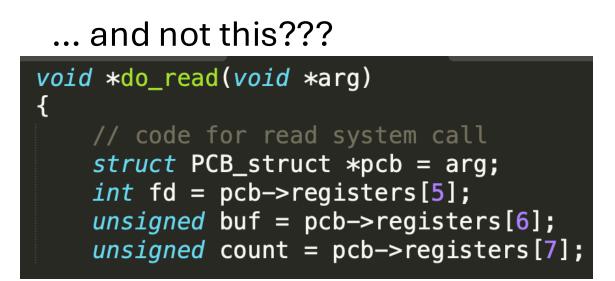
Ryan Wenger

First, a humble request

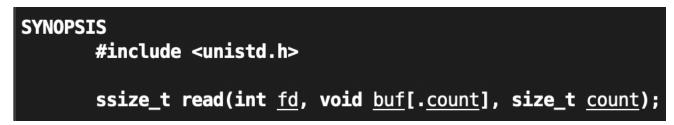
Why this

void *do_read(void *arg)

// code for read system call
struct PCB_struct *pcb = arg;
int arg1 = pcb->registers[5];
int arg2 = pcb->registers[6];
int arg3 = pcb->registers[7];



Are we in CS8?? No?? I am begging you to *please* give your variables useful names. They're right there at the top of the man page!



We want to run non-basic complex programs

- i.e, ones that #include stuff
 - Typically, these use the C standard library ("libc")
- These programs need to allocate heap memory, interact with other processes, and learn about the rest of the system
- Most C library functions aren't system calls, but make use of system call(s) to perform their function
 - e.g. malloc() calls sbrk() when it runs out of space on the FreeList, printf() calls write() to send data to the console

Many, many new system calls

- ioctl() (limited, see cookbook)
- fstat() (limited, see cookbook)
- getpagesize() get memory page size
- getpid() get caller's process ID
- getppid() get caller's parent's process
 ID
- getdtablesize() return 64
- close() return EBADF (*until next lab)

All of these are trivial in lab 2 and don't require much more than syscall_return()

Non-trivial system calls

- sbrk() "shift break": increment the process' break pointer. Called by libc malloc() to request additional heap memory from the kernel.
- execve() "execute (with vector argv & environment)": replace the current process image with a new one. i.e., stop the current program and start running a new one from the beginning.
- fork() "fork": spawn a duplicate of the current process
- wait() "wait": wait until another (child) process exits

Virtual Memory

- >1 process occupying a chunk of memory == very bad
- Divide simulator's main_memory[] into 8 slices, allowing 8 processes to run simultaneously without overlapping each other's memory
- PCB stores the starting/base index of its slice of main_memory[] and the total size of that slice
- Right before calling run_user_code(), set global vars User_Base and User_Limit to (base, size) so that the simulator knows which slice of memory to use while executing
 - When converting from user addresses/pointers in your system calls, don't forget to add PCB->base

Process memory layout

```
* A process's slice of main memory[],
* once InitUserProcess completes but
* before calling run user code():
* +----+ <- base_addr (0x0 to the program)</pre>
          [program code & static data]
  +---+ <- break pointer returned by
               load_user_program(), grows 🕓
          [unallocated. initially very large]
  +---+ <- stack pointer (grows 🟠)
          [stack contents]
    . . . .
* +----+ <- base addr + addr size - 1
*!! SEGFAULT !! (belongs to another process)
```

* A process's slice of main memory[], * after it has been running for a while * and performed several sbrk() calls: * + - - - + < - base addr (0x0 to the program) ... [program code & static data] +----+ <- return value of load user program() [dynamic data/heap] +---+ <- break pointer, grows 🕓 ... [unallocated] +----+ <- stack pointer (grows 🟠) [stack contents] * +----+ <- base_addr + addr_size - 1 * !! SEGFAULT !! (belongs to another process) */

void *sbrk(int increment);

- Shifts the **break pointer** (stored in the PCB) by *increment*
 - i.e., adjusts the size of the heap
- On success, returns the value of the break pointer before incrementing
- sbrk != malloc
 - If you wanted, you could replace the BigBuffer array from your lab0 with calls to sbrk() whenever the FreeList is empty
- Don't allow the break pointer to pass the stack pointer!

- Completely replaces the calling program with a new one.
 - zeros the calling process's memory slice and registers
 - loads the program file at pathname
 - sets the program counter to the start of the new program
 - passes in argv and begins executing (envp is ignored in KOS)
 - Process ID, parent/child relationships, and file descriptors are left untouched
- Save a copy of pathname and argv somewhere safe *before* zeroing the caller's memory (envp will be NULL)
- Other than its break pointer and registers, don't modify the calling PCB variable **at all**

Saving arguments to execve()

- pathname: pointer to the 1st character in the file path
 - (pathname+1): pointer to the 2nd character in the file path
 - → treat this the same as arguments to write, but save a copy of it somewhere. (hint: see strdup() from <string.h>)
- argv: pointer to a pointer to the 1st character of the 1st argument to the new program
 - (argv[0]): pointer to the 1st character of the 1st argument to the new program
 - (argv[0] + 1): pointer to the 2nd character of the 1st argument to the new program
 - (argv + sizeof(char*)): pointer to a pointer to the 1st character of the 2nd argument to the new program

Handling execve() arguments

<inside shell spawned by 'strace --trace=execve -ff /bin/sh' on CSIL> sh-5.2\$ Is -I ~rich/cs170 strace: Process 56562 attached [pid 56562] execve("/usr/bin/ls", ["ls", "-I", "/cs/faculty/rich/cs170"], /* 59 vars */) = 0

- to iterate over all characters in each element of argv, convert argv itself from a simulator to kernel pointer
- and convert each pointer within the resulting array from sim to kernel
- Then malloc() space (don't use the stack) to store copies of all this & make sure none of the strings go past the process's address space

pid_t fork(void);

- Completely duplicates the calling process and resumes execution of both
 - New process receives a different PID and the original becomes the new one's parent
- Allocate new PCB and new slice of main_memory[], then copy everything from the caller into the new one
 - Don't forget to assign modified base_addr, PID, and parent/children records
- kt_fork a new thread that has the child syscall_return(0)
- Parent returns child's PID

pid_t wait(int *wstatus);

- Waits for any of the caller's child processes to exit, then stores the exit code in *wstatus* (if *wstatus* isn't NULL)
 - To implement, have the waiter call P() on a semaphore
 - When any of its children exit, they will V() this semaphore, waking up the parent so that it may return

Cleaning up processes

- If a parent hasn't called wait() before its child exits, child becomes zombie
 - exit code, pid, etc. must be preserved in the kernel until the parent calls wait() or exits itself
 - wait() call needs to handle resource cleanup (free() PCB, recycle PID, etc)
- If parent exits before child, child becomes an orphan and is adopted by the init process (which in turn shuts down the entire system when it exits)
- It's **init**'s job to reap **orphaned zombie children** (call free on their PCB's and handle cleanup)

questions

• ?