SYSTEM AND DEVICE PROGRAMMING

Solved exercises on OS161 (taken from past exams)

NOTE: comments regarding corrections of answers to the exam or indications of possible/frequent errors have been deliberately left in the following text

1. Given an OS161 operating system.

   Suppose that the `syscall()` function, with a `SYS__exit` value in `callno`, calls the `my_sys_exit()` function, define the prototype of this function and write the call (with actual parameters) in `syscall()` .
   Observe the following (incomplete) snippet of `my_sys_exit()` code,

```
my_sys_exit(...                    ) {


  ... = status;
  ...

  cv_signal(...);

  ...
  ...
  ...

  thread_exit();
}
```

   tell where the `status` value can or should come from, and what variable or struct field it should be assigned to. Assume that `sys_waitpid()` has been created in the kernel, and that the latter uses a condition variable as the synchronization primitive. Complete `my_sys_exit()`, providing a brief explanation:

```
my_sys_exit(int status) {     // status is received as a parameter
  struct proc *p = curproc;   // it is used to access the process after detaching it
                              // from curthread (curproc no longer valid)
  p->p_status = status;       // save the return status for waitpid()
  proc_remthread(curthread);  // detach the process from the thread

  // signals the waitpid process
  // (to use cv_signal you need to have the related lock)
  lock_acquire(p->p_lock);
  cv_signal(p->p_cv,p->p_lock);
  lock_release(p->p_lock);

  // better NOT to call as_destroy() here (the proc_destroy() will do it)
  // the thread ends here (becomes zombie)
  thread_exit();
}
```

   The call will be like:
   `my_sys_exit((int)tf->tf_a0);`

2. Why is it necessary to create a copy of `argv` and `argc` in OS161 in order to handle arguments to the main?

   It is necessary because the arguments to the main must be in user memory, so that the user program can access them. Since the arguments to the main are originally in kernel memory, it is necessary to make a copy.

   Where should this copy be made?

   It must be created in user memory, in particular, in an accessible part of the address space: the simplest solution is the beginning (high addresses) of the (user) stack.

   Why aren't the original values nargs and args sufficient? They are passed to cmd_prog (*menu.c*: having prototype `static int cmd_prog(int nargs, char **args)`, starting from a command string.

   The main reason has already been mentioned: the original values are in kernel memory, therefore not accessible to the user process.
   one could however add that, even if the process could access them, they would be data in the stack of another (kernel) thread, the menu kernel thread, therefore to be duplicated in any case, to guarantee its consistency and accessibility from another thread.

   (For completeness, see for example the function `cmd_dispatch`, whose array (local) `args` will be received as `argv` from `cmd_prog`)

3. A) Is it possible to implement `sys_waitpid` using a lock for waiting, on which to make `lock_acquire`, while the signal operation is done by the `sys_exit` function with `lock_release` of the same lock? (motivate the answer)
   NO. A lock cannot be used for wait-signal synchronizations, because of the ownership problem (signal/release is not done by the owner of the lock): the lock is used for mutual exclusion. For wait-signal we recommend semaphores or condition variables (any lock possibly associated to the process could be used for mutual exclusion, not to handle wait-signal).

   B) Suppose you want to implement the `sys_open()` and `sys_close()` system calls. Is it necessary to associate the concept of ownership by a thread to a file descriptor, so that only the thread that called `open()` on that file is authorized to call `close()` on that file, too? (motivate the answer)
   NO. A file does not have ownership concepts of this type: a file can be closed by a thread different from the one that opened it. Indirectly, however, a file has a concept of ownership linked to the process, since a file descriptor is associated with the context of a process (and its table of open files).

   What are the OS161 `copyin()` and `copyout()` functions for?

   They are used to copy data between user and kernel memory: copyin() has as destination the kernel memory, copyout the user memory. The main difference from other forms of memory-to-memory copying is that any errors/exceptions related to invalid user addresses/pointers are handled consistently, thus preventing the kernel from terminating abnormally (e.g. crash/panic).

   Suppose to replace a call `copyin(src,dst,size);`
   with `memmove(dst,src,size);`
   Is this a legitimate replacement? Do you lose or gain something, or are they equivalent instructions?

   YES. This replacement is possible/correct. Other solutions are possible, but protection from exceptions / errors is lost.

4. Given the portion of the `load_elf()` function represented in the following figure, in which you want to read the header of the ELF file, having as its destination the `struct eh`.

```
load_elf(struct vnode *v, vaddr_t *entrypoint)
{
     Elf_Ehdr eh;    /* Executable header */
     int result;
     struct iovec iov;
     struct uio ku;
     /*
      * Read the executable header from offset 0 in the file.
      */
     result = VOP_READ(v, &eh, sizeof(eh));
    ...
}
```

The program reported is incorrect. Explain why and propose the necessary correction(s).

The call to VOP_READ is incorrect. Reading must be done with a different strategy: first we define the operation to be performed, using `uio_kinit` (for kernel memory operation), using ku and iov variables, then VOP_READ is called, using ku. The correct version is (the "exact" order of the `uio_kinit` parameters is not essential in terms of evaluation):

```
uio_kinit(&iov, &ku, &eh, sizeof(eh), 0, UIO_READ);
result = VOP_READ(v, &ku);
```

Then tell (in short) what they are (and what they are used for in the proposed program):

- ***the v parameter**: it is the pointer to the vnode (the file control block) of the ELF file from which the read is performed*
- ***the ku variable**: the struct in which the I/O operation must be set (via uio_kinit) before carrying it out (via VOP_READ). It points to iov, it contains fields to define read/write kernel/user, and more*
- ***the iov variable**: the struct in which the address in memory and size are actually loaded, for the destination of a VOP_READ (or source of a VOP_WRITE)*

5. Given an OS161 operating system.

Parts of the `uio_kinit`, `load_elf` and `load_segment` functions are shown in the following figure.

```
void uio_kinit (struct iovec *iov, struct uio *u,
              void *kbuf, size_t len,
               off_t pos, enum uio_rw rw) {
  iov->iov_kbase = kbuf;
  iov->iov_len = len;
  u->uio_iov = iov;
  u->uio_iovcnt = 1;
  u->uio_offset = pos;
  u->uio_resid = len;
  u->uio_segflg = UIO_SYSSPACE;
  u->uio_rw = rw;
  u->uio_space = NULL;
}
int load_elf (struct vnode *v, vaddr_t *entrypoint) {
  Elf_Ehdr eh;   /* Executable header */
  Elf_Phdr ph;  /* "Program header" = segment header */
  int result;
  struct iovec iov;
  struct uio ku;
  ...
  uio_kinit(&iov, &ku, &eh, sizeof(eh), 0, UIO_READ);
  result = VOP_READ(v, &ku);
  ...
}
```

```
load_segment (struct addrspace *as, struct vnode *v,
     off_t offset, vaddr_t vaddr,
     size_t memsize, size_t filesize,
     int is_executable) {

  struct iovec iov;
  struct uio u;
  int result;

  iov.iov_ubase = (userptr_t)vaddr;
  iov.iov_len = memsize;  // length of the memory space
  u.uio_iov = &iov;
  u.uio_iovcnt = 1;
  u.uio_resid = filesize; // amount to read from the file
  u.uio_offset = offset;
  u.uio_segflg = is_executable ? UIO_USERISPACE :
                                 UIO_USERSPACE;
  u.uio_rw = UIO_READ;
  u.uio_space = as;

  result = VOP_READ(v, &u);
  ...
}
```

Briefly explain the role of `struct iovec` and `struct uio`, in relation to the subsequent VOP_READ.

R The struct iovec contains the pointer to the destination memory area of the read and its size: &eh and sizeof(eh) in load_elf, vaddr and memsize in the load_segment.
The struct uio contains all the information necessary for the IO:

- the pointer to a (or possibly an array of) stuct iovec;
- the offset and the number of bytes to be read in the file;
- Information on the virtual space (kernel/user) and type of I/O (R/W) to be made.

Before making an I/O operation in kernel space, simply call the uio_kinit, to prepare and connect the two struct, before an I/O in user space, the two structures must be loaded explicitly, as there is no function equivalent to the uio_kinit for the user space.

Why does `load_segment` use UIO_USERISPACE / UIO_USERSPACE, while in the initial part of `load_elf` we use (via `uio_kinit`) UIO_SYSSPACE?

R Because the first part of load_elf acquires from the elf file, in a local variable in kernel memory, the header of the elf file: it is therefore an I/O of type UIO_SYSSPACE. The load_segment instead, must acquire the actual segments from the elf file to the user memory partitions just allocated for the process: the I/O is therefore of the UIO_USERISPACE type for the code (instructions) and UIO_USERSPACE for the data.

Why is the `u->uio_space` field assigned NULL in one case, while `as` in the other? (*What is this assignment for?*)

R	The assignment is used to provide the information necessary for translation between logical and physical addresses. Nothing is needed for the kernel space (so you leave a NULL pointer) as the translation simply consists of adding/subtracting MIPS_KSEG0. For the user space, the pointer to the struct addrspace of the process is needed, in which the logical-physical mappings of the two segments and the stack are defined.