Esercizi proposti e svolti su OS161 (tratti da compiti di esame)

NOTA: si sono volutamente lasciati commenti relativi a correzioni di risposte all'esame o indicazioni di errori possibili/frequenti

1. Sia dato un sistema operativo OS161.

   Si supponga che la funzione `syscall()`, in corrispondenza al valore `SYS__exit` in `callno`, chiami la funzione `my_sys_exit()`, si definisca il prototipo di tale funzione e se ne scriva la chiamata (con parametri attuali) in `syscall()`.

   Si osservi il seguente frammento di codice (incompleto) della `my_sys_exit()`,

```
my_sys_exit(...                    ) {


  ... = status;
  ...

  cv_signal(...);

  ...
  ...
  ...

  thread_exit();
}
```

   si dica da dove può o deve provenire il valore `status`, e a che variabile o campo di struct va assegnato. Si supponga che nel sistema si sia realizzata la `sys_waitpid()`, e che quest'ultima utilizzi come primitiva di sincronizzazione una condition variable. Si completi la `my_sys_exit` riportata nel seguito, fornendone una breve spiegazione:

```
my_sys_exit(int status) {  // lo stato viene ricevuto come parametro
  struct proc *p = curproc;  // serve per poter accedere al processo dopo averlo
                             // staccato da curthread (curproc non più valido)

  p->p_status = status;  // salva lo stato di ritorno per la waitpid

  proc_remthread(curthread);  // stacca il processo dal thread

  // segnala al processo che fa la waitpid (per usare cv_signal è
  // necessario possedere il relativo lock)
  lock_acquire(p->p_lock);
  cv_signal(p->p_cv,p->p_lock);
  lock_release(p->p_lock);

  // meglio NON fare as_destroy qui (lo farà la proc_destroy)
  // il thread finisce qui (diventa zombie)
  thread_exit();
}
```

   La chiamata dovrà essere del tipo:
   `my_sys_exit((int)tf->tf_a0);`

2. Perché in OS161, per gestire gli argomenti al main, è necessario creare una copia di `argv` e `argc`?

E' necessario in quanto gli argomenti al main debbono essere in memoria user, affinchè il programma utente possa farvi accesso. Siccome gli argomenti al main sono in origine in memoria kernel, si rende necessario farne una copia

Dove va creata tale copia?

Va fatta in memoria user, In particolare, in una parte accessibile dell'address space: la soluzione più semplice è l'inizio (indirizzi alti) dello (user) stack.

Perché non sono sufficienti i valori originali nargs e args, passate alla cmd_prog (menu.c: avente prototipo `static int cmd_prog(int nargs, char **args)`), a partire da una stringa di comando?

Il motivo principale è già stato citato: i valori originali sono in memoria kernel, quindi non accessibili al processo user.
Si potrebbe tuttavia aggiungere che, quand'anche il processo potesse accedervi, si tratterebbe di dati nello stack di un altro (kernel) thread, quello del menu, quindi da duplicare in ogni caso, a meno di garantirne la consistenza e accessibilità da un altro thread.

(Per completezza, si veda ad esempio la funzione `cmd_dispatch`, il cui vettore (locale) `args` sarà quello che viene ricevuto come `argv` da `cmd_prog`)

3. A) E' possibile realizzare la `sys_waitpid` utilizzando per l'attesa un lock, su cui fare `lock_acquire`, mentre la segnalazione, da parte della `sys_exit` viene realizzata con `lock_release` dello stesso lock? (motivare la risposta)
NO. Un lock non può essere usato per trasmettere sincronizzazioni di tipo wait-signal, proprio per il problema dell'ownership: il lock serve per mutua esclusione. Per wait-signal si consigliano semafori o condition variable (l'eventuale lock associato serve per mutua esclusione, non per gestire wait-signal).

   B) Si vogliono realizzare le system calls `sys_open` e `sys_close`. E' necessario associare a un file descriptor il concetto di ownership da parte di un thead, in modo tale che solo il thread che ha fatto la `open` di un file sia autorizzato a chiamare la `close` del file? (motivare la risposta)
NO. Un file non ha concetti di ownership di questo tipo: un file può essere chiuso da un thread diverso da quello che lo ha aperto. Indirettamente invece, un file ha un concetto di ownership legato al processo, in quanto un file descriptor è associato al contesto di un processo (e della relativa tabella dei file aperti).

   A cosa servono le funzioni OS161 `copyin` e `copyout`?
Servono a effettuare copie di dati tra memoria user e kernel, copyin ha come destinazione la memoria kernel, dualmente la copyout. La principale differenza rispetto ad altre forma di copia da memoria a memria consiste nel fatto che vengono gestite in modo consistente eventuali errori/eccezioni legate a indirizzi/puntatori user non validi, impedendo così al kernel di terminare in modo anomalo (es. crash/panic).

   SI supponga di sostituire una chiamata `copyin(src,dst,size);`
   con `memmove(dst,src,size);`
   Si tratta di una sostituzione lecita? Si perde o guadagna qualcosa, oppure sono istruzioni equivalenti?
SI. La sostituzione è lecita (sono possibili altre soluzioni) ma si perde la protezione da eccezioni/errori.

4. Sia data la porzione della funzione load_elf rappresentata in figura, in cui si vuole leggere dal file ELF l'header del file, avente come destinazione la 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));
   ...
}
```

Il programma riportato è errato. Si spieghi perché e si proponga la correzione necessaria.

La chiamata a VOP_READ è errata. La lettura va fatta con una strategia diversa: prima si definisce l'operazione da effettuare, mediante `uio_kinit` *(per operazione in memoria kernel), usando le variabili ku e iov, poi si chiama VOP_READ, utilizzando ku. La versione corretta è (non è fondamentale l'ordine "esatto" dei parametri a* `uio_kinit`*):*

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

Si dica poi (in breve) che cosa sono (e a cosa servono nel programma proposto):

- *il parametro v: è il puntatore a vnode (il file control block) del file ELF da cui si legge*
- *la variabile ku: la struct nella quale va impostata l'operazione di IO (mediante uio_kinit) prima di effettuarla (mediante VOP_READ). Punta a iov (seguente), contiene campi per definire lettura/scrittura kernel/user, e altro*
- *la variabile iov; la struct in cui effettivamente vengono caricati indirizzo in memoria e dimensione, per la destinazione di una VOP_READ (oppure sorgente di una VOP_WRITE)*

5. Sia dato un sistema operativo OS161.

A.

Si riportano in figura parti delle funzioni `uio_kinit`, `load_elf` e `load_segment`.

```
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);
  ...
}
```

Si spieghi brevemente il ruolo della "`struct iovec`" e della "`struct uio`", In relazione alla successiva VOP_READ.

R	La struct iovec contiene il puntatore all'area di memoria destinazione della read e la relativa dimensione: &eh e sizeof(eh) nella load_elf, vaddr e memsize nella load_segment. La struct uio contiene tutte le informazioni necessarie per l'IO: • Il puntatore a una (o eventualmente un vettore di) stuct iovec • L'offset e il numero di byte da leggere nel file • Le informazioni sullo spazio virtuale (kernel/user) e tipo di I/O (R/W) da effettuare Prima di effettuare un IO in spazio kernel, è sufficiente chiamare la uio_kinit, per predisporre e collegare le due struct, prima di un IO in spazio user, le due strutture vanno caricate in forma esplicita, in quanto non c'è una funzione equivalente alla uio_kinit per lo spazio user.

Perché `load_segment` utilizza UIO_USERISPACE/UIO_USERSPACE, mentre nella parte iniziale della `load_elf` si usa (tramite `uio_kinit`) UIO_SYSSPACE?

R	Perché la prima parte di load_elf acquisisce dal file elf, in una variabile locale in memoria kernel, l'header del file elf: si tratta quindi di un IO di tipo UIO_SYSSPACE. La load_segment invece, deve acquisire i segmenti

	veri e propri dal file elf alle partizioni di memoria user appena allocate per il processo: l'IO è quindi di tipo UIO_USERISPACE per il codice (istruzioni) e UIO_USERSPACE per i dati.

Perché al campo `u->uio_space` in un caso viene assegnato NULL, mentre nell'altro si assegna `as`? *(a cosa serve questa assegnazione?)*

R	L'assegnazione serve per fornire le informazioni necessarie alla traduzione tra indirizzi logici a fisici. Per lo spazio kernel non serve nulla (quindi si lascia un puntatore NULL) in quanto la traduzione consiste semplicemente nel sommare/sottrarre MIPS_KSEG0. Per lo spazio user serve invece il puntatore alla struct addrspace del processo, in cui sono definite le mappature logico-fisiche dei due segmenti e dello stack.