PdS 2020 - Laboratorio OS161 - 5

Per affrontare questo laboratorio è necessario:

- *aver svolto (e capito) i laboratori 2 e 4*
  - *il laboratorio 2 serve per rivedere le system call (parzialmente implementate) read e write*
  - *il laboratorio 4 per la gestione di un processo.*
- *Aver visto e capito la lezione os161-userprocess, nella quale, oltre a una visione complessiva sui processi utente e a una implementazione (parziale, limitata alla console) delle system call read/write) si descrivono, nell'ambito di load_elf, le operazioni necessarie per leggere/scrivere su un file.*

Realizzare il supporto per il file system: system call open/close e completamento di read/write

In OS161 manca il supporto per il file system, inteso come insieme delle system calls che forniscono operazioni sui file, quali: `open`, `read`, `write`, `lseek`, `close`, `dup2`, `chdir`, `__getcwd`. Tale supporto, oltre a richiedere la realizzazione delle singole funzioni, necessita opportune strutture dati (tabelle dei file aperti, direttori, ecc.) che permettano la ricerca dei file e la loro identificazione a partire da `id` interi.

Si chiede di realizzare in forma parziale e semplificata tale supporto (`open`, `close`, `read` e `write`), dopo aver familiarizzato con le strutture dati e le funzioni del kernel coinvolte:

- Il Virtual File System (VFS): .c in `kern/vfs` e .h `kern/include/vfs.h`. In tale ambito la `struct vnode` rappresenta un file. Le funzioni `vfs_open` e `vfs_close` servono ad aprire e chiudere il file (si veda ad es. `runprogram`).

- Il supporto per trasferimento dati tra file, memoria kernel e memoria user: modulo `uio`. Le funzioni di principale interesse in tale contesto sono: `uio_kinit` (seguita da VOP_READ o VOP_WRITE) per IO di kernel (per IO con buffer in userspace occorre una variante manuale di `uio_kinit`) e `uiomove` (chiamata da `copyout`, `copyin` e `copystr` (che trasferisce dati tra memoria user e kernel). Si noti che `copyout` e `copyin` servono per effettuare copia di dati tra memoria user e kernel in modo sicuro, cioè senza che il kernel vada in crash nel caso di un puntatore user errato.
  Ad esempio, nel caso in cui, data una stringa di lunghezza `len` in spazio user (`userptr_t stru`) da copiare in `strk` (in spazio kernel: `char *strk`), la copia andrebbe fatta con
  ```
  copyin(stru, strk, len+1);
  ```
  anzichè con
  ```
  for (i=0; i<=len; i++) {
    strk[i] = ((char *)stru)[i];
  }
  ```
  oppure
  ```
  memcpy (strk, (void *)stru, len+1);
  ```
  Si sono considerate le stringhe già correttamente allocate. Si è ipotizzato inoltre di copiare anche il terminatore di stringa (`'\0'`).

Si consiglia di osservare come viene gestito l'IO in `runprogram/load_elf/load_segment`, dalle quali si può prendere spunto per ulteriori ricerche sulle funzioni coinvolte. In particolare, si suggerisce di notare come venga aperto e chiuso un file in `runprogram`, mentre si guardi la `load_segment` per realizzare (in `sys_read` e `sys_write`) la lettura/scrittura tra file (dato un `vnode`) e buffer utente (un `vaddr_t`). In particolare, lettura e scrittura si realizzano con VOP_READ e VOP_WRITE, precedute da un'opportuna inizializzazione di una `struct iovec` e di una `struct uio` (si veda `load_segment`).

Per testare `open/close` si consiglia di utilizzare `testbin/filetest` (senza argomenti al main, se si volessero usare argomenti al main si dovrebbe prima realizzarne il supporto, cone indicato nella parte opzionale che segue).

Si consiglia di realizzare il supporto per `open` e `close` negli stessi file già usati per `read` e `write`. Occorre generare, per un dato processo user, una tabella di puntatori a `vnode` (per semplicità, realizzare un vettore di tali puntatori in cui si salva un puntatore a vnode per ogni nuovo file creato). Non si richiede ancora (sebbene possibile) di realizzare la doppia tabella (user e kernel)., che permetterebbe lo sharing di file tra processi. Un file viene aperto e chiuso con le funzioni `vfs_open/vfs_close` (si veda `runprogram`). La constante `OPEN_MAX (limits.h)` definisce il massimo numero di file aperti per un processo. Ad ogni file aperto va assegnato in file descriptor, un intero non negativo (il minimo tra quelli non occupati al momento della open).

Attenzione: `stdin`, `stdout` e `stderr`, a meno che siano rediretti su file, sono associati alla console (`kern/dev/generic/console.c`) gestita mediante `kprintf` (che indirettamente chiama `putch`) e `kgets` (che chiama `getch`). La tabella dei file va gestita in modo tale che la presenza o meno di un file su cui sia stato rediretto `stdin`, `stdout` o `stderr`, determini il tipo di IO da effettuare (a console o a file).

Passaggio degli argomenti al main (ARGOMENTO FACOLTATIVO)

Per passare gli argomenti (`argv`) a un programma utente occorre caricare gli argomenti nello spazio di indirizzamento del programma (nello stack dell'address space user): sia le stringhe che il vettore di argomenti `argv (un vettore di puntatori).`

Attenzione! Un char può essere collocato in un qualunque indirizzo virtuale, mentre un puntatore deve essere a un indirizzo multiplo di 4 (padding). Un vincolo simile esiste per lo stack pointer, che deve essere a un indirizzo multiplo di 8 (in quanto i dati più grandi rappresentabili (double) sono di 8 byte). Per comprendere come funziona il passaggio di `argv` and `argc,` si consideri, ad esempio, `testbin/tail`, che vuole due argomenti. Il `main()` di `tail` è:

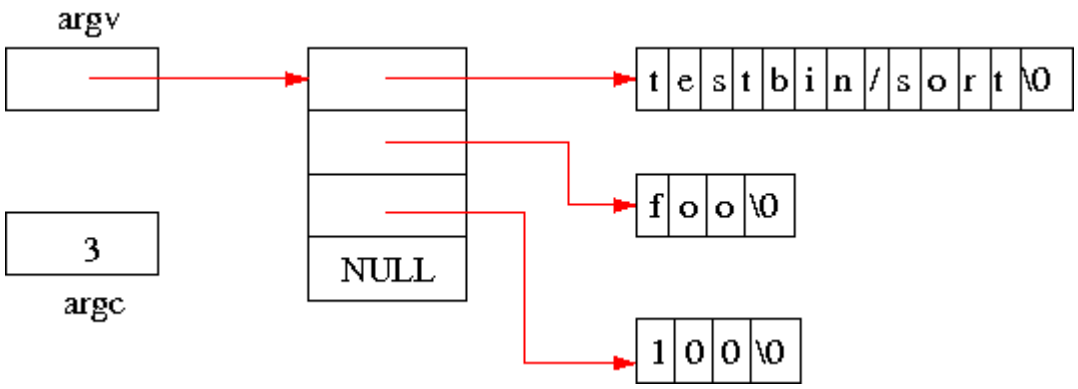
```
int main(int argc, char **argv) {

        int file;
        if (argc < 3) {
               errx(1, "Usage: tail  ");
        }
        file = open(argv[1], O_RDONLY);
        if (file < 0) {
               err(1, "%s", argv[1]);
        }
        tail(file, atoi(argv[2]), argv[1]);
        close(file);
        return 0;
}
```

`argv` è un puntatore a puntatore a char – punta a un vettore di puntatori a carattere, ognuno dei quali punta a uno degli argomenti. Ad esempio, se `tail` fosse chiamato con una linea di comando:

```
OS/161 kernel [? for menu]: p testbin/sort foo 100
```

Le variabili `argv` and `argc` sarebbero come nella figura:


argv
3
argc
NULL
t e s t b i n / s o r t \0
f o o \0
1 0 0 \0