Progetto OS161 C2: Shell

Giacomo Fantino s310624
Giacomo Cauda s317977

Introduzione

Il progetto scelto prevedere lo sviluppo di una serie di system call per il sistema operativo OS161, specificatamente progettato per poter essere programmato e completato per includere funzionalità dei moderni sistemi operativi.

La nostra configurazione iniziale includeva già ciò che è stato fatto nei 5 laboratori del corso, specificatamente:

- Un sistema di gestione della memoria principale (*kmalloc* e *kfree*).
- Primitive per la mutua esclusione (lock, semafori e condition variable).
- System calls per la gestione dei file (*open*, *read* e *write*).
- System call per la sincronizzazione (*waitpid*).

Queste implementazioni non saranno spiegate perché sono state analizzate e implementate a lezione, ma saranno mostrate nel caso sia stato necessario andare a modificarle per il progetto.

Il mezzo principale con cui sono gestiti i cambiamenti alle funzioni e le nuove implementazioni è mediante l'utilizzo delle **opzioni**. In OS161 quando si crea una nuova configurazione si sceglie quali opzioni includere, le quali cambiano quali sono i pezzi di codice che saranno compilati e quindi come l'applicazione si comporta.

Nel nostro caso tutte le nuove aggiunte sono state fatte utilizzando l'opzione **OPT_SHELL**.

Infine OS161 implementa un sistema di errori dato dalla definizione nel file *kern/errno.h* di un insieme di macro le quali associano un nome ad un codice che identifica univocamente l'errore. Un esempio è il caso in cui non si può più allocare memoria perché tutta occupata: con la macro **ENOMEM** (associata al numero 3) si può fare riferimento a questo errore.

Obiettivi

Lo scopo di questo progetto è la realizzazione di una serie di System Call che ci permettano di espandere le funzionalità di OS161, tra cui la possibilità di eseguire più processi contemporaneamente o avere a disposizione tutte le system call funzionanti per poter aprire, leggere o scrivere e salvare un file all'interno del File System.

Nella parte finale abbiamo utilizzato la shell contenuta in *bin/shell* e i programmi contenuti in *testbin/* per testare le nostre implementazioni e verificarne la correttezza.

File systems

Per la realizzazione delle system call relative alla gestione dei file abbiamo deciso di utilizzare un approccio a due tabelle: una **System File Table** globale che mantiene le informazioni riguardo a tutti i file aperti, e una **Open File Table** per ogni processo che punta alle entry della System File Table in base ai file aperti dal processo.

La definizione della System File Table è la seguente:

```
struct systemFileTable {
    struct vnode *vn;
    off_t offset;
    int mode_open;
    unsigned int count_refs;
    struct lock *lock;
};
```

Oltre a memorizzare il puntatore al file aperto (che in OS161 sono gestiti mediante vnode) memorizziamo anche le seguenti informazioni:

- offset del file aperto.
- modalità con cui è stato aperto.
- il numero di processi che fanno riferimento a questa entry.
- un lock per le modifiche in mutua esclusione.

Per la open file table abbiamo aggiunto alla **struct proc** un array di riferimenti alle entry della System File Table. Gli indici del array faranno da file descriptor dei rispettivi file aperti.

Per la gestione dei file descriptor insieme a *stdin*, *stdout* e *stderr* abbiamo deciso di trattarli come fossero dei **VNODE**. I vari input e output sono memorizzati in accordo ai valori delle costanti *stdin*, *stdout* e *stderr*: nella Open File Table in posizione 0 di default ci sarà il riferimento ad una entry di System File Table la quale punta al vnode dello standard input. Questa apertura dei vnode e inserimento nella Open File Table viene fatta all'avvio di ogni processo (così che l'utente non debba esplicitamente aprirli per scriverci/leggerci).

Per le system calls (contenute nel file file_syscalls.c):

- ***open****(path, flag, mode, err)*: il file a cui si riferisce path viene aperto mediante *vfs_open*. Il suo vnode viene inserito nella system file table e il riferimento alla rispettiva entry viene inserita nella open file del processo. Si noti che la ricerca di una entry libera non usa mai le prime 3 posizioni perché occupate da *stdin*, *stdout* e *stderr*.

- ***read****(fd, buf, size)*: viene richiamata la *file_read* per la gestione di file e di stdout/stderr. Tramite *curproc* facciamo riferimento alla openFile table e prendiamo il puntatore al vnode nella System File Table. Mediante *VOP_READ* andiamo a scrivere per poi aggiornare l'offset del file.

- ***write****(fd, buf, size)*: funzionamento simile alla read, semplicemente in questo caso si va a richiamare la funzione *VOP_WRITE*.

- ***close****(fd)*: permette di chiudere un file aperto. Diminuisce il contatore di file aperti e va a mettere a *null* il campo della Open File Table che faceva riferimento al file.

- ***lseek****(fd, pos, whence)*: modifica l'offset memorizzato nella entry della system file table del file in base ai valori di pos e whence.

Per la lseek abbiamo dovuto gestire la posizione dell'offset di un file (valore a 64 bit) per la system call *lseek*. Essendo che il trapframe memorizza valori a 32 bit il valore di pos viene memorizzato in due campi diversi (a2 e a3) mentre il valore di whence viene salvato nello stack.

Inoltre il valore di ritorno di lseek è anch'esso un valore a 64 bit. Per essere inserito dentro il trapframe abbiamo usato i due campi *tf_v0* e *tf_v1*, ma per non andare a modificare il codice di syscall.c i 32 bit più piccoli sono inseriti dentro retval (il cui valore sarà poi inserito dentro *tf_v0*).

```
err = sys_lseek((int)tf->tf_a0, (((off_t)tf->tf_a2 << 32) | ((off_t)tf->tf_a3)),
    *(int32_t *)(tf->tf_sp+16),
    &retval_low, &retval_up);

if(!err){
    retval = retval_low;
    tf->tf_v1 = retval_up;
}
```

- ***dup2****(oldfd, newfd)*: andiamo a modificare il file descriptor oldfd in modo tale che vada a puntare alla entry della system file table puntata da *newfd*. In questo modo il processo può andare a scrivere e leggere su uno stesso file da entrambi i file descriptor.

OS161 include già le funzionalità per gestire la current directory di un processo usando il campo ***p_cwd***.

- ***getcwd****(buf, buflen)*: inserisce dentro buf il nome della current directory del processo. Per ottenere questa informazione viene utilizzata *vfs_getcwd*.

```
uio_kinit(
  &iov,
  &u,
  (userptr_t)buf,
  buflen,
  0,
  UIO_READ
);

u.uio_space = curthread->t_proc->p_addrspace;
u.uio_segflg = UIO_USERSPACE;
```

  Per specificare a *vfs_getcwd* di andare a memorizzare il nome della current directory nel buffer utente andiamo a creare un *uio* mediante *uio_kinit*.
  Dobbiamo modificare però uio_segflg poiché kinit assume si voglia scrivere in memoria kernel ma noi utilizziamo *UIO_USERSPACE* (il buffer si trova nella memoria dell'utente).

- **chdir**(pathname): mediante *vfs_setcurdir* mette come current directory del processo il vnode della cartella indicato da pathname.

Processi

Andiamo ora a vedere ora come abbiamo espanso OS161 per implementare le system calls relative ai processi.

Gestione PID

Abbiamo introdotto una tabella in cui vengono memorizzate le seguenti informazioni:

- Active indica se la tabella è già stata istanziata dal bootstrap (utile nel caso dei processi cerchino di accedervi quando non è stata ancora creata).
- proc è un array che ad ogni pid associa un puntatore a un processo. (se una posizione vale *null* il pid non è stato assegnato).
- last_i indica l'ultimo pid assegnato.
- lk è uno spinlock per poter accedere e modificare la tabella in mutua esclusione.

```
#define MAX_PROC 100
static struct _processTable
{
    int active;
    struct proc *proc[MAX_PROC + 1];
    int last_i;
    struct spinlock lk;
} processTable;
```

La *proc_destroy* è stata modificata per fare in modo che il pid utilizzato da un processo venga restituito, e quindi sia riutilizzabile in futuro, non appena un altro processo richiama waitpid sullo stesso.
Inoltre con questa tabella abbiamo creato una funzione *proc_search_pid* la quale dato un pid ci viene restituito il riferimento del processo che possiede quel pid (la quale ci è stata fondamentale per waitpid).
Infine alla creazione di un processo viene richiamata la funzione *proc_init* la quale assegna al processo il primo pid che trova libero nella tabella.

Possiamo ora andare a vedere come funzionano le system calls per la gestione dei pid:

- ***getpid()*:** restituisce il Pid del processo che la chiama.
- ***waitpid(** pid, statusp, options)***:** Risaliamo ad un processo tramite il pid che ci viene fornito, attraverso la funzione *proc_search_pid(pid_t)*. Una volta ottenuto lo mettiamo in attesa con il semaforo contenuto in proc. Questa funzione è fondamentale perché va a richiamare la *proc_destroy*, la quale elimina definitivamente un processo dalla memoria principale e restituisce il thread.

Gestione e creazione di processi

Per quanto riguarda le System Call relative ai processi, abbiamo dovuto introdurre alcune strutture ausiliarie necessarie alla gestione dei processi figli:

```
struct lista_child {
    pid_t pid;              /*
    struct lista_child * next;
} lista_child;
```

Andando a mettere nella struttura *proc* una lista concatenata contenente i pid dei processi figli del processo. Inoltre nella struttura *proc* abbiamo utilizzato un campo *parent_pid* che contiene il pid del processo che lo ha creato.

Quando un processo muore vengono fatte due cose:

- Viene tolto dalla lista dei child del padre (**remove_child**).
- Tutti i processi figli vengono resi orfani, ovvero *parent_pid* viene impostato ad 1.

Possiamo ora andare a vedere come operano le due system call per creare dei nuovi processi:

- ***fork****(*trapframe, *retval)*: funzione che crea un nuovo processo con lo stesso address space del padre ma diverso pid. Permette di creare due processi paralleli i quali si possono coordinare condividendo le strutture dati. I passi sono i seguenti:
  - ➔ Creiamo un processo nuovo, vuoto, senza nessun address space. Per farlo utilizziamo la funzione *proc_create_runprogram(p_name)* a cui assegniamo lo stesso nome del processo padre.
  - ➔ Con *as_copy* andiamo ad assegnargli lo stesso address space del padre.
  - ➔ Eseguiamo una copia del **Trap Frame** del padre ed una copia della tabella dei file aperti.
  - ➔ Ora gestiamo l'aggiunta del processo figlio alla lista dei processi figli del padre. Inoltre inseriamo come *parent_pid* del processo figlio il pid del padre.
  - ➔ Infine, con *thread_fork*, possiamo fare partire il nuovo processo in modalità utente.

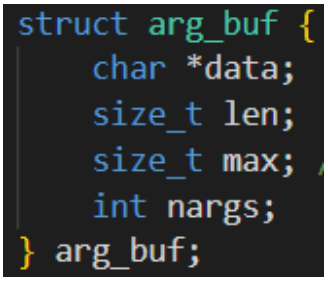
execv (program, args)

Per execv abbiamo avuto necessità di occuparci dello spostamento della stringa program e del vettore di stringhe args dalla memoria del processo originale al kernel (per gestire la creazione del nuovo processo) e al nuovo processo.

Per la stringa di program è bastato utilizzare la funzione *kmalloc* seguita da una *copyinstr*. Per il vettore di argomenti abbiamo utilizzato una struct *arg_buf* per:

- memorizzare in data gli argomenti.

- memorizzare informazioni aggiuntive quali il numero di argomenti (**nargs**), la lunghezza degli argomenti (**len**) e la dimensione in byte allocata per data (**max**).



Descritta la struttura utilizzata possiamo ora vedere come opera *execv*:

Per quanto riguarda il caricamento del nuovo programma, istanziamo con una chiamata a *kmalloc* della memoria all'interno di *kpath*, che conterrà quindi il percorso dell'eseguibile.

- ***copyinstr()***: funzione nativa di OS161 per copiare dalla memoria user alla memoria kernel.
- Calcoliamo la dimensione degli elementi all'interno del vettore di stringhe args, in modo da allocare memoria sufficiente per contenerli.
- ***argbuf_fromuser****(&kargv, uargv, num_args, len_args)*: funzione che prende il vettore di stringhe in memoria utente *uargv*, lo memorizza in *kargv->data* e aggiorna i rispettivi campi.
- ***load_program****(kpath, &entrypoint, &stackptr) :* carica il file elf riferito da program e crea un address space vuoto.
- ***argbbuf_touser****(&kargv, &stackptr, &uargv)* : carica i parametri dalla memoria kernel alla memoria utente. Questi sono memorizzati nelle prime posizioni libere dello stack.
- ***enter_new_process****(kargv, nargs, uargv, NULL, stackptr, entrypoint)*: funzione che lancia un nuovo processo. Andiamo a specificare il numero di argomenti (primo parametro) e la loro locazione nella memoria utente (secondo parametro).

Exit

Abbiamo modificato la *thread_exit*, in modo da gestire il riutilizzo del pid di un processo alla sua terminazione.
Inoltre, andiamo a cancellare tutti i processi figli dal processo padre se quest'ultimo muore; lo facciamo con la funzione *proc_remove_child* (questo solo nel caso di *fork* sia stata implementata).

```c
void sys__exit(int status)
{
#if OPT_WAITPID
  struct proc *p = curproc;

  p->p_status = status & 0xff; /* just lower 8 bits returned */
  proc_remthread(curthread);
  V(p->p_sem);
#else
  /* get address space of current process and destroy */
  struct addrspace *as = proc_getas();
  as_destroy(as);
#endif

  #if OPT_FORK
    //rimuovi il processo dal padre
    remove_child(proc_search_pid(p->parent_pid), p->p_pid);
  #endif

  /* thread exits. proc data structure will be lost */
  thread_exit();

  panic("thread_exit returned (should not happen)\n");
}
```

Gestione errori

La precedente gestione degli errori prevedeva il lancio di un panic richiamando la funzione *killcurthread*. L'implementazione migliorata prevede che un thread che va in errore venga terminato e venga impostato uno status di errore al processo che ha generato il thread.
Per prima cosa impostiamo lo status di errore (fornitoci dal codice di errore *code*), dopodichè rimuoviamo il thread dal processo attraverso la *proc_remthread(curthread)*. Viene anche eseguita la signal sul semaforo che gestisce la *proc_destroy* e per ultima operazione eseguiamo *thread_exit()*.

Per impostare lo status andiamo ad inserire nei 30 bit più significativi il codice di errore e nei 2 bit meno significativi il valore 1, che indica che un processo è stato bloccato perché ha ricevuto una trap (informazioni incluse dentro *include/kern/wait*).

```c
//per evitare crash del sistema operativo andiamo a chiamare la exit sul thread
//questo porterà ad un context switch e alla rimozione del processo errato
#if OPT_SHELL
    struct proc *p = curthread->t_proc;
    //Ogni processo mantiene un info: Status (32 bit)
    //SIG --> mi da l'errore per questa situazione specifica
    //devo shiftare sig di 2 per settarlo come stato del processo
    //i due bit più bassi dicono cosa è successo, i restanti 30 bit sono il codice di uscita
    //l'OR finale con 1 serve per segnalare che l'uscita è dovuta ad un errore
    //spiegazione in include/kern/wait.
    p->p_status = (sig<<2 | 1); //shift di due a sinistra per WEXITSTATUS
    proc_remthread(curthread);
    V(p->p_sem); //quando il semaforo arriva a zero viene eseguita la proc_destroy
    thread_exit();
#else
    panic("I don't know how to handle this\n");
#endif
```

Analisi delle implementazioni

Ecco un elenco di test che abbiamo usato per verificare la correttezza delle nostre implementazioni:

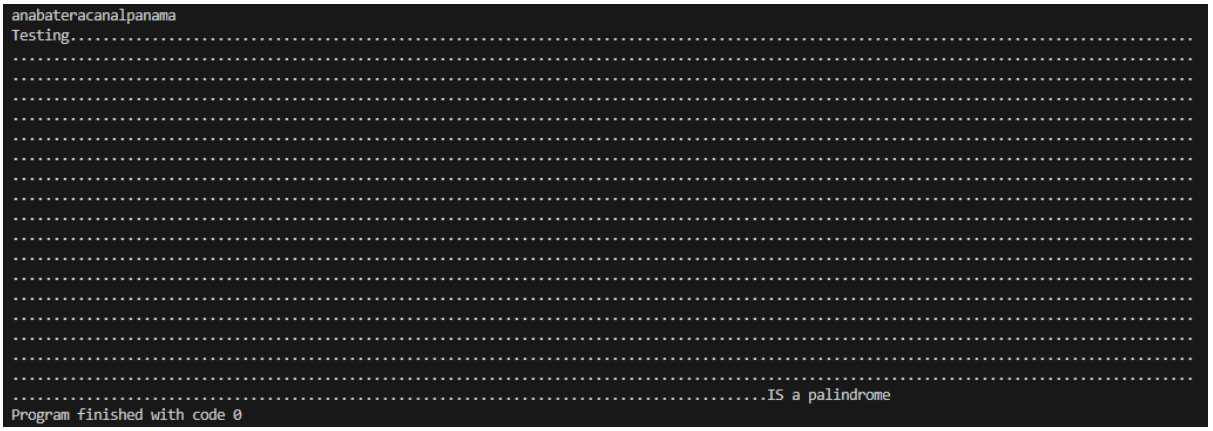
testbin/filetest

Abbiamo usato questo test andandolo a modificare perché non usi la system call *remove*, per poi andare a ricompilarlo usando *bmake* e *bmake install*. Scopo del test è aprire un file, scrivere dentro una stringa per poi riaprire il file in lettura per verificare che ciò che è stato scritto è stato correttamente memorizzato. Mediante debugger abbiamo verificato la correttezza dell'esecuzione.

```
OS/161 kernel [? for menu]: p testbin/filetest
(program name unknown): No arguments - running on "testfile"
Passed filetest.
Program finished with code 0
Operation took 0.326098457 seconds
```

testbin/palin

Test utilizzato per verificare la correttezza della write usando lo standard output (dopo la modifica per la sua gestione come *vnode*).



testbin/bigseek

Questo test prevede l'utilizzo di tutte le system call per la lettura e scrittura su un file e il posizionamento dell'offset di un file.
Il test include anche varie situazioni di errore sia in caso di *lseek* ma anche per quanto riguarda *read* e *write*.
Avendo gestito correttamente tutti i valori di ritorno da passare alle system call il test è superato correttamente.

```
OS/161 kernel [? for menu]: p testbin/bigseek
Creating file...
Writing something at offset 0
Seeking to (and near) 0x1000
Writing something else
Seeking to (and near) 0x0
Checking what we wrote
Seeking to (and near) 0x1000
Checking the other thing we wrote
Seeking to (and near) 0x20
Seeking to (and near) 0x7fffffff
Seeking to (and near) 0x80000000
Seeking to (and near) 0x80000020
Seeking to (and near) 0x100000000
Seeking to (and near) 0x100000020
Seeking to (and near) 0x180000000
Seeking to (and near) 0x180000020
Now trying to read (should get EOF)
Now trying to write (should get EFBIG)
Seeking to (and near) 0x100000000
Trying to read again (should get EOF)
Passed.
Program finished with code 0
```

chdir + getcwd

Data la loro semplicità essendo che richiamano le funzioni già definite in *vfs* abbiamo deciso di non testare la loro funzionalità ma di provare a verificare gli input errati mediante *testbin/badcall*.
Questi vanno a richiamare le system calls con input errati per verificare che siano gestiti e che siano restituiti gli errori corretti.

```
Choose: s
badcall: chdir with NULL path... Bad memory reference                passed
badcall: chdir with invalid-pointer path... Bad memory reference     passed
badcall: chdir with kernel-pointer path... Bad memory reference      passed
badcall: chdir to empty string... Invalid argument                   passed
Choose: z
badcall: getcwd with NULL buffer... Bad memory reference             passed
badcall: getcwd with invalid buffer... Bad memory reference          passed
badcall: getcwd with kernel-space buffer... Bad memory reference     passed
```

Fork

Essendo una system call importante e molto usata in diversi applicativi abbiamo deciso di utilizzare più test per verificarne l'utilizzo e la correttezza in diverse situazioni.

Il primo test utilizzato è *testbin/forktest*, nel quale viene fatta una chiamata a fork per ogni processo precedentemente generato per 4 volte. Iniziando da un processo iniziale andremo ad averne 2, poi 4 e infine 8. Ogni processo andrà a stampare una lettera e per verificarne la correttezza le lettere devono seguire il pattern di generazione (2 A, 4 B ecc.). SI noti che ogni processo padre alla fine delle stampe andrà ad aspettare sui figli richiamando waitpid.

```
OS/161 kernel [? for menu]: p testbin/forktest
(program name unknown): Starting. Expect this many:
|--------------------------|
AABBBBCCCCCCCCDDDDDDDDDDDDDDDD
(program name unknown): Complete.
Program finished with code 0
Operation took 1.168476991 seconds
```

Il secondo test che abbiamo utilizzato è *testbin/parallelvm*, il quale va a lanciare 24 processi in parallelo con stessa priorità per testare se il kernel vada a gestire correttamente la presenza di più processi e la parallelizzazione.

```
Process 10 answer -182386335: passed
Process 11 answer -364554240: passed
Process 12 answer 251084843: passed
Process 13 answer -61403136: passed
Process 14 answer 295326333: passed
Process 15 answer 1488013312: passed
Process 16 answer 1901440647: passed
Process 17 answer 0: passed
Process 18 answer -1901440647: passed
Process 19 answer -1488013312: passed
Process 20 answer -295326333: passed
Process 21 answer 61403136: passed
Process 22 answer -251084843: passed
Process 23 answer 364554240: passed
Test complete
```

Ultimo test che abbiamo utilizzato è *testbin/bigfork* il quale può essere visto come un misto dei due test precedenti: i processi sono divisi come in *forktest* ma l'algoritmo procede fino ad una profondità di 6 divisioni. In contemporanea i processi fanno delle operazioni come *parallelvm*.

```
Stage 5 #7 done: 62977821
Stage 5 #39 done: 62977821
Stage 5 #23 done: 62977821
Stage 5 #55 done: 62977821
Stage 5 #15 done: 62977821
Stage 5 #47 done: 62977821
Stage 5 #31 done: 62977821
Stage 5 #63 done: 62977821
Stage 5 #35 done: 62977821
Stage 5 #3 done: 62977821
Stage 5 #19 done: 62977821
Stage 5 #51 done: 62977821
Stage 5 #11 done: 62977821
Stage 5 #43 done: 62977821
Stage 5 #27 done: 62977821
Stage 5 #59 done: 62977821
Done.
Program finished with code 0
Operation took 31.897302606 seconds
```

execv

Il primo test che abbiamo eseguito per testare il funzionamento della execv è *testbin/bigexec*.
Questo ultimo richiama execv provando input diversi sia in numero che in dimensione.

```
OS/161 kernel [? for menu]: p testbin/bigexec
(program name unknown): Starting.
(program name unknown): 1. Execing with one 8-letter word.
/testbin/bigexec: 2. Execing with one 4050-letter word.
/testbin/bigexec: 3. Execing with two 4050-letter words.
/testbin/bigexec: 4. Execing with 16 4050-letter words.
/testbin/bigexec: 5. Execing with one 16320-letter word.
/testbin/bigexec: 6. Execing with two 16320-letter words.
/testbin/bigexec: 7. Execing with four 16320-letter words.
/testbin/bigexec: 8. Execing with one 65500-letter word.
/testbin/bigexec: 9. Execing with 300 8-letter words.
/testbin/bigexec: 10. Execing with 3850 8-letter words.
/testbin/bigexec: Complete.
```

Un altro modo di testare la execv è semplicemente utilizzando la shell. Questo ci permette di verificare la correttezza di execv e la ricezione dei parametri da parte della funzione chiamata.

Test dei programmi false, true, pwd della shell.

```
OS/161$ pwd
emu0:
(program name unknown): subprocess time: 0.095872584 seconds
OS/161$ true
(program name unknown): subprocess time: 0.077485751 seconds
OS/161$ false
(program name unknown): subprocess time: 0.077761678 seconds
Signal 0
```

Test programma CAT

```
OS/161 kernel [? for menu]: p bin/sh
(program name unknown): Timing enabled.
OS/161$ cat cat_test/file_cat.txt
File prova di output cat \n(program name unknown): subprocess time: 0.177430304 seconds
```

Test programma CP.
Il file *file_cat* contiene “File prova di output cat \n” mentre file_cp contiene nulla.
Eseguiamo la cp e successivamente la cat su *file_cp* per verificare che il contenuto del primo file sia stato copiato correttamente nel secondo

```
OS/161 kernel [? for menu]: p bin/sh
(program name unknown): Timing enabled.
OS/161$ cat cat_test/file_cp.txt
(program name unknown): subprocess time: 0.114931663 seconds
OS/161$ cp cat_test/file_cat.txt cat_test/file_cp.txt
(program name unknown): subprocess time: 0.142884228 seconds
OS/161$ cat cat_test/file_cp.txt
File prova di output cat \n(program name unknown): subprocess time: 0.162856771 seconds
```