

SYSTEM NETWORK PROGRAMMING

NOTES

Libro di testo -> Advanced Programming in the UNIX Environment, W. Richard Stevens
Per la parte di networking, Stevens ha fatto una parte per protocollo TCP/IP fatta bene, una sull'implementazione, ma sulla programmazione di rete vedi UNIX Network Programming (The Sockets Networking API)

Poi c'è The Linux Programming Interface di Kerrisk, che riprende il libro di Stevens e lo specifica maggiormente su Linux.

Libro facilmente scaricabile in italiano sulle system call è GaPiL Guida alla Programmazione in Linux, Simone Piccardi, in italiano.

Come si trasferiscono file da mac a linux? Usando "vai a" potresti aprire in gui la schermata di linux.

Altrimenti usa scp (che si appoggia sul server ssh quindi bisogna specificare la porta)

```
ssh fdenunzio859@hplinux3.unisalento.it -p 2552
scp -P 2552 -r testdir fdenunzio859@hplinux3.unisalento.it:testdir
```

ricordati che il livello di sicurezza è alto
ma su moodlis no N4QzWEyZ

Esiste una fondamentale differenza tra i linguaggi interpretati e i linguaggi compilati:

- I linguaggi compilati vengono convertiti direttamente nel codice macchina che viene poi eseguito dal processore. Di conseguenza, tendono ad essere più veloci ed efficienti da eseguire rispetto ai linguaggi interpretati. Inoltre, permettono allo sviluppatore di avere un maggior controllo sugli aspetti legati all'hardware, come la gestione della memoria e l'uso della CPU. I linguaggi compilati necessitano di una fase di "costruzione": devono essere compilati manualmente.
- Gli interpreti funzionano attraverso un programma che esegue ogni comando riga per riga. I linguaggi interpretati erano notevolmente più lenti dei linguaggi compilati. Adesso, con lo sviluppo della compilazione just-in-time, il divario sta diminuendo.

C'è da fare una distinzione tra kernel e sistema operativo.

Il ruolo principale di ogni sistema operativo è quello di coordinare tutte le risorse permettendo il corretto funzionamento di ogni dispositivo attraverso continui scambi tra componenti software e hardware. Il ruolo del kernel, invece, è quello di gestire la memoria, la gestione dei processi, delle attività e del disco.

Il sistema operativo è basato sul kernel, ed aggiunge ad esso le interfacce per permettere all'utente di interagire.

Noi conosciamo le utilities di un sistema operativo (come lo shell), invece ora faremo le system call che permettono di programmare le utilities e i comandi già visti.

Le system call si presentano al programmatore come le funzioni di libreria in C. La differenza sta in come viene chiamata la system call; nei programmi compilati, la riga di codice viene trasformata in istruzioni per il processore: sequenze di bit che corrispondono a particolari istruzioni. I programmatori usano il linguaggio assembly per avere un linguaggio più vicino alle sequenze di bit. In C spesso le systemcall hanno un wrap sotto forma di funzione che ci permette di usarle facilmente.

Il codice utente utilizza funzioni, che per esempio in C possono rappresentare delle system call (non essendo direttamente loro, ma sono "contenute" nelle funzioni). L'utente può chiamare funzioni di C che utilizzano a loro volta delle system call, ma anche direttamente una system call.

In genere il codice assembly mette gli argomenti della system call da qualche parte, poi mette un particolare numero da un'altra parte. Il numero corrisponde alla system call. In alcuni registri c'è ciò che serve per effettuare la system call (argomenti), e in un registro particolare viene proprio inserito il numero che identifica la system call.

Per esempio

```
mov ebx, 1 ; file descriptor (stdout)
mov eax, 4 ; numero della system call, in questo caso è una sys_write
int 0x80 ; viene chiamato il kernel
Cio che succede dopo l'istruzione int 0x80 è il codice del kernel che noi non vediamo.
```

System call più semplice è la exit:

```
mov eax, 1 ; system call number of exit
int 0x80 ; viene chiamato il kernel
```

Esempio di una write:

```
mov edx, 4 ; lunghezza del messaggio
mov ecx, msg ; messaggio da scrivere
mov ebx, 1 ; file descriptor (stdout)
```

Il file descriptor è un valore che specifica un file che proviene dall'output di un'altra systemcall, che in questo caso è open (che avrà come argomento il percorso del file).

```
mov eax,4 ; (system call number della write)
int 0x80 ; viene chiamato il kernel
```

```
#include <stdio.h>
int main(void) {
    printf("hello, world!\n");
    return 0;
}
```

The above program uses `printf`, which under the hood makes a system call to write those bytes to stdout. We can see this using `strace`:

```
$ cc hello.c
$ strace ./a.out
...
write(1, "hello, world!\n", 14)      = 14
...
```

`strace` conveniently shows us these system calls using C syntax. We can use that expression in our program instead of using `printf`:

DIRETTIVE DEL PREPROCESSORE

Ci sono due compilatori che sono `gcc` e `clang`, noi ci occuperemo di `gcc`. Esso permette di compilare qualsiasi estensione e qualsiasi linguaggio. Da notare che in realtà macos dice di usare `gcc` ma poi chiama `clang`.

Quando si compila, una cosa fondamentale sono le direttive del preprocessore, ovvero tutto ciò che è nel codice sorgente ma che viene fatto dal preprocessore prima di compilare. In C le direttive del preprocessore sono precedute dal carattere `#`. Per dettagli controllare su wikipedia.

1 Directives	
1.1 #include	
1.1.1 Headers	
1.2 #pragma	
1.3 #define	
1.4 macros	
1.5 #error	
1.6 #warning	
1.7 #undef	
1.8 #if,#else,#elif,#endif (conditionals)	
1.9 #ifdef,#ifndef	
1.10 #line	

https://en.wikibooks.org/wiki/C_Programming/Preprocessor_directives_and_macros

Un classico esempio sono gli `#include`, i `#define` e gli `#ifdef`.
 Define -> sostituisce tutte le occorrenze di una stringa (chiamata macro) con un valore prima di compilare. Inoltre, possono essere utilizzate per effettuare debug in fase di compilazione. Le macro, infatti, sono spesso combinate con la `ifdef` per decidere se compilare o meno una parte di codice, per esempio per compilare per il giusto sistema operativo. Per questo motivo è anche utile creare una macro senza associargli un valore.

Ifdef -> una normale if che viene però verificata prima della compilazione. Viene utilizzata per compilare delle parti di codice solo al verificarsi di alcune condizioni. Le condizioni utilizzate sono l'esistenza o meno di una macro.

Include -> Includere un file equivale a copiarne il codice all'interno dell'attuale file, in modo da conoscerne le funzioni. Gli include usualmente avvengono a cascata, ovvero, ogni file può includere a sua volta molti altri file e così via. Qui, infatti, torna utile l'utilizzo della #ifdef, per sapere se un file è stato già incluso o meno.

Ci sono due differenti tipi di include:

1. `#include "apue.h"` → utilizzando le virgolette
2. `#include <dirent.h>` → utilizzando le parentesi triangolari

Nel secondo caso viene incluso un file include di sistema: il sistema offre infatti delle librerie allo sviluppatore. Di queste librerie alcune sono standard, invece altre non fanno parte dello standard ma sono comunque offerte allo sviluppatore. I file di intestazione di queste librerie si trovano in alcune directory standard. Su linux la directory standard è /usr/include.

Infatti, se si vuole sapere dove si trova il file `<dirent.h>` (su Linux), sapendo che è tra i file di intestazione e che è una libreria di sistema, possiamo utilizzare il comando:

`find /usr/include -name dirent.h`. Si nota che ce ne sono due, ma uno è in una sottodirectory. Nel file ci sono molte direttive del preprocessore, una dice: se non è definito dirent_H, allora definiscilo. Se la macro non è definita, la definisce e continua con il codice necessario, altrimenti tutto il file viene ignorato perché significa che sarà già stato incluso: questo serve per evitare doppie inclusioni.

Dopodiché ogni sviluppatore può usare le sue librerie o scaricarne fatte da altri, e qui entrano in gioco le virgolette: per importare una propria libreria si include come nel primo caso.

Per poter utilizzare librerie sono necessari i prototipi (o intestazioni) che mostrano la struttura della funzione, e il file compilato con le definizioni delle funzioni usate.

Bisogna distinguere infatti la definizione delle funzioni, dove è presente il codice sorgente di esse, e i loro prototipi, che invece mostrano solo la loro intestazione in modo da fornire al compilatore informazioni sufficienti per sapere cosa aspettarsi.

Il file .h che viene incluso è quello che contiene tutti i prototipi delle funzioni. Poi in fase di compilazione del main, tramite il linking si ricollegano queste intestazioni al file (già compilato) che contiene le definizioni.

Su linux, i file di intestazione sono in /usr/include, le librerie sono in un'altra directory di default /usr/lib

Per verificare `find /usr -name libc.a`

Dove sono i file di intestazione del mac? Per questione di sicurezza dovrebbero essere nella cartella di XCODE, in modo da "non renderli disponibili se non necessario".

Cerchiamo stdio.h nella cartella di XCODE poiché lì ci sono tutti i prototipi delle funzioni che utilizziamo. Ovviamente dentro le librerie di sistema ci sarà la definizione delle funzioni.

È possibile includere il codice staticamente, oppure a Runtime dinamicamente. Vediamo prima come farlo staticamente.

NOTA: Per questioni di sicurezza le librerie dinamiche sono soggette alla casualizzazione di

dove vengono caricate nella memoria per questioni di sicurezza

Effettuiamo ora la nostra **prima compilazione**, tramite i file nella directory "First compilation":

Innanzitutto, ricordiamo che un programma c non linka se non c'è una funzione che si chiama main, quindi solo program.c potrà diventare un programma.

Ci sono due file che definiscono una funzione (bill.c e fred.c), e un file lib.h con le intestazioni che viene incluso in program.c

```
---> gcc -c fred.c
---> gcc -c bill.c
---> gcc -o my_exe program.c bill.o fred.o
---> ./my_exe
```

```
program.c
#include <stdlib.h>
#include "lib.h"

int main()
{
    bill("Hello World");
    fred(99999);
    exit(0);
}

francescodenu@Francescos-MacBook-Pro First compilation % ls
Compilation commands.txt      lib.h
bill.c                         program.c
fred.c

francescodenu@Francescos-MacBook-Pro First compilation % gcc -c bill.c
francescodenu@Francescos-MacBook-Pro First compilation % gcc -c fred.c
francescodenu@Francescos-MacBook-Pro First compilation % gcc -o my_exe program.c bill.o fred.o
francescodenu@Francescos-MacBook-Pro First compilation % ./my_exe
bill: you passed Hello World
fred: you passed 99999
francescodenu@Francescos-MacBook-Pro First compilation % ls -l
total 128
-rw-r--r--@ 1 francescodenu  staff    282 Sep 28 19:00 Compilation commands.txt
-rw-r--r--@ 1 francescodenu  staff     87 May 20  2007 bill.c
-rw-r--r--  1 francescodenu  staff    752 Sep 28 19:01 bill.o
-rw-r--r--@ 1 francescodenu  staff     85 May 20  2007 fred.c
-rw-r--r--  1 francescodenu  staff    752 Sep 28 19:01 fred.o
-rw-r--r--@ 1 francescodenu  staff    111 May 20  2007 lib.h
-rwxr-xr-x  1 francescodenu  staff  33523 Sep 28 19:01 my_exe
-rw-r--r--@ 1 francescodenu  staff    109 Sep 28 18:58 program.c
```

gcc -c crea il file object, ovvero trasforma il codice in linguaggio macchina (non ancora eseguibile) che sarà integrato con altri .o per creare l'eseguibile di program.c

Utilizzando l'opzione -c si specifica che si vuole solo il file object e non l'eseguibile.

Per creare l'eseguibile bisogna compilare program.c, che contiene il main, ed effettuare il linking. Il linking si occupa di associare al prototipo delle funzioni, contenute in lib.h e importate nel file program.c, con le loro definizioni, che sono compilate all'interno dei file .o. Con l'opzione -o ovviamente si specifica il file di output.

FunFact: Compilazione e linking sono due fasi che oggi sono fatte dallo stesso compilatore, ma una volta erano due programmi separati.

Quando si compila le istruzioni vengono tradotte in byte; volendo si può utilizzare l'opzione -S che mostra le istruzioni in assembly per renderle comprendibili agli esseri umani.

Seguono una serie di **opzioni utili** in fase di compilazione:

```
A few useful compilation commands

# To see the values of implicit rules
make -p | bbedit

# To see the include directories
clang -x c -v -E /dev/null # macOS
gcc -x c -v -E /dev/null # Linux

gcc -E file.c # To see the effect of preprocessor directives
gcc -E -P file.c # To see the effect of preprocessor directives (without comments)
gcc -E -dM ls1.c # To list all "define"
gcc -H # To see the hierarchy of the include files
gcc -E -P -nostdinc ls1.c # To see the hierarchy of the include files (without system includes)

gcc -S ls1 # To see the assembler (Linux and macOS)
otool -tv ls1 # To see the assembler (macOS)

nm ../lib/libapue.a # To see the symbols of a library
nm ls1.o # To see the symbols of an object file
```

NOTA: con -I si dice dove sono i file di include, ovvero i .h

Con l'opzione -P evitiamo i commenti

È possibile però, oltre che includere il codice staticamente come appena fatto, includerlo a runtime dinamicamente.

Alla fine della precedente compilazione, non ci sono più riferimenti a funzione perché diventa una sequenza di istruzioni di processori tutte incluse in un unico file. Nel caso dinamico lo scioglimento dei pezzi di libreria viene fatto man mano quando serve, il che ovviamente alleggerisce l'eseguibile.

Consideriamo ora la cartella lib_example.

useprova.c utilizza una funzione definita in libprova.c, la quale intestazione è in libprova.h. Si nota che dentro useprova.c, sono presenti due inclusioni del file libprova.h, ma grazie all'utilizzo delle istruzioni del preprocessore nel file libprova.h, l'inclusione viene fatta solo una volta.

Siccome questa libreria l'abbiamo costruita noi, abbiamo anche il codice sorgente.

Altrimenti in genere si ha solo il .lib contenente i prototipi, e un file con le funzioni compilate.

Quale è il problema nelle librerie statiche? Il fatto che spesso esse vengano aggiornate e quindi si necessita della ricompilazione del codice.

Tramite le dynamic library, il richiamo al codice della libreria succede a runtime: quando è necessario il file viene caricato nella memoria.

Questa operazione viene fatta costantemente con le librerie di sistema, infatti moltissimi aggiornamenti dei sistemi operativi riguardano le librerie di sistema; in questo modo siccome il file è aggiornato dal sistema operativo e quando viene chiamato dinamicamente,

arriva corretto. Se invece la libreria fosse inglobata staticamente all'interno dell'eseguibile, sarebbe necessario ricompilare il codice ogni volta che cambia il contenuto di una libreria. Da notare però che il prototipo delle funzioni non può essere cambiato, altrimenti bisognerebbe ricompilare il programma, in quanto questi sono negli header di cui viene fatto l'include.

Le librerie statiche sono pericolose, perché si rende più complicato poter aggiustare bug nei programmi (dovendoli ricompilare per intero); alcuni sistemi operativi, infatti, le hanno abolite (per quanto riguarda le librerie di sistema).

Per esempio, lib.c (che si chiama anche in maniera diversa) è stata bandita in versione statica da macos.

I file compilati delle librerie combinati tra loro compongono un archivio (.a).

Della funzione printf, per esempio, non abbiamo il codice sorgente, abbiamo direttamente il compilato. Dove è il compilato? Nelle librerie. Le librerie sono infatti collezioni di oggetti. La famosa Clibrary è una collezione di molti oggetti, che contengono le definizioni delle funzioni (in codice macchina).

Da notare che precedentemente l'inclusione della nostra libreria era effettuata in maniera statica; invece, quelle di sistema erano incluse dinamicamente: il sistema sa dove andare a pescare le librerie di sistema e effettuare il linking dinamico autonomamente.

```

Version 202110040942

# Static Library (Linux and MacOS)

gcc -c libprova.c # Builds the object file
ar rcs libprova.a libprova.o # Adds the object to the library (creates the library if not existing)
gcc -Wall -g -c useprova.c -o useprova.o # Builds the main object file
gcc -g -o useprova useprova.o -L. -lprova # Links object files and library

# Dynamic Library (Linux)

gcc -fPIC -Wall -g -c libprova.c
gcc -g -shared -Wl,-soname,libprova.so.0 -o libprova.so.0.0 libprova.o -lc
ln -sf libprova.so.0.0 libprova.so.0
ln -sf libprova.so.0 libprova.so # All the 4 lines to build the shared library

# Dynamic Library (MacOS)

gcc -dynamiclib libprova.c -o libprova.dylib # Builds the shared library

# Use of the Dynamic Library

gcc -Wall -g -c useprova.c -o useprova.o
gcc -g -o useprova useprova.o -L. -lprova

LD_LIBRARY_PATH=`pwd` ./useprova # Variable definition needed only in Linux
LD_LIBRARY_PATH=`pwd` ldd useprova # Same for ldd command (Linux)

# The -static flag may be added in compilation to use only static libraries

# Mac OS X does not have a static C lib (libSystem.B.a).
# So in most cases the -static flag cannot be used in compilation.

# Linux instead does have a static version of the System Library
# /usr/lib/x86_64-linux-gnu/libc.a
# While the default location for libraries is /usr/lib, the linker knows about
# this directory and finds the library anyway
# see: https://askubuntu.com/questions/52617/what-is/usr-lib-i386-linux-gnu-for

# Steps to build the static library in Linux and to compile statically:
gcc -c libprova.c
ar crv libtest.a libprova.o
gcc -c useprova.c
gcc -o useprova useprova.o libtest.a
gcc -static -o useprova useprova.o libtest.a

# Very good intro to libraries for Linux is in
# http://tldp.org/HOWTO/Program-Library-HOWTO/introduction.html
# See also: https://medium.com/@dkwok94/the-linking-process-exposed-static-vs-dynamic-libraries-977e92139b5f

```

The screenshot shows a code editor with two tabs open:

- useprova.c**: This file contains C code that includes the `libprova.h` header and defines a `main` function. It uses the `print_hello()` function from the library.
- libprova.c**: This file contains the implementation of the `print_hello()` function, which prints the value of variable `i`.

```

useprova.c
Documents/UNISALENTOMAGISTRALE/SYSTEM_NETWORK_PROGRAMMING/lib_example/useprova.c
/*
 * useprova.c -- uses print_hello() function from libprova.so */
#include "libprova.h"
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include "libprova.h"

/*A program to try a shared library.*/

int c=7, f=5;
int d;
int e;

int
main(int argc,char **argv) {
    print_hello();
    e=c*fatoi(argv[1]); /* ricordarsi di passare un argomento */
    printf("la variabile e vale: %d\n", e);
    return 0;
}

libprova.c
Documents/UNISALENTOMAGISTRALE/SYSTEM_NETWORK_PROGRAMMING/lib_example/libprova.c      (no function selected)
/*
 * libprova.c -- provides print_hello() function for libprova.so */
#include <stdio.h>

void print_hello(void) {
    int i;
    i=346;
    printf("la variabile i vale %d (messaggio stampato da una funzione di libreria).\n", i);
}

```

```
libprova.h
documents/UNISALENTOMAGISTRALE/SYSTEM_NETWORK_PROGRAMMING/lib_
#ifndef LIBHELLO_H_
#define LIBHELLO_H_

void print_hello(void);

#endif /* LIBHELLO_H_ */
```

```
francescodenu@Francescos-MacBook-Pro lib_example % gcc -c useprova.c
francescodenu@Francescos-MacBook-Pro lib_example % ar rcs libprova.a libprova.o
francescodenu@Francescos-MacBook-Pro lib_example % gcc -Wall -g -c useprova.c useprova.o
clang: warning: useprova.o: 'linker' input unused [-Wunused-command-line-argument]
francescodenu@Francescos-MacBook-Pro lib_example % ./my_exe useprova.o -L. -lprova
francescodenu@Francescos-MacBook-Pro lib_example % ls
libprova.a    libprova.h    my_exe        useprova.o
libprova.c    libprova.o    useprova.c
francescodenu@Francescos-MacBook-Pro lib_example % ./my_exe 3
la variabile i vale 346 (messaggio stampato da una funzione di libreria).
la variabile e vale: 105
francescodenu@Francescos-MacBook-Pro lib_example %
```

COMPILAZIONE STATICÀ CON CREAZIONE DI UN ARCHIVIO

Prima si costruiscono i due file oggetto con l'opzione -c.
ar rcs permette di costruire la libreria. Si potrebbero anche aggiungere più file alla libreria, ma bisogna fare attenzione ai nomi delle funzioni poiché ovviamente non ce ne possono essere due uguali.

Nota: -Wall significa che voglio i warning nella compilazione.

L'ultimo comando linka manualmente le librerie. Da notare che quando si linka, per passare la libreria, chiamata libprova.a, si scrive -l(nomelibreriasenza"lib"): infatti la libreria si chiamava libprova.a e si scrive -lprova (togliendo anche l'estensione).

Quando si usano librerie di sistema non bisogna specificare dove stanno.

Questa compilazione è stata statica per quanto riguarda la nostra libreria, e dinamica per la libreria di sistema. Su macos non ho scelta. Su linux avrei potuto linkarle entrambe staticamente.

Ora, dove è la funzione atoi utilizzata nel programma? Deve esserci il prototipo da qualche parte. In quale file di intestazione è dichiarata? Si va nelle pagine di manuale della funzione.

RICORDA: le pagine di manuale sono divise in sezioni.

1 -> comandi su riga di comando

2-> sezione delle system call

3-> sezione delle librerie di sistema

Atoi quindi sicuramente non è una systemcall, allora si cerca nella sezione delle librerie di sistema. Si vede che essa è nella libreria libc.

COMPILIAMO ORA LINKANDO LIBRERIE DINAMICAMENTE

NOTA: DEVI FARLO SOLO SU LINUX

Su MAC:

per creare la libreria dinamica è sufficiente
gcc -dynamiclib libprova.c -o libprova.dylib

Il linking si fa allo stesso modo del caso statico con un archivio,
gcc -o use_prova useprova.c -L. -lprova

Partendo ora dall'eseguibile, ci chiediamo **quali librerie questo sono usate:**
otool -L nomeprogrammacompilato

Vengono ovviamente così mostrate le librerie dinamiche incluse, in quanto quelle statiche è come se fossero semplicemente copiate e incollate nel codice. Da notare l'estensione utilizzata, su linux si usa ldd, invece qui le estensioni sono .so (anche se su mac vedo una cosa strana come dylib).

Nota che anche le librerie dinamiche sono pezzi di codice che usano altre librerie dinamiche, potresti chiamare questo comando su una libreria e vedere quante ne usa.

NOTA SICUREZZA: questo comando può essere usato per controllare se nel tuo eseguibile è presente una libreria con dei bug

Costruiamo ora una libreria dinamica su **LINUX**:

-lc alla fine significa che usa la libreria c

Ln -sf serve per aggiornare librerie senza aggiornare gli eseguibili:

Grazie ai link simbolici quando viene chiamata la libreria si viene portati ai file giusti, in caso di problemi è sufficiente infatti cambiare solo i link ai file.

Se cambia il numero di versione piccolo è ok, se cambia il numero di versione grande devi ricompilare.

Nota: per default si compila dinamico se non si dà l'opzione static.

Ora quando viene eseguito il programma, bisogna prima specificare dove è la libreria.

Quando poi quando siamo sicuri funzioni, la installiamo della directory giusta (su linux /usr/lib) e verrà trovata automaticamente.

Infatti, se si usa il comando sull'eseguibile per vedere quali librerie sono usate e dove sono, non troverà la nostra libreria se non specifichiamo dove sia.

Per questo si usa il costrutto:

Definizionevariabile lanciocomandocondefinizioneveriable

Mac automaticamente cerca la libreria nella current working directory, invece su linux bisogna specificare. Se però si sposta il programma bisogna ovviamente specificare dove sia... a meno che non sia di sistema e sappia già dove trovarla.

RIASSUNTO PRATICO COMPILAZIONI:

Supponiamo di avere un file con il main che include una nostra libreria. Questa libreria sarà un .h e conterrà i prototipi delle funzioni utilizzate, necessari per permettere la compilazione. La definizione di queste funzioni sarà all'interno di altri file oggetto già compilati (.o)

Per poter compilare bisogna prima compilare ogni singolo file con l'opzione -c per creare il file object compilato .o

Se si vuole si può creare un archivio con il comando

```
ar rcs libnomearchivio.a file1.o file2.o file3.o
```

Ora che tutto è stato compilato, bisogna fare il linking:

```
gcc -o nomefileoutput fileconmain.o (oppure anche il .c) file1.o file2.o file3.o
```

oppure si può passare l'archivio

```
gcc -o nomefileoutput fileconmain.o -L. -Inomearchiviosenzainiziolibesenzaestensione
```

In questo modo abbiamo linkato staticamente. Da notare che le librerie di sistema sono già compilate e presenti in /usr/lib, mentre i file di intestazione sono su /usr/include

Ci sono dei vantaggi visti sopra nel fare il linking dinamico.

Per creare libreria dinamiche:

Su MAC:

```
gcc -dynamiclib libreria.c -o libnomelibreria.dynamiclib
```

Per il linking si procede come se avessimo un archivio

```
gcc -o myexe nomefileconmain.c/o -L. -Inomelibreriasenzainiziolibestensione - Inomelibreria2senzalibeestensione
```

Però per utilizzarla, bisogna specificare dove si trova la libreria quando si lancia l'eseguibile.

Su mac si da per scontato possa stare nella cwd.

MAKE

Make è un utility, sviluppata sui sistemi operativi della famiglia UNIX, che automatizza il processo di creazione di file che dipendono da altri file, risolvendo le dipendenze e invocando programmi esterni per il lavoro necessario. Esso permette una più comoda compilazione dei file.

Nota: su macOS le applicazioni sono delle directory che contengono i file necessari per fare funzionare il programma.

Security note: su macOS i programmi e le librerie per sviluppare sono all'interno di xcode, poiché sono la prima cosa che gli attaccanti cercano dopo aver preso il controllo della macchina. In questo modo non sono installati su tutti i computer se non sono necessari, a differenza dei sistemi linux.

L'opzione -i permette di ignorare gli errori e continuare la compilazione.

Quando si scrive make, bash cerca nella cwd un file chiamato " Makefile ", altrimenti " makefile ". Se si specifica -f si prende quel file in particolare.

La struttura di un Makefile è:

Target:lista dei prerequisiti

Regole (cosa fare per costruire dai prerequisiti il target, comandi da shell)

Esempio di funzionamento di un banale makefile:

```
Test.txt: pippo.txt pluto.txt
          cat pippo.txt pluto.txt > test.txt

pippo.txt:
          echo "sono pippo " > pippo.txt

pluto.txt:
          echo "sono pluto " > pluto.txt

clean:
          rm pippo.txt pluto.txt
```

Si parte dal primo target: test.txt. I prerequisiti sono pippo.txt e pluto.txt, se essi non sono già esistenti, si vede cosa è previsto dal makefile per creare pippo.txt e pluto.txt.

Quindi prima si crea pippo.txt scrivendo "sono pippo" nel file, e lo stesso per pluto.txt.

Quando i prerequisiti sono soddisfatti, si torna al primo target e si esegue, creando un file test.txt contenente "sono pippo sono pluto".

Si può usare il target clean che rimuove i file, ma bisogna specificarlo quando si usa make (make clean), altrimenti verrà ignorato se non viene già chiamato da nessun altro target (per esempio essendo nei prerequisiti)

Ci sono una serie di abbreviazioni importanti presenti in [Automatic Variables in GNU MakeURL](#), per esempio, \$@ è la scorciatoria per scrivere il target

Diamo ora un'occhiata al makefile di apue:

```
DIRS = lib intro sockets advio daemons datafiles db environ \
       fileio filedir ipc1 ipc2 proc pty relation signals standards \
       stdio termios threadctl threads printer exercises

all:
    for i in $(DIRS); do \
        (cd $$i && echo "making $$i" && $(MAKE) ) || exit 1; \
    done

clean:
    for i in $(DIRS); do \
        (cd $$i && echo "cleaning $$i" && $(MAKE) clean) || exit 1; \
    done
```

All'interno del primo loop: si lancia ogni volta una subshell (poiché il comando è tra parentesi) e viene fatto cd nella directory passata (elencate in DIRS) e viene chiamato il comando make (\$MAKE corrisponde alla directory di make). In pratica viene chiamato il comando make in tutte le directory elencate.

\$\$ serve a fare l'escape di \$ con il \$

Vediamo allora per esempio il makefile di intro:

```
ROOT=..
PLATFORM=$(shell $(ROOT)/systype.sh)
include $(ROOT)/Make.defines.$(PLATFORM)

PROGS = getcpuc hello ls1 mycat shell1 shell2 testerror uidgid

all:    $(PROGS)

%: %.c $(LIBAPUE)
        $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS) $(LDLIBS)

clean:
        rm -f $(PROGS) $(TEMPFILES) *.o

include $(ROOT)/Make.libapue.inc
```

Si associa alla variabile PLATFORM il nome del sistema operativo \$(shell \$(ROOT)/systype.sh), Dove systype.sh è

```

case `uname -s` in
"FreeBSD")
    PLATFORM="freebsd"
;;
"Linux")
    PLATFORM="linux"
;;
"Darwin")
    PLATFORM="macos"
;;
"SunOS")
    PLATFORM="solaris"
;;
*)
    echo "Unknown platform" >&2
    exit 1
esac
echo $PLATFORM
exit 0

```

PROGS contiene i nomi dei programmi che vorremmo costruire.

Include è come l'include di c, e dice di mettere al posto della riga, il file che si chiama/Make.defines.macos (o il particolare sistema operativo) come si può vedere in apue.3e esiste un file per ogni sistema operativo.

-  Make.defines.freebsd
-  Make.defines.linux
-  Make.defines.macos
-  Make.defines.solaris

```

CC=gcc
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) -c
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
LDFLAGS=
LDDIR=-L$(ROOT)/lib
LDLIBS=$(LDDIR) -lapue $(EXTRALIBS)
CFLAGS=-ansi -I$(ROOT)/include -Wall -DMACOS -D_DARWIN_C_SOURCE $(EXTRA)
RANLIB=ranlib
AR=ar
AWK=awk
LIBAPUE=$(ROOT)/lib/libapue.a

```

Ciò che avviene in questo tipo di file è le ridefinizioni di alcune variabili usate da Make per lo specifico sistema operativo. Ci sono anche una serie di macro predefinite, usate in alcune regole predefinite.

Queste specificano per esempio quali opzioni utilizzare nel compilare un file, etc.

In questo caso queste macro vengono modificate, anche se probabilmente non aggiungono nulla di nuovo... così è però facile poter modificare il tipo di operazioni da compiere. Make chiamerà queste variabili per eseguire le operazioni sui file, per compilare, per linkare, per costruire archivi, etc.

Per esempio, CC dice quale compilatore usare, CFLAGS quali opzioni usare quando si compila, LINK si occupa del linking, AR il comando da usare per creare archivi e così via.

Per vedere queste regole implicite, si usa l'opzione -p (make -p). In questo caso, si vedrà che quando Make ha come target un file.c, questo sarà il comando che userà.

Esistono infatti serie di regole implicite, si possono vedere con -p, ovvero make -p
Qui, per esempio, si trova la regola per COMPILE.c

```
francescodenu@Francescos-MacBook-Pro ~ % make -p | grep "COMPILE.c ="  
make: *** No targets specified and no makefile found. Stop.  
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
```

Per esempio, nel makefile di intro, per avere all, servono tutti i prerequisiti (presenti in PROGS). I prerequisiti in questo caso devono essere i compilati, e make sa già per conto suo cosa fare: prendere le macro predefinite (che in questo caso abbiamo leggermente modificato) per eseguire il comando

Per esempio, infatti, per fare la compilazione tramite make di First Compilation, è sufficiente creare un file chiamato Makefile con al suo interno

```
program: program.c fred.o bill.o
```

Vedendo le estensioni dei file capisce che si vogliono ottenere i compilati, e per ciascuno ha la regola già salvata in memoria. Make capisce che se vuoi program e hai program.c, deve compilare e fare poi il linking con i file fred.o e bill.o

Da notare però che nel makefile di intro, la riga su cosa fare è specificata e non presa in automatico. Questo per poter avere la possibilità di aggiungere opzioni manualmente.

Vediamo ora il Makefile di lib

```
ROOT=..  
PLATFORM=$(shell $(ROOT)/systype.sh)  
include $(ROOT)/Make.defines.$(PLATFORM)  
  
LIBMISC = libapue.a  
OBJS    = bufargs.o cliconn.o clrfl.o \  
          daemonize.o error.o errorlog.o lockreg.o locktest.o \  
          openmax.o pathalloc.o popen.o prexit.o prmask.o \  
          ptyfork.o ptyopen.o readn.o recvfd.o senderr.o sendfd.o \  
          servaccept.o servlisten.o setfd.o setfl.o signal.o signalintr.o \  
          sleepus.o spipe.o tellwait.o ttymodes.o writen.o  
  
all:    $(LIBMISC) sleep.o  
  
$(LIBMISC): $(OBJS)  
    $(AR) rv $(LIBMISC) $?  
    $(RANLIB) $(LIBMISC)  
  
clean:  
    rm -f *.o a.out core temp.* $(LIBMISC)  
  
include $(ROOT)/Make.libapue.inc
```

Come si può vedere le istruzioni servono a costruire una libreria, infatti il target è \$(LIBMISC), ovvero libapue.a

OBJS contiene tutti i compilati che bisogna aggiungere alla libreria

Il target è una libreria (.a), e i prerequisiti sono i file .o che la compongono.

Grazie alle regole prestabilite, sa cosa fare quando si vuole un .a con tanti .o

\$? Sono tutti i prerequisiti più nuovi del target, ovvero prende tutti gli oggetti che sono più nuovi della libreria e li rimette dentro per aggiornare la libreria. Tutto il make si basa sulla data di modifica, su modification date. Non si fa nulla se tutto è aggiornato.

Ovviamente però se si cambia un file di quelli delle librerie, di cui lui fa i .o, ricompilerà tutti i file che usano la libreria. Viene aggiustato solo un pezzo della libreria, però di conseguenza cambia la libreria, e quindi tutti quelli che usano la libreria devono ricompilare. La libreria come file è cambiata nonostante in realtà solo un libro sia cambiato.

ranlib Aggiunge un indice a ogni funzione nella libreria

Iniziamo analizzando il codice di ls1.c

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR          *dp;
    struct dirent  *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Ricordo innanzitutto che la dimensione di un indirizzo della ram dipende all'architettura, può essere a 32 o 64 bit.

In questo caso dp contiene l'indirizzo a una variabile di tipo DIR, invece dirp è il puntatore a una struttura dirent.

Vogliamo ottenere ora informazioni riguardo la struttura dirent. Poiché fa parte di una libreria di sistema, la possiamo conoscere tramite il manuale (man dirent)

```
struct dirent { /* when _DARWIN_FEATURE_64_BIT_INODE is NOT defined */
    ino_t      d_ino;           /* file number of entry */
    __uint16_t  d_reclen;       /* length of this record */
    __uint8_t   d_type;         /* file type, see below */
    __uint8_t   d_namlen;       /* length of string in d_name */
    char       d_name[255 + 1]; /* name must be no longer than this */
};
```

Da notare che c'è anche un file dirent.h nella cwd che dice questo...

```
#if !__DARWIN_64_BIT_INO_T
struct dirent {
    ino_t d_ino;           /* file number of entry */
    __uint16_t d_reclen;   /* length of this record */
    __uint8_t d_type;      /* file type, see below */
    __uint8_t d_namlen;    /* length of string in d_name */
    char d_name[__DARWIN_MAXNAMLEN + 1]; /* name must be no longer than this */
};
#endif /* !__DARWIN_64_BIT_INO_T */
```

In realtà però, essendo una libreria di sistema, per conoscere informazioni sulla struttura dirent si dovrebbe fare una grep ricorsiva tra i file include di sistema:

grep -rw "struct dirent" directory

Su linux la directory è /usr/include

Dove sono i file di libreria su Mac? In teoria le librerie su mac dovrebbero essere in applications/xcode... ma a me funziona anche senza xcode. Infatti, facendo un find di stdio ricorsivo in tutto il sistema con find / -name stdio si nota che molti prototipi sono in /Library/Developer/CommandLineTools/SDKs/MacOSX11.3.sdk/usr/include/

Il file dirent.h è infatti in

/Library/Developer/CommandLineTools/SDKs/MacOSX11.3.sdk/usr/include/sys/dirent.h

```
#if !__DARWIN_64_BIT_INO_T
struct dirent {
    ino_t d_ino;           /* file number of entry */
    __uint16_t d_reclen;   /* length of this record */
    __uint8_t d_type;      /* file type, see below */
    __uint8_t d_namlen;    /* length of string in d_name */
    char d_name[__DARWIN_MAXNAMLEN + 1]; /* name must be no longer than this */
};
#endif /* !__DARWIN_64_BIT_INO_T */

#pragma pack()

#define __DARWIN_MAXPATHLEN      1024

#define __DARWIN_STRUCT_DIRENTRY { \
    __uint64_t d_ino;           /* file number of entry */ \
    __uint64_t d_seekoff;       /* seek offset (optional, used by servers) */ \
    __uint16_t d_reclen;        /* length of this record */ \
    __uint16_t d_namlen;        /* length of string in d_name */ \
    __uint8_t d_type;          /* file type, see below */ \
    char     d_name[__DARWIN_MAXPATHLEN]; /* entry name (up to MAXPATHLEN bytes) */ \
}

#if __DARWIN_64_BIT_INO_T
struct dirent __DARWIN_STRUCT_DIRENTRY;
#endif /* __DARWIN_64_BIT_INO_T */
```

Nota che nel file in cui è definito dirent, c'è prima una #if che controlla se il pc è a 64 bit. Se non lo è viene definita la struttura, altrimenti si crea prima una macro che contiene una stringa utilizzata successivamente per costruire la struttura per i pc a 64 bit.

Dove sta ora err_quit? Si vede che non è di libreria di sistema perché non è in dirent.h, o in sottolibrerie, allora si cerca in apue.h

In ls1, c'è la funzione opendir: dopo che gli passi la directory della directory, restituisce un puntatore ad essa. In seguito, c'è un loop: passa il puntatore alla directort (dp) alla funzione

readdir, che legge le entry della directory a una a una, e restituisce una struttura che si chiama dirp di tipo dirent.

Legge finchè la lettura non restituisce null, ovvero quando ha finito la lista degli elementi della directory. In seguito, si chiude la directory.

Viene poi stampato il campo d_name della struttura dirent.

Se vogliamo modificare questo programma per far sì che veda il numero di inode della directory entry (dato salvato nella struttura dirent), ovvero l'equivalente di ls -i

Bisogna solo aggiungere la print `printf("%lld\n", dirp->d_ino);`

Da notare che l'inode dovrebbe essere un unsigned long int. In questo caso, però, come si vede dalla struttura sopra, è un `_uint64_t`

Dove è ora questa definizione? Cerca `_uint64_t` nella directory dove ci sono tutti i file di include, e così si trova che deriva da typedef ed è un unsigned long long

Ora, per ricompilare devi compilare il .c :

gcc -ansi -I./include -Wall -DMACOS -D_DARWIN_C_SOURCE \${1}.c -o \${1} -L./lib -lapue

Dove, al posto di \${1} metti il file .c da compilare, ansi indica il tipo di c, -I./ indica la directory con i prototipi .h, -Wall ignora gli errori. -L indica l'archivio. Poi ci sono altre opzioni varie ma vabbe.

NOTA: su mac ci sono degli attributi estesi che sono indicati con una @ tra gli attributi visibili con il comando ls -la. Per vedere questi attributi si lancia il comando ls -l@

Per eliminare un attributo speciale, xattr -f nomeattributo nomefile

Ora, opendir è una systemcall o una funzione?? In C le systemcall hanno lo stesso prototipo delle funzioni, allora per capirlo utilizziamo il manuale (man opendir).

In generale per capire la distinzione tra una chiamata a funzione e una systemcall si capisce con l'assembly, vedendo cosa succede quando viene chiamata la funzione, ovvero se il comportamento è quello di una systemcall o no. In ogni caso essite un file chiamato syscall.h per vedere tutte le systemcall.

I file oggetto contengono il codice binario eseguibile, ma nel momento in cui si chiama la funzione rimane il suo nome finché non è fatto il linking che lo sostituisce con le istruzioni. Si potrebbe notare questo con una grep sui .o, oppure ancora meglio utilizzare il comando nm file.a sull'archivio:

```
francescodenu@Francescos-MacBook-Pro lib % nm libapue.a | grep err_sys
          U _err_sys
00000000000018c T _err_sys
```

Per esempio, qui si vede che la definizione di err_sys si trova a un determinato offset dall'inizio libreria. Il codice è quello che inizia con la T (sta per testo, e si intende l'eseguibile). Gli offset sono però nell'ambito del libro, ovvero rispetto all'inizio del libro.

```
[francescodenu@Francescos-MacBook-Pro lib % nm libapue.a | head

libapue.a(bufargs.o):
    U __stack_chk_fail
    U __stack_chk_guard
0000000000000000 T _buf_args
    U _strtok
0000000000000018 s l_.str
0000000000000000 t ltmp0
0000000000000018 s ltmp1
00000000000000120 s ltmp2
```

Infatti qui dice che a partire dal libro libapue.a, le definizioni delle funzioni presenti sono a questo offset dall'inizio

Quando si apre un file, viene salvato l'indirizzo e viene fatto corrispondere un numero, a cui si fa riferimento per leggere il file. Questo si chiama **File Descriptor**. Quando si chiude un file con close, il numero corrispondente al file è liberato. Ogni volta che un file viene aperto, il numero più piccolo possibile gli viene assegnato, e viene conservato fino alla chiusura.

Analizziamo ora il codice di figura 1.4, mycat.c

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int      n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

C'è un loop infinito che essenzialmente esegue più volte l'istruzione if -> write
Viene eseguita la chiamata read (che è una systemcall) con tre parametri: il primo è il file da cui si vuole leggere, poi il puntatore all'array (dove sono salvati i byte letti), e il terzo parametro è quanto si vuole leggere.

La read vuole leggere 4096 byte, ma se un file è più piccolo ritorna tranquillamente i valori che ha letto; infatti, è una systemcall che fa eccezione: invece di ritornare 0 se va a buon fine, si ritorna il numero di byte letti. Il ciclo va avanti finché il numero di byte letti è maggiore di 0.

Quando si apre un file, il processo di apertura riserva un'area di memoria che indica fino a che punto è stato letto il file.

Per vedere i file aperti da ciascun processo: **lsof** che mostra tutti i file di tutti i tipi che il processo sta usando... compresa la directory che viene vista come un file, e tutte le librerie dinamiche che sta utilizzando.

Insomma, quando il file regolare è stato letto tutto, ovviamente la read ritorna zero, e la condizione non sarà più soddisfatta.

A ogni lettura si legge dentro l'area di memoria e si mette ciò che c'è dentro buf.

Poi c'è la write, che somiglia molto a una read (sia come argomenti che come valore ritornato).

La if stabilisce giustamente che se si vogliono scrivere n byte, ma ne viene scritto un numero minore (si sa grazie al valore ritornato), significa che c'è stato un errore.

In questo caso STDIN_FILENO e STDOUT_FILENO sono file descriptor. Cosa sono questi numeri? Chi glieli da? Abbiamo detto che la open ritorna il file descriptor, ma in questi casi chi glieli ha dati? Che valori hanno?

Cercando all'interno dei prototipi di sistema troviamo che il valore è zero.

```
francescodenu@Francescos-MacBook-Pro include % grep -rw "STDIN_FILENO" /Library/Developer/CommandLineTools/SDKs/MacOSX11.3.sdk/usr/include
/Library/Developer/CommandLineTools/SDKs/MacOSX11.3.sdk/usr/include/unistd.h:#define      STDIN_FILENO    0          /* standard
input file descriptor */
/Library/Developer/CommandLineTools/SDKs/MacOSX11.3.sdk/usr/include/asl.h: * asl_log_descriptor(c, m, ASL_LEVEL_NOTICE, STD
IN_FILENO, ASL_LOG_DESCRIPTOR_READ);
francescodenu@Francescos-MacBook-Pro include %
```

Ma come fa a conoscere questo valore?

Ricorda che ogni processo nasce dalla fork di un processo parent. Successivamente il child, identico al parent, si evolve facendo una exec e procedendo per la sua strada.

Nel nostro caso specifico, quando eseguiamo il codice, il parent del nostro processo sarà Bash. Questo tramite una fork tira fuori un altro bash, e poi tramite l'exec farà girare il nostro codice.

Come è possibile ora che questo processo si ritrovi dei file descriptor funzionanti senza aver fatto una open? La open l'ha fatta il parent. I child ereditano i file descriptor del parent. In questo caso chi ha fatto la open davvero? Potrebbe essere bash, oppure potrebbe anche essere a sua volta suo padre.

Da notare che mycat.c legge dallo stdin e scrive sullo stdout. Lo shell ci permette di reindirizzare lo stdout tramite > su un file. Questa ridirezione chi la fa, e come la fa? Quando si lancia il programma con la ridirezione, dopo la fork e prima della exec, è chiuso il file descriptor 1 ed in questo modo si crea un buco nell'elenco dei file descriptor disponibili.

Poiché quando serve un file descriptor si assegna il valore più piccolo disponibile, quando si fa la open di outfile.txt, viene infatti assegnato 1. Invece di trovare il vecchio file quando si va a scrivere a 1, si trova il nuovo (outfile.txt) che è stato aperto e ha trovato il valore 1 disponibile. In questo modo, utilizzando lo stesso valore del file descriptor, è stato ridirezionato l'output.

Ricordando che su linux **everything is a file**, quale è il file che rappresenta il terminale? Per scoprirlo il comando è tty.

```
francescodenu@Francescos-MacBook-Pro include % tty
/dev/ttys000
```

Ovviamente se non si ridirezionasse l'output quando si lancia mycat, si scriverebbe da terminale e si l'output sarebbe il terminale stesso. Questo perché bash legge dallo standard input e scrive sullo standard output.

Infatti, se si fa cat > /dev/ttys00, ciò che si scrive arriva sul terminale.

RICORDA: i file descriptor sono relativi al processo.

Ricordando che everything is a file.....

Il file è un meccanismo, riassunto dalle operazioni: apri leggi scrivi chiudi

Il device driver dice cosa significa open, read, write, close.

Analizziamo ora l'esempio 1.7 che in meno di 30 righe crea uno shell.

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE]; /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% "); /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) { /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

I due %% perché % è l'escape del %

fgets è una funzione della std library, che aspetta che tu scriva qualcosa e dia invio, mettendo ciò che hai scritto dentro lo spazio di memoria buf. In questo caso legge dallo stdin.

Dove è definito invece MAXLINE? Troviamo che è dentro apue.h, ed è 4000 caratteri.

Da dove viene la parola stdin?

La stdio library usa un modo diverso di accedere ai file.... La std library usa fopen che apre un file ma non restituisce il file descriptor, ma un puntatore a una struttura che si chiama FILE. Questa struttura è ciò che serve per accedere al file. Per utilizzare il file si passa allora il puntatore alla struttura invece del file descriptor. Tra le operazioni delle stdio library, c'è la definizione di stdin. Essa è una struttura di tipo file che però accede allo standard input.

Fgets restituisce una stringa, che è quello che hai scritto. Questa stringa termina con il carattere a capo (poiché necessario per dare l'invio). La if controlla se l'ultimo carattere è \n e lo sostituisce con null.

Dopo accade la fork. Essa viene chiamata una volta e ritorna due diversi valori, uno nel parent e uno nel child. Grazie alla fork il codice capisce chi è il parent e il child: ritorna al parent il pid del child e al child il valore di ritorno è 0; ciò fa capire ad esso di essere un child.

Dopo la exec il child si distingue dal parent ed esegue il proprio codice, ovvero quello contenuto in buf. Il codice del child allora da qui in poi cambia, e solo il parent proseguirà con l'esecuzione di questo codice.

Waitpid è una funzione che aspetta che il child finisca di eseguire il suo codice per ritornare un valore al parent, in modo che quest'ultimo sappia che il child ha eseguito il suo codice. Il parent aspetta la fine dell'esecuzione del child, e poi ha informazioni su come è andata l'esecuzione del figlio, grazie a &status.

Cosa succede dopo la exec? Vedi "come bash esegue i processi" su libro di sistemi operativi. Spazio di memoria (nella ram ovviamente):

I processi sono i programmi in esecuzione. Un programma quando esegue un processo ha un suo spazio di memoria. I processi vedono questo spazio di memoria (virtuale) grande quanto permette l'indirizzamento della macchina su cui siamo. Se siamo su una macchina a 32 bit, lo spazio di memoria è 2^{32} bit (4gb). Se siamo invece su una macchina a 64 bit è 2^{64} , che è molto di più. Un processo è così pazzo da pensare di avere tutto questo spazio di memoria a sua disposizione. Cosa c'è in questo spazio? Sicuramente il codice da eseguire, il "testo", i bit che vanno in pasto al processore. Poi ci sono le variabili, lo stack, e ciò di cui il processo ha bisogno.

In sostanza: cosa succede quando è eseguito un programma? Il processo iniziale che lancia questo eseguibile fa una fork, creando una copia identica di se stesso ma con un altro pid. In questo caso, quindi, ovviamente viene creato un altro bash. Il parent rimane in sospeso ma continua a vivere comunque. Viene poi chiamata una exec: questa distingue i due processi. Il processo figlio ora azzera lo spazio di memoria e inizia a eseguire il suo codice.

RICORDA: Quando c'è la fork, il parent ha come valore di ritorno il pid del figlio, invece il figlio ha 0, che gli fa capire di essere figlio.

In waitpid, se ha avuto successo, il valore di ritorno è di nuovo il pid del child.

Nota: è il kernel che orchestra il comportamento di parent e child ovviamente.

Ogni processo pensa di avere tutto il processore per sé e tutta la ram per sé ahah che carino 😊. È il kernel che riporta le cose alla ragione, che quando ci sono più processi, gli da questa illusione, ma fa il time sharing. Ovvero c'è uno scheduling dei processi. Ovviamente per un sistema destinato a automazione industriale, si dà più spazio al calcolo che a ciò che è interattivo, come il click del mouse. Altrimenti oggi i computer danno più spazio a ciò che è interattivo, perché dà l'idea all'utente che il computer sia più veloce.

Gestione degli errori:

La gestione degli errori di una certa funzione la scopriamo guardando il manuale di essa. In genere c'è una convenzione, per esempio se ritorna 0 è andato ok. Però ci sono eccezioni, come la read che ritorna il numero di byte letti. Oppure in genere ritornano un valore negativo in base all'errore che c'è stato. Insomma, non si scappa dal leggere il manuale.

Le system call invece, quando c'è un errore, avvalorano una certa variabile. La variabile è chiama err_no, e va messa nel proprio codice e dichiarata come extern.

Extern -> significa che è globale e può essere modificata da funzioni presenti in altri file; ci può essere prima la dichiarazione (extern int err_no) e poi viene assegnato il valore dalle systemcall.

Cos'è questa variabile? È avvalorata quando c'è un errore, e infatti va letta subito dopo la system call. C'è una lista di numeri, che dice per ogni numero quale è stato l'errore.

Vediamo ora quali sono questi numeri e questi errori. Si possono vedere in man 2 intro. (su linux invece è man errno, per trovare il numero corrispondente find /usr/include -name errno.h, oppure direttamente grep EACCES \$(!!), ovvero grep di EACCESS nella lista di file ottenuta prima. Non sembra avere successo, allora grep -rw EACCES /usr/include, e si trova)

NOTA: err_no non viene resettata. Bisogna leggerla quando sai che c'è stato un errore, altrimenti potrebbe essere l'errore della systemcall di prima.

Per esempio, se essa è 1 significa che si sta facendo una waitpid a un processo che non esiste. Nota che nessuna delle costanti in err_no è uguale a zero.

```
#include <string.h>
char *strerror(int errnum);
```

La funzione perror, passando il numero di errore, dice la corrispondente descrizione che c'era nel manuale, nulla di più.

```
#include <stdio.h>
void perror(const char *msg);
```

La funzione perror invece stampa un messaggio relativo all'errore che c'è effettivamente stato.

Vediamo esempio 1.8

```
#include "apue.h"
#include <errno.h>

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

Nota: per usare err_no bisogna includere il file errno.h

Fprintf ha come argomento un puntatore a una struttura di tipo file, in questo caso predefinito in stderr. Dove è definito stderr? Cercatelo ☺
E' stampato il valore della stringa in output di sterror, l'errore corrispondente al nome "EACCES", che corrisponde al numero 13 (si sarebbe potuto mettere il numero).
Poi si forza un errore che non c'è stato, giusto per usare la perror, passando il nome del programma (argv[0]) e descrivendo l'errore ENOENT che abbiamo forzato.
Quindi:
Sterror dice cosa significa una determinata costante
Perror spiega invece meglio cosa è successo.

Cosa vuol dire la keyword **const**?

Ricorda che ogni funzione vede le sue stesse variabili e basta. Se invece una funzione vuole lavorare sul mondo esterno ci sono 3 modi: passaggio dei parametri, passaggio degli indirizzi, e l'uso delle variabili globali (ovvero quelle definite al di fuori di un blocco).
Const significa che anche se si passa l'indirizzo di una variabile, non si può modificare il valore della variabile, si può solo leggere. Chi si occupa di fare il poliziotto? Il compilatore.

Cosa vuol dire invece la keyword **restrict**? Si applica a dei puntatori, e significa che questi puntatori non possono essere copiati in altre variabili. Si può accedere al dato solo attraverso il puntatore con la keyword restrict. Questo perché a volte il compilatore cerca di fare delle ottimizzazioni e vuole quindi sapere esattamente i puntatori utilizzati. Un esempio di ottimizzazione è quello di copiare ciò che viene molto utilizzato in un registro in modo da renderlo più veloce da raggiungere.

Figura 1.9:

```
#include <apue.h>

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Con getuid si ottiene l'user id del proprio utente, e con getgid si ottiene il numero corrispondente al gruppo di appartenenza.

Questo da terminale si può leggere usando il comando id.

Parliamo ora di **segnali**, figura 1.10

```

#include "apue.h"
#include <sys/wait.h>

static void sig_int(int); /* our signal-catching function */

int
main(void)
{
    char    buf[MAXLINE]; /* from apue.h */
    pid_t   pid;
    int     status;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal error");

    printf("%s ", /* print prompt (printf requires % to print %) */
    while ((fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = \0; /* replace newline with null */

        if ((pid = fork()) < 0)
            err_sys("fork error");
        } else if (pid == 0) { /* child */
            execvp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%s ", /* print prompt */
    }
    exit(0);
}

void
sig_int(int signo)
{
    printf("interrupt\n% ");
}

```

Somiglia molto a fig 1.7

Rispetto al codice precedente, sono stati aggiunti dei pezzi che riguardano la gestione dei segnali.

La dichiarazione della funzione `sig_int` con la sua definizione in fondo al file, e l'`if` statement. La funzione `sig_int` prende un intero e non restituisce nulla: questo è il nostro gestore di segnale. Il gestore di segnale in questo caso è molto scemo, semplicemente stampa una linea. La funzione prende un intero che non viene però passato manualmente, ma è il kernel che lo assegna automaticamente mettendo il numero del segnale ricevuto.

Notare che il valore di ritorno di `signal()` è l'indirizzo del precedente handler per questo specifico segnale, altrimenti ritorna `SIG_ERR` (in caso di "errore").

È facile vederne il funzionamento perché SIGINT è il segnale mandato quando si fa control-c. Ora però il processo ignora il contro-c. Per uscire dal programma ora si manda il segnale sigquit con control \.

NOTA: A volte viene stampato un qualcosa quando si fa control c o control \, chi stampa questa cosa?? I segnali di loro se ne fregano. Booh vedremo. Nota: stty -a dice caratteri cose

ANOMALIA: se si interrompe il processo di questo shell e si fa ripartire, non riesce a ripartire e muore. Questo succede perché la read ritorna 0 per il segnale, e quindi anche la fget ritorna zero. Sembra quindi che non ci sia nulla da leggere e si esce dal loop.

Ricordo che i processi pensano di avere tutto il processore e tantissima memoria solo per sé, ma in realtà non è così: il kernel si occupa di "limitare il processo" tramite dei meccanismi di comunicazione. Infatti, esistono una serie di interfacce chiamate IPC (inter process communication) che permettono ai processi di comunicare con il mondo esterno, e l'interfaccia più basilare è quella dei segnali.

I segnali vengono chiamati anche software interrupt.

Cosa sono invece gli hardware interrupt? Hardware interrupt: l'interazione dell'utente con il computer genera degli interrupt. Tutte le periferiche sono in grado di fare arrivare al processore i cosiddetti interrupt, che hanno associato un valore chiamato interrupt number: quando arriva questo "segnale" con questo numero, il sistema ha questa tabella degli interrupt, e ha una routine che il processore esegue (interrupt handler), e poi torna a fare quello che stava facendo. Esempi di hw interrupt possono essere banalmente causati dalla tastiera quando si digitano delle parole.

Nel caso dei software interrupt, i segnali vengono mandati dal kernel e non da una periferica hardware. Mentre viene eseguito il codice macchina step by step, il kernel è capace di interromperne l'esecuzione grazie dell'arrivo di un segnale. Se il segnale non causa danni, poi il processo potrà riprendere da dove era stato interrotto. È sempre il kernel a mandare segnali, ma spesso sono i processi che chiedono al kernel di mandare un particolare segnale ad un altro processo.

Molto spesso i segnali sono causati dal verificarsi di alcune situazioni particolari, per esempio la divisione per 0, oppure se un processo cerca di accedere a uno spazio di memoria che però non può visitare.

Come dicevamo, il processo pensa di avere tutta la memoria per sé, ma deve sempre chiedere kernel prima di poter accedere: se si chiede di accedere ad un indirizzo a cui non si può accedere, viene mandato un segnale (segmentation fault fondamentalmente è un segnale).

Un'altra situazione è quella della digitazione di control-z: noi chiediamo al kernel di mandare al processo un segnale di stop; allo stesso modo si può fare ripartire con il segnale di continuazione.

Quando arriva un segnale al processo, cosa succede? Dipende anche dal processo. Se il processo non era preparato, succede l'azione di default, altrimenti è possibile ignorare i segnali o gestirli chiamando una funzione all'arrivo di essi.

Come con i segnali hardware, la gestione dei segnali software avviene tramite un numero che indica il particolare segnale. Quando un processo riceve tale segnale, viene fatto partire il signal handler. Il codice si occupa allora di gestire il segnale tramite le istruzioni date precedentemente dal programmatore, e poi continua con l'esecuzione.

Per vedere i segnali man signal.

	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGNALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal

Nota: "create core image", significa che il processo viene terminato, ma viene creata una fotografia della memoria usata dal determinato processo al momento della sua dipartita. Così è possibile fare una specie di autopsia del processo per capire cosa ha scatenato il segnale.

Time values

Time values: il tempo viene gestito in secondi dal primo gennaio 1970, dalla epoch (circa la nascita di linux).

Per ogni processo sono presi in considerazione 3 tempi. Lanciando il comando time + nomecomando possiamo vedere i vari tempi utilizzati dal processo per la sua esecuzione.

I tempi sono real, user e sys.

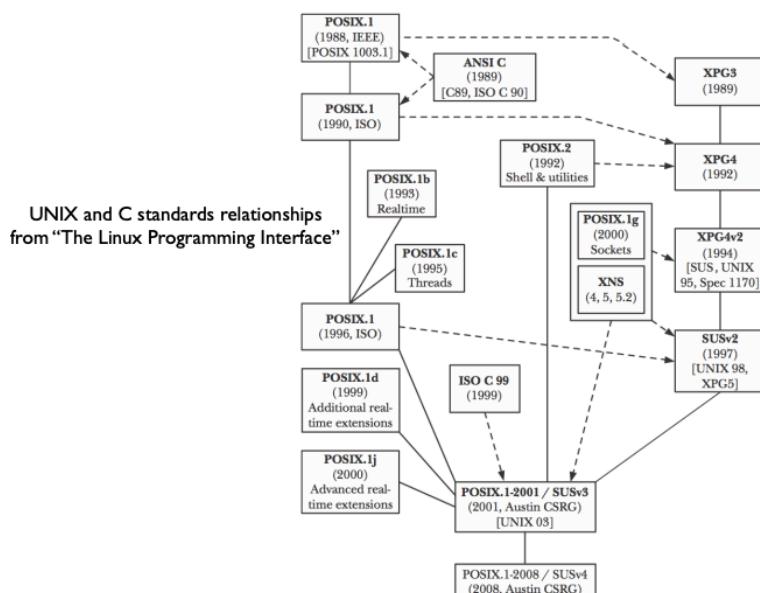
Real -> tempo di orologio per completare il processo.

User -> tempo che non richiede intervento del kernel, per esempio operazioni matematiche.

Sys -> tempo del kernel a eseguire le systemcall.

NOTA: real non è user + sys, perchè ci sono anche altri processi di cui il kernel si deve occupare.

Parliamo ora di **standard**.



Quando uscii unix, tutti si fecero il proprio ed erano tutti incompatibili. Nel 1988 viene fuori POSIX 1. Gli standard volevano standardizzare le interfacce ovvero i prototipi delle funzioni. Siccome il C anche aveva delle funzioni standard, era fondamentale avere anche un C standard, nota infatti che anche nel 99 è stato rifatto uno standard del c. Lo standard si occupava solo delle interfacce: ingresso, uscita e semantica.

Da POSIX poi si crea una versione che contiene shell e utilities.

Le thread vengono standardizzate nel 1995. Dal '94 c'è un consorzio proprietario del marchio unix, il che ha risolto molti problemi. Vengono definite le SUS (single unix specifications).

I due standard importanti sono POSIX 1 SUS v3 del 2001 e la SUS v4 nel 2008.

Ciò che usiamo ora è standardizzato in queste versioni. Mac è certificato SUS v3, invece Linux segue SUS v4 nonostante non ne abbia la certificazione. Linux ha seguito tutto questo mondo ma senza ottenere la vera certificazione, è stata fatta la propria certificazione linux standard base. Mentre SUS si riferisce solo alle interfacce e non alle implementazioni, linux

standard base contiene anche delle implementazioni. Linux, infatti, dice per esempio anche come deve essere fatto l'eseguibile finale.

Quasi nessuno è però sus v4, che contiene parti sul funzionamento dei segnali real time, per esempio, che complicano notevolmente i segnali di sus v3 in cui i segnali hanno solo l'istante di tempo in cui sono lanciati, e il numeretto corrispondente. Questo può tornare utile per esempio in ambito industriale.

Ora, come si fa a essere sicuri che un programma possa essere eseguito contemporaneamente su mac a linux? Innanzitutto, si parte da sus v3 per mac. Poi linux nonostante non sia certificato segue sus v4, bisogna quindi trovare l'intersezione.

Ovviamente è fondamentale l'utilizzo di macro per fare eseguire solo alcune parti di codice in alcune situazioni in cui determinate macro sono definite o hanno particolari valori. C'è una macro particolare `_XOPEN_SOURCE` che si usa per dire per che standard stiamo compilando in base al diverso valore che gli assegniamo. Quindi si fa `gcc -d` e si passa `XOPEN_SOURCE 600` e si compila per un determinato standard

In apue.h per esempio c'è un `#if` che stabilisce con quale sus compilare. Si usa `_XOPEN_SOURCE 600` o `700` in base se sia definita la macro `SOLARIS` oppure no.

Nota: conviene compilare per sus v4, tanto è retrocompatibile. In generale, infatti, si compila per lo standard più alto, l'importante è che se non si è compatibili con sus v4 non vengano utilizzate funzioni appartenenti a quello standard.

Per controllare gli standard:

SUSv3 -> <http://pubs.opengroup.org/onlinepubs/009604599/>

SUSv4 -> <http://pubs.opengroup.org/onlinepubs/9699919799/>

Certified UNIXes -> <http://www.opengroup.org/openbrand/register/>

Linux Standard Base Core Specification - Generic 5.0 ->

http://refspecs.linuxfoundation.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic.pdf

C'è un altro programma interessante `ftm.c` che quando compilato mostra tutte le macro che sono state definite. Qui non viene imposto nulla. Dove stanno però queste macro? VATTELE A CERCARE RETINO. Nota che sto file non sta in apue

Ci sono due tipi di limiti: runtime limit e compilation limit. I compilation limit li trovi negli header, invece i runtime limit sono controllati tramite la funzione `sysconf`.

DICIAMO CHE POTRESTI ESSERE STATO PIU ATTENTO QUI.

ci SONO ANCche dei posix limits, che dice per esempio che un processo può avere un massimo numero di figli.

Esistono alcune systemcall come `sysconf` e `pathconf` che danno informazioni su cosa si può fare con il proprio sistema: per esempio numero di child massimi per un processo, lunghezza massima di un path, e tante cose.

Ci sono due chiamate che danno informazioni sulle configurazioni di sistema.

```
#include <unistd.h>
long sysconf(int name);
long pathconf(const char *pathname, int name);
long fpathconf(int fd, int name);
```

In pathconf il path è quello del filesystem (della chiavetta) di cui si vogliono informazioni in fpathconf invece del pathname si passa il file descriptor che si ottiene da pathconf.

sysconf(), passandogli un argomento ritorna il valore. Per esempio, si può sapere la lunghezza massima di un argomento passato a exec, il numero di file che posso aprire, e tante altre informazioni passando degli argomenti fissati.

Alcune delle informazioni che si possono conoscere in questo modo sono presenti nella tabella che segue:

Per sysconf:

Name of limit	Description	<i>name</i> argument
ARG_MAX	maximum length, in bytes, of arguments to the <code>exec</code> functions	<code>_SC_ARG_MAX</code>
ATEXIT_MAX	maximum number of functions that can be registered with the <code>atexit</code> function	<code>_SC_ATEXIT_MAX</code>
CHILD_MAX	maximum number of processes per real user ID	<code>_SC_CHILD_MAX</code>
clock ticks/second	number of clock ticks per second	<code>_SC_CLK_TCK</code>
COLL_WEIGHTS_MAX	maximum number of weights that can be assigned to an entry of the <code>LC_COLLATE</code> order keyword in the locale definition file	<code>_SC_COLL_WEIGHTS_MAX</code>
DELAYTIMER_MAX	maximum number of timer expiration overruns	<code>_SC_DELAYTIMER_MAX</code>
HOST_NAME_MAX	maximum length of a host name as returned by <code>gethostname</code>	<code>_SC_HOST_NAME_MAX</code>
IOV_MAX	maximum number of <code>iovec</code> structures that can be used with <code>readv</code> or <code>writev</code>	<code>_SC_IOV_MAX</code>
LINE_MAX	maximum length of a utility's input line	<code>_SC_LINE_MAX</code>
LOGIN_NAME_MAX	maximum length of a login name	<code>_SC_LOGIN_NAME_MAX</code>
NGROUPS_MAX	maximum number of simultaneous supplementary process group IDs per process	<code>_SC_NGROUPS_MAX</code>
OPEN_MAX	one more than the maximum value assigned to a newly created file descriptor	<code>_SC_OPEN_MAX</code>
PAGESIZE	system memory page size, in bytes	<code>_SC_PAGESIZE</code>
PAGE_SIZE	system memory page size, in bytes	<code>_SC_PAGE_SIZE</code>
RE_DUP_MAX	number of repeated occurrences of a basic regular expression permitted by the <code>regexec</code> and <code>regcomp</code> functions when using the interval notation <code>\{m, n\}</code>	<code>_SC_RE_DUP_MAX</code>
RTSIG_MAX	maximum number of real-time signals reserved for application use	<code>_SC_RTSIG_MAX</code>
SEM_NSEMS_MAX	maximum number of semaphores a process can use at one time	<code>_SC_SEM_NSEMS_MAX</code>
SEM_VALUE_MAX	maximum value of a semaphore	<code>_SC_SEM_VALUE_MAX</code>
SIGQUEUE_MAX	maximum number of signals that can be queued for a process	<code>_SC_SIGQUEUE_MAX</code>
STREAM_MAX	maximum number of standard I/O streams per process at any given time; if defined, it must have the same value as <code>FOPEN_MAX</code>	<code>_SC_STREAM_MAX</code>
SYMLINK_MAX	number of symbolic links that can be traversed during pathname resolution	<code>_SC_SYMLINK_MAX</code>
TIMER_MAX	maximum number of timers per process	<code>_SC_TIMER_MAX</code>
TTY_NAME_MAX	length of a terminal device name, including the terminating null	<code>_SC_TTY_NAME_MAX</code>
TZNAME_MAX	maximum number of bytes for a time zone name	<code>_SC_TZNAME_MAX</code>

Per pathconf:

Name of limit	Description	<i>name</i> argument
FILESIZEBITS	minimum number of bits needed to represent, as a signed integer value, the maximum size of a regular file allowed in the specified directory	<code>_PC_FILESIZEBITS</code>
LINK_MAX	maximum value of a file's link count	<code>_PC_LINK_MAX</code>
MAX_CANON	maximum number of bytes on a terminal's canonical input queue	<code>_PC_MAX_CANON</code>
MAX_INPUT	number of bytes for which space is available on terminal's input queue	<code>_PC_MAX_INPUT</code>
NAME_MAX	maximum number of bytes in a filename (does not include a null at end)	<code>_PC_NAME_MAX</code>
PATH_MAX	maximum number of bytes in a relative pathname, including the terminating null	<code>_PC_PATH_MAX</code>
PIPE_BUF	maximum number of bytes that can be written atomically to a pipe	<code>_PC_PIPE_BUF</code>
_POSIX_TIMESTAMP_RESOLUTION	resolution in nanoseconds for file timestamps	<code>_PC_TIMESTAMP_RESOLUTION</code>
SYMLINK_MAX	number of bytes in a symbolic link	<code>_PC_SYMLINK_MAX</code>

Da pag 42 di unix programming.

Vedi <https://www.ibm.com/docs/en/aix/7.2?topic=s-sysconf-subroutine>

Vedi file conf.c che credo sia creato da un file .awk. Questo file stampa tutti i valori di queste macro che esplicitano i limiti del sistema.

Nel file conf.c c'è un paragone tra gli header di sistema e quello che restituisce la syscall.

Ovvero si controlla se il valore è stato modificato.

Se la trova con sysconf bene, altrimenti il valore viene cercato nei file di intestazioni.

Vedi options.c e conf.c che usano la systemcall sysconf e pathconf

Vedi per esempio le funzionalità `POSIX_SAVED_ID` e `_POSIX_SEMAPHORES` che possono essere controllate da sysconf

Per esempio, se vuoi allocare della memoria per manipolare una string path, conviene allocare la massima lunghezza di un path che si trova con pathconf in modo da sapere con certezza di non avere problemi.

NOTA: pathconf perché dipende dal particolare filesystem. Il valore è NAME_MAX

Per sapere la lunghezza massima del nome di un file si fa `pathconf(_PC_NAME_MAC)` E SI OTTIENE NAME_MAX che sarà 255

Scopriamo insomma alla fine chi è pid_t:

tu cerchi grep -rn "off_t;"

/Library/Developer/CommandLineTools/SDKs/MacOSX11.3.sdk/usr/include

ma non trovi..... CERCALOOOO

```

#include "apue.h"
#include <errno.h>
#include <limits.h>

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <dirname>");

#ifndef ARG_MAX
    printf("ARG_MAX defined to be %ld\n", (long)ARG_MAX+0);
#else
    printf("no symbol for ARG_MAX\n");
#endif
#ifndef _SC_ARG_MAX
    pr_sysconf("ARG_MAX =", _SC_ARG_MAX);
#else
    printf("no symbol for _SC_ARG_MAX\n");
#endif

/* similar processing for all the rest of the sysconf symbols... */

#ifndef MAX_CANON
    printf("MAX_CANON defined to be %ld\n", (long)MAX_CANON+0);
#else
    printf("no symbol for MAX_CANON\n");
#endif
#ifndef _PC_MAX_CANON
    pr_pathconf("MAX_CANON =", argv[1], _PC_MAX_CANON);
#else
    printf("no symbol for _PC_MAX_CANON\n");
#endif

/* similar processing for all the rest of the pathconf symbols... */

    exit(0);
}

static void
pr_sysconf(char *mesg, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = sysconf(name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs("(not supported)\n", stdout);
            else
                err_sys("sysconf error");
        }
        else {
            fputc((char)(val / 1000), stdout);
            fputc((char)(val % 1000 / 100), stdout);
            fputc((char)(val % 100 / 10), stdout);
            fputc((char)(val % 10), stdout);
        }
    }
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = pathconf(path, name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs("(not supported)\n", stdout);
            else
                err_sys("pathconf error, path = %s", path);
        }
        else {
            fputc((char)(val / 1000), stdout);
            fputc((char)(val % 1000 / 100), stdout);
            fputc((char)(val % 100 / 10), stdout);
            fputc((char)(val % 10), stdout);
        }
    }
}

```

```

        }
    }
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = pathconf(path, name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs("(not supported)\n", stdout);
            else
                err_sys("pathconf error, path = %s", path);
        }
        else {
            fputc((char)(val / 1000), stdout);
            fputc((char)(val % 1000 / 100), stdout);
            fputc((char)(val % 100 / 10), stdout);
            fputc((char)(val % 10), stdout);
        }
    }
}

```

FILE I/O

NOTA: quando si apre un file, la prima read parte dal primo byte, le scritture invece partono dalla fine. Dopo che hanno fatto la prima scrittura, la read riparte a leggere da dove si era fermata. Se si sposta la read, la prossima write parte dalla fine della read.

Esiste una sola posizione che tiene conto della posizione all'interno del file per read e write.

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Both return: file descriptor if OK, -1 on error

La open ritorna il fd del file aperto.

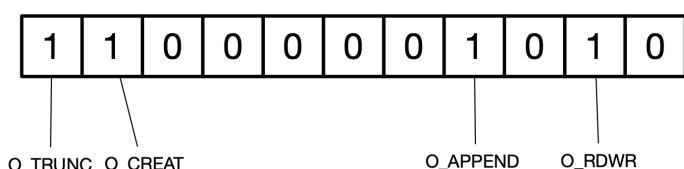
La `open` normale prende come parametri un path assoluto (o relativo rispetto alla cwd), poi dei flag e la modalità di apertura.

Da notare che la open è una variadic function, ovvero con un numero di argomenti che può variare.

Esiste anche `openat` dove il `fd` è il file descriptor di una directory (poiché anche le directory possono essere aperte con una `open`, e non solo `opendir`, ma utilizzando il flag apposito `O_DIRECTORY`). `openat` allora prenderà un path relativo a quella directory e permette di “costruire un mondo separato” per il processo. Questo è molto utile perché altrimenti con la `open` servirebbero i privilegi di lettura e attraversamento di tutte le directory. Così invece si parte dalla directory di lavoro.

`chdir` -> serve a cambiare la root del processo. Ogni processo scioglie i path a partire dalla root.

Vediamo ora il funzionamento dei flag.



`open("a file", O_RDWR | O_APPEND | O_CREAT | O_TRUNC, FILE_MODE);`

Flags use in the open() system call

```
#define O_RDWR     0x0002    /* open for reading and writing */
#define O_APPEND    0x0008    /* set append mode */
#define O_CREAT     0x0200    /* create if nonexistant */
#define O_TRUNC     0x0400    /* truncate to zero length */
```

Questi flag però vanno consultati, perché oltre alcuni di base, ce ne sono alcuni che sono utilizzati solo su alcuni sistemi operativi... bisogna quindi ricorrere agli standard. Notare che per esempio il flag O_DIRECTORY non fa parte dello standard.

Tornando a prima, i flag per decidere come aprire il file, se scrittura, lettura o cosa, li troviamo nello stesso file aperto prima.

```
#define O_RDONLY      0x0000      /* open for reading only */
#define O_WRONLY       0x0001      /* open for writing only */
#define O_RDWR         0x0002      /* open for reading and writing */
#define O_ACCMODE      0x0003      /* mask for above modes */
```

O_TRUNC -> quando apro un file nuovo ed esiste un file con quel nome quel file viene raso al suolo

O_EXCL -> risolve il problema O_TRUNC , se il file esisteva fa fallire la chiamata

Ce ne sono anche altri come O_APPEND, oppure O_CREAT. vedi pag 62 per i flag disponibili.

Mode invece indica i permessi del file .

Quali sono i permessi di un file appena creato? Tramite **umask** vengono dati i permessi, ovvero vengono assegnati per differenza i permessi 666-umask.

Ogni processo ha la sua umask che eredita dal parent. È possibile controllare la umask di bash con l'omonimo comando.

NOTA: non si prende proprio in considerazione l'esecuzione, infatti si sottrae la umask a 666 e non 777

NOTA: non è detto che la richiesta che si fa nella open riguardo i permessi venga rispettata, perché le libertà che si hanno dipendono dal valore della umask, che indica i massimi permessi che si possono assegnare al file.

C'è una chiamata che preso il file descriptor cambia questi flag.

NOTA: quando si apre un file con determinati privilegi, anche se poi magari la umask del processo cambia, la modalità di apertura del file rimane quella iniziale.

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Returns: new file offset if OK, -1 on error

È possibile cambiare la posizione del "puntatore" all'interno del file, tramite la lseek.

Il primo argomento è il file descriptor, il secondo è l'offset, il terzo parametro indica il punto di riferimento. Questo parametro può avere 3 valori, SEEK_SET che corrisponde al valore assoluto, SEEK_CUR corrisponde alla posizione attuale, SEEK_END corrisponde alla posizione finale.

Nota: SEEK_END farà usare un offset negativo probabilmente perché si vuole scrivere prima della fine del file, a meno che non si voglia saltare oltre.

Se non c'è il flag O_APPEND, si può comunque scrivere alla fine del file con write preceduta da una lseek con SEEKEND e offset 0.

NOTA: ci sono dei file seekable e non seekable, cioè cercabili e non cercabili. Ovvero se si può fare lseek o no... per esempio stdin non è seekable.

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

Analizziamo adesso la read:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
```

Returns: number of bytes read, 0 if end of file, -1 on error

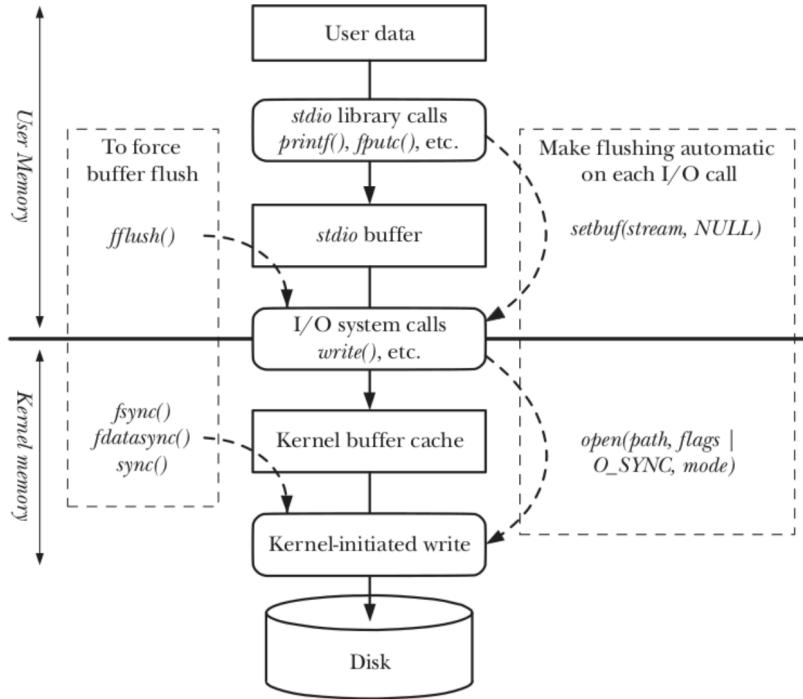
La read ritorna il numero di byte letti, quindi se ritorna zero significa è arrivata alla fine del file. Inoltre, quando la read legge da un terminale, ritorna quando viene dato invio (per esempio è quello che fa cat, fa una read finchè non si fa invio). Quando si legge da una rete, il buffering può causare una lettura di meno byte rispetto a quelli richiesti. Nota però che la read da network rimane appesa, se non arriva nulla non viene ritornato 0, ma aspetta. La stessa cosa succede da una pipe, se si mette cat dopo una pipe, se non si scrive nulla rimane appesa. Se c'è qualcosa nella pipe la read ritorna il numero di byte che ha potuto leggere. Quando interrotta da un segnale, la read ritorna comunque il numero di ciò che è stato letto, a meno che la system call non viene resettata.

Analizziamo ora la write:

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Returns: number of bytes written if OK, -1 on error

Ha un prototipo simile alla read, ma in questo caso se si ritorna un numero di byte scritti diversi da quelli previsti significa che c'è stato un errore. Un classico caso è quello del disco pieno.



Unbuffered IO -> input output senza buffer, si scrive direttamente.

Allo scopo di migliorare le prestazioni spesso si usano i buffer, ovvero prima si raccolgono un po' di cose da scrivere, e poi si effettua l'operazione. La libreria accumula le scritture e in base ai suoi criteri, in base a dei timeout o a quanto si è scritto, decide quando tirare fuori tutto

Questa idea è alla base della libreria di utente stdio library. Un esempio è la `printf`.

Qui c'è una grande differenza tra `write` e `printf`: `printf` usa dei buffer che vengono gestiti in maniera che prima di effettuare l'operazione, ci sia un minimo di "riflessione". La stdio ha quindi dei buffer interni per ottimizzare le prestazioni. Infatti, c'è la funzione `fflush` in C che caccia tutto ciò che ha nel buffer.

Cio non toglie che anche il kernel possa avere un suo buffer, infatti lo ha. Esiste quindi `sync()` a livello di kernel che svuota il buffer (del kernel).

Parliamo ora di **file sharing**, quando più processi vogliono lavorare con lo stesso file.

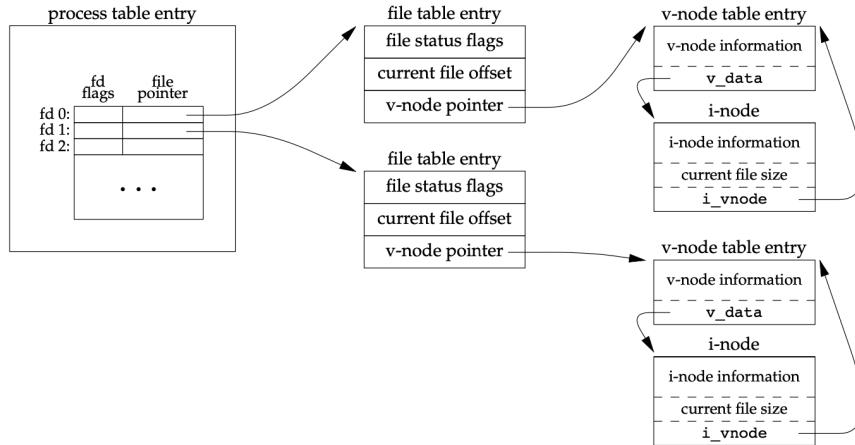


Figure 3.7 Kernel data structures for open files

C'è una tabella con i file descriptor flags e i puntatori ai file, uno per ogni file descriptor. I fd flag dicono come il processo può trattare il file: leggi scrivi o quel che è. Ogni fd ha un puntatore che punta a una file table entry diversa. La table entry ha il current file offset, e dei flag chiamati status flag relativi al file. Infine c'è scritto nel puntatore come arrivare ai dati del file. Nel caso sia un file regolare dice come arrivare ai blocchi del file.

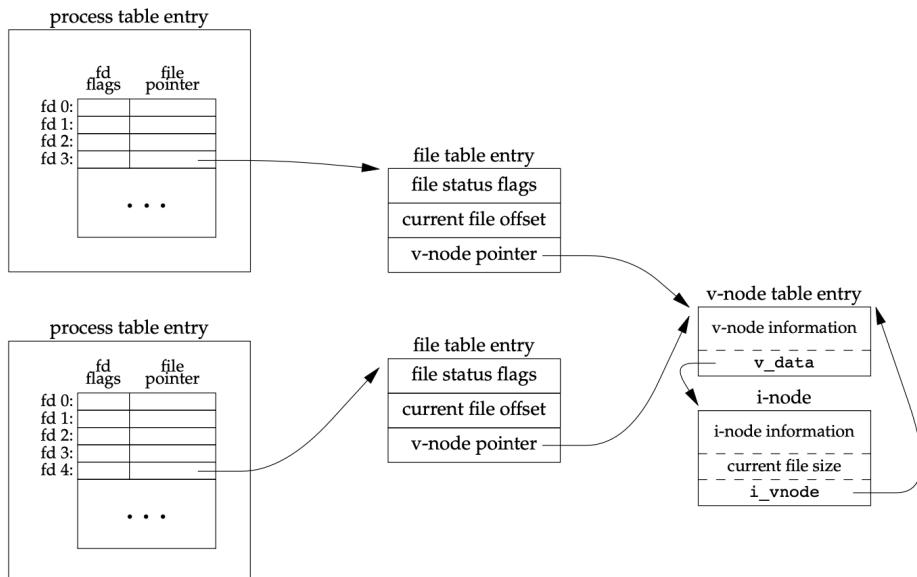


Figure 3.8 Two independent processes with the same file open

Quando ci sono più processi che vogliono accedere agli stessi file, ognuno avrà la sua file table entry che quindi permetterà di avere degli offset e flag diversi.

RICORDA: se qualcuno scrive in mezzo, sovrascrive dey byte.

Ricordiamo il meccanismo delle systemcall, in cui si mette un numero in uno scatolino, si riempiono i registri corrispondenti con i dati, e poi il sistema esegue la systemcall e restituisce il risultato. Se due processi però vogliono fare la stessa cosa, si rischia di sovrascrivere a vicenda i dati. Per questo torna utile il flag O_APPEND, che permette di fare una scrittura atomica: evita che altre systemcall si possano mettere in mezzo, fa automaticamente lseek alla fine del file + write in un solo colpo.

Il problema deriva dal fatto che se faccio lseek e write, ma qualcuno fa una lseek dopo la mia lseek e prima della mia write si crea un pasticcio.

Nello stesso modo, se si vuole scrivere nel mezzo del file, si può usare la chiamata p_write (o p_read per leggere) che è come una write ma che prende un quarto parametro che è l'offset, e vengono fatte un un solo istante la lseek e la write/read.

A proposito di questo problema, torna utile anche il flag O_EXCL: quando creo il file, se esiste già non toccare nulla.

Che succede se quando io controllo il file non esiste, ma nel frattempo un altro processo lo crea? Significa che quando io creerò il mio pensando che non ne esiste un altro, andrò a sovrascrivere quello dell'amico mio. Con O_CREAT e O_EXCL si ottiene l'atomicità.

Se si usa O_EXCL e O_CREAT, e il file esiste già, allora si ha errore, non si fa nulla (?)

NOTA: un'idea simpatica per sincronizzare due processi potrebbe essere la creazione di un file. Ovvero per svolgere determinate operazioni, si crea un file e il che significa che il secondo processo vedrà che il file esiste già e aspetterà. Quando il file verrà distrutto dal primo processo, allora il secondo processo capirà di poter svolgere determinate operazioni.

Vediamo ora due stupidissime funzioni, pietra miliare del sistema unix.

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, -1 on error

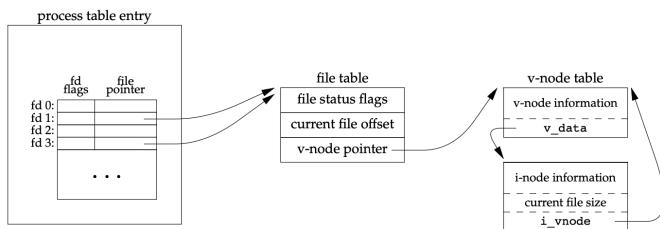


Figure 3.9 Kernel data structures after dup(1)

Un processo che fa la dup ottiene il duplicato di un file descriptor. C'è poi la dup2 che fa la stessa cosa specificando su quale numero voglio il file descriptor. Se il numero di fd richiesto è già occupato, però viene chiuso quello precedente e poi viene dato.

```

#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);

void sync(void);

```

Queste systemcall invece servono a forzare i flush del buffer del kernel. In genere vengono automaticamente fatte prima dello shutdown del sistema.

Analizziamo ora, dal capitolo 8, cosa succede quando c'è una fork nell'ambito dei file aperti da un processo.

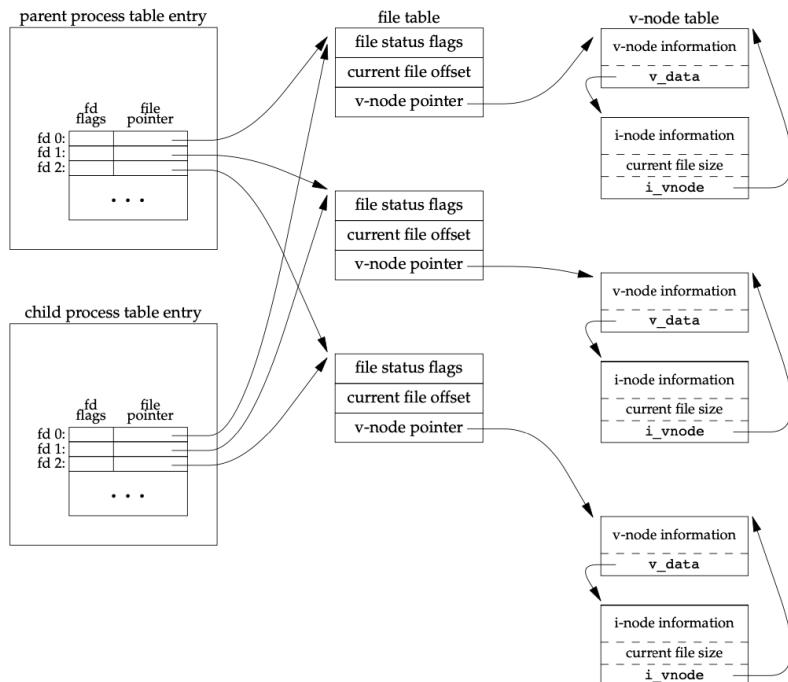


Figure 8.2 Sharing of open files between parent and child after `fork`

Quando si fa la fork, prima delle exec, il figlio avrà la process table che sarà una copia del parent, ma in particolare condivideranno la stessa file table. Ciò significa che se il parent o il child fanno una scrittura o spostano l'offset del puntatore al file, anche l'altro processo ne risentirà.

Tutti questi meccanismi visti rendono possibile il funzionamento delle pipe o delle ridirezioni.

Quando si fa una pipe, o una ridirezione, bash analizza il comando e grazie a dei caratteri speciali capisce alcune operazioni da fare. I comandi che noi lanciamo sanno solo di dover scrivere su fd 1 (stdout) o per esempio di leggere da fd 0 (stdin). Bash permette le ridirezioni grazie alla systemcall dup2: bash fa una fork, e prima della exec viene aperto il nuovo file su

cui si vuole scrivere e viene scambiato il suo fd con quello dello stdin o stdout. In questo modo il comando lanciato senza saperlo invece di usare stdout o stdin utilizzerà altri file. Tutto questo è possibile grazie ai fd flags visti nelle process table entry: se il valore corrispondente al fd è 0, allora il child erediterà il fd dal parent.

Il flag CLOEXEC è usato per attivare il fd flag.

NOTA: in realtà il flag è uno, non so perché dice fd "flags"

Se si vogliono cambiare le proprietà di un file già aperto, si usa la funzione **fcntl**, a cui passando un parametro si determina il comportamento della funzione.

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* int arg */ );
```

Returns: depends on *cmd* if OK (see following), -1 on error

Qui l'effetto coltellino svizzero sta nel parametro cmd. I comandi più frequenti sono i seguenti:

1. Duplicate an existing descriptor (*cmd* = F_DUPFD or F_DUPFD_CLOEXEC)
 2. Get/set file descriptor flags (*cmd* = F_GETFD or F_SETFD)
 3. Get/set file status flags (*cmd* = F_GETFL or F_SETFL)
 4. Get/set asynchronous I/O ownership (*cmd* = F_GETOWN or F_SETOWN)
 5. Get/set record locks (*cmd* = F_GETLK, F_SETLK, or F_SETLKW)
-
- 1) F_DUPFD duplica il filedescriptor. Rispetto alla dup questo ci permette di scegliere il file descriptor passando il secondo argomento. Se il secondo argomento è 10 significa che viene assegnato il più basso fd disponibile dopo il 10. Non è molto usato. Il nuovo file descriptor potrebbe avere un file descriptor flag diverso. Quando viene creato però viene prima settato a zero. Invece F_DUPFD_CLOEXEC invece lascia settato il fd flag se settato. Poco usati questi ahaha sfigati
 - 2) FGETFD permette di leggere il file descriptor, invece F_SETFD permette di settarlo
 - 3) F_GETFL e F_SETFL Questo è molto utile, cambia i file status flag. Non si lavora più nella tabella del processo ma nella file table entry nel kernel. Questa funzione si usa se voglio cambiare i flag in corsa. Se l'avevo aperto in lettura, ora posso cambiare e permettere la scrittura. Oppure se non c'è append, mettere append.

Vedi pagina 82 3.14 per vedere i possibili cmd values.

NOTA: dal manuale di fcntl si nota che ogni sistema operativo ha le sue lame che non sono standard.

NOTA: i file in unix non sono strutturati, sono solo una successione di byte, non c'è distinzione tra i file di testo e quelli binari. La distinzione è fatta dai programmi che li utilizzano.

Vediamo insieme esempio 3.11

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int      val;

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write");
        break;

    default:
        err_dump("unknown access mode");
    }

    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
    if (val & O_SYNC)
        printf(", synchronous writes");

#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC) && (O_FSYNC != O_SYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif

    putchar('\n');
    exit(0);
}
```

E' utilizzato F_GETFL che prende il file status flag (che indica in che modalità è aperto il file). Restituisce un intero che indica la modalità e viene salvato dentro val.

Nello switch c'è un and bit a bit con la maschera O_ACCMODE che da ritorna 1 nel caso della modalità con cui il file è stato aperto.

O_ACCMODE è una maschera per analizzare se il file è aperto in scrittura, lettura, o entrambi. Poi si controllando gli altri flag.

Si puo lanciare il codice passando per esempio il terminale.

./fileflag 0 </dev/ttys002

Vediamo read only perche è passato con <

Se facciamo ./fileflag 1 > prova.txt vediamo in prova il risultato che è write only.

Grazie ai metacaratteri si fanno aperture diverse dei file

```

$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read write

```

Nell'ultimo caso si legge il file descriptor 5, e poi si ridirige il file descriptor 5 a un file.

Vediamo ora 3.12

```

#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int      val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;      /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}

```

Figure 3.12 Turn on one or more of the file status flags for a descriptor

Notiamo che c'è solo una funzione e manca il main.

Gli si passa il fd e dei flag: prima si salvano in val i flag attualmente attivi, poi si aggiungono quelli passati alla funzione e infine si settano i flag di val.

Notare che con |= si fa un or e quindi si aggiungo i flag, invece con fa &= ~ si fa un and della not e quindi spegneremmo quei bit.

```
val &= ~flags;      /* turn flags off */
```

Esiste inoltre una funzione chiamata ioctl che si occupa di gestire le operazioni di lettura e scrittura sui file che non sono gestite dalla banale read e write. Considerando che su unix everything is a file, questa è la funzione usata per scrivere su file particolari come robot, o qualsiasi device.

```

#include <unistd.h>      /* System V */
#include <sys/ioctl.h>    /* BSD and Linux */

int ioctl(int fd, int request, ...);

```

Ogni device ha il suo set di ioctl commands, ad esempio:

Category	Constant names	Header	Number of ioctls
disk labels	DIOXXX	<sys/disklabel.h>	4
file I/O	FIOXXX	<sys/filio.h>	14
mag tape I/O	MTIOXXX	<sys/mtio.h>	11
socket I/O	SIOXXX	<sys/sockio.h>	73
terminal I/O	TIOXXX	<sys/ttycom.h>	43

FILES AND DIRECTORIES

Rivediamo brevemente la struttura dei filesystem.

Ogni memoria di massa può essere suddivisa in più partizioni, e ogni partizione è in grado di ospitare un filesystem: esso permette l'organizzazione dei file per ottenere un rapido accesso ai dati da parte del sistema e delle sue applicazioni.

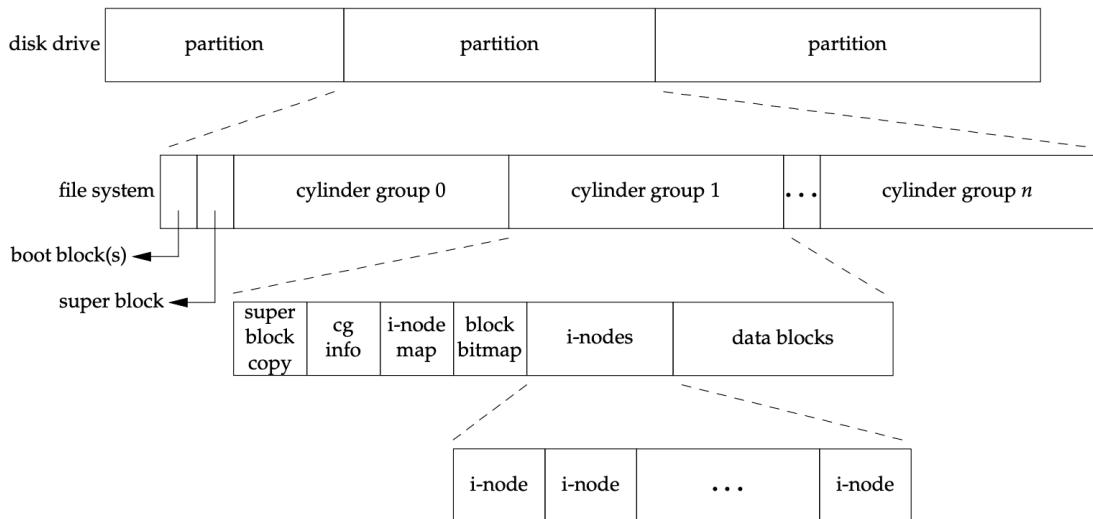


Figure 4.13 Disk drive, partitions, and a file system

Ogni partizione è composta da:

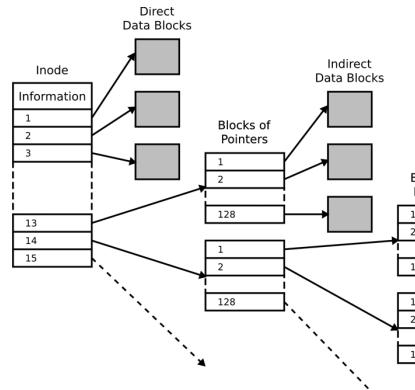
- Un boot sector o boot block: porzione di memoria che ospita il primo codice da eseguire in caso la partizione sia scelta come partizione di avvio
- Un superblocco: che contiene informazioni indispensabili per la lettura dell'intera partizione
- Block groups o cylinder groups: gruppi di blocchi nei quali il filesystem raggruppa i blocchi di memoria

Analizziamo ora la struttura di ogni singolo blocco:

- Ogni gruppo di blocchi inizia con una copia del superblocco. Questo blocco viene ripetuto identicamente all'inizio di tutti i gruppi di blocchi.
- Poi c'è la block bitmap: sequenza di bit in corrispondenza biunivoca con i blocchi del gruppo. Ogni bit è 1 se il blocco corrispondente è occupato da un file.
- Poi c'è la inode bitmap: un'altra bitmap dedicata a registrare l'occupazione degli inode: spazi della partizione che contengono le inode table.

Seguono poi gli inode, che sono delle tabelle che occupano uno spazio fisso e contengono due tipi di informazioni: i metadati del file, e i puntatori ai blocchi di dati che contengono il file (utilizzati per poter finalmente raggiungere il contenuto nei blocchi)

Sono presenti anche dei blocchi di dati destinati a contenere esclusivamente i puntatori (indirect blocks) in modo che con un solo puntatore a blocco si possano raggiungere più blocchi. Questo è comodo quando abbiamo un file molto grande e non vogliamo sprecare la memoria, poiché dovremmo ogni volta ripetere le informazioni relative ai metadati.



Inoltre, le directory sono l'unico luogo dove esiste il nome del file. Esse sono una tabella che mettono in corrispondenza nome del file con il numero di inode. In questo modo uno stesso file può essere in più directory senza comportare la duplicazione dei dati del file:

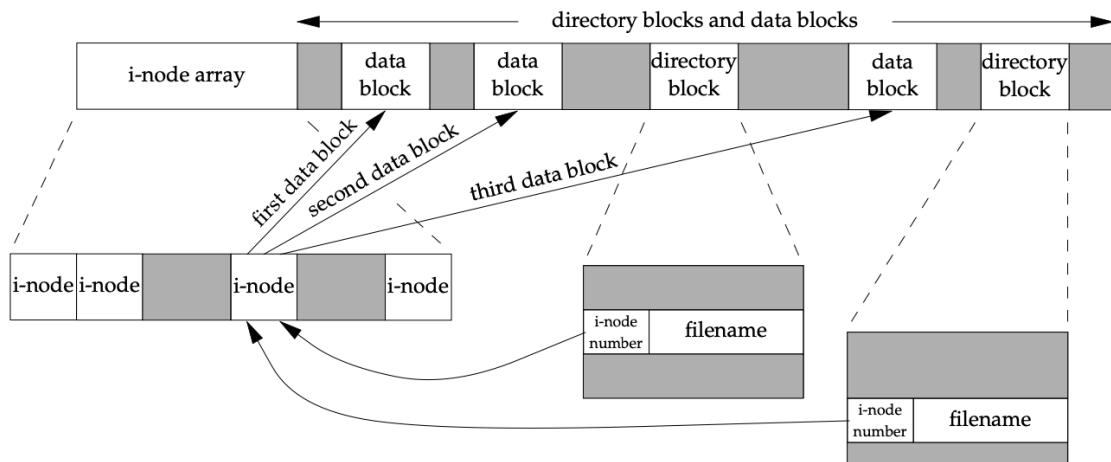


Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

Per leggere l'inode e vedere i metadati relativi a un file si usa su unix il comando stat (su mac stat -x). Da notare che non vengono mostrate informazioni "inutili" come i puntatori ai blocchi che in genere non servono all'utente.

Usando il comando dumpe2fs (solo ext2 ext3 ext3 su linux credo) fa il dump di tutti i dati nella partizione che passiamo. Vengono mostrate prima informazioni sul filesystem e poi per ogni gruppo di blocchi diverse informazioni interessanti. Nel mezzo spesso si vedono delle copie dei superblocks.

Un bel esercizio sarebbe quello di trovare i puntatori ai blocchi i dati.

Per trovare allora i puntatori ai blocchi di un file, dal nome vedo l'inode del file e poi vedo in quale gruppo di blocchi si trova l'inode.

```
#include <sys/stat.h>

int stat(const char *restrict pathname, struct stat *restrict buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *restrict pathname, struct stat *restrict buf);
int fstatat(int fd, const char *restrict pathname,
            struct stat *restrict buf, int flag);
```

All four return: 0 if OK, -1 on error

Esiste anche la funzione stat, che mostra i metadati del file. Essa prende il path del file e una struttura che verrà usata come parametro di uscita. Poi c'è la versione fstatat dove invece del path si passa il file descriptor della directory in cui è contenuto e il path del file relativo ad essa. Invece lstat che nel caso sia stato passato un link simbolico lo considera come file di tipo link senza arrivare al file a cui punta.

```
struct stat {
    mode_t          st_mode;      /* file type & mode (permissions) */
    ino_t           st_ino;       /* i-node number (serial number) */
    dev_t           st_dev;       /* device number (file system) */
    dev_t           st_rdev;      /* device number for special files */
    nlink_t         st_nlink;     /* number of links */
    uid_t           st_uid;       /* user ID of owner */
    gid_t           st_gid;       /* group ID of owner */
    off_t           st_size;      /* size in bytes, for regular files */
    struct timespec st_atim;     /* time of last access */
    struct timespec st_mtim;     /* time of last modification */
    struct timespec st_ctim;     /* time of last file status change */
    blksize_t        st_blksize;    /* best I/O block size */
    blkcnt_t        st_blocks;    /* number of disk blocks allocated */
};
```

Si nota subito la struttura timespec che è utilizzata per memorizzare i tempi, in particolare è composta come segue:

```
time_t tv_sec;  
long tv_nsec;
```

Il campo st_mode contiene informazioni riguardo il tipo di file ed i suoi privilegi.

Seguono i possibili tipi di file.

1. Regular file. The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary. Any interpretation of the contents of a regular file is left to the application processing the file.

One notable exception to this is with binary executable files. To execute a program, the kernel must understand its format. All binary executable files conform to a format that allows the kernel to identify where to load a program's text and data.

2. Directory file. A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write directly to a directory file. Processes must use the functions described in this chapter to make changes to a directory.
3. Block special file. A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.

Note that FreeBSD no longer supports block special files. All access to devices is through the character special interface.

4. Character special file. A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.
5. FIFO. A type of file used for communication between processes. It's sometimes called a named pipe. We describe FIFOs in Section 15.5.
6. Socket. A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host. We use sockets for interprocess communication in Chapter 16.
7. Symbolic link. A type of file that points to another file. We talk more about symbolic links in Section 4.17.

Per conoscere il tipo di file si usano delle function like macro, che ritornano true o false in base al tipo.

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

Figure 4.1 File type macros in <sys/stat.h>

Vediamo ora l'esempio 4.3

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int          i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}

$ ./a.out /etc/passwd /etc /dev/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: block special
/dev/cdrom: symbolic link
```

Semplicemente per ogni argomento viene chiamata la funzione lstat, e viene poi analizzato il campo st_mode per conoscerne il tipo.

Real user id e real group id indicano chi è l'utente, ma spesso si differenziano dall'effective user id del processo, che invece indica “da chi è stato lanciato il processo”. Nel caso in cui il set user id bit è settato, il processo si presenta al mondo come se fosse eseguito dal proprietario del file, nonostante l'utente che lo esegue non lo sia. I suoi privilegi e le sue libertà sono quindi calcolati rispetto all'effective user id e non al real.

Un esempio è il comando ps che ha lo sticky bit settato infatti ha effective user id root nonostante l'utente che l'ha lanciato non sia root.

```
~ > ls -l $(which ps)
-rwsr-xr-x  1 root  wheel  203504 May  9 23:30 /bin/ps
```

Poi ci sono delle altre caratteristiche che identificano il processo, ovvero “saved set user id” e “saved set group id” che hanno a che fare con il bollino delle discoteche. Essi contengono copie dell'effective user id e group id quando il processo è eseguito, in modo da permettere di lasciare i permessi ad un certo punto e poterli riottenere successivamente.

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by <code>exec</code> functions

I permessi sono verificabili tramite queste macro:

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

Queste macro assumono dei valori che in binario hanno acceso un solo bit, e tramite l'operatore & con st_mode se l'output è 1 significa che il privilegio c'è.

NOTA: qui si cono 3 terne invece di 4, ma ci sono anche le altre definizioni nello stesso file in cui sono definite queste.

Il significato di read, write e execute sono i seguenti:

- The read permission for a file determines whether we can open an existing file for reading: the O_RDONLY and O_RDWR flags for the `open` function.
- The write permission for a file determines whether we can open an existing file for writing: the O_WRONLY and O_RDWR flags for the `open` function.
- We must have write permission for a file to specify the O_TRUNC flag in the `open` function.
- We cannot create a new file in a directory unless we have write permission and execute permission in the directory.
- To delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself.
- Execute permission for a file must be on if we want to execute the file using any of the seven `exec` functions (Section 8.10). The file also has to be a regular file.

NOTA: permesso di esecuzione per directory significa possibilità di attraversarla. Ciò è diverso dal permesso di lettura che invece dà la possibilità di elencarne i file all'interno.

NOTA: per cancellare un file esistente, ci vuole solo permesso di esecuzione e scrittura sulla directory e non sul file. Effettivamente si scrive sulla directory e non sul file in quanto la cancellazione, non corrisponde alla eliminazione del file, ma semplicemente della sua entry in quella directory.

NOTA: con le ACL è possibile dare dei specifici permessi a utenti particolari. Per vedere queste ACL su mac si usa ls -e.

I test eseguiti dal kernel prima di accedere ad un file sono i seguenti:

1. If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free rein throughout the entire file system.
2. If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied. By *appropriate access permission bit*, we mean that if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on.
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.
4. If the appropriate other access permission bit is set, access is allowed. Otherwise, permission is denied.

NOTA: se il privilegio di “accesso” a un file è concesso a tutti, ma tu ne sei il proprietario ed il privilegio non è esplicitamente concesso a te, non potrai accedere sul file. Questo perché prima si controllano i privilegi del proprietario, e se non ha i privilegi si blocca l’accesso senza continuare con i controlli.

Esistono delle systemcall che permettono di sapere in anticipo se sarà possibile svolgere o meno determinate azioni su un file.

```
#include <unistd.h>
int access(const char *pathname, int mode);
int faccessat(int fd, const char *pathname, int mode, int flag);
```

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission

Analizziamo ora il concetto della umask.

```
mode_t umask(mode_t cmask);
```

Questa maschera determina i massimi privilegi che avrà un nuovo file appena creato.

Può assumere valori da 000 ovvero che limita ogni tipo di privilegio, a 666 che eventualmente permette di concederli tutti, tranne il permesso di esecuzione che viene escluso a priori.

Tutti i processi hanno una umask associata, che si può verificare con il comando umask nomeprocesso.

NOTA: la umask è più forte della open, se la umask è più restrittiva non se ne fa nulla.

```
#include "apue.h"
#include <fcntl.h>

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

```
$ umask                                first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw----- 1 sar                          0 Dec  7 21:20 bar
-rw-rw-rw- 1 sar                          0 Dec  7 21:20 foo
$ umask                                see if the file mode creation mask changed
002
```

Ci sono poi tutte le versioni di chmod in versione da programmatore che hanno il comportamento atteso.

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
```

NOTA: nel seguente script si crea una situazione anomala, ovvero è tolta l'esecuzione al gruppo ma viene aggiunto il set group id. Ovviamente non ha senso, però a volte vengono usate per indicare qualcosa di specifico nel sistema operativo.

```

#include "apue.h"
int
main(void)
{
    struct stat      statbuf;
    /* turn on set-group-ID and turn off group-execute */
    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");
    /* set absolute mode to "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");
    exit(0);
}
$ ls -l foo bar
-rw-r--r--  1 sar          0 Dec  7 21:20 bar
-rw-rwSr-w  1 sar          0 Dec  7 21:20 foo

```

Lo sticky bit si dà alle directory (indicato da una t) quando più utenti le condividono, in modo da non dare la possibilità di eliminare i file da utenti che non li hanno creati. Per esempio la directory Shared.

```

~ > ls -l /Users
total 0
drwxrwxrwt  5 root          wheel  160 Jun 21 23:39 Shared
drwxr-x---+ 31 francescodenu staff  992 Oct 19 09:29 francescodenu

```

4.11 chown, fchown, fchownat, and lchown Functions

The `chown` functions allow us to change a file's user ID and group ID, but if either of the arguments *owner* or *group* is `-1`, the corresponding ID is left unchanged.

```

#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int fchownat(int fd, const char *pathname, uid_t owner, gid_t group,
             int flag);
int lchown(const char *pathname, uid_t owner, gid_t group);

```

All four return: 0 if OK, -1 on error

Le funzioni `chown` permettono di cambiare l'user ID e il group ID di un file o una directory.

Recall from Section 2.6 that the `_POSIX_CHOWN_RESTRICTED` constant can optionally be defined in the header `<unistd.h>`, and can always be queried using either the `pathconf` function or the `fpathconf` function. Also recall that this option can depend on the referenced file; it can be enabled or disabled on a per file system basis. We'll use the phrase "if `_POSIX_CHOWN_RESTRICTED` is in effect," to mean "if it applies to the particular file that we're talking about," regardless of whether this actual constant is defined in the header.

If `_POSIX_CHOWN_RESTRICTED` is in effect for the specified file, then

1. Only a superuser process can change the user ID of the file.
2. A nonsuperuser process can change the group ID of the file if the process owns the file (the effective user ID equals the user ID of the file), *owner* is specified as `-1` or equals the user ID of the file, and *group* equals either the effective group ID of the process or one of the process's supplementary group IDs.

Per troncare la coda di un file esiste la funzione `truncate`:

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
```

Per creare un hard link si usa la funzione `link`.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
int linkat(int efd, const char *existingpath, int nfd, const char *newpath,
           int flag);

int unlink(const char *pathname);
int unlinkat(int fd, const char *pathname, int flag);
int remove(const char *pathname);

int rename(const char *oldname, const char *newname);
int renameat(int oldfd, const char *oldname, int newfd,
             const char *newname);
```

Il link simbolico ha una sua chiamata ad hoc.

```
int symlink(const char *actualpath, const char *sympath);
int symlinkat(const char *actualpath, int fd, const char *sympath);
```

Cosa succede una funzione prende come argomento un file che è in realtà un link simbolico? Alcune funzioni arrivano fino al file a cui linkano, altre invece considerano il file come un link. Nella seguente tabella si distinguono questi due tipi di funzioni:

Function	Does not follow symbolic link	Follows symbolic link
<code>access</code>		•
<code>chdir</code>		•
<code>chmod</code>		•
<code>chown</code>		•
<code>creat</code>		•
<code>exec</code>		•
<code>lchown</code>	•	
<code>link</code>		•
<code>lstat</code>	•	
<code>open</code>		•
<code>opendir</code>		•
<code>pathconf</code>		•
<code>readlink</code>	•	
<code>remove</code>	•	
<code>rename</code>	•	
<code>stat</code>		•
<code>truncate</code>		•
<code>unlink</code>	•	

Figure 4.17 Treatment of symbolic links by various functions

Le altre funzioni lavorano sul link in quanto file e non in quanto collegamento.

Considerando che la open come abbiamo visto segue il link simbolico, se vogliamo leggere un link esiste la chiamata apposita read link, avente come input pathname e come output buf.

```
#include <unistd.h>
ssize_t readlink(const char* restrict pathname, char *restrict buf,
                 size_t bufsize);
ssize_t readlinkat(int fd, const char* restrict pathname,
                   char *restrict buf, size_t bufsize);
```

Vediamo ora quali sono i tempi che vengono presi in considerazione quando si parla di file.

Field	Description	Example	<code>ls(1)</code> option
<code>st_atim</code>	last-access time of file data	<code>read</code>	<code>-u</code>
<code>st_mtim</code>	last-modification time of file data	<code>write</code>	<code>default</code>
<code>st_ctim</code>	last-change time of i-node status	<code>chmod, chown</code>	<code>-c</code>

Figure 4.19 The three time values associated with each file

Da notare che last-change è riferito all'inode, e include modifiche come la modifica dei permessi o del proprietario del file.

E nella prossima tabella vediamo quali operazioni modificano quali tempi.

Function	Referenced file or directory			Parent directory of referenced file or directory	Section	Note
	a	m	c			
chmod, fchmod			•		4.9	
chown, fchown			•		4.11	
creat	•	•	•	•	3.4	
creat		•	•		3.4	O_CREAT new file
exec	•				8.10	O_TRUNC existing file
lchown			•		4.11	
link			•	•	4.15	parent of second argument
mkdir	•	•	•	•	4.21	
mkfifo	•	•	•	•	15.5	
open	•	•	•	•	3.3	O_CREAT new file
open		•	•		3.3	O_TRUNC existing file
pipe	•	•	•		15.2	
read	•				3.7	
remove			•	•	4.15	remove file = unlink
remove				•	4.15	remove directory = rmdir
rename			•	•	4.16	for both arguments
rmdir				•	4.21	
truncate, ftruncate	•	•			4.13	
unlink			•	•	4.15	
utimes, utimensat, futimens	•	•	•		4.20	
write	•	•	•		3.8	

Figure 4.20 Effect of various functions on the access, modification, and changed-status times

Da notare che la open ha situazioni diverse in base ai flag utilizzati; infatti, se si usa il flag O_TRUNC non viene modificato l'ultimo tempo di accesso al file.

Da notare che remove non cancella un file ma un link, per questo i tempi che cambiano sono quelli dall'ultima modifica dell'inode.

Esistono anche delle funzioni per cambiare i tempi di accesso dei file. È possibile cambiare i tempi forzatamente.

```
#include <sys/stat.h>
int futimens(int fd, const struct timespec times[2]);
int utimensat(int fd, const char *path, const struct timespec times[2],
              int flag);
```

Si passa il file descriptor e un array di due strutture timespec che indicano il tempo di accesso e di ultima modifica.

NOTA: Da linea di comando si possono cambiare con il comando touch, passando come argomento un parametro che esiste già. Per esempio, con l'opzione -A si cambia il modification time.

Nel codice che segue, si prende il file da linea di comando, lo si apre con O_TRUNC per modificare i tempi, ma poi vengono reimpostati quelli iniziali prima della apertura.

```

int
main(int argc, char *argv[])
{
    int             i, fd;
    struct stat     statbuf;
    struct timespec times[2];

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) { /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;
        }
        times[0] = statbuf.st_atim;
        times[1] = statbuf.st_mtim;
        if (futimens(fd, times) < 0)      /* reset times */
            err_ret("%s: futimens error", argv[i]);
        close(fd);
    }
    exit(0);
}

```

```

$ ls -l changemode times           look at sizes and last-modification t
-rwxr-xr-x  1 sar    13792 Jan 22 01:26 changemode
-rwxr-xr-x  1 sar    13824 Jan 22 01:26 times
$ ls -lu changemode times          look at last-access times
-rwxr-xr-x  1 sar    13792 Jan 22 22:22 changemode
-rwxr-xr-x  1 sar    13824 Jan 22 22:22 times
$ date                                print today's date
Fri Jan 27 20:53:46 EST 2012
$ ./a.out changemode times          run the program in Figure 4.21
$ ls -l changemode times            and check the results
-rwxr-xr-x  1 sar      0 Jan 22 01:26 changemode
-rwxr-xr-x  1 sar      0 Jan 22 01:26 times
$ ls -lu changemode times          check the last-access times also
-rwxr-xr-x  1 sar      0 Jan 22 22:22 changemode
-rwxr-xr-x  1 sar      0 Jan 22 22:22 times
$ ls -lc changemode times          and the changed-status times
-rwxr-xr-x  1 sar      0 Jan 27 20:53 changemode
-rwxr-xr-x  1 sar      0 Jan 27 20:53 times

```

ATTENZIONE: su mac non compila perché la struttura di stat è diversa: bisognerebbe mettere st_mtimespec e st_atimespec. Lo si vedeva nel manuale (cap. 2) di stat.

```

struct stat { /* when _DARWIN_FEATURE_64_BIT_INODE is defined */
    dev_t          st_dev;           /* ID of device containing file */
    mode_t         st_mode;          /* Mode of file (see below) */
    nlink_t        st_nlink;         /* Number of hard links */
    ino_t          st_ino;           /* File serial number */
    uid_t          st_uid;           /* User ID of the file */
    gid_t          st_gid;           /* Group ID of the file */
    dev_t          st_rdev;          /* Device ID */
    struct timespec st_atimespec;   /* time of last access */
    struct timespec st_mtimespec;   /* time of last data modification */
    struct timespec st_ctimespec;   /* time of last status change */
    struct timespec st_birthtimespec; /* time of file creation(birth) */
    off_t          st_size;          /* file size, in bytes */
    blkcnt_t       st_blocks;         /* blocks allocated for file */
    blksize_t      st_blksize;        /* optimal blocksize for I/O */
    uint32_t       st_flags;          /* user defined flags for file */
    uint32_t       st_gen;            /* file generation number */
    int32_t        st_lspare;         /* RESERVED: DO NOT USE! */
    int64_t        st_qspare[2];      /* RESERVED: DO NOT USE! */
};


```

NOTA: Il trucco di modifica dei tempi di accesso e di modifica si scopre però controllando l'ultima modifica dell'inode, che non si può manomettere.

Seguono poi i classici mkdir e rmdir

```

#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
int mkdirat(int fd, const char *pathname, mode_t mode);
int rmdir(const char *pathname);


```

Per cambiare la current working directory di un processo si usa la chiamata chdir:

```

#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int fd);


```

Per aprire una directory e leggerne il contenuto ci sono le seguenti funzioni:

```
#include <dirent.h>
DIR *opendir(const char *pathname);
DIR *fdopendir(int fd);

struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
int closedir(DIR *dp);
long telldir(DIR *dp);
void seekdir(DIR *dp, long loc);
```

Both return: pointer if OK, NULL on error

Returns: pointer if OK, NULL at end of directory or error

Returns: 0 if OK, -1 on error

Returns: current location in directory associated with *dp*

fopendir converte il file descriptor di una directory (aperta con open e flag O_DIRECTORY) in una struttura DIR.

La readdir permette di leggere il contenuto di una directory, e ogni volta si posiziona alla prossima lettura. Per questo motivo torna utile rewinddir che sposta il puntatore all'inizio della directory.

Telldir invece dice in che punto ci troviamo nella directory, e restituisce un long che va poi passato alla seekdir.

NOTA: non esiste writedir. Questo perché i modi in cui si "scrive" su una directory sono vari, tra cui una open con O_CREAT, oppure un semplice mkdir, e altri.

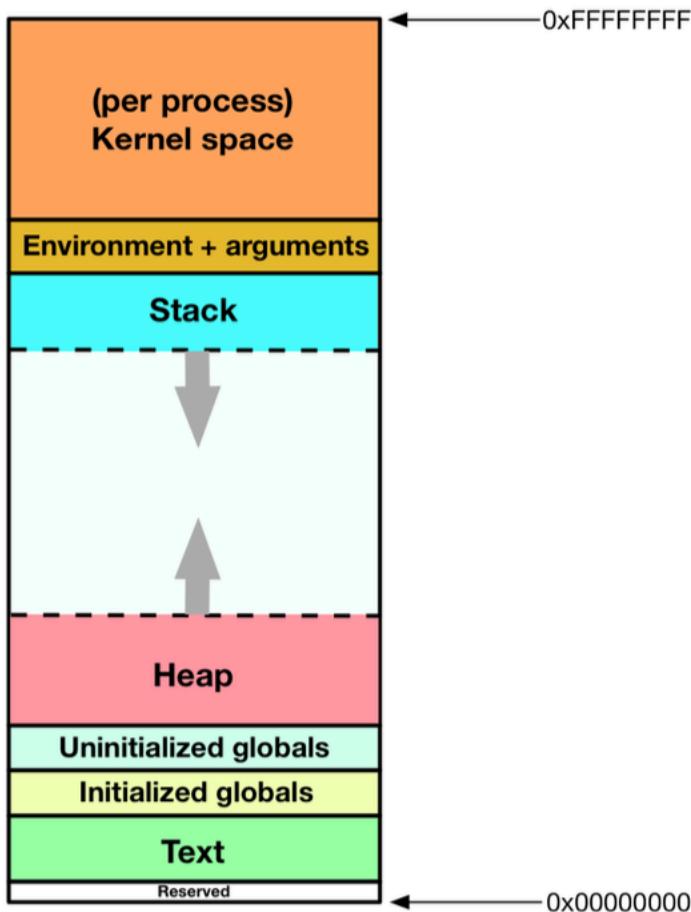
Facciamo una review sul funzionamento della **memoria** per ogni processo.

Se un processo cerca di accedere a una parte di memoria senza autorizzazione riceve un segnale di errore di segmentation fault. Se invece è usata la malloc per chiedere l'accesso alla memoria il kernel glielo da.

Il processo ragione con i 2^64 bit di memoria, ma poi deve sempre chiede al kernel il permesso di accedere alla memoria, e quindi lo spazio concesso alla fine sarà sempre molto meno. I programmati, infatti, ragionano su uno spazio di memoria virtuale di 2^64 bit di memoria (se siamo su una cpu a 64bit). Per questo è possibile che due processi pensino di avere due variabili diverse allo stesso indirizzo, perché la memoria virtuale è relativa al processo.

Vedremo poi come sarà possibile collegare due processi ognuno con i suoi spazi virtuali tramite la condivisione di variabili in posti della memoria fisica realmente esistenti, tramite la shared memory.

Figura 2.8. Organizzazione della memoria virtuale di un processo



NOTA: Immagine pensata per sistemi a 32 bit, quindi ci sono 8 cifre esadecimale 0x00000000

Sulla base la parte bianca è riservata, su mac os sono per esempio i primi 4gg, poi segue il testo (text) che indica il codice eseguibile. Tra i registri del processore ce n'è uno chiamato program counter (pc), che è memorizza l'indirizzo della prossima istruzione da eseguire. Esiste poi l'area delle initialized globals e uninitialized globals, che è dove sono memorizzate le variabili globali inizializzate e non.

Partendo dall'alto, invece, ci sono gli arguments, ovvero gli argomenti passati sulla riga di comando.

Poi c'è l'environnement: ovvero tutte le variabili "d'ambiente" del processo. In quello spazio di memoria vedremmo i caratteri ascii che compongono il nome della variabile, il carattere uguale e poi il valore della variabile scritta come ascii. Le variabili d'ambiente si susseguono una dopo l'altra e sono divise dal carattere null.

Se il programmatore usa nel codice una determinata variabile d'ambiente, si può cambiare il comportamento ridefinendo la variabile prima di lanciare il processo.

I processi figli ereditano l'ambiente dal padre.

Ci sono poi lo stack e l'heap. Lo stack è il metodo usato per lavorare con le funzioni, si comporta come una tendina: man mano che le funzioni sono chiamate, tutta la memoria relativa a loro viene aggiunta in questo spazio di memoria una dopo l'altra.

Se una funzione chiama un'altra funzione, la tendina si allunga. Questo meccanismo della tendina è necessario perché non si può prevedere quanto spazio sarà necessaria alla funzione, in quanto dipende dallo svolgimento.

NOTA: la dimensione dello stack è in realtà fissata a propri, ma può essere cambiata. E' possibile vederla con ulimit -a sul terminale. In questo caso si vedrebbe il limite di bash, ma è comunque ereditato dai processi lanciati da esso.

Quando la memoria si satura si può fare lo "swapping". Vuoi 200 giga di ram? Okay ma non ti servono tutti insieme, allora le cose non utilizzate le prendo e le porto sulla memoria di massa che è molto più ampia anche se ovviamente molto più lenta.

Non si può accedere all'heap senza chiedere l'autorizzazione. Un modo per chiedere l'autorizzazione è la malloc, che riserva una parte di memoria su richiesta per memorizzare delle variabili che persisteranno anche dopo la exit della funzione. Nell'heap vanno a finire anche le librerie dinamiche che sono caricate a runtime. Spesso questi codici sono condivisi: più processi le usano contemporaneamente e ognuno avrà nel suo spazio di memoria quel pezzo di codice (in sola lettura). Ovviamente questi codici nella memoria virtuale poi puntano sulla stessa memoria fisica, infatti vengono chiamati shared library.

NOTA: le librerie dinamiche vengono poste in parti inaspettate della memoria per non renderle accessibili agli attaccanti.

Debugger

Il debugger classico a linea di comando è gdb, mac ne usa uno più nuovo che è llDb.

```
[~ > which llDb  
/usr/bin/llDb  
[~ > which gdb  
gdb not found  
~ > ]
```

Vedi gdb reference card

Essential Commands

<code>gdb program [core]</code>	debug <i>program</i> [using coredump <i>core</i>]
<code>b [file:]function</code>	set breakpoint at <i>function</i> [in <i>file</i>]
<code>run [arglist]</code>	start your program [with <i>arglist</i>]
<code>bt</code>	backtrace: display program stack
<code>p expr</code>	display the value of an expression
<code>c</code>	continue running your program
<code>n</code>	next line, stepping over function calls
<code>s</code>	next line, stepping into function calls

Quando si compila il flag da dare su ggdb è -ggdb

Dopo che si compila li lancia il comando llDb passando come argomento il compilato

NOTA: premendo invio è lanciato il comando precedente

Comandi utili:

- l vedi il codice sorgente
- b numero riga per aggiunge un breakpoint
- r arg per fare il run passando gli argomenti r arg1 arg2...
- p var per printare la variabile var
- si usano i comandi n per andare alla nuova riga saltando la funzione, s per entrarci dentro

Usefuldbcommands permettono di leggere la memoria a un determinato indirizzo. Copiati i comandi del file txt per vederne il funzionamento.

Esplorando la memoria si vedono prima gli argomenti passati, e poi se si va oltre ci sono le variabili d'ambiente che sono memorizzate come stringhe, a differenza delle variabili normali che sono una corrispondenza tra indirizzo e valore

C'è il comando backtrace bt che mostra gli stackframe che sono presenti.

Si notano due stack frame libdyld che è una libreria dinamica chiamata per gestire gli eseguibili, e nota che si trova molto in alto nello spazio del kernel. Si vede così anche l'indirizzo della funzione main per esempio, che è molto in basso perché fa parte del "testo", dell'eseguibile.

Per vedere l'indirizzo delle variabili p &var e vedi l'indirizzo di una variabile, che potrebbe essere la prima dello stack

Qual è la punta dello stack? Fin dove arriva lo stack? Si mette l'indirizzo dello stack pointer \$sp

Si possono anche leggere i registri con l'istruzione register read. Quando li leggiamo la r prima del nome del registro significa che leggiamo l'intero registro, e significa che vogliamo i 4 byte meno significativi, etc

rax è il registro dove si mette il numero della systemcall

quando si fa una chiamata a funzione normale in genere semplicemente si salta all'indirizzo della funzione, e c'è una chiamata del processore che ti riporta all'istruzione seguente alla call della funzione. Poi c'è il registro rsp che contiene lo stack pointer. Da rbx a rsi sono i passaggi dei parametri a funzione e a systemcalls. Poi rbp è la base dello stackframe, rsp è la punta dello stack. Per dimensioni stack si fa base – punta. Un altro registro importante è rip che è l'istruzione pointer, ovvero dice l'indirizzo della prossima istruzione a essere eseguita che sarà nel testo in basso. C'è poi un istruzione disasseble che mostra il codice eseguibile come istruzioni del processore. Fa vedere le istruzioni della funzione nella quale mi trovo. Quando si fa una chiamata a systemcall c'è una callq a un indirizzo che è uno stub che è un pezzo di codice che fa da segnaposto della systemcall, e poi si arriva dopo nelle librerie (che se sono dinamiche non sono dove c'è il testo ovviamente).

E' possibile eseguire il codice istruzioni di processore una per una ovvero con le istruzioni assembly una per volta? Sì, tramite l'istruzione si

NOTA: in questo caso si mette prima il sorgente quando si fa mov per copiare

NOTA: a te su mac con processore arm i registri si chiamano diversamente quindi il set di istruzioni può cambiare ma la logica deve essere quella

NOTA: lo stack si scrive dall'alto verso il basso in maniera crescente, però sono visualizzati al contrario nel debugger.

Si nota che dalla punta dello stack a dove ci sono le poche variabili descritte nella nostra piccola main c'è un sacco di spazio, perché? Perché in questo codice in particolare c'era un buffer di 4096 byte.

NOTA: c'è un sito carino chiamato compiler explorer (goldbolt.org) che mostra la corrispondenza tra un'istruzione in un linguaggio di programmazione in assembly, figooo

Vediamo ora il big programma di cui parlavamo, ftw8.c .

C'è una variabile di tipo Myfync, e prima di questo c'è un typedef in cui viene creato un tipo che è una funzione. Successivamente viene passata come argomento a un'altra funzione, questo è il meccanismo dei plugin. In base a come si definisce questa funzione, si dice cosa fare a ogni nodo. Dove si definisce la funzione? In zona. La funzione lavora su un pathname, poi prende un puntatore a una struttura di tipo stat, e un tipo (che potrà assumere una serie di valori). C'è anche uno switch annidato che prende dalla struttura stat il campo st_node (prende il campo di una struttura tramite il puntatore, per questo la freccia). Nel campo st_node ci sono i permessi e il tipo di file. Si fa però un & con una maschera S_IFMT: si cerca quindi di capire il tipo di file. Viene quindi incrementata una variabile, che però non è dichiarata né tra i parametri né nella funzione, infatti è globale. Essendo globale è inutile che sia static (che serve a evitare di disalocarla quando esce dal blocco, quando esce dallo stack rimane lì. Rimane utilizzabile solo dalla funzione, però quando una funzione esce e rientra riesce a leggere il valore).

Questo plugin fa semplicemente una statistica dei file.

Anche qui il nome di una funzione è un puntatore, come con gli array e le stringhe.

Vediamo myftw che è dove si passa la funzione. Dentro questa funzione viene utilizzata e data a sua volta a un'altra funzione dopath.

NOTA: si passa sempre il puntatore più e più volte finché qualcuno non lo utilizza e chiama la funzione

Ricorda con il debugger: quando stai per entrare nella funzione, se facciamo sì entra nella funzione step by step, con n si tratta come una sola riga

Nota: la dopath chiama se stessa

```

#include "apue.h"
#include <dirent.h>
#include <limits.h>

/* function type that is called for each filename */
typedef int Myfunc(const char *, const struct stat *, int);

static Myfunc myfunc;
static int myftw(char *, Myfunc *);
static int dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int ret;

    if (argc != 2)
        err_quit("usage: ftw <starting-pathname>");

    ret = myftw(argv[1], myfunc); /* does it all */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1; /* avoid divide by 0; print 0 for all counts */
    printf("regular files = %7ld, %5.2f %%\n", nreg,
        nreg*100.0/ntot);
    printf("directories = %7ld, %5.2f %%\n", ndir,
        ndir*100.0/ntot);
    printf("block special = %7ld, %5.2f %%\n", nblk,
        nblk*100.0/ntot);
    printf("char special = %7ld, %5.2f %%\n", nchr,
        nchr*100.0/ntot);
    printf("FIFOs = %7ld, %5.2f %%\n", nfifo,
        nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink,
        nslink*100.0/ntot);
    printf("sockets = %7ld, %5.2f %%\n", nsock,
        nsock*100.0/ntot);
    exit(ret);
}

```

Analizziamo ora il codice con il debugger.

lldb ftw8

mettiamo un break alla riga 19 del controllo degli argomenti: b 19

facciamo il run del programma con un argomento: r .

con bt vedi lo stato dello stack

nota fullpath così avrà sia l'indirizzo della directory sia il path del corrispondente indirizzo

```

/* The caller's func() is called for every file.
 */
#define FTW_F 1      /* file other than directory */
#define FTW_D 2      /* directory */
#define FTW_DNR 3    /* directory that can't be read */
#define FTW_NS 4     /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */
static size_t pathlen;

static int             /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullPath = path_alloc(&pathlen);    /* malloc PATH_MAX+1 bytes */
                                         /* ({Prog pathalloc}) */
    if (pathlen <= strlen(pathname)) {
        pathlen = strlen(pathname) * 2;
        if ((fullPath = realloc(fullPath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    strcpy(fullPath, pathname);
    return(dopath(func));
}

/*
 * Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return. For a directory, we call ourself
 * recursively for each name in the directory.
 */
static int             /* we return whatever func() returns */
dopath(Myfunc* func)
{
    struct stat    statbuf;
    struct dirent  *dirp;
    DIR            *dp;
    int             ret, n;

    if (lstat(fullPath, &statbuf) < 0) /* stat error */
        return(func(fullPath, &statbuf, FTW_NS));
    if (S_ISDIR(statbuf.st_mode) == 0) /* not a directory */
        return(func(fullPath, &statbuf, FTW_F));

    /*
     * It's a directory. First call func() for the directory,
     * then process each filename in the directory.
     */
    if ((ret = func(fullPath, &statbuf, FTW_D)) != 0)
        return(ret);

    /*
     * It's a directory. First call func() for the directory,
     * then process each filename in the directory.
     */
    if ((ret = dopath(fullPath, func)) != 0)
        return(ret);

    n = strlen(fullPath);
    if (n + NAME_MAX + 2 > pathlen) { /* expand path buffer */
        pathlen *= 2;
        if ((fullPath = realloc(fullPath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    fullPath[n++] = '/';
    fullPath[n] = 0;

    if ((dp = opendir(fullPath)) == NULL) /* can't read directory */
        return(func(fullPath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0 || 
            strcmp(dirp->d_name, "..") == 0)
            continue; /* ignore dot and dot-dot */
        strcpy(&fullPath[n], dirp->d_name); /* append name after "/" */
        if ((ret = dopath(func)) != 0) /* recursive */
            break; /* time to leave */
    }
    fullPath[n-1] = 0; /* erase everything from slash onward */

    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullPath);
    return(ret);
}

```

```
static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;      break;
        case S_IFBLK:   nblk++;      break;
        case S_IFCHR:   nchr++;      break;
        case S_IFIFO:   nfifo++;     break;
        case S_IFLNK:   nslink++;    break;
        case S_IFSOCK:  nsock++;     break;
        case S_IFDIR:   /* directories should have type = FTW_D */
            err_dump("for S_IFDIR for %s", pathname);
        }
        break;
    case FTW_D:
        ndir++;
        break;
    case FTW_DNR:
        err_ret("can't read directory %s", pathname);
        break;
    case FTW_NS:
        err_ret("stat error for %s", pathname);
        break;
    default:
        err_dump("unknown type %d for pathname %s", type, pathname);
    }
    return(0);
}
```

Time and Date Routines

Permettono nel codice di contare il passaggio del tempo, ed è utile, per esempio, per fare i profiling del programma, ovvero per capire dove il programma è più lento e dove è più veloce.

```
#include <time.h>
time_t time(time_t *calptr);
```

time ritorna il tempo in secondi dall'epoch (1 Gennaio 1970), e l'output è memorizzato nel puntatore, oppure se si passa null è il valore di ritorno.

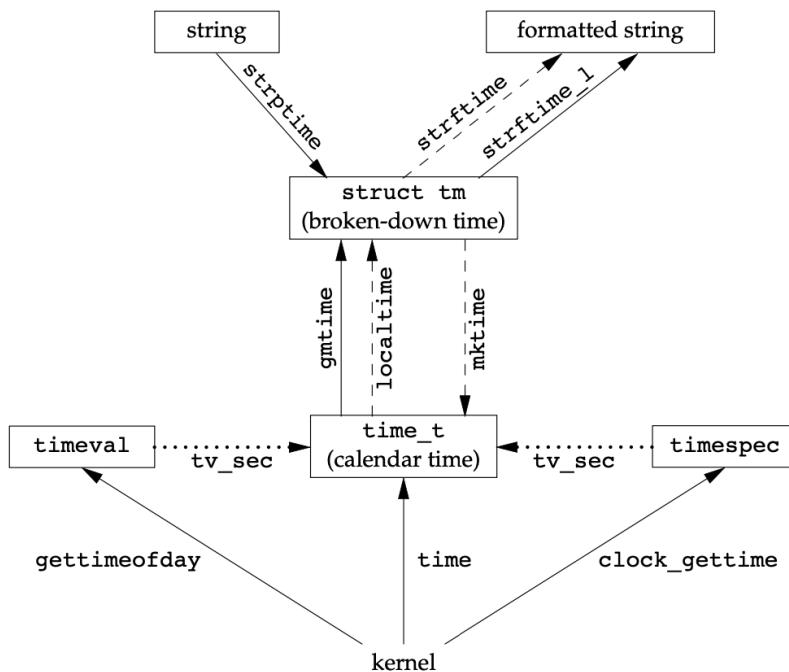


Figure 6.9 Relationship of the various time functions

Poi ci sono due altri funzioni fondamentali: `gettimeofday` e `clock_gettime`.

`Clock_gettime` ritorna i nanosecondi, invece `gettimeofday` restituisce secondi e nanosecondi. La freccia `tv_sec` indica che il valore restituito indica la stessa cosa con una precisione differente. Ci sono poi delle funzioni che lavorano sul numero di secondi in

mondo da renderli più semplici da leggere ad un essere umano, per esempio gmtime lo indica come UTC ma in ogni caso è utilizzata una struttura di tipo tm (vedi man)

```
struct tm {      /* a broken-down time */
    int tm_sec;   /* seconds after the minute: [0 - 60] */
    int tm_min;   /* minutes after the hour: [0 - 59] */
    int tm_hour;  /* hours after midnight: [0 - 23] */
    int tm_mday;  /* day of the month: [1 - 31] */
    int tm_mon;   /* months since January: [0 - 11] */
    int tm_year;  /* years since 1900 */
    int tm_wday;  /* days since Sunday: [0 - 6] */
    int tm_yday;  /* days since January 1: [0 - 365] */
    int tm_isdst; /* daylight saving time flag: <0, 0, >0 */
};
```

NOTA: controlla i possibili valori assunti dalle variabili della struttura, perché gennaio corrisponde a 0, domenica corrisponde a 0, e gli anni partono dal 1900

Per quanto riguarda la formattazione è utilizzata la funzione strftime.

Quando compiamo un miliardo di secondi? Cerca su moodle onebillion.

```
#include "apue.h"
#include <time.h>
#include <sys/time.h>
#include <sys/utsname.h>

int
main(void)
{
    struct tm *birthdate, *newdate;
    time_t a_time;
    int seconds;
    char str[100];

    birthdate = malloc(sizeof(struct tm));
    newdate = malloc(sizeof(struct tm));

    printf("Please write your birth's year (- 1900)\n");
    scanf("%d", &birthdate->tm_year);
    printf("Please write your birth's month (0-11)\n");
    scanf("%d", &birthdate->tm_mon);
    printf("Please write your birth's day (1-31)\n");
    scanf("%d", &birthdate->tm_mday);
    printf("Please write your birth's hour (0 - 23)\n");
    scanf("%d", &birthdate->tm_hour);
    printf("Please write your birth's minute (0 - 59)\n");
    scanf("%d", &birthdate->tm_min);
    printf("You were born: %s\n", asctime(birthdate));
    a_time=mktime(birthdate);
    printf("Please write how many seconds you want to add\n");
    scanf("%d", &seconds);
    a_time=a_time+seconds;
    newdate=localtime(&a_time);
    strftime(str ,100 , "The requested time is %+.\\n", newdate);
    printf("%s\\n", str);

    exit(0);
}
```

NOTA: occhio alle operazioni di sottrazione, potresti dover gestire tu il riporto

Process environment

NOTA: Quando si lancia un programma c'è sempre una routine che prepara l'ambiente e poi chiama il main, e ciò si può vedere tramite il debugger nello stack

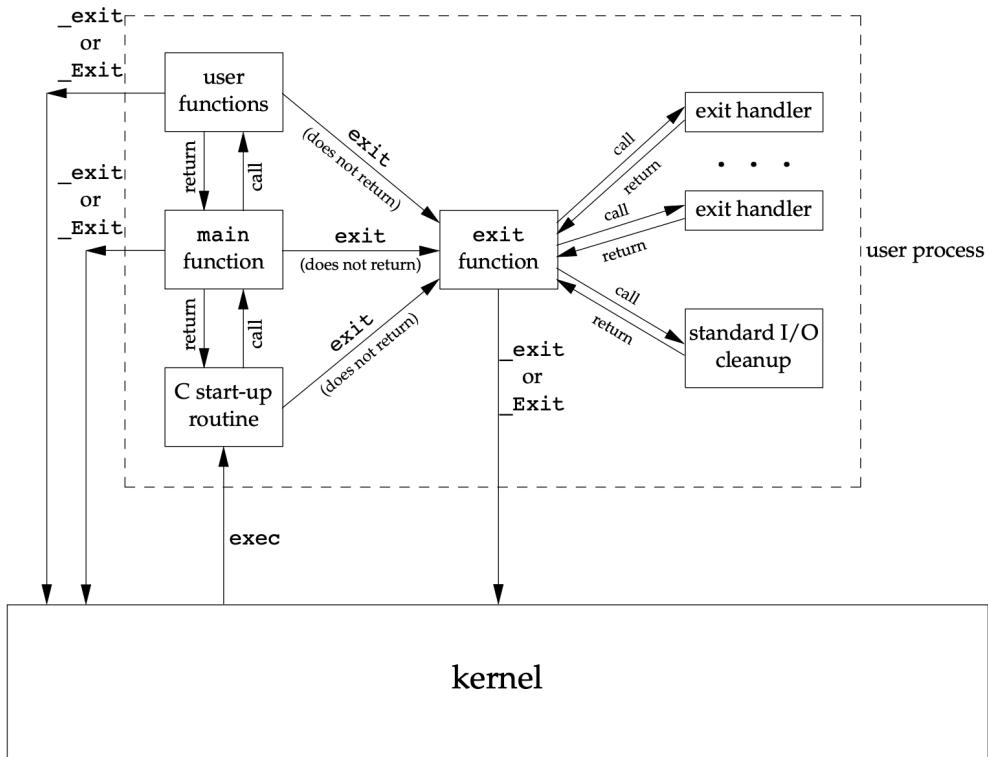
There are eight ways for a process to terminate. Normal termination occurs in five ways:

1. Return from `main`
2. Calling `exit`
3. Calling `_exit` or `_Exit`
4. Return of the last thread from its start routine (Section 11.5)
5. Calling `pthread_exit` (Section 11.5) from the last thread

Abnormal termination occurs in three ways:

6. Calling `abort` (Section 10.17)
7. Receipt of a signal (Section 10.2)
8. Response of the last thread to a cancellation request (Sections 11.5 and 12.7)

Bisogna infatti notare che se ci sono diversi thread, nonostante il main abbia finito il programma continua a girare finchè non hanno concluso tutti i thread.



Il kernel fa la exec, poi dopo la startup routine e si entra nella main function, la quale a sua volta farà chiamate a funzioni di utente e così via.

C'è una distinzione tra le exit e le _exit: le _exit sono sulla sinistra vanno direttamente al kernel, e tutto quello che riguarda il processo è perduto, se non qualche informazione che il kernel vuole dare al parent su come è finito il processo figlio. Se si usa invece exit si da la possibilità di gestire l'uscita dal programma, tramite le exit handler, in modo da dare la libertà di fare qualcosa prima di uscire dal programma. Da notare che a differenza del gestore di segnale che viene evocato in maniera asincrona, queste sono evocate in maniera sincrona. Prima della fine di un processo, c'è sempre la standard I/O cleanup.

La exit riguarda tutto il processo.

Esiste una funzione chiamata abort () che permette di terminare un processo, ma dando la possibilità di effettuare alcune operazioni prima della terminazione, tramite i gestori di exit. Bisogna gestire il segnale SIGABRT (mandato da questa funzione) con le operazioni da fare prima della terminazione. Infatti il segnale causerà in ogni caso la terminazione del programma.

NOTA: SIGABRT non può nemmeno essere bloccato da un cancello

NOTA: corrisponde a fare raise(SIGABRT)

```
#include <stdlib.h>
void abort(void);
```

Exit chiude la singola thread, abort chiude tutto.

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

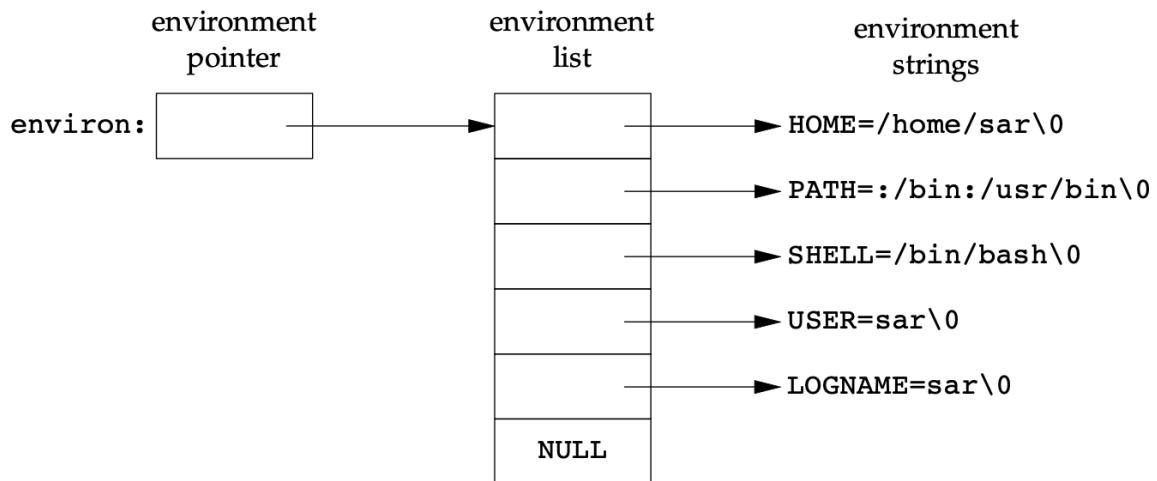
static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

NOTA: vengono eseguiti prima gli handler installati per ultimi.

NOTA: con i segnali si può mettere un solo gestore, qui se ne possono mettere di più.



Quando bash fa la exec, lo spazio di memoria del child è azzerato. Se si vuole fare conoscere al child una variabile, tramite la keyword export questa viene messa nell'environment dei child.

Ci sono due chiamate per vedere le variabili d'ambiente, env e set.

La differenza fondamentale è che set è un builtin, quindi viene eseguito nel bash corrente senza una exec, e mostrerà quindi solo le variabili d'ambiente del bash corrente. Env invece farà una fork e un exec creando un child, e mostrerà quindi le variabili d'ambiente di questo child di bash. Ci si accorge di questo comportamento quando si fa export di una variabile: con env la vediamo nelle variabili d'ambiente (perché crea un child), invece con set no perché rimaniamo nel bash iniziale.

La routine che lancia il programma piazza le variabili d'ambiente come stringhe nella memoria. In questo modo si può cambiare il valore di una variabile d'ambiente, ma rimanendo nel numero di caratteri iniziali.

Guardando la figura di sopra vediamo che le stringhe sono referenziate in un array di puntatori a stringhe. Attraverso questi puntatori arriviamo alle variabili d'ambiente.

In tutto questo i programmi possono dichiarare un puntatore all'array, (** si chiama handle). Questa variabile viene dichiarata come external, ovvero si dichiara ma poi sarà definita da qualche altra parte in un altro file.

Le variabili d'ambiente si raggiungono tramite extern char** environ, e storicamente unix prevede un terzo argomento alla main che è char * [] envp.

Per una variabile d'ambiente basta fare getenv path semplicemente.

La chiamata per conoscere una variabile d'ambiente è getenv, invece per settarla si usa putenv o setenv.

```

#include <stdlib.h>
char *getenv(const char *name);

#include <stdlib.h>
int putenv(char *str);

```

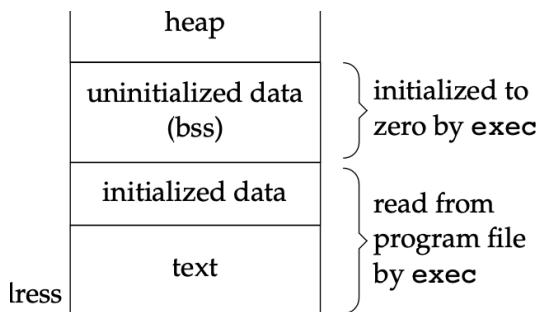
Returns: 0 if OK, nonzero on error

```

int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);

```

Both return: 0 if OK, -1 on error



Da notare che le variabili globali già inizializzate possono essere presenti già nell'exec.

Con size \$(which ls) si ottengono informazioni sulla dimensione del codice (text), sulle variabili già inizializzate (data), sulle variabili non inizializzate (bss).

```
$ size /usr/bin/cc /bin/sh
   text     data      bss      dec      hex  filename
346919      3576    6680  357175  57337  /usr/bin/cc
102134     1776   11272  115182  1c1ee  /bin/sh
```

NOTA: la malloc gestisce una lista delle parti di memoria allocate, in modo da non assegnare indirizzi che si accavallano. Inoltre, la memoria deve essere globale poiché altrimenti la perderemmo alla fine della funzione in cui è stata effettuata la malloc. Esisterà quindi una double linked list che memorizzerà tutti gli indirizzi assegnati dalla malloc.

C'è un programma semplice che stampa tutti gli indirizzi ed è "much simpler way to print addresses"

È interessante analizzare il programma memory_dump.c su moodle.

Il programma stampa tutte le variabili d'ambiente due volte. Nel mezzo però ne viene modificata una (term), e create due nuove.

Nel main viene prima chiamata la funzione dump, in cui viene chiamata la popen che restituisce un puntatore che è l'ingresso a una pipe. Questo comando servirà a ordinare l'ambiente.

Prima viene stampato l'initial environment con un sort della terza colonna (1 e 3 colonna indicano la stessa cosa in base diverse).

Viene sempre stampato il carattere & prima delle variabili per indicare che si sta mostrando l'indirizzo.

Poi sono stampati gli indirizzi delle funzioni fun1 e fun2 e si nota che sono in basso, il che conferma l'immagine che raffigurava la memoria, che posizionava il testo in basso. Invece fprintf è in alto, perché fa parte di una libreria dinamica.

Le complicazioni in questo programma vengono dal fatto che vogliamo vedere ciò che succede alle variabili d'ambiente quando le si creano o modificano. Per crearle o modificarle lui usa la chiamata putenv e setenv. Putenv però è più comoda perché prendere una stringa e quella stringa è tutto della variabile d'ambiente (nome = valore) e la mette direttamente nell'environment, invece setenv si passano due parametri nome e valore però questa stringa non può essere allocata sullo stack altrimenti quando la funzione in cui è memorizzata esce questo va in crisi.

Vengono quindi create due variabili d'ambiente. Dopo putenv l'effetto è ottenuto. Dove stanno però queste variabili d'ambiente?

Poi c'è il loop sugli elementi degli array. Sul banco di lavoro buf mettiamo alla prima colonna & che è il motivo per cui lo vediamo alla stampa iniziale.

La funzione strncpy si usa per copiare da buf + 1 perché il primo carattere lo abbiamo già copiato, e copiamo poi la stringa della variabile d'ambiente con size di buf – 2 (per l' eof finale), e si posiziona lo 0 al 32 carattere.

La funzione mkPrintable è utilizzata per permettere di stampare alcuni caratteri “non stampabili”, i quali vengono sostituiti con @.

buf è il nome della variabile d'ambiente di cui è stampato l'indirizzo, che viene passato come terzo argomento di dumpaddr.

NOTA: la stampa la fa quando è chiusa la pipe con pclose(p). Alla printf si butta dentro qualcosa, poi è nella logica della pipe di aspettare a stampare. Mette tutti i dati nella pipe fino alla fine della pipe in cui poi effettivamente stampa tutto.

NOTA: esiste una struttura dati chiamata pipe che è un array di 2 file descriptor. Se apri una pipe per esempio hai 2 fd che sono 3 e 4. Quando il processo fa una fork e crea un child ha anche lui 3 e 4 come file descriptor. Ora se il parent scrive a 3 il child può leggere da 4.

Lo vedremo bene dopo comunque.

Nella popen uno dei due processi siamo noi, l'altro lo posso dire come comando e quindi non bisogna fare manualmente la gestione di questi due file. Si dice manualmente ciò che c'è dall'altra parte della pipe.

Questa chiamata non è sicura, sarebbe meglio fare chiamata pipe, poi una fork e il child fa una exec del sort.

Quindi insomma in quel for di prima si passa in tutti gli elementi di envp iniziale, ogni volta si stampa quella roba.

Che succede poi? Strchr cerca dentro buf il carattere = e mette a 0 quello che era un =
Insomma, si è isolato il nome della variabile e poi vediamo se il nome della variabile è TERM
Sapendo che noi prima avevamo cambiato la variabile term, se noi ora stiamo passando da TERM, vogliamo capire dove sta. Viene usata la funzione getenv che dice se il valore è definito, mette in cp il puntatore a valore. Ora confronta i due valori, quello preso da getenv e quello a cui arriva attraverso il vettore envp. Se tutte queste condizioni sono verificate, ovvero se il puntatore vero è diverso da quello a cui puntava envp significa che la variabile è cambiata. Allora mette all'inizio la stringa @TERM e ci copia dentro il nuovo valore.

Ricorda che la getenv dice esattamente dove è la variabile, se hai cambiato la variabile la sposterà.

Tramite getenv e setenv succedono sotto il cofano cose che il sistema non dice: viene cambiato l'indirizzo delal variabile e questa variabile non la troviamo piu tramite l'array iniziale.

Questa gestione avviene in aree globali perché poi viene conosciuto ovunque.

NOTA: nell'array iniziale rimane anche il valore vecchio, ma poi ne viene salvata una da un'altra parte.

Nel main quindi c'è il dump dell'ambiente iniziale, poi le due creazioni delle variabili.

Ora, tutto sto programma è cosi complicato perche si vuole vedere come viene gestita sta cosa delle variabili d'ambiente, c'è un altro programma per vedere gli indirizzi delle cose c'è l'altro programma show_mem.c

Per vedere una mappa del processo c'è il comando `vmmmap -resident -interleaved`

`pidprocesso`

In linux invece: `cat /proc/pid_processo/maps`

Qui si possono vedere le aree di memoria per cui un processo ha permesso di accedere.

Si nota una grande distanza tra lo stack e la heap, chiamata STACK GUARD, che fa sì che non ci sia overlap tra questi spazi di memoria.

Si nota anche che ci sono una serie di preallocazioni piccole della malloc.

I `__TEXT` indicano le librerie dinamiche, e si può notare che sono presenti nella versione “leggi e scrivi” e “leggi ed esegui”, la quale è molto più grande. Per scrittura si intende il fatto che le librerie standard di I/O scrivano su dei buffer, per esempio.

Se ci si trova in un punto dello stack preceduto da molti stackframe, e si vuole cancellare tutto e tornare a una funzione in particolare, si possono usare le funzioni **setjmp** e **longjmp**.

In questo modo si velocizza il processo di “chiusura delle tendine”.

Si fa la chiamata **setjmp** nella funzione a cui si vuole tornare, e poi si chiama **longjmp** nel momento in cui vogliamo tornare indietro, cancellando tutti gli stackframe.

Prima si crea un oggetto globale **jmp_buf**, che si passa alla **setjump**. La prima volta che si chiama **setjmp** ritorna 0, al contrario del valore di **output** che avrà quando chiameremo la **longjmp**. Nella chiamata **longjmp** è infatti è passata prima la fotografia passata anche a **setjmp**, e poi un valore che è quello che poi ritornerà la **setjmp** quando torneremo indietro nel tempo. In base a questi valori è possibile sapere da che situazione stiamo tornando indietro, in modo da poter avere comportamenti diversi.

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD      5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

. . .

void
cmd_add(void)
{
    int    token;

    token = get_token();
    if (token < 0)    /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

LIMITI

Tutti i processi hanno dei limiti su ciò che possono fare, e possono essere visualizzati tramite le seguenti funzioni:

```
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);
```

Queste funzioni prendono un intero che indica la risorse da confrontare/modificare, e poi una struttura.

```
struct rlimit {
    rlim_t rlim_cur; /* soft limit: current limit */
    rlim_t rlim_max; /* hard limit: maximum value for rlim_cur */
};
```

Ci sono due tipi di limite, il soft limit e l'hard limit.

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

Alcuni esempi di limiti che un processo ha sono:

<code>RLIMIT_AS</code>	The maximum size in bytes of a process's total available memory. This affects the <code>sbrk</code> function (Section 1.11) and the <code>mmap</code> function (Section 14.8).
<code>RLIMIT_CORE</code>	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
<code>RLIMIT_CPU</code>	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the <code>SIGXCPU</code> signal is sent to the process.
<code>RLIMIT_DATA</code>	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap from Figure 7.6.
<code>RLIMIT_FSIZE</code>	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the <code>SIGXFSZ</code> signal.
<code>RLIMIT_MEMLOCK</code>	The maximum amount of memory in bytes that a process can lock into memory using <code>mlock(2)</code> .
<code>RLIMIT_MSGQUEUE</code>	The maximum amount of memory in bytes that a process can allocate for POSIX message queues.
<code>RLIMIT_NICE</code>	The limit to which a process's nice value (Section 8.16) can be raised to affect its scheduling priority.
<code>RLIMIT_NOFILE</code>	The maximum number of open files per process. Changing this limit affects the value returned by the <code>sysconf</code> function for its <code>_SC_OPEN_MAX</code> argument (Section 2.5.4). See Figure 2.17 also.
<code>RLIMIT_NPROC</code>	The maximum number of child processes per real user ID. Changing this limit affects the value returned for <code>_SC_CHILD_MAX</code> by the <code>sysconf</code> function (Section 2.5.4).
<code>RLIMIT_NPTS</code>	The maximum number of pseudo terminals (Chapter 19) that a user can have open at one time.
<code>RLIMIT_RSS</code>	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
<code>RLIMIT_SBSIZE</code>	The maximum size in bytes of socket buffers that a user can consume at any given time.
<code>RLIMIT_SIGPENDING</code>	The maximum number of signals that can be queued for a process. This limit is enforced by the <code>sigqueue</code> function (Section 10.20).
<code>RLIMIT_STACK</code>	The maximum size in bytes of the stack. See Figure 7.6.
<code>RLIMIT_SWAP</code>	The maximum amount of swap space in bytes that a user can consume.
<code>RLIMIT_VMEM</code>	This is a synonym for <code>RLIMIT_AS</code> .

NOTA: un limite interessante è quello per il nicevalue, che indica la priorità che può avere un processo nello scheduling.

Vediamo ora il programma 7.16

```

#include "apue.h"
#include <sys/resource.h>

#define doit(name) pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
#ifdef RLIMIT_AS
    doit(RLIMIT_AS);
#endif
    #endif
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
#ifdef RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    #endif
    #endif
    doit(RLIMIT_MSGQUEUE);
    doit(RLIMIT_NICE);
    #endif
    doit(RLIMIT_NOFILE);
#ifdef RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
    #endif
    #endif
    doit(RLIMIT_NPTS);
    doit(RLIMIT_RSS);
    #endif
    #endif
    doit(RLIMIT_SBSIZE);
}

```

NOTA: #name significa che viene passato il nome della risorsa senza sostituire il valore

Per esempio l'output potrebbe essere:

```

$ ./a.out
RLIMIT_AS      (infinite) (infinite)
RLIMIT_CORE    (infinite) (infinite)
RLIMIT_CPU     (infinite) (infinite)
RLIMIT_DATA    536870912 536870912
RLIMIT_FSIZE   (infinite) (infinite)
RLIMIT_MEMLOCK (infinite) (infinite)
RLIMIT_NOFILE  3520      3520
RLIMIT_NPROC   1760      1760
RLIMIT_NPTS    (infinite) (infinite)
RLIMIT_RSS     (infinite) (infinite)
RLIMIT_SBSIZE  (infinite) (infinite)
RLIMIT_STACK   67108864 67108864
RLIMIT_SWAP    (infinite) (infinite)
RLIMIT_VMEM    (infinite) (infinite)

```

NOTA: ulimit -a può essere usato per mostrare i limiti di bash. Per cambiare un limite si può fare per esempio ulimit -c 1000 . Ulimit mostra solo i soft, ma se cambiamo un valore forse cambiano sia i soft che gli hard.

Process control

Seguono alcune funzioni fondamentali:

```
#include <unistd.h>
pid_t getpid(void);
>Returns: process ID of calling process

pid_t getppid(void);
>Returns: parent process ID of calling process

uid_t getuid(void);
>Returns: real user ID of calling process

uid_t geteuid(void);
>Returns: effective user ID of calling process

gid_t getgid(void);
>Returns: real group ID of calling process

gid_t getegid(void);
>Returns: effective group ID of calling process
```

Cosa succede con i file descriptor quando un parent fa una fork?

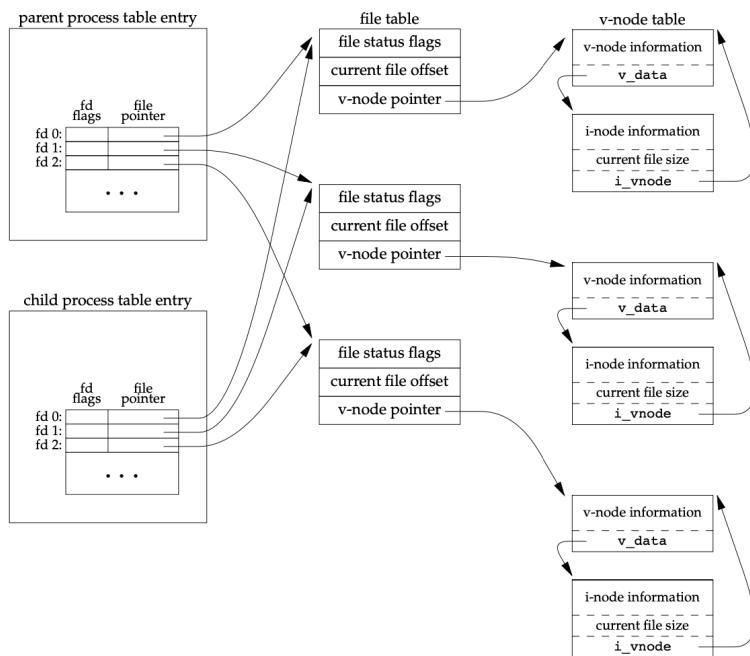


Figure 8.2 Sharing of open files between parent and child after `fork`

Bisogna notare che i file descriptor vengono ereditati (copiati) nel child, ma puntano alle stesse file table. Se quindi si cambiano i flag (tramite `fcntl`), o qualsiasi valore appartenente a quella tabella, child e parent ne risentiranno.

La funzione `fork` è la seguente, e ritorna 0 nel child e il pid del child al parent.

```
#include <unistd.h>
pid_t fork(void);
```

Ci sono una serie di elementi ereditati dai child:

- Real user ID, real group ID, effective user ID, and effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

Le differenze invece sono

- The return values from `fork` are different.
- The process IDs are different.
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change.
- The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are set to 0 (these times are discussed in Section 8.17).
- File locks set by the parent are not inherited by the child.
- Pending alarms are cleared for the child.
- The set of pending signals for the child is set to the empty set.

Una funzione fondamentale è la wait.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or -1 on error

Quando il parent fa un child, questo avrà vita indipendente. Se il parent vuole avere però informazioni su come è finito il processo child, può usare la funzione wait.

È quello che succede quando lanciamo un comando su bash: questo fa una fork, poi la exec e una wait per aspettare la fine del programma e mostrarcici l'output.

Finchè il parent non fa la wait, il kernel non può buttare l'informazione sul processo child. Questa situazione porta alla creazione di processi zombie, ovvero tutte quelle informazioni sulla fine dei child, che però non essendo state richieste dal parent, il kernel deve mantenere in memoria.

NOTA: la wait è una chiamata bloccante, ovvero aspetta finchè il child non muoia

NOTA: quando mandiamo un programma in background su bash, fondamentalmente evitiamo di fare la wait bloccante che altrimenti ci farebbe aspettare la terminazione del processo child

NOTA: una tecnica per evitare di fare tante wait quanti child, è fare una doppia fork. Ovvero il primo child fa uno o più child. Questo perché basterebbe fare la wait del child di prima generazione. Quando un parent muore, i child vengono adottati dal processo 1, ma non si sarà necessario fare la wait per ognuno dei child orfani. Vedi figura 8.8.

Un modo per non dover fare la wait e aspettare che il child termini l'esecuzione, è l'utilizzo di un segnale: quando il parent riceve questo segnale dal child, significa che esso ha terminato l'esecuzione e può quindi fare una wait per ottenere informazioni sul child ed eliminare il processo zombie, senza dover attendere la sua terminazione.

Il segnale è SIGCHILD, che viene mandato al parent quando un child termina l'esecuzione.

NOTA: La wait non fa nessun riferimento al processo che stiamo aspettando, appena un qualsiasi child termina, la wait è completata.

Se si è interessati a un particolare processo, si può usare la funzione waitpid. Questa funzione, può essere usata sia come bloccante che no.

Il primo parametro della wait pid può essere uno dei seguenti, mentre l'ultimo decide se la chiamata deve essere bloccante o meno.

The interpretation of the *pid* argument for `waitpid` depends on its value:

- | | |
|------------------|--|
| <i>pid</i> == -1 | Waits for any child process. In this respect, <code>waitpid</code> is equivalent to <code>wait</code> . |
| <i>pid</i> > 0 | Waits for the child whose process ID equals <i>pid</i> . |
| <i>pid</i> == 0 | Waits for any child whose process group ID equals that of the calling process. (We discuss process groups in Section 9.4.) |
| <i>pid</i> < -1 | Waits for any child whose process group ID equals the absolute value of <i>pid</i> . |

Constant	Description
<code>WCONTINUED</code>	If the implementation supports job control, the status of any child specified by <i>pid</i> that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI option).
<code>WNOHANG</code>	The <code>waitpid</code> function will not block if a child specified by <i>pid</i> is not immediately available. In this case, the return value is 0.
<code>WUNTRACED</code>	If the implementation supports job control, the status of any child specified by <i>pid</i> that has stopped, and whose status has not been reported since it has stopped, is returned. The <code>WIFSTOPPED</code> macro determines whether the return value corresponds to a stopped child process.

Figure 8.7 The *options* constants for `waitpid`

Per esaminare l'output del child, vengono utilizzate alcune macro che analizzano il parametro (*status*) passato come input.

Macro	Description
<code>WIFEXITED(status)</code>	True if <i>status</i> was returned for a child that terminated normally. In this case, we can execute <code>WEXITSTATUS(status)</code> to fetch the low-order 8 bits of the argument that the child passed to <code>exit</code> , <code>_exit</code> , or <code>_Exit</code> .
<code>WIFSIGNALED(status)</code>	True if <i>status</i> was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute <code>WTERMSIG(status)</code> to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro <code>WCOREDUMP(status)</code> that returns true if a core file of the terminated process was generated.
<code>WIFSTOPPED(status)</code>	True if <i>status</i> was returned for a child that is currently stopped. In this case, we can execute <code>WSTOPSIG(status)</code> to fetch the signal number that caused the child to stop.
<code>WIFCONTINUED(status)</code>	True if <i>status</i> was returned for a child that has been continued after a job control stop (XSI option; <code>waitpid</code> only).

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`

```

#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t    pid;
    int      status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)           /* child */
        exit(7);

    if (wait(&status) != pid)    /* wait for child */
        err_sys("wait error");
    pr_exit(status);            /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)           /* child */
        abort();                  /* generates SIGABRT */
    if (wait(&status) != pid)    /* wait for child */
        err_sys("wait error");
    pr_exit(status);            /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)           /* child */
        status /= 0;              /* divide by 0 generates SIGFPE */
    if (wait(&status) != pid)    /* wait for child */
        err_sys("wait error");
    pr_exit(status);            /* and print its status */

    exit(0);
}

```

Figure 8.6 Demonstrate various `exit` statuses

Segue la `pr_exit`, che prende un intero ricevuto dalla `wait` e ne stampa lo status.

```

#include "apue.h"
#include <sys/wait.h>

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
               WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
               WTERMSIG(status),
               #ifdef WCOREDUMP
               WCOREDUMP(status) ? " (core file generated)" : "");
    #else
               "");
    #endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
               WSTOPSIG(status));
}

```

Dove `WEXITSTATUS` è una macro che funziona come una maschera.

Esistono delle situazioni chiamate **Race Conditions**.

In queste situazioni succede che il risultato delle elaborazioni cambia ogni volta che viene lanciato il programma poiché dipende da come il kernel ha gestito lo scheduling.

C'è un esempio di programma che ha questo problema, figura 8.12:

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t    pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char      *ptr;
    int       c;
    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

charattime stampa un carattere alla volta, ma nel time slice dato a un processo non si riesce a stampare tutta la stringa.

ESERCIZIO: inserire dentro la funzione un piccolo ritardo in modo da rallentare il computer fino a capire quanto si deve rallentare in modo da fare emergere la race condition.

Un modo per evitare le race condition è una sincronizzazione tramite l'utilizzo di segnali.

Dentro apue sono definite delle funzioni che permettono la comunicazione tra parent e child.

La soluzione è in tellwait2.c

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t pid;

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        WAIT_PARENT(); /* parent goes first */
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
        TELL_CHILD(pid);
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char *ptr;
    int c;

    setbuf(stdout, NULL); /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Dove

```
static volatile sig_atomic_t sigflag; /* set nonzero by sig handler */

static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo) /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
}

void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    /* Block SIGUSR1 and SIGUSR2, and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2); /* tell parent we're done */
}
```

```

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* and wait for parent */
    sigflag = 0;

    /* Reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}

void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1);           /* tell child we're done */
}

void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* and wait for child */
    sigflag = 0;

    /* Reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}

```

Seguono le exec functions:

```

#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);

```

Sono caratterizzate dalle ultime lettere del nome, ma cosa indicano queste lettere? Indicano

il tipo di parametri di cui hanno bisogno:

L -> lista

v -> vettore

e -> environment, ovvero l'ambiente è passato tramite un vettore invece che tramite la variabile ENVIROM

p -> path, ovvero si utilizza la variabile d'ambiente path per trovare l'eseguibile

Function	<i>pathname</i>	<i>filename</i>	<i>fd</i>	Arg list	<i>argv[]</i>	<i>environ</i>	<i>envp[]</i>
<code>exec1</code>	•			•		•	
<code>execlp</code>		•		•		•	
<code>execle</code>	•			•			•
<code>execv</code>	•				•	•	
<code>execvp</code>		•			•	•	
<code>execve</code>	•		•		•		•
<code>fexecve</code>					•		•
(letter in name)		p	f	l	v		e

Figure 8.14 Differences among the seven `exec` functions

NOTA: gli argomenti iniziano con un puntatore ad `argv[0]` e finiscono con il cast di 0 a carattere per esplicitare la terminazione della stringa.

NOTA: il nome viene quindi ripetuto due volte, una per il path, e una per `argv0`.

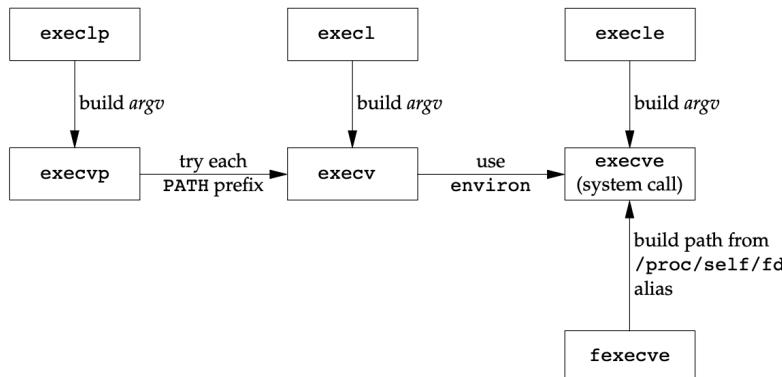


Figure 8.15 Relationship of the seven `exec` functions

We've mentioned that the process ID does not change after an `exec`, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Nice value (on XSI-conformant systems; see Section 8.16)
- Values for `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`

RICORDA: la gestione dei file aperti dipende dal close on exec e dal file descriptor flag

Vediamo ora l'esempio 8.16

```
#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                   "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

NOTA: bisogna cambiare il path del programma della exec.

La prima volta non eredita l'ambiente e lo diciamo noi, la seconda volta invece lo eredita.

Nota che echoall visita le variabili d'ambiente e le stampa:

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int      i;
    char    **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++) /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++) /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

Interpreter files

L'interpreter file viene definito dalla she bang line #!

Se noi mettiamo un programma chiamato pippo, si cerca di fare eseguire a pippo questo file di testo con la she bang. Poi se pippo sa fare qualcosa sono affari suoi.

NOTA: si possono anche passare opzioni all'interprete

NOTA: in qualche modo è un modo di avere un programma che fa delle cose anche se manca il compilatore mettendo un interpreter diverso????? Boh

Esiste una funzione per niente sicura, ovvero la system.

```
#include <stdlib.h>
int system(const char *cmdstring);
```

Process Times

Vediamo ora il process times, che si ricollegano al risultato del comando time su bash.

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

NOTA: si chiama times a differenza da time (che dà i secondi passati dalla epoch)

```
struct tms {
    clock_t tms_utime; /* user CPU time */
    clock_t tms_stime; /* system CPU time */
    clock_t tms_cutime; /* user CPU time, terminated children */
    clock_t tms_cstime; /* system CPU time, terminated children */
};
```

Time lavora con i clock ticks, e non con i secondi. Cosa sono i ticks? Sono una suddivisione del secondo in ticks, che dipende dalla macchina ma in genere indicano i centesimi del secondo. Si può verificare tramite la sysconf, oppure controllando negli standard.

I primi due campi della struttura sono gli stessi mostrati dal comando time, gli altri due invece indicano i tempi consumati dai children.

NOTA: il tempo di orologio viene ritornato dalla funzione direttamente, ma è sempre in ticks.

NOTA: il valore di output di questa funzione è inutile se non calcolato per differenza rispetto a un altro punto nel codice.

Vediamo l'esempio 8.31

```

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int
main(int argc, char *argv[])
{
    int      i;

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]); /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd)      /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmssend;
    clock_t     start, end;
    int         status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1) /* starting values */
        err_sys("times error");

    if ((status = system(cmd)) < 0)          /* execute command */
        err_sys("system() error");

    if ((end = times(&tmssend)) == -1) /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmssend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmssend)
{
    static long      clkck = 0;

    if (clkck == 0) /* fetch clock ticks per second first time */
        if ((clkck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");

    printf("  real:  %7.2f\n", real / (double) clkck);
    printf("  user:  %7.2f\n",
           (tmssend->tms_utime - tmsstart->tms_utime) / (double) clkck);
    printf("  sys:   %7.2f\n",
           (tmssend->tms_stime - tmsstart->tms_stime) / (double) clkck);
    printf("  child user: %7.2f\n",
           (tmssend->tms_cutime - tmsstart->tms_cutime) / (double) clkck);
    printf("  child sys:  %7.2f\n",
           (tmssend->tms_cstime - tmsstart->tms_cstime) / (double) clkck);
}

```

\$./a.out "sleep 5" "date" "man bash >/dev/null"

command: sleep 5
real: 5.01
user: 0.00
sys: 0.00
child user: 0.00
child sys: 0.00
normal termination, exit status = 0

command: date
Sun Feb 26 18:39:23 EST 2012
real: 0.00
user: 0.00
sys: 0.00
child user: 0.00
child sys: 0.00
normal termination, exit status = 0

command: man bash >/dev/null
real: 1.46
user: 0.00
sys: 0.00
child user: 1.32
child sys: 0.07
normal termination, exit status = 0

Process Control Chapter 8

Figure 8.31 Time and execute all command-line arguments

Questo programma esegue una serie di comandi passati come argomento, mandando in output i tempi di esecuzione.

Esiste un ulteriore funzione chiamata getrusage, che da molte altre informazioni riguardo un processo.

```
struct rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long ru_maxrss;          /* max resident set size */
    long ru_ixrss;           /* integral shared text memory size */
    long ru_idrss;           /* integral unshared data size */
    long ru_isrss;           /* integral unshared stack size */
    long ru_minflt;          /* page reclaims */
    long ru_majflt;          /* page faults */
    long ru_nswap;           /* swaps */
    long ru_inblock;          /* block input operations */
    long ru_oublock;          /* block output operations */
    long ru_msgsnd;          /* messages sent */
    long ru_msgrcv;          /* messages received */
    long ru_nssignals;        /* signals received */
    long ru_nvcsw;           /* voluntary context switches */
    long ru_nivcsw;          /* involuntary context switches */
};
```

E' infatti così possibile vedere tutta la memoria utilizzata dal processo (condivisa e non).

Per page fault si intende quando il processo ha bisogno di una pagina (pezzo di memoria di 4096 byte) ma non è stata mappata, non si trova.

NOTA: questa funzione non da però il tempo di orologio.

NOTA: anche qui i valori devono essere calcolati relativamente a un altro momento

NOTA: per avere il tempo di orologio si dovrebbe usare la gettimeofday

SCUSA MA SE POSSO USARE GETTIMEofday SIGNIFICA CHE NON E' RELATIVO IL TEMPO,

GIUSTO????

Oggi iniziamo a vedere il codice di kerrisk:

dopo aver fatto il make, per non dover rifare il make ogni volta ci copiamo una riga di compilazione:

```
cc -std=c99 -D_XOPEN_SOURCE=600 -D_DEFAULT_SOURCE -g -I../lib  
-pedantic -Wall -W -Wmissing-prototypes -Wno-sign-compare -  
Wimplicit-fallthrough -Wno-unused-parameter    ${1}.c  
../liblpi.a   -o $1
```

oppure

```
cc -std=c99 -D_XOPEN_SOURCE=600 -D_DEFAULT_SOURCE -g -I../lib  
-pedantic -Wall -W -Wmissing-prototypes -Wno-sign-compare -  
Wimplicit-fallthrough -Wno-unused-parameter  
ptmr_sigev_thread.c ../liblpi.a  ../liblpi.a -lrt -pthread -  
o ptmr_sigev_thread
```

PRENDI QUESTA CHE HA FUNZIONATO LE ALTRE MI SA CHE HANNO DATO ERRORE

```
cc -std=c99 -D_XOPEN_SOURCE=600 -D_DEFAULT_SOURCE -g -I../lib  
-pedantic -Wall -W -Wmissing-prototypes -Wno-sign-compare -  
Wimplicit-fallthrough -Wno-unused-parameter -pthread  
thread_cancel.c ../liblpi.a  ../liblpi.a -pthread -o  
thread_cancel
```

oppure quello di Filippo

```
cc -std=c99 -D_XOPEN_SOURCE=600 -D_DEFAULT_SOURCE -g -I../lib -  
pedantic -Wall -W -Wmissing-prototypes -Wno-sign-compare -Wimplicit-  
fallthrough -Wno-unused-parameter ${1}.c ../libtlpi.a ../libtlpi.a -pthread -o  
${1}
```

Andiamo ora a togliere dal makefile.inc l'opzione -lrt che serve per segnali a realtime

NOTA: hai molti meno eseguibili degli altri, forse ti servono le librerie. Su linux dovrebbe andare tutto linearmente.

Core Files

L'azione di default di molti segnali ha come azione di default quella di rilasciare un core, ovvero l'immagine di uno stato del processo.

NOTA: per vedere la dimensione massima del core ulimit -a. Questa caratteristica è una caratteristica del processo, e siccome ulimit è un builtin ci dà le informazioni di bash, ma in ogni caso i limiti sono ereditati anche dai children. Ulimit -c unlimited per settare a illimitata la dimensione massima delle core image.

Dove viene rilasciato il core?

Bisogna prima controllare sia attiva la possibilità di rilasciare un corefile, tramite sysctl

```
./tlpi-dist > sysctl -a | grep coredump
kern.coredump: 1
kern.sugid_coredump: 0
./tlpi-dist > sysctl -a | grep corefile
kern.corefile: /cores/core.%P
kern.drivercorefile: /cores/core.%P
./tlpi-dist >
```

In moodlis c'è "a program which causes a core dump" e "commands to examine core files with llDb and gdb"

```

/*
Un programma che, cercando di dereferenziare
un puntatore nullo p (all'interno della funzione a)
causa il dumping di un core.

perché il core sia rilasciato occorre che
ulimit -c non ritorni 0 nello shell.
Si può dare il comando
ulimit -c unlimited

Inoltre kern.coredump = 1

I core si trovano in /cores

linea di compilazione

gcc -Wall -ggdb producecore.c -o producecore

*/
int a (int *p);

int
main (void)
{
    int *p = 0; /* null pointer */
    return a (p);
}

int
a (int *p)
{
    int y = *p;
    return y;
}

```

Commands to examine core files with lldb and gdb:

First of all, for both tools, the following command must be executed:

ulimit -c unlimited (be careful: the limitation is PER PROCESS, changing window will not work)
On Mac OS also:
sysctl kern.coredump=1

Then, for lldb:

- compile the offending program with -ggdb
- execute it, then:
- lldb ./producecore -c /cores/core.24592
(the last argument is the path to core file; Mac OS X put them in /cores)
- make sure the core has been produced exactly by that executable

Another way to do the same is:

lldb ./producecore

then, once in lldb:

target create -c /cores/core.24592

Beware: lldb commands are different from gdb's.

The gdb command is

gdb EXECUTABLE-FILE CORE-FILE

NOTA: su linux usa lldb, su mac gdb

Poiché si è compilato il file con il debugger è possibile conoscere l'istruzione problematica

Esiste un comando molto utile per vedere tutte le systemcall effettuate da un programma, è chiamato **strace**.

Su mac strace però si chiama dtruss, il cui utilizzo è complicato per questioni di sicurezza.

NOTA: da vedere su linux dovrebbe essere meno problematico

Utilizzando l'opzione -t vengono mostrati i tempi di esecuzione, e più t vengono messe più i tempi sono precisi (fino a 3 t)

E' possibile stampare maggiore informazioni tramite le opzioni -v o -s (credo)

La chiamata access è utilizzata per vedere se si hanno i privilegi di fare una determinata operazione.

Nella openat si vede che il sistema usa le cache per ricordare dove sono le librerie.

Process group and sessions

E' utile scaricare le slide "process group and sessions review" da moodle.

Ora si spiegherà come funziona il meccanismo che permette con un solo segnale (control c) di interrompere più processi (come nel caso di una pipe).

Ogni processo ha un process group id, che definisce il gruppo al quale appartiene. In genere il processo eredita il process group id del parent, nonostante sia poi possibile cambiare gruppo. Lo spostamento da un gruppo all'altro lo può fare il parent fino a chè non è fatta una exec, o direttamente il child tramite la chiamata setpgid.

NOTA: se si setta pid 0, il processo che deve cambiare gruppo è sé stesso, e se si setta il gruppo a 0, si crea un nuovo gruppo con il proprio pid (di cui il processo relativo diventa leader)

Esiste poi il concetto di sessione: c'è una gerarchia a due livelli, dove prima c'è una sessione che contiene diversi gruppi, all'interno dei quali ci sono i processi.

Per poter cambiare process group id, infatti, è necessario che entrambi i gruppi (di sorgente e destinazione) facciano parte della stessa sessione.

Esiste anche il session leader, che nel nostro caso su macos è spesso il "login" che genera la shell.

Un esempio: la finestra di terminale è la sessione, i gruppi sono composti dai processi mandati in pipe.

Tutti i processi di un job (comando o pipeline) devono essere all'interno di un singolo process group, in modo che sia possibile mandare segnali a tutto il gruppo. Se infatti viene

mandato un segnale tramite il comando kill con un pid negativo, significa che il segnale

verrà inviato a tutti i processi appartenenti al gruppo

Bash, in base a come vengono dati i comandi, prima di fare una exec gestisce i gruppi... i

programmi di tutto questo non ne sanno nulla.

Nella realtà, quando bash fa una fork, prima della exec, crea un altro gruppo apposito per

questo processo e lo mette dentro. Chi fa questo cambiamento del gruppo? In realtà lo

fanno entrambi, perché quando hai una fork non sai quale dei due processi verrà eseguito

prima e quale dopo; quindi, per sicurezza il codice per il cambiamento di gruppo si mette in

entrambi i processi, e poi viene fatta la exec. Non si può infatti essere sicuri che il child

cambi pgid prima che il parent inizi a mandargli segnali, oppure che invece lo faccia il parent

prima che il child faccia la exec: ci troveremmo in una race condition in questi casi

Segue un codice di esempio:

```
1 pid_t childPid;
2 pid_t pipelinePgid; /* PGID to which processes in a pipeline are to be assigned */
3
4 /* Other code */
5
6 childPid = fork();
7 switch (childPid) {
8 case -1: /* fork() failed */
9     /* Handle error */
10 case 0: /* Child */
11     if (setpgid(0, pipelinePgid) == -1)
12         /* Handle error */
13     /* Child carries on to exec the required program */
14 default: /* Parent (shell) */
15     if (setpgid(childPid, pipelinePgid) == -1 && errno != EACCES)
16         /* Handle error */
17     /* Parent carries on to do other things */
18 }
```

Da notare che pipelinePgid ci è dato in qualche modo.

```

10:31:13 -> find / -name weqweewqwe 2> /dev/null &
[1] 55024
10:31:31 -> ./test_setpgid 55024

parent before the change: pid=55039      pgid=55039      psid=29155      ppid=29156
parent after the change: pid=55039      pgid=55024      psid=29155      ppid=29156
child: pid=55040      pgid=55039      psid=29155      ppid=55039
10:32:22 -> █

```

```

10:31:22 -> ps -aj
USER      PID  PPID  PGID   SESS  JOBC  STAT    TT      TIME COMMAND
root     15211  608 15211      0    0 Ss  s002   0:00.03 login -pf franco
franco  15212 15211 15212      0    1 S  s002   0:00.03 -bash
root    55029 15212 55029      0    1 R+  s002   0:00.00 ps -aj
root    29155  608 29155      0    0 Ss  s003   0:00.04 login -pfl franco /bin/bash -c exec -la bash /bin/
franco 29156 29155 29156      0    1 S+  s003   0:00.18 -bash
franco 55024 29156 55024      0    1 R  s003   0:02.19 find / -name weqweewqwe

```

```

.../francescodenu > find / -name "djenedjnde" 2> /dev/null &
[1] 3476
.../francescodenu > ps -aj
USER      PID  PPID  PGID   SESS  JOBC  STAT    TT      TIME COMMAND
root     3341  3340 3341      0    0 Ss  s000   0:00.01 login -pf francescodenu
francescodenu 3342 3341 3342      0    1 S  s000   0:00.04 -zsh
francescodenu 3476 3342 3476      0    1 UN  s000   0:01.15 find / -name djenedjnde
root     3478  3342 3478      0    1 R+  s000   0:00.00 ps -aj

```

NOTA: Perché find su mac ci mette più tempo? Forse cerca sulla rete, perché c'è un'opzione che fa specificare se si vuole cercare solo sul filesystem attuale. Con strace potremmo vedere in termini di systemcall cosa fa e dove cerca. In questo caso comunque ci fa comodo la sua “lentezza”.

NOTA: ogni volta che si lancia un programma, viene creato un gruppo con quel processo, nel caso di una pipe invece di creare un gruppo per tutti i processi.

NOTA: ps -aj per vedere tutte informazioni riguardo i process groups e sessions

NOTA: su macos la sessione non è stampata, ma siccome c'è una corrispondenza biunivoca tra sessione e terminale, capiamo che è la stessa sessione in base al campo TT o TTY che indica il terminale.

Per interpretare le lettere nel campo stat, si può fare un man ps cercando la parola “leader”.

```

I      Marks a process that is idle (sleeping for longer than
      about 20 seconds).
R      Marks a runnable process.
S      Marks a process that is sleeping for less than about 20
      seconds.
T      Marks a stopped process.
U      Marks a process in uninterruptible wait.
Z      Marks a dead process (a "zombie").

```

Additional characters after these, if any, indicate additional state information:

```

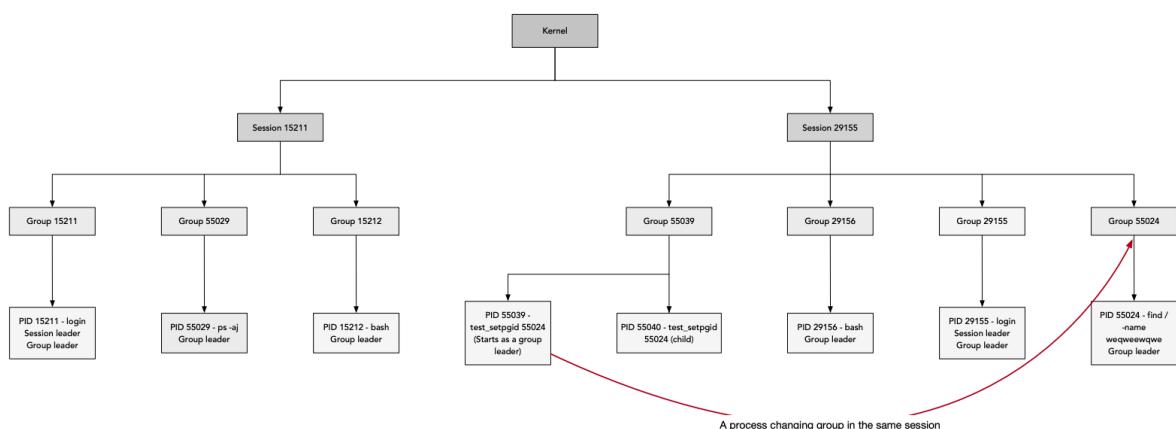
+      The process is in the foreground process group of its
      control terminal.
<      The process has raised CPU scheduling priority.
>      The process has specified a soft limit on memory
      requirements and is currently exceeding that limit;
      such a process is (necessarily) not swapped.
A      the process has asked for random page replacement
      (VA_ANOM, from vadvise(2), for example, lisp(1) in a
      garbage collect).
E      The process is trying to exit.
L      The process has pages locked in core (for example, for
      raw I/O).
N      The process has reduced CPU scheduling priority (see
      setpriority(2)).
S      The process has asked for FIFO page replacement
      (VA_SEQL, from vadvise(2), for example, a large image
      processing program using virtual memory to sequentially
      address voluminous data).
s      The process is a session leader.
V      The process is suspended during a vfork(2).

```

NOTA: il session leader su mac è il login e devi fare ps -e per vederlo perche non è un processo del tuo utente, su linux invece è bash

Segue la gerarchia di gruppi e sessioni, dell'esempio precedente in cui il parent si sposta in un altro gruppo (del find in questo caso).

Sessions and groups hierarchy

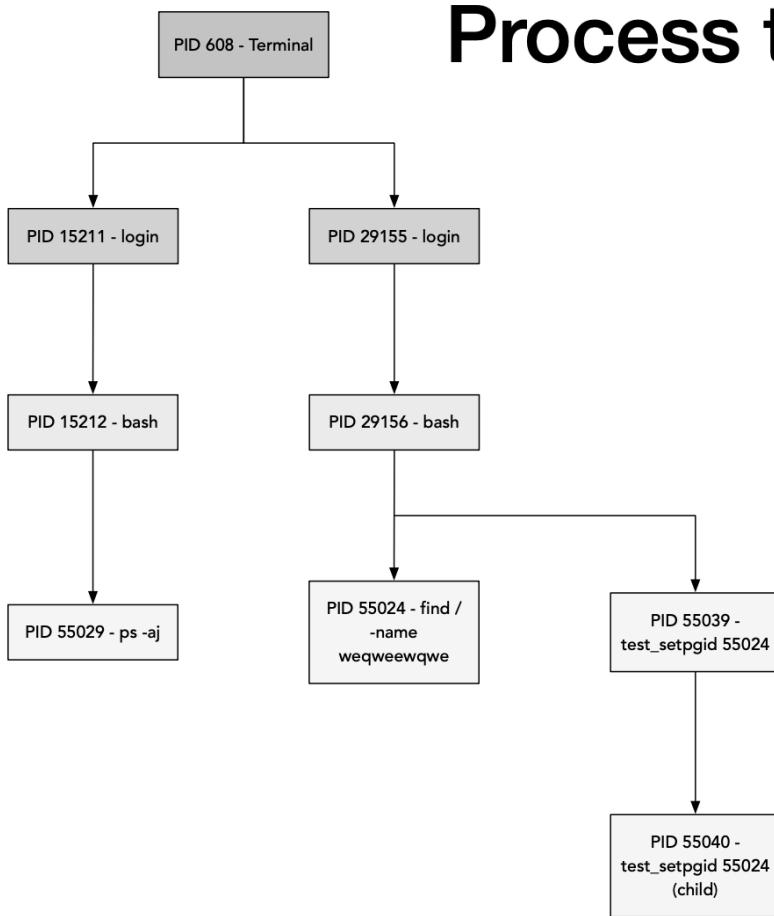


Sulla sinistra c'è ciò che accade normalmente: il kernel crea una sessione per il terminale, e il primo gruppo creato è quello di login, perché in realtà bash è figlio di login. Poi è creato un gruppo per bash, il quale normalmente crea un gruppo per ogni processo che viene lanciato, quindi si può vedere il gruppo del processo ps.

Sulla destra invece c'è una complicazione: viene lanciato il nostro programma (e creato quindi un gruppo ad'hoc) il quale avrà un child, che automaticamente farà parte dello stesso gruppo del parent. Quello che il nostro programma fa, è spostare poi il parent in un altro gruppo, in questo caso quello del fine.

segue ora invece la gerarchia di processi che già conoscevamo, che è tutt'altro discorso rispetto a quello dei gruppi e delle sessioni.

Process tree



Il programma lanciato è il seguente:

https://moodliis.unisalento.it/pluginfile.php/7276/mod_resource/content/2/test_setpgid.c

```
#include "apue.h"
#include <errno.h>

int main(int argc, char *argv[])
{
    pid_t pid;
    int pgid;
    if (argc != 2)
        err_quit("Usage: test_setpgid <An existing PGID in the same session>");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) { /* parent */
        perror("fork failed");
        pid = getpid();

        /* Print data about the parent before group switch */
        fprintf(stdout,"parent before the change: \tpid=%d\tpgid=%d\tps= %d\tppid=%d\n",
                pid,(int)getpgid(pid),(int)getsid(pid),(int)getppid());

        /* The parent tries to change its group to that passed as an argument */

        if ((pgid = setpgid(0,atoi(argv[1])) < 0)) { err_sys("creat error"); }

        /* Print data about the parent after group switch */

        fprintf(stdout,"parent after the change: \tpid=%d\tpgid=%d\tps= %d\tppid=%d\n",
                pid,(int)getpgid(pid),(int)getsid(pid),(int)getppid());
        sleep(35);           /* to allow reading IDs in another window by "ps -aj" */
    } else {               /* child */
        pid = getpid();

        /* Print child data */

        fprintf(stdout,"child (IDs do not change): \tpid=%d\tpgid=%d\tps= %d\tppid=%d\n",
                pid,(int)getpgid(pid),(int)getsid(pid),(int)getppid());
        sleep(35);
    }
    exit(0);
}
```

NOTA: questo codice usa err_quit, bisogna quindi metterlo dentro apue

NOTA: ps non dà il numero di sessione su mac, ma la chiamata getsid si

Insomma, tutto questo meccanismo permette la gestione del job control, in modo che

quando si fa per esempio control z e poi si fa bg, tutti i processi del gruppo vengano stoppati
e poi rimessi in bg.

```

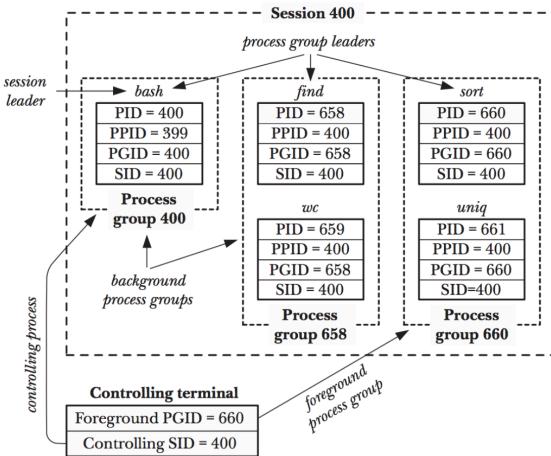
$ echo $$                                Display the PID of the shell
400
$ find / 2> /dev/null | wc -l &        Creates 2 processes in background group
[1] 659
$ sort < longlist | uniq -c          Creates 2 processes in foreground group

```

Figure 1

At this point, the shell (*bash*), *find*, *wc*, *sort*, and *uniq* are all running.

How a job-control shell sets the process group ID of a child process



Le systemcall utilizzate sono quindi:

```
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

Returns: process group ID if OK, -1 on error

If *pid* is 0, the process group ID of the calling process is returned. Thus

`getpgid(0);`

is equivalent to

`getpgrp();`

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

Returns: 0 if OK, -1 on error

This function sets the process group ID to *pgid* in the process whose process ID equals *pid*. If the two arguments are equal, the process specified by *pid* becomes a process group leader. If *pid* is 0, the process ID of the caller is used. Also, if *pgid* is 0, the process ID specified by *pid* is used as the process group ID.

```
#include <unistd.h>
pid_t getsid(pid_t pid);
```

Returns: session leader's process group ID if OK, -1 on error

If *pid* is 0, *getsid* returns the process group ID of the calling process's session leader. For security reasons, some implementations may restrict the calling process from obtaining the process group ID of the session leader if *pid* doesn't belong to the same session as the caller.

```
#include <unistd.h>
pid_t setsid(void);
```

Returns: process group ID if OK, -1 on error

If the calling process is not a process group leader, this function creates a new session. Three things happen.

1. The process becomes the *session leader* of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
2. The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
3. The process has no controlling terminal. (We'll discuss controlling terminals in the next section.) If the process had a controlling terminal before calling *setsid*, that association is broken.

Vediamo ora quali sono le chiamate per gestire i gruppi in foreground e in background

We need a way to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals (Figure 9.7).

```
#include <unistd.h>
pid_t tcgetpgrp(int fd);
```

Returns: process group ID of foreground process group if OK, -1 on error

```
int tcsetpgrp(int fd, pid_t pgrp_id);
```

Returns: 0 if OK, -1 on error

The function *tcgetpgrp* returns the process group ID of the foreground process group associated with the terminal open on *fd*.

If the process has a controlling terminal, the process can call *tcsetpgrp* to set the foreground process group ID to *pgrp_id*. The value of *pgrp_id* must be the process group ID of a process group in the same session, and *fd* must refer to the controlling terminal of the session.

NOTA: ovviamente i segnali da tastiera sono mandati al gruppo in foreground

Come lo shell gestisce il fatto che anche chi è in background vuole usare terminale (come input o output) ? Quando un processo in background vuole usare il terminale in lettura, questo riceve il segnale SIGTTIN, che normalmente stoppa il processo, ma può essere gestito. Allo stesso modo quando un processo in bg vuole scrivere sul terminale, riceve il segnale SIGTTOU.

Il processo puo decidere di ignorare questo segnale e scrivere lo stesso sullo stdout.

NOTA: Potrebbe essere bash a ignorare questo segnale, tanto sarebbe ereditato dal child

Vediamo degli esempi presi da moodlis: CELEBT

Sono 3 programmi, in cui il primo ha un problema e gli altri cercano di risolverlo.

```
int main() {
    pid_t pid;
    int status;

    if (fork() == 0)
        if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
            perror("tcgetpgrp() error");
        else {
            printf("original foreground process group id of stdout was %d\n",
                   (int) pid);
            if (setpgid(getpid(), 0) != 0)
                perror("setpgid() error");
            else {
                printf("now setting to %d\n", (int) getpid());
                if (tcsetpgrp(STDOUT_FILENO, getpid()) != 0)
                    perror("tcsetpgrp() error");
                else if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
                    perror("tcgetpgrp() error");
                else
                    printf("new foreground process group id of stdout was %d\n",
                           (int) pid);
            }
        }
    else wait(&status);
}
```

Viene stampato dal child chi è in fg, poi il child crea un altro gruppo dove si mette, che poi setta in fg e viene stampato.

Questo programma si blocca a "now setting...", perché? Perche quando dopo vuole stampare sul terminale, riceve il segnale SIGTTOU che per default fa lo stop. Si suppone che se ora si manda il segnale SIGCONT, il processo dovrebbe ripartire. Quello che succede però è un interrupted system call. Comportamento un po inaspettato che dopo grandi ricerche non siamo riusciti a spiegarci...

Il problema potrebbe essere che è il child a fare il cambiamento di gruppo.

Sono proposte allora 2 soluzioni, nella prima sia child che parent fanno il cambiamento di gruppo del child.

```
int main() {
    pid_t pid, pid2;
    int status;

    if ((pid2 = tcgetpgrp(STDOUT_FILENO)) < 0)
        perror("tcgetpgrp() error");
    else {
        printf("original foreground process group id of stdout was %d\n", (int) pid2);
    }
    pid = fork();
    if (pid == 0)
    {
        if (setpgid(getpid(), 0) != 0)
            perror("setpgid() error");
    }
    else {
        if (setpgid(pid, 0) != 0)
            perror("setpgid() error");
        printf("now setting to %d\n", pid);
        if (tcsetpgrp(STDOUT_FILENO, pid) != 0)
            perror("tcsetpgrp() error");
        else if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
            perror("tcgetpgrp() error");
        else
            printf("new foreground process group id of stdout was %d\n", (int) pid);
        fflush(stdout);
        wait(&status);
    }
}
```

L'altra soluzione è più radicale, è quello che in genere fa uno shell: ignorare il segnale SIGTTOU

```
int main() {
    pid_t pid;
    int status;

    if (fork() == 0)
    {
        signal(SIGTTOU, SIG_IGN);
        if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
            perror("tcgetpgrp() error");
        else {
            printf("original foreground process group id of stdout was %d\n",
                   (int) pid);
            if (setpgid(getpid(), 0) != 0)
                perror("setpgid() error");
            else {
                printf("now setting to %d\n", (int) getpid());
                if (tcsetpgrp(STDOUT_FILENO, getpid()) != 0)
                    perror("tcsetpgrp() error");
                else if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
                    perror("tcgetpgrp() error");
                else
                    printf("new foreground process group id of stdout was %d\n", (int) pid);
                fflush(stdout);
            }
        }
    }
    else wait(&status);
}
```

NOTA: se fai control z lo stoppi e devi ancora decidere se mandarlo in bg o fg

Si cerca ora di risolvere le anomalie riscontrate nel primo codice, si scarica allora da

moodlelist “a better versione of CELEBT10.c”

Sono state aggiunte le gestioni dei segnali SIGTTOU e SIGTTIN in modo che stampino la ricezione del segnale, e ora il comportamento su mac a linux è lo stesso.

Mostra il primo fg process group, poi lo vuole portare a 2255, ma il processo 2255 riceve il segnale 22 SIGTTOU, però la systemcall fatta dal child tcsetpgrp è interrotta e non ripare..

Poi il foreground process group rimane lo stesso, e il child scrive exiting.

RICORDA:SIGTTOU in genere stoppa il processo

Quello che non si capisce, è

- 1) Non è chiaro perché tcsetpgrp sia interrotta.

Facciamo allora su linux strace per vedere quali sono le vere systemcall e cosa succede.

Cosa succede a strace quando c'è un child? Usiamo l'opzione -f per seguire il child, che è quello che vogliamo seguire perché fa la chiamata strana che causa l'errore.

Strace -ttt -f ./a_better_versionof_celebt10

Quando arriva il child, il processo fa una ioctl (classica chiamata a coltellino svizzero) e sarà legata alla tcget

Poi ne fa un'altra dove in ioctl mette un'altra costante allo scopo di fare la set invece della get

Mentre fa la seconda chiamata per settare, si vede che succede il pasticcio

Il problema è che questa system call fatta dal child per qualche motivo è lenta e "riceve un interrupt" (?)

Ora abbiamo provato a togliere il gestore di segnale e vogliamo vedere se succedono le stesse cose, e anche qui è coerente comunque. Su linux invece dopo che si blocca il processo quando si manda SIGCONT in realtà, invece di bloccarsi il child a casa si sigtout, non lo fa

La chiamata setsid crea una nuova sessione e il processo che la chiama diventa leader di questa. Process group id, id della sessione e pid del leader saranno allora uguali.

Quando un processo parte ed è leader della propria sessione non ha controlling terminal, e anche se lo aveva in precedenza, lo perde.

Tutti i processi di una sessione possono avere un solo controlling terminal.

Quando si crea una nuova sessione, per stabilire il terminale bisogna aprire un file terminale tramite una open con il flag “O_NOCTTY”. Questo flag si setta quando si vuole aprire un terminale ma non si vuole che sia controlling terminal.

Un esempio di terminale non di controllo, è ciò che succede quando si scrive su un terminale tramite cat > /dev/ttyterminaleascelta

Quando un session leader apre un controlling terminal, diventa il processo di controllo per il terminale

NOTA: Se il processo che chiama setsid è un process group leader, fallisce

Quando si chiude il teminale di controllo, tutti i processi lanciati tramite esso terminano, perché viene mandato il segnale **SIGHUP** che di default li uccide.

Se vogliamo che un comando lanciato continui a girare anche dopo la chiusura del terminale, si mette prima del comando “nohup”, che permette di ignorare il segnale.

NOTA: nel caso dei daemon, non avendo terminale, SIGHUP per convenzione gestisce il segnale e rilegge i file di configurazione. In questo modo è possibile cambiare la configurazione del server senza doverlo riavviare.

Il terminale ha una serie di configurazioni, che possono essere gestite da riga di comando con stty, stty -a si vede la configurazione corrente. Nota che – significa che quella caratteristica è stata tolta.

Segnali

Per mandare segnali a un processo si usa la kill, invece per mandarli a sé stesso si usa la raise:

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

NOTA: esiste anche il segnale numero 0, che serve solo a sapere se un processo esiste ancora

In genere i segnali sono asincroni, perché arrivano in maniera inaspettata, tranne alcune situazioni, per esempio nel caso di una divisione per 0, in cui sono sincroni.

Ci sono due segnali che non possono essere né ignorati, né gestiti: sigkill e sigstop.

Alcuni segnali sono:

SIGILL -> quando c'è un eseguibile con istruzione illegale per il processore.

SIGQUIT -> control \ ed è un'alternativa a control c

SIGSEGV -> accedi a memoria di cui non hai possibilità di accedere segmentation fault

RICORDA: su bash il gestore di segnali è trap

Ci sono due segnali particolari il cui utilizzo è riservato all'utente, e sono SIGUSR1 e SIGUSR2

È fondamentale ricordare che l'ignorare un segnale viene ereditato dai child anche dopo la exec.

La funzione per gestire i segnali è la signal:

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

Nonostante sembra intricata, in realtà prende semplicemente un intero e la funzione da eseguire. Questa funzione ritorna la disposizione precedente del segnale, ovvero SIG_DFL.

Per ignorare il segnale si passa la costante `SIG_IGN`, e per settare il default `SIG_DFL`.

In generale, c'è da fare una distinzione tra le funzioni e le systemcall. La funzione è una sequenza di istruzioni utente e viene interrotta all'arrivo di un segnale senza che essa venga completata. Le systemcall invece sono eseguite direttamente dal kernel e sono più veloci, tanto che spesso non possono essere interrotte poiché terminano velocemente. Ci sono però anche delle systemcall lente, come la `read`, che ci mettono molto tempo, e possono essere interrotte. Se la systemcall fallisce perché interrotta da un segnale, `errno` assume il valore `EINTR`.

La signal prima era problematica perché dopo la ricezione di un segnale gestito, c'era un frangente di tempo in cui il gestore di segnali si reinstallava, e in cui quindi il segnale non era gestito.

Ci sono una serie di funzioni che possono essere usate senza problemi all'interno dei gestori di segnale, ovvero le reentrant functions (funzioni rientranti). Queste funzioni sono quelle in cui entra qualcosa nella black box, ed il risultato che esce è basato solo sull'input e sul contenuto della blackbox, ovvero non necessitano di nulla al di fuori di sé stesse (come variabili o strutture globali). Le funzioni non reentrant (come la `malloc`) non dovrebbero essere usate nei gestori di segnali.

Alcune funzioni che sicuramente sono sicure da usare:

abort	faccessat	linkat	select	socketpair
accept	fchmod	listen	sem_post	stat
access	fchmodat	lseek	send	symlink
aio_error	fchown	lstat	sendmsg	symlinkat
aio_return	fchownat	mkdirat	sendto	tcdrain
aio_suspend	fcntl	mkdirat	setgid	tcflow
alarm	fdatasync	mkfifo	setpgid	tcflush
bind	fexecve	mknodat	setsid	tcgetattr
cfgetispeed	fork	mknodat	setsockopt	tcgetpgrp
cfgetospeed	fstat	open	setuid	tcsendbreak
cfsetispeed	fstatat	openat	shutdown	tcsetattr
cfsetospeed	fsync	pause	sigaction	tcsetpgrp
chdir	ftruncate	pipe	sigaddset	time
chmod	futimens	poll	sigdelset	timer_getoverrun
chown	getegid	posix_trace_event	sigemptyset	timer_gettime
clock_gettime	geteuid	pselect	sigfillset	timer_settime
close	getgid	raise	sigismember	times
connect	getgroups	read	signal	umask
creat	getpeername	readlink	sigpause	uname
dup	getpgrp	readlinkat	sigpending	unlink
dup2	getpid	recv	sigprocmask	unlinkat
execl	getppid	recvfrom	sigqueue	utime
execle	getsockname	recvmsg	sigset	utimensat
execv	getsockopt	rename	sigsuspend	utimes
execve	getuid	renameat	sleep	wait
_Exit	kill	rmdir	socketmark	waitpid
_exit	link		socket	write

Figure 10.4 Reentrant functions that may be called from a signal handler

Esiste il concetto di segnali pendenti: un processo ha la possibilità di bloccare la consegna di un segnale a sé stesso, il segnale è stato quindi generato ma non consegnato. Questo segnale rimane pendente finché il processo non lo sblocca.

Il sistema è in grado di decidere come comportarsi quando arriva un segnale ma viene bloccato.

Questo meccanismo a volte è usato per gestire sezioni critiche di codice che non devono essere assolutamente bloccate.

Ci sono due utili funzioni, la alarm e la pause:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Alarm è la funzione “sveglia a orologeria”. Dopo un tot di secondi scatta un segnale che se non viene gestito uccide il programma. E’ possibile disinnescare la sveglia con alarm(0)

```
#include <unistd.h>
int pause(void);
```

La pause invece ferma un processo finché non gli viene mandato un segnale qualsiasi.

Una tecnica è: alarm -> genera segnale -> gestione che mette in pausa il programma.

Si può però creare un problema di race condition, se a causa dello scheduling del kernel, il segnale che dovrebbe svegliare il processo dalla “pause” arriva prima che la pausa sia effettivamente stata fatta.

Vediamo un semplice esempio:

```
#include <signal.h>
#include <unistd.h>

static void
sig_alarm(int signo)
{
    /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int seconds)
{
    if (signal(SIGALRM, sig_alarm) == SIG_ERR)
        return(seconds);
    alarm(seconds);      /* start the timer */
    pause();            /* next caught signal wakes us up */
    return(alarm(0));   /* turn off timer, return unslept time */
}
```

Analizziamo ora un tipo di dato signal set, che verrà poi utilizzato come maschera per i segnali. Ha il funzionamento di una rastrelliera.

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);

All four return: 0 if OK, -1 on error

int sigismember(const sigset_t *set, int signo);

Returns: 1 if true, 0 if false, -1 on error
```

Per modificare la maschera sei segnali si usa la funzione sigprocmask.

RICORDA: la signalprocmask è ereditata dai processi anche dopo la exec, non vengono invece ereditate le disposizioni.

SIGPROCMASK(2)	System Calls Manual	SIGPROCMASK(2)
----------------	---------------------	----------------

NAME
sigprocmask – manipulate current signal mask

SYNOPSIS

```
#include <signal.h>

int
sigprocmask(int how, const sigset_t *restrict set,
            sigset_t *restrict oset);
```

DESCRIPTION

The **sigprocmask()** function examines and/or changes the current signal mask (those signals that are blocked from delivery). Signals are blocked if they are members of the current signal mask set.

If set is not null, the action of **sigprocmask()** depends on the value of the parameter how. The signal mask is changed as a function of the specified set and the current mask. The function is specified by how using one of the following values from <signal.h>:

SIG_BLOCK	The new mask is the union of the current mask and the specified <u>set</u> .
SIG_UNBLOCK	The new mask is the intersection of the current mask and the complement of the specified <u>set</u> .
SIG_SETMASK	The current mask is replaced by the specified <u>set</u> .

If oset is not null, it is set to the previous value of the signal mask. When set is null, the value of how is insignificant and the mask remains unset providing a way to examine the signal mask without modification.

Il primo parametro determina cosa bisogna fare con la maschera che si sta passando, può essere uno dei 3 mostrati sopra (block, unblock setmask). Poi si passa la maschera da settare, e nel terzo parametro viene salvata la vecchia maschera.

Per ispezionare la vecchia maschera, si usa la funzione vista sopra sigismember.

Tramite la funzione sigpending è possibile sapere quali sono i segnali pendenti.

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Vediamo ora l'esempio 10.15 che mostra come si può proteggere dai segnali una parte critica di codice.

```
#include "apue.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t      newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    /*
     * Block SIGQUIT and save current signal mask.
     */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5); /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

    /*
     * Restore signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");

    sleep(5); /* SIGQUIT here will terminate with core file */
    exit(0);
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
}
```

NOTA: la `sig_quit` la prima volta gestisce il segnale ma poi si disinstalla

In realtà come dicevamo prima la `signal` non era sicura poiché c'era un intervallo di tempo in cui si doveva reinstallare, per questo ora in realtà sotto il cofano usa la `sigaction`.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act,
             struct sigaction *restrict oact);
```

Returns: 0 if OK, -1 on error

Come primo parametro c'è il numero di segnale che vogliamo gestire, poi si passano due strutture sigaction.

```
struct sigaction {
    void     (*sa_handler)(int); /* addr of signal handler, */
                                /* or SIG_IGN, or SIG_DFL */
    sigset_t sa_mask;          /* additional signals to block */
    int      sa_flags;         /* signal options, Figure 10.16 */
};

/* alternate handler */
void     (*sa_sigaction)(int, siginfo_t *, void *);
};
```

Alcuni campi si rifanno ai segnali realtime, con non portano con sé solo il numero e l'istante in cui sono arrivati, ma anche altre informazioni.

I campi che interessano a noi sono 3: il primo che è la funzione da chiamare, poi sigset che permette di aggiungere altri segnali da bloccare (spesso NULL), e poi una serie di flag.

I flag che possono essere settati sono i seguenti:

Option	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
SA_INTERRUPT			•			System calls interrupted by this signal are not automatically restarted (the XSI default for <code>sigaction</code>). See Section 10.5 for more information.
SA_NOCLDSTOP	•	•	•	•	•	If <code>signo</code> is <code>SIGCHLD</code> , do not generate this signal when a child process stops (job control). This signal is still generated, of course, when a child terminates (but see the <code>SA_NOCLDWAIT</code> option below). When the XSI option is supported, <code>SIGCHLD</code> won't be sent when a stopped child continues if this flag is set.
SA_NOCLDWAIT	•	•	•	•	•	If <code>signo</code> is <code>SIGCHLD</code> , this option prevents the system from creating zombie processes when children of the calling process terminate. If it subsequently calls <code>wait</code> , the calling process blocks until all its child processes have terminated and then returns -1 with <code>errno</code> set to <code>ECHILD</code> . (Recall Section 10.7.)
SA_NODEFER	•	•	•	•	•	When this signal is caught, the signal is not automatically blocked by the system while the signal-catching function executes (unless the signal is also included in <code>sa_mask</code>). Note that this type of operation corresponds to the earlier unreliable signals.
SA_ONSTACK	XSI	•	•	•	•	If an alternative stack has been declared with <code>sigaltstack(2)</code> , this signal is delivered to the process on the alternative stack.
SA_RESETHAND	•	•	•	•	•	The disposition for this signal is reset to <code>SIG_DFL</code> , and the <code>SA_SIGINFO</code> flag is cleared on entry to the signal-catching function. Note that this type of operation corresponds to the earlier unreliable signals. The disposition for the two signals <code>SIGILL</code> and <code>SIGTRAP</code> can't be reset automatically, however. Setting this flag can optionally cause <code>sigaction</code> to behave as if <code>SA_NODEFER</code> is also set.
SA_RESTART	•	•	•	•	•	System calls interrupted by this signal are automatically restarted. (Refer to Section 10.5.)
SA_SIGINFO	•	•	•	•	•	This option provides additional information to a signal handler: a pointer to a <code>siginfo</code> structure

Sono interessanti `SA_RESTART`, che risegue una systemcall se interrotta dal segnale, e `SA_RESETHAND`, che disinstalla il gestore una volta che il segnale viene chiamato.

Seguono alcuni esempi di sigaction da sigaction_example.

```
#include <unistd.h>
#include <signal.h>

static int count = 0;

static void
catch_sigint(int signo) {
    ++count;
    write(1,"CATTURATO SIGINT!\n",18);
}

int
main(int argc,char *argv[]) {
    struct sigaction sa_old;
    struct sigaction sa_new;

    sa_new.sa_handler = catch_sigint;
    sigemptyset(&sa_new.sa_mask);
    sa_new.sa_flags = 0;
    sigaction(SIGINT,&sa_new,&sa_old);

    puts("STARTED:");

    do {
        sleep(1);
    } while ( count < 3 );

    puts("ENDED.");
    return 0;
}
```

```
static void
catch_sig(int signo) {
    if ( signo == SIGINT ) {
        alarm(0); /* Cancel the timer */
        write(1,"CAUGHT SIGINT.\n",15);
    } else if ( signo == SIGALRM )
        write(1,"CAUGHT SIGALRM.\n",16);
}

int
main(int argc,char *argv[])
{
    sigset(SIGINT+SIGALRM); /* SIGINT + SIGALRM */
    struct sigaction sa_old;
    struct sigaction sa_new;

    sa_new.sa_handler = catch_sig;
    sigemptyset(&sa_new.sa_mask);
    sigadd(SIGALRM,&sa_new);
    sigadd(SIGINT,&sa_new);
    sa_new.sa_flags = 0;

    sigaction(SIGINT,&sa_new,&sa_old); /* Catch SIGINT */
    sigaction(SIGALRM,&sa_new,0); /* Catch SIGALRM */

    sigfill(SIGINT);
    sigdel(SIGINT);
    sigdel(SIGALRM);

    puts("You have 3 seconds to SIGINT:");

    alarm(3); /* Timeout in 3 seconds */
    sigsuspend(SIGINT); /* Wait for SIGINT or SIGALRM */
```

Qui è usata la **sigsuspend** che è simile alle pause ma evita le race conditions, poiché atomica: non esiste più una finestra tra l'attesa e lo sblocco, ovvero non può succedere che aspetti dopo che il segnale è già arrivato.

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

NOTA: secondo me all'inizio quando installa gestore per sigint e sigalarm, blocca anche i segnali sigint e sigalarm, in caso arrivino nel frangente di tempo che c'è fino alla chiamata `sigsuspend`, che cambierà la maschera, svegliandosi in caso di sigint e sigalarm.

Viene sostituita temporaneamente la maschera dei segnali bloccati con quella della `sigsuspend`, e si attende l'arrivo di un segnale (che quindi non sia nella maschera dei segnali bloccati ADESSO, e che invece potrebbe essere nella maschera di prima (messa installando gli handler) che ora sono liberi di arrivare, poiché `sigsuspend` ha cambiato la maschera)

Vediamo l'esempio 10.22

```
#include "apue.h"
static void sig_int(int);
int
main(void)
{
    sigset_t     newmask, oldmask, waitmask;
    pr_mask("program start: ");
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    sigemptyset(&waitmask);
    sigaddset(&waitmask, SIGUSR1);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /*
     * Block SIGINT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    /*
     * Critical region of code.
     */
    pr_mask("in critical region: ");
    /*
     * Pause, allowing all signals except SIGUSR1.
     */
    if (sigsuspend(&waitmask) != -1)
        err_sys("sigsuspend error");
    pr_mask("after return from sigsuspend: ");

    /*
     * Reset signal mask which unblocks SIGINT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    static void
    sig_int(int signo)
    {
        pr_mask("\nin sig_int: ");
    }
}
```

NOTA: pr_mask stampa la maschera dei segnali bloccati

NOTA: un segnale pendente non sparisce, appena cambia la maschera viene ricevuto

```
$ ./a.out
program start:
in critical region: SIGINT
^c                           type the interrupt character
in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
program exit:
```

Nota che se sigint inizialmente bloccato, se arriva durante la sigsuspend non è bloccato perché sigsuspend cambia la maschera dei segnali bloccati.

Quello che succede è che mandando sigusr1 non succede nulla perché bloccato dalla sigsuspend. Quando però si esce dalla sigsuspend, se è stato mandato sigusr1, questo verrà lasciato passare e ucciderà il programma. Altrimenti, dopo essere usciti dalla sigsuspend, qualsiasi segnale diverso da sigusr1 potrà passare, e sigusr1 verrà bloccato.

Vediamo ora l'esempio 10.23

```
#include "apue.h"
volatile sig_atomic_t quitflag; /* set nonzero by signal handler */

static void
sig_int(int signo) /* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1; /* set flag for main loop */
}

int
main(void)
{
    sigset(SIGINT, sig_int);
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /*
     * Block SIGQUIT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /*
     * SIGQUIT has been caught and is now blocked; do whatever.
     */
    quitflag = 0;

    /*
     * Reset signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    exit(0);
}
```

\$./a.out
interrupt
^c
interrupt
^c
interrupt
^c
interrupt
^c
now terminate with the quit character

NOTA: volatile -> A volte per ottimizzare alcune variabili che vengono utilizzate spesso (come quelle su cui si itera) vengono messe nei registri invece che nella memoria per averle più a portata di mano. Questa keyword serve ad evitare ciò.

In questo caso qualsiasi segnale sveglia la sigsuspend, e ognuno riceve la sua gestione. Se è SIGINT, semplicemente viene fatta una stampa, se è SIGQUIT, cambia la variabile quitflag e non si farà più la suspend, e il codice concluderà.

Ci sono poi le funzioni di sleep:

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);

#include <time.h>
int nanosleep(const struct timespec *reqtp, struct timespec *remtp);
```

NOTA: nella directory status che si trova in /proc/pid del processo, si trovano molte informazioni sul processo, tra cui l'utilizzo della memoria, per esempio, e i segnali bloccati, ignorati o gestiti. Per sapere i segnali gestiti, si considera il valore mostrato e ricordiamo che ogni bit corrisponde un segnale. Si parte da esadecimale, poi si scrive in binario e si vede quali bit sono a 1, e in base al numero di bit so quali segnali sono nella maschera. Per esempio il primo bit indica il segnale numero 1, 4o bit indica il segnale numero 4

Threads

I thread sono trame di esecuzione dello stesso programma, sono “programmi” che vengono eseguiti nello stesso spazio di memoria del main thread. Anche i thread ragionano nella stessa memoria virtuale “infinita”, ma ogni thread ha il suo stack.

NOTA: Tutte le funzioni che riguardano le thread iniziano con pthread, dove p sta per posix.

Le thread sono fondamentalmente delle funzioni che si lanciano in parallelo al main, e che quindi su un processore multicore permettono di migliorare le prestazioni del programma.

Nel caso ci sia un solo core, i thread possono sempre essere utilizzati ma non miglioreranno le prestazioni perché le funzioni non verrebbero comunque eseguite in parallelo, ma farebbero parte tutte dello scheduling dello stesso processore.

NOTA: lanciando il comando time a un programma che usa i thread potrebbe mostrare un real time di esecuzione maggiore del tempo reale necessario ad eseguire il programma, poiché vengono sommati i tempi dei singoli thread.

I thread sono identificati da un id, che si ottiene tramite pthread_self:

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Per vedere se due tid (thread id) sono uguali, si usa la funzione equal:

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Per creare una thread la funzione è pthread_create, e restituisce il tid:

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tidp,
                  const pthread_attr_t *restrict attr,
                  void *(*start_rtn)(void *), void *restrict arg);
```

Le thread possono avere degli attributi che vengono passati alla creazione. L'attributo può essere visto come un oggetto, di cui passeremo il puntatore. Nei nostri esempi per iniziare questo valore sarà null.

Poi si passa la funzione da eseguire, ovvero fondamentalmente il puntatore al codice. Infine, c'è arg, in cui passiamo gli argomenti della funzione. Nota che arg è un puntatore a void, sarà la funzione che dovrà sapere cosa farci.

RICORDA: restrict significa che il puntatore non può essere copiato.

Vediamo ora l'esempio 11.2

```

#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t      pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
           (unsigned long)tid, (unsigned long)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int      err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "can't create thread");
    printids("main thread:");
    sleep(1);
    exit(0);
}

```

Nel main si crea la thread, si passa l'ndirizzo della variabile globale di tipo pthread dove verrà salvato il tid, come attributi si passa null, e poi si passa la funzione senza argomenti.

Nel preciso istante in cui la thread viene creata, essa avrà vita propria e nel frattempo si procede con le istruzioni del main.

In ciascun thread si possono chiamare anche le stesse funzioni, ma bisogna fare attenzione al contenuto di queste. Se infatti scrivono sugli stessi file, oppure fanno contemporaneamente delle malloc che danno spazi di memoria che si accavallano, si fa un casotto. Ci sono infatti delle funzioni che possono essere eseguite senza porsi questi problemi, e sono le reentrant (le stesse che si possono usare nei gestori di segnale)

Una thread può terminare tre diversi modi:

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.
2. The thread can be canceled by another thread in the same process.
3. The thread can call `pthread_exit`.

```

#include <pthread.h>
void pthread_exit(void *rval_ptr);

```

The `rval_ptr` argument is a typeless pointer, similar to the single argument passed to the start routine. This pointer is available to other threads in the process by calling the `pthread_join` function.

Il valore passato `rval_ptr` è il valore di uscita della thread.

NOTA: una thread non può morire se ha un mutex.

Una thread può aspettare un'altra thread usando la `pthread_join`.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);
```

NOTA: come valore di ritorno si ha un puntatore a puntatore, poiché non siamo sicuri se la thread voglia ritornare un intero, un array, o altri tipi di strutture. Infatti, nella `exit` il valore di uscita della thread è un puntatore, quindi è necessario un puntatore a puntatore per puntare ad esso (credo).

Vediamo l'esempio 11.3

```
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int          err;
    pthread_t    tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

\$./a.out
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2

Esiste poi la `pthread_cancel`, che dà la possibilità a una thread di eliminarne un'altra:

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

NOTA: se il thread vuole ritornare solo un intero, lo qualifica (casta) come un puntatore in quanto è ciò che si aspetta la join.

Un altro metodo per passare valori di output dalle thread è valorizzare delle variabili globali.

È fondamentale quando si parla di thread l'argomento della sincronizzazione, consideriamo allora l'esempio di kerrisk in threads.

Per risolvere il problema della sincronizzazione sono stati creati i **mutex**, al purale: mutexes:

Il mutex è un “oggetto” che solo una thread per volta può avere, e che quindi è usato per sincronizzare operazioni.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Con la funzione lock si cerca di prendere il mutex. Se qualcuno vuole prendere il mutex mentre questo lo ha già un'altra thread, dovrà aspettare. La seconda lock avrà successo solo quando il primo thread avrà fatto l'unlock.

La trylock è una variante più dolce, che cerca di prendere il mutex, ma in caso non riuscisse continua a eseguire il resto del codice senza aspettare.

NOTA: se più thread vogliono lo stesso mutex e sono in “fila”, chi riuscirà a ottenerlo per primo quando si libererà è “casuale”.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

L'oggetto mutex deve essere inizializzato tramite un nome, ed eventualmente diversi attributi. Per semplicità però il mutex può essere inizializzato tramite una costante invece di fare la init: pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

Analizziamo ora il codice thread_incr_mutex.c di kerrisk.

```
#include <pthread.h>
#include "tlpi_hdr.h"

static volatile int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *                         /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j, s;
```

```

for (j = 0; j < loops; j++) {
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    loc = glob;
    loc++;
    glob = loc;

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
}

return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}

```

I mutex introducono però un nuovo problema, quello dei deadlock: ovvero una situazione di stallo incrociato.

I deadlock sono quelle situazioni in cui due thread cercano di sincronizzarsi utilizzando due mutex, e ciascuna ne ha uno e cerca di prendere l'altro prima di abbandonare il proprio: si entra quindi in un loop. La trylock aiuta questa situazione, ma è un problema che deve essere risolto dal programmatore ragionando sulle possibili situazioni che si possono creare.

Esiste poi una via di mezzo tra la lock e la trylock, che permette di non abbandonare subito il tentativo di prendere il lock, ma spetta del tempo, indicato tramite la struttura timespec.

```

#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict tspt);

```

E' possibile anche fare una distinzione tra lock in lettura e in scrittura.

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                      const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

In questo modo i "lettori" possono condividere un lock, permettendo però la scrittura solo ad un thread. Ovvero se qualcuno ha un lock in scrittura, nessuno può ottenere nessun altro lock, se invece qualcuno lo ha in lettura anche altre thread possono ottenerlo in lettura.

NOTA: che l'utilizzo dei lock sta al programmatore, ovvero hanno significato solo in base a come lui decide di usarli e interpretarli.

Come al solito è possibile fare l'inizializzazione tramite PTHREAD_RWLOCK_INITIALIZER.

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

In questo modo si richiedono i lock per la read e per la write, con i corrispondenti try:

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tspr);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tspr);
```

NOTA: le liste e le code sono un tipico caso di applicazione dei lock

Esiste poi l'idea delle **condition variables**, nonostante il nome non suggerisca assolutamente il loro utilizzo.

Normalmente un thread dovrebbe ogni volta accedere a una variabile e vedere che valore ha e in base a questo comportarsi in un determinato modo o meno. Questo comporterebbe di prendere mutex, controllare e sbloccare il mutex.

La soluzione a questo problema sono le condition variables, un meccanismo che permette di aspettare che il valore della variabile sia cambiato senza dover controllare costantemente.

Scarichiamo ora il file "A Note On Condition Variables"

Come al solito è possibile inizializzare con un'unica riga:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

La thread che vuole essere avvertita del cambiamento ("la sedia è pronta"), fa la chiamata `pthread_cond_wait()`, e verrà notificata tramite le funzioni `pthread_cond_signal()` o `pthread_cond_broadcast()`.

NOTA: che `pthread_cond_signal()` almeno una delle thread bloccate riceverà il messaggio, con `broadcast()` invece lo riceveranno tutte.

Nota che le condition variables viaggiano in coppia con un mutex, infatti queste sono protette da essi. Il funzionamento è il seguente: se una thread vuole sapere quando una variabile è cambiata, deve prima prendere il mutex corrispondente, e fare la wait per la condition variable. Fare la wait significa sbloccare il mutex, e riprenderlo automaticamente quando un thread farà la signal (e il mutex sarà libero). Quindi, la thread che vuole essere avvertita quando "la sedia è pronta", fa la chiamata `pthread_cond_wait()`, che atomicamente fa la unlock di quel mutex e l'attesa che un altro thread faccia la `signal()` o la `broadcast()`.

NOTA: le operazioni di andare a dormire e lasciare il mutex, o svegliarsi e prenderlo, sono atomiche. Non è detto però che quando una thread fa l'unlock, qualcun altro oltre a chi sta facendo la wait non possa prendere prendere il lock.

Nota che non tutto c'è sia per mac che per linux conviene quindi sempre controllare il manuale, gli standard, oppure una grep -rw funzione directoryinclude

Vediamo ora l'esempio di kerrisk, cartella thread, prod_no_condvar.c (SENZA COND. VAR.)

```
#include <pthread.h>
#include <stdbool.h>
#include "tlpi_hdr.h"

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static int avail = 0;

static void *
producer(void *arg)
{
    int cnt = atoi((char *) arg);

    for (int j = 0; j < cnt; j++) {
        sleep(1);

        /* Code to produce a unit omitted */

        int s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        avail++;           /* Let consumer know another unit is available */

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    time_t t = time(NULL);

    int totRequired = 0;          /* Total number of units that all
                                    threads will produce */
    /* Create all threads */

    for (int j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);

        pthread_t tid;
        int s = pthread_create(&tid, NULL, producer, argv[j]);
        if (s != 0)
            errExitEN(s, "pthread_create");
    }

    /* Use a polling loop to check for available units */

    int numConsumed = 0;          /* Total units so far consumed */
    bool done = false;

    for (;;) {
        int s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        while (avail > 0)           /* Consume all available units */

            /* Do something with produced unit */

            numConsumed++;
            avail--;
            printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t),
                   numConsumed);

            done = numConsumed >= totRequired;
    }

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");

    if (done)
        break;

    /* Perhaps do other work here that does not require mutex lock */

} exit(EXIT_SUCCESS); }
```

Vengono create tante thread quanti argomenti passati, e a ognuno viene passato un argomento diverso.

In questo caso tutte le thread cercano costantemente di prendere il lock per aumentare la variabile avail, per poi fare l'unlock. La main thread invece è “consumatrice”, e controlla se è stato prodotto qualcosa e diminuisce la variabile avail aumentando consumed.

E’ necessario notare che spesso c’è un lock-unlock inutile poiché la variabile avail è ancora uguale a zero. (credo sia per questo)

Si può lanciare il programma per esempio come ./prod_no_condvar 3 5 8 9

Anlizziamo ora la soluzione con la condition variable:

```
#include <time.h>
#include <pthread.h>
#include <stdbool.h>
#include "tlpi_hdr.h"

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int avail = 0;

static void *
producer(void *arg)
{
    int cnt = atoi((char *) arg);

    for (int j = 0; j < cnt; j++) {
        sleep(1);

        /* Code to produce a unit omitted */

        int s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        avail++;      /* Let consumer know another unit is available */

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");

        s = pthread_cond_signal(&cond);           /* Wake sleeping consumer */
        if (s != 0)
            errExitEN(s, "pthread_cond_signal");
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    time_t t = time(NULL);

    int totRequired = 0;          /* Total number of units that all threads
                                    will produce */

    /* Create all threads */

    for (int j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);

        pthread_t tid;
        int s = pthread_create(&tid, NULL, producer, argv[j]);
        if (s != 0)
            errExitEN(s, "pthread_create");
    }

    /* Loop to consume available units */

    int numConsumed = 0;          /* Total units so far consumed */
    bool done = false;

    for (;;) {
        int s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");
```

```

while (avail == 0) { /* Wait for something to consume */
    s = pthread_cond_wait(&cond, &mtx);
    if (s != 0)
        errExitEN(s, "pthread_cond_wait");
}

/* At this point, 'mtx' is locked... */

while (avail > 0) { /* Consume all available units */

    /* Do something with produced unit */

    numConsumed++;
    avail--;
    printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t),
           numConsumed);

    done = numConsumed >= totRequired;
}

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");

if (done)
    break;

/* Perhaps do other work here that does not require mutex lock */
}

exit(EXIT_SUCCESS);
}

```

NOTA: In questo caso essendoci un solo consumatore invece del while avail == 0 andrebbe bene anche solo una if

NOTA: I tempi di utente sono sensibilmente diminuiti perché erano dovuti dal loop che faceva continuamente il lock.

Parliamo ora di **spinlocks**.

Cosa sono? Sono alla base del meccanismo con cui sono gestiti i mutex. La differenza con i mutex sta nel tempo che ci mette un thread a rendersi conto che un lock si è liberato, e questa thread ci mette molto a risvegliarsi, in quanto l'operazione non è atomica. Il kernel fa l'operazione di controllo con meno frequenza rispetto a quanto potrebbe essere necessario, allora grazie agli spinlock è come se si ciclasse continuamente sulla variabile in modo da accorgersi immediatamente quando si libera. Ovviamente questa operazione avrà un “costo” maggiore, ma a volte è necessaria.

Da notare che un'idea interessante potrebbe essere quella di usare uno spinlock per qualche secondo, e poi in caso di “fallimento” tornare a usare il mutex normale.

NOTA: su mac potrebbero non esserci gli spinlock

Parliamo ora di **barriere**.

Spesso i thread si suddividono il lavoro di preparazione di un certo materiale, ma per andare avanti è necessario che tutti abbiano finito.

NOTA: le barriere sono opzionali in susv3, infatti mac non le implementa, e obbligatorie in susv4

Vediamo l'esempio pthread_barrier_demo.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static pthread_barrier_t barrier;
    /* Barrier waited on by all threads */

static int numBarriers;           /* Number of times the threads will
                                    pass the barrier */

static void *
threadFunc(void *arg)
{
    long threadNum = (long) arg;

    printf("Thread %ld started\n", threadNum);

    /* Seed the random number generator based on the current time
       (so that we get different seeds on each run) plus thread
       number (so that each thread gets a unique seed). */

    srand(time(NULL) + threadNum);

    /* Each thread loops, sleeping for a few seconds and then waiting
       on the barrier. The loop terminates when each thread has passed
       the barrier 'numBarriers' times. */

    for (int j = 0; j < numBarriers; j++) {

        int nsecs = random() % 5 + 1;    /* Sleep for 1 to 5 seconds */
        sleep(nsecs);

        /* Calling pthread_barrier_wait() causes each thread to block
           until the call has been made by number of threads specified
           in the pthread_barrier_init() call. */

        printf("Thread %ld about to wait on barrier %d "
              "after sleeping %d seconds\n", threadNum, j, nsecs);
        int s = pthread_barrier_wait(&barrier);

        /* After the required number of threads have called
           pthread_barrier_wait(), all of the threads unblock, and
           the barrier is reset to the state it had after the call to
           pthread_barrier_init(). In other words, the barrier can be
           once again used by the threads as a synchronization point.

           On success, pthread_barrier_wait() returns the special value
           PTHREAD_BARRIER_SERIAL_THREAD in exactly one of the waiting
           threads, and 0 in all of the other threads. This permits
           the program to ensure that some action is performed exactly
           once each time a barrier is passed. */

        if (s == 0) {
            printf("Thread %ld passed barrier %d: return value was 0\n",
                   threadNum, j);

        } else if (s == PTHREAD_BARRIER_SERIAL_THREAD) {
            printf("Thread %ld passed barrier %d: return value was "
                  "%d\n", threadNum, j, s);

            /* In the thread that gets the PTHREAD_BARRIER_SERIAL_THREAD
               return value, we briefly delay, and then print a newline
               character. This should give all of the threads a chance
               to print the message saying they have passed the barrier,
               and then provide a newline that separates those messages
               from subsequent output. (The only purpose of this step
               is to make the program output a little easier to read.) */

            usleep(100000);
        }
    }
}
```

```

        printf("\n");

    } else { /* Error */
        errExitEN(s, "pthread_barrier_wait (%ld)", threadNum);
    }
}

/* Print out thread termination message after a briefly delaying,
so that the other threads have a chance to display the return
value they received from pthread_barrier_wait(). (This simply
makes the program output a little easier to read.)*/

usleep(200000);
printf("Thread %ld terminating\n", threadNum);

return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s num-barriers num-threads\n", argv[0]);

    numBarriers = atoi(argv[1]);
    int numThreads = atoi(argv[2]);

    /* Allocate array to hold thread IDs */

    pthread_t *tid = calloc(sizeof(pthread_t), numThreads);
    if (tid == NULL)
        errExit("calloc");

    /* Initialize the barrier. The final argument specifies the
       number of threads that must call pthread_barrier_wait()
       before any thread will unblock from that call. */

    int s = pthread_barrier_init(&barrier, NULL, numThreads);
    if (s != 0)
        errExitEN(s, "pthread_barrier_init");

    /* Create 'numThreads' threads */

    for (long threadNum = 0; threadNum < numThreads; threadNum++) {
        s = pthread_create(&tid[threadNum], NULL, threadFunc,
                           (void *) threadNum);
        if (s != 0)
            errExitEN(s, "pthread_create");
    }

    /* Each thread prints a start-up message. We briefly delay,
       and then print a newline character so that an empty line
       appears after the start-up messages. */

    usleep(100000);
    printf("\n");

    /* Wait for all of the threads to terminate */

    for (int threadNum = 0; threadNum < numThreads; threadNum++) {
        s = pthread_join(tid[threadNum], NULL);
        if (s != 0)
            errExitEN(s, "pthread_join");
    }

    exit(EXIT_SUCCESS); }

```

Questo esecizio permette di avere più barriere e più thread.

Tid sarà un array dove si da l'indice del numero della thread. Per tutte le thread si creano le thread una ad una e si passa a tutte ola stessa funzione e un argomento, Questo argomento che si passa è il suo numero. Alla 4 thread si passa il numero 4.

Una volta create le thread, ogni thread parte e va per fatti suoi.

Poi c'è una join fatta a tutte quante, che comunque si potrebbe togliere perché succederà comunque quando ogni thread sarà terminata. Trall'altro le aspetta in sequenza...

Vediamo ora la thread function, prende come argomento il suo stesso numero, poi genera un generatore di numeri casuali (per evitare di avere sempre lo stesso generatore). Poi si fa un loop sul numero di barriere, che nel nostro caso sarà 1. Poi in questa esecuzione dorme un numero casuale da 1 a 5 secondi.

NOTA: la barriera quando viene creata ha il numero delle thread che deve aspettare, e verrà risvegliata quando la wait sarà fatta a quel numero di thread.

Dopo che ha dormito, dice che aspetta ad aspettare la barriera e si mette ad aspettare. Quando tutti i thread avranno fatto tutti la wait (che significherà che sono arrivati alla barriera), tutti potranno andare avanti

C'è un ultima particolarità da descrivere: può darsi che il programmatore abbia creato tante thread che fanno il loro lavoro. Può darsi che il programmatore desideri che all'uscita delle thread della barriera, solo una faccia una cosa e le altre facciano altro. Cioè tutte le thread che magari come in questo caso sono la stessa funzione, dopo la barriera si devono comportare diversamente.

Questa idea è la base del valore restituito dalla wait. In genere quando si esce dalla wait si ha come valore di ritorno 0, ma solo una thread riceve un valore particolare che è PTHREAD_BARRIER_SERIAL_THREAD, e in questo modo si può mettere una sezione condizionale che in quel caso particolare una sola thread può avere un comportamento diverso.

Se invece le thread fossero diverse non ci sarebbe bisogno di questo modo per distinguerle, semplicemente si mette il codice nella thread corrispondente.

Quando si mettono più barriere semplicemente vengono fatte in sequenza per questo si usa la stessa variabile &barrier.

NOTA: l'esempio di Stevens ha un sacco di pezzi, li ordina prima in parti e poi controlla quando finiscono, vedi barrier.c

Attributi

Analizziamo ora gli attributi che le thread possono avere.

Per assegnare un'attributo a una thread, prima si costruisce l'"oggetto" e poi viene assegnato.

NOTA: si può usare lo stesso attributo per più thread

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

Both return: 0 if OK, error number on failure
```

Alcuni attributi sono per esempio

Name	Description	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<i>detachstate</i>	detached thread attribute	•	•	•	•
<i>guardsize</i>	guard buffer size in bytes at end of thread stack	•	•	•	•
<i>stackaddr</i>	lowest address of thread stack	•	•	•	•
<i>stacksize</i>	minimum size in bytes of thread stack	•	•	•	•

Detachedstate significa che non si è interessati all'esito che avrà la thread e che non ci sarà bisogno quindi di fare una join per aspettarne i risultati.

Guardsize invece indica la terra di nessuno tra stack e heap.

Per aggiungere per esempio il primo attributo si usano le seguenti funzioni:

```
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

Both return: 0 if OK, error number on failure
```

Il valore dell'intero può essere

PTHREAD_CREATE_DETACHED oppure PTHREAD_CREATE_JOINABLE a seconda se lo vogliamo joinable o meno.

Esempio 12.4

```
#include "apue.h"
#include <pthread.h>

int
makethread(void *(*fn)(void *), void *arg)
{
    int             err;
    pthread_t       tid;
    pthread_attr_t  attr;
    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

Un altro attributo utile nell'ambito delle thread è il cancelstate.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Returns: 0 if OK, error number on failure

Questo permette o meno la possibilità che un'altra thread possa cancellare quella con questo attributo.

NOTA: esiste il concetto di cancellazione pendente, ovvero se un thread non è cancellabile e riceve una richiesta di cancellazione, questo verrà cancellato appena si renderà cancellabile.

Esiste poi il concetto di cancellation type. Ci sono due tipi di cancellazioni, DEFERRED e ASYNCHRONOUS. Nel caso deferred la cancellazione avviene quando il thread usa una funzione appartenente alla tabella 12.15, nel caso asynchronous invece avviene immediatamente.

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

accept	mq_timedsend	pthread_join	sendto
aio_suspend	msgrecv	pthread_testcancel	sigsuspend
clock_nanosleep	msgsnd	pwrite	sigtimedwait
close	msync	read	sigwait
connect	nanosleep	readv	sigwaitinfo
creat	open	recv	sleep
fcntl	openat	recvfrom	system
fdatasync	pause	recvmsg	tcdrain
fsync	poll	select	wait
lockf	pread	sem_timedwait	waitid
mq_receive	pselect	sem_wait	waitpid
mq_send	pthread_cond_timedwait	send	write
mq_timedreceive	pthread_cond_wait	sendmsg	writenv

Figure 12.14 Cancellation points defined by POSIX.1

Se c'è una richiesta di cancellazione pendente, il thread è “cancellabile”, ma non viene terminato, c'è la possibilità aggiungere un punto di cancellazione.

```
#include <pthread.h>

void pthread_testcancel(void);
```

E' infatti possibile aggiungere funzioni tra le seguenti nei cancellation point:

access	freeobj	getchar	getchar
catione	freeport	glon	getchar
catgets	frstat	iconv_close	getchar
catopen	iconv_fopen	iconv_fclose	getchar
chmod	ioctl	ioctcl	getchar
close	link	linkat	getchar
connect	lstat	lstatat	getchar
creat	lseek	lseekat	getchar
fcntl	linkat	lremovat	getchar
fdatasync	lstat	lremovat	getchar
fsync	lstatat	lremovat	getchar
lockf	lstatat	lremovat	getchar
mq_receive	lstatat	lremovat	getchar
mq_send	lstatat	lremovat	getchar
mq_timedreceive	lstatat	lremovat	getchar

Figure 12.15 Optional cancellation points defined by POSIX.

La stessa logica degli attributi vale anche per le condition variable e per i mutex.

Analizziamo brevemente gli attributi nei mutex.

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Un attributo molto importante per i mutex è il seguente:

```
#include <pthread.h>
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
                                 restrict attr,
                                 int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);
Both return: 0 if OK, error number on failure
```

Grazie a questo attributo è possibile usare i mutex tra processi differenti, tramite la shared memory. Si posiziona il mutex in una zona di memoria mappata su entrambi i processi in modo che sia la stessa. Il mutex viene memorizzato proprio nell'area condivisa, in modo che entrambi i processi usino lo stesso mutex.

Se come intero si passa il valore PTHREAD_PROCESS_SHARED, allora il mutex a cui sarà dato quell'attributo sarà condivisibile.

Come fanno più funzioni a condividere le stesse variabili? Utilizzando l'area globale.

Supponiamo di lanciare lo stesso thread più volte, e che questo utilizzi delle variabili globali in modo che le funzioni da lui utilizzate le possano conoscere. Il problema è che tutti i thread andrebbero a modificare e leggere le stesse variabili.

Esiste un modo di creare delle variabili globali, che però sono diverse in base alla thread che le utilizza: ogni thread gemella che utilizza la variabile globale ha la propria con il proprio valore: si creano quindi delle variabili globali “per thread”.

C'è "a note on thread local storage" che spiega il problema, e poi scarica thread examples

Vedi ATEST22TLS

Sono utilizzate due variabili TLS_data1 e TLS_data2 che hanno il prefisso __thread, che significa che quelle variabili siano globali ma per thread. In questo modo ogni thread ha una copia personale di questa variabile globale.

Quello che succede è che vengono creati 8 thread, e a ognuno viene passata una struttura che viene riempita con valori diversi. Si nota però che in questo modo ogni thread vedrà un valore diverso della stessa variabile.

NOTA: salta la parte di stevens di questo argomento

```

#define _MULTI_THREADED
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void foo(void); /* Functions that use the TLS data */
void bar(void);

#define checkResults(string, val) { \
    if (val) { \
        printf("Failed with %d at %s", val, string); \
        exit(1); \
    } \
}

__thread int TLS_data1;
__thread int TLS_data2;

#define NUMTHREADS 8

typedef struct {
    int data1;
    int data2;
} threadparm_t;

void *theThread(void *parm)
{
    int rc;
    threadparm_t *gData;

    printf("Thread %.16lx: Entered\n", pthread_self());

    gData = (threadparm_t *)parm;

    TLS_data1 = gData->data1;
    TLS_data2 = gData->data2;

    foo();
    return NULL;
}

void foo() {
    printf("Thread %.16lx: foo(), TLS data=%d %d\n",
           pthread_self(), TLS_data1, TLS_data2);
    bar();
}

void bar() {
    printf("Thread %.16lx: bar(), TLS data=%d %d\n",
           pthread_self(), TLS_data1, TLS_data2);
    return;
}

int main(int argc, char **argv)
{
    pthread_t thread[NUMTHREADS];
    int rc=0;
    int i;
    threadparm_t gData[NUMTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create/start threads\n");
    for (i=0; i < NUMTHREADS; i++) {
        /* Create per-thread TLS data and pass it to the thread */
        gData[i].data1 = i;
        gData[i].data2 = (i+1)*2;
        rc = pthread_create(&thread[i], NULL, theThread, &gData[i]);
        checkResults("pthread_create()\n", rc);
    }

    printf("Wait for the threads to complete, and release their resources\n");
    for (i=0; i < NUMTHREADS; i++) {
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    printf("Main completed\n");
    return 0;
}

```

Thread e Segnali

Può essere utile consultare il file “Main points on Signals and Threads”, da telegram.

I gestori di segnali innanzitutto sono condivisi tra tutte le thread. Ogni thread però può avere la propria signal mask che determina i segnali bloccati.

E’ comunque possibile consegnare un segnale a una thread in 3 modi:

- 1) In maniera asincrona, ovvero il segnale arriva a una thread che non lo ha bloccato.
- 2) In maniera sincrona, ovvero se una thread fa divisione per 0, o cerca di accedere a una parte di memoria non autorizzata, solo quella thread riceverà il segnale.
- 3) In maniera diretta, esiste una chiamata anloga alla kill (pthread_kill)

```
#include <signal.h>
int pthread_kill(pthread_t thread, int signo);
```

La maschera dei segnali può essere cambiata tramite pthread_sigmask che funziona come la sigprocmask:

```
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

NOTA: cosa dovrebbe usare la mainthead? Sigprocmask o pthread_sigmask? Bella domanda

NOTA: una nuova thread eredita una copia della signalmask del creatore

NOTA: Una buona idea è fare in modo che tutti i segnali siano bloccati, e una sola thread riceva tutti i segnali e li gestisca.

NOTA: se il segnale è letale, anche se lo riceve solo una thread, fa terminare tutto il processo

NOTA: se arriva un segnale mentre si cerca di fare un lock, la chiamata è riavviata.

```
#include <signal.h>
int sigwait(const sigset_t *restrict set, int *restrict signop);
```

C’è un modo di mettersi in attesa di un segnale, tramite la sigwait (corrispondente alla sigsuspend, ma cambia la maschera del solo thread). Come abbiamo già visto precedentemente, si può utilizzare una maschera personale, e si può anche gestire il segnale in maniera asincrona: la thread sarà a conoscenza dell’arrivo del segnale, ma deciderà lei cosa fare e quando farlo, senza fretta.

NOTA: Si può usare la funzione sigpending per avere l’unione di tutti i segnali pendenti.

La funzione **sigwait** prende la maschera e un puntatore a un intero usato come output.

NOTA: la sigwait vuole un segnale nella maschera, a differenza da sigsuspend che ne vuole

uno che non è nella maschera per sbloccarsi

NOTA: se c'era un segnale pendente, la sigwait lo considera e si sblocca.

NOTA: se si aspetta un segnale ignorato si aspetterà per sempre.

Analizziamo sigwait4 da Esperimento sigwait:

```
sigset_t mask;
int main(void)
{
    int                 signo, err;
    sigset_t oldmask;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGTERM);

    err = sigwait(&mask, &signo);
    if (err != 0) {
        printf("Fallita sigwait\n");
        exit(0);
    }
    switch (signo) {
    case SIGINT:
        printf("Received %d\n", signo);
        break;

    case SIGTERM:
        printf("Received %d\n", signo);
        break;

    default:
        printf("unexpected signal %d\n", signo);
        exit(1);
    }
    exit(0);
}
```

Se si manda un segnale nella maschera che lo sblocca dalla attesa bene. Se si manda prima un segnale come SIGQUIT, prima non succede nulla, ma appena si manda un segnale che lo fa uscire dalla wait, la prima cosa che succede è il risultato della SIGQUIT.

In sigwait5 invece, prima si bloccano i segnali che poi vogliamo aspettare, e poi ci si mette ad aspettare. In questo modo anche se i segnali arrivano prima di quando li aspettiamo, poi li riceviamo al “momento giusto” quando si fa la wait.

In sigwait6 invece inizialmente si bloccano tutti i segnali.

Ci sono però delle differenze su come si comportano linux e macos in presenza di questi codici:

			SIGINT	SIGQUIT	SIGQUIT & SIGINT
test_sigwait4	Linux	Nessun blocco	Errore 130	Terminato	
test_sigwait5	Linux	Solo blocco mask sigwait	Received 2	Terminato	
test_sigwait6	Linux	Tutti bloccati	Received 2	Bloccato	Received 2
test_sigwait4	macOS	Nessun blocco	Received 2	Bloccato	Terminato
test_sigwait5	macOS	Solo blocco mask sigwait	Received 2	Bloccato	Terminato
test_sigwait6	macOS	Tutti bloccati	Received 2	Bloccato	Received 2

L'anomalia è il comportamento quando si invia sigquit: su macos questo è sempre bloccato se stiamo facendo una wait, invece su linux si risponde sempre ad esso.

Potrebbe essere un bug di macos, ovvero in realtà anche durante la wait il processo dovrebbe reagire alla ricezione dei segnali.

Vediamo ora l'esempio di stevens che è 12.16

```
#include "apue.h"
#include <pthread.h>

int      quitflag;    /* set nonzero by thread */
sigset_t mask;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waitloc = PTHREAD_COND_INITIALIZER;

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            err_exit(err, "sigwait failed");
        switch (signo) {
        case SIGINT:
            printf("\ninterrupt\n");
            break;

        case SIGQUIT:
            pthread_mutex_lock(&lock);
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&waitloc);
            return(0);

        default:
            printf("unexpected signal %d\n", signo);
            exit(1);
        }
    }
}

int
main(void)
{
    int      err;
    sigset_t oldmask;
    pthread_t tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "SIG_BLOCK error");

    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&waitloc, &lock);
    pthread_mutex_unlock(&lock);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;

    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    exit(0);
}
```

NOTA: sigwait vale solo per il singolo thread, e ASPETTA UNO DEI SEGNALI NELLA MASCHERA, a differenza di sigsuspend che ne aspetta uno che non ne fa parte (perche blocca tutti gli altri)

Il main blocca i segnali sigint e sigquit, mentre invece il thread li “aspetta”. Il thread chiamato dalla main eredita la maschera che blocca i segnali, il che è buono perché assicura che cattureremo qualsiasi segnale anche nel frattempo sigwait ancora non è stata chiamata.

Se si manda sigint non succede niente, invece se si manda sigquit viene settato a 1 il flag e la main thread smette di aspettare.

NOTA: in questo caso il loop while quitflag == 0 è inutile, era sufficiente aspettare la chiamata della condition variable.

Cosa succede nel caso in cui un thread faccia una fork? Viene creato un altro processo con lo spazio di memoria della thread. Ha senso? Solo se si vuole fare una exec.

Thread and I/O è semplice grazie alle funzioni pread e pwrite, poiché i thread condividono i file descriptor. Bisognerebbe infatti fare attenzione in caso thread diversi spostino la posizione del cursore su un file e si pestino i piedi, ma grazie alla atomicità di queste funzioni il problema non si pone. Infatti la pread e pwrite fanno automaticamente prima una lseek e poi la read o la write.

Daemon

La parola daemon arriva dai greci, che per demoni intendevano dei “genietti” o delle entità soprannaturali con funzione ispiratrice.

I daemon sono processi eseguiti in background e non hanno accesso a un terminale. In genere i daemon sono dei server.

Si possono riconoscere i daemon dal fatto che non abbiano un terminale di controllo associato, o dal nome che finisce per d. Inoltre, i daemon hanno pid bassi poiché in genere vengono lanciati all'avvio della macchina.

Su linux, i daemon sono tra parentesi quadre, il chè significa che non sono processi generati con il normale meccanismo di generazione. I normali processi sono tutti discendenti del processo 1, invece nel caso dei daemon sono generati direttamente dal kernel senza passare dalla fork-exec.

NOTA: molte operazioni che fa il kernel hanno delle chiamate alternative, per esempio invece di printf il kernel usa printk (senza usare infatti la stdio library).

Ci sono delle regole per fare diventare un processo un deamon:

- 1) Viene settata la umask a 0 in modo da non ereditare le restrizioni di chi ha lanciato il processo.
- 2) Si chiama la fork e il parent fa la exit in modo da evitare processi zombie.
- 3) Si usa setsid per creare una nuova sessione, di cui il daemon diventa leader. È poi disassociato il terminale di controllo.
- 4) Sostituire la current working directory con la root. Si utilizza la chiamata la chdir.
- 5) Chiudere i file descriptor non utilizzati devono essere chiusi. Ricorda che a meno che non c'è il flag vengono ereditati. Tramite getrlimit si potrebbe vedere quanti sono i file descriptor e chiuderli tutti.
- 6) Reindirizzare i fd 0,1,2 a /dev/null in modo che quando si voglia leggere o scrivere a 0 e da 1 non ci sia errore

È stata preparata la funzione “demonize” che rende un comando un “daemon” effettuando automaticamente tutte queste operazioni. Dopo averla chiamata si fa la exec.

Il daemon per comunicare con il mondo utilizza dei file di log. All'interno di questi file scrive tutto ciò che vuole fare sapere al mondo. In genere per questioni di sicurezza i log sono mandati su un'altra macchina (o più).

Esiste infatti un daemon chiamato syslogd che sostanzialmente fa due cose: scrive i file locali, e manda ad un altro syslogd le informazioni.

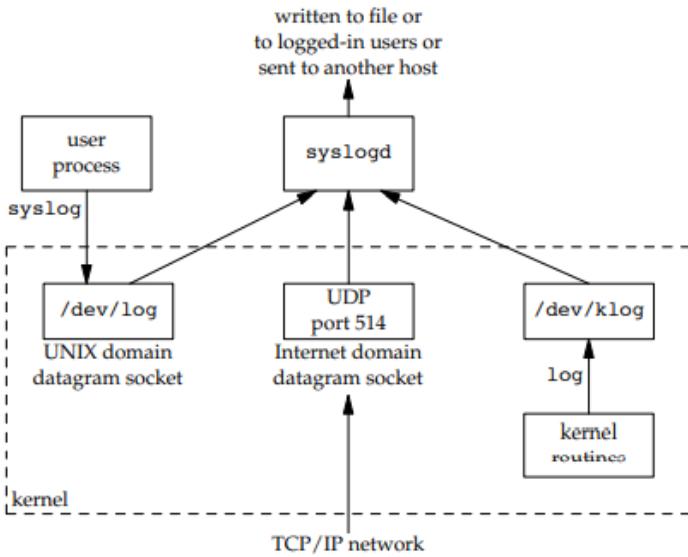


Figure 13.2 The BSD syslog facility

Si nota infatti la freccia tcp/ip entrante e uscente al sistema.

C'è la chiamata openlog che mette in piedi il meccanismo e syslog fa la scrittura.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
int setlogmask(int maskpri);
```

Su moodle c'è un file zip “syslog” con una descrizione del funzionamento di syslog e un file di configurazione tipo con spiegazione e un codice. Il programma di esempio semplicemente manda i messaggi, che poi leggiamo con tail -f sul file in questione.

NOTA: quando si lancia un daemon nel sistema, si vuole essere sicuri non ce ne siano altri uguali. Si usa allora il trucco del lockfile, oppure di fa una open con OEXCL in modo da controllare l'eventuale presenza di un file.

Quando si usano i daemon ci sono delle convenzioni, per esempio sul significato del segnale sighup che viene interpretato come “rileggere i file di configurazione”. In genere i file di configurazione sono nella directory /etc.

Sebbene i daemon possano essere lanciati dalla command line di bash, spesso sono invece avviati dal sistema unix tramite script in sh che si trovano nei file di inizializzazione di unix. Questi file su linux si trovano in /etc/rc* dove * indica un indice che li distingue.

Nella figura 13.7 si mostra l'esempio di utilizzo di sighup

Advanced I/O

Esistono delle systemcall che si possono chiamare in maniera che non “blocchino” il sistema, nonblocking I/O. Per esempio, se si fa la open con il flag O_NONBLOCK: se c’è da leggere o scrivere, o lo fa subito o non rimane bloccata.

Vediamo esempio 14.1

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>

char    buf[500000];

int
main(void)
{
    int      ntwrite, nwrite;
    char    *ptr;

    ntwrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntwrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK); /* set nonblocking */

    ptr = buf;
    while (ntwrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntwrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntwrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */
    exit(0);
}
```

Figure 14.1 Large nonblocking write

Dopo essere stato messo il flag O_NONBLOCK, in un loop si cerca di scrivere quello che lo stdin legge. Se non ci fosse il flag, si aspetterebbe finchè non si scrive qualcosa.

Nota che la if serve per aggiornare il cursore.

In questo caso, se si scrive il file grosso su un file tutto una volta, non ci sono problemi. Il problema sorge quando si scrive sul termiale.

```
$ ls -l /etc/services          print file size
-rw-r--r-- 1 root    677959 Jun 23 2009 /etc/services
$ ./a.out < /etc/services > temp.file      try a regular file first
read 500000 bytes
nwrite = 500000, errno = 0           a single write
$ ls -l temp.file                verify size of output file
-rw-rw-r-- 1 sar     500000 Apr  1 13:03 temp.file
```

Per creare il nostro file grosso:

```
dd if=/dev/urandom bs=1 count=1000000 | tr -dc [:alnum:][:punct:]
```

il -d cancellerebbe tutti gli alfanumerici ma il -c fa il complementare, quindi è cancellato tutto ciò che non è alfanumerico.

NOTA: tr si aspetta utf-8, allora si fa LANG=C prima di tr

Se invece scriviamo sul terminale:

```
$ ./a.out < /etc/services 2>stderr.out          output to terminal  
$ cat stderr.out                                lots of output to terminal ...  
read 500000 bytes  
nwrite = 999, errno = 0  
nwrite = -1, errno = 35  
nwrite = 1001, errno = 0  
nwrite = -1, errno = 35  
nwrite = 1002, errno = 0  
nwrite = 1004, errno = 0  
nwrite = 1003, errno = 0  
nwrite = 1003, errno = 0  
nwrite = 1005, errno = 0  
nwrite = -1, errno = 35                           61 of these errors  
:  
nwrite = 1006, errno = 0  
nwrite = 1004, errno = 0  
nwrite = 1005, errno = 0  
nwrite = 1006, errno = 0  
nwrite = -1, errno = 35                           108 of these errors  
:  
nwrite = 1006, errno = 0  
nwrite = 1005, errno = 0  
nwrite = 1005, errno = 0  
nwrite = -1, errno = 35                           681 of these errors  
:  
and so on ...  
nwrite = 347, errno = 0
```

Il programma cerca di scrivere sempre tutto quello che deve (500000), nel primo caso riesce a scrivere 999, poi non riesce più perché il terminale si riempie. Cerca infatti costantemente di scrivere ma non riesce e ritorna errore 35. Il problema è che non “aspetta si svuoti il terminale”, e cerca di scrivere costantemente, ma non ce la fa perché lo trova pieno. A un certo punto però ce la farà, e così’ via.

Record locking

Come lo stevens fa notare, parlare di “record locking” è sballato, perché non ci sono record nei file unix: i file non sono strutturati.

Cosa si intende per record? Porzioni di file che noi definiamo tali.

A che servono? Sono un meccanismo di sincronizzazione tra processi. Il file è infatti un punto di incontro tra più processi per scambiarsi dei messaggi in maniera atomica.

Per esempio, dato un file, dico che su quel file prendo un lock dal byte 10 al byte 20. Se un altro processo cerca di prendere lo stesso lock non lo riuscirà a prendere finché non lo lascio io. Il lock di questo tipo è chiamato advisory (consiglio) perché nulla impedisce al secondo processo di fare quello che vuole comunque... ovvero è il programmatore che decide le regole. Se il secondo processo volesse, potrebbe fregarsene e scrivere lo stesso. È quindi lo stesso funzionamento dei lock. Spesso l'obiettivo, infatti, non è proteggere il file, ma magari una sezione di codice.

Per effettuare il lock si usa la funzione a coltellino svizzero fcntl:

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );
```

Returns: depends on cmd if OK (see following), -1 on error

For record locking, *cmd* is F_GETLK, F_SETLK, or F_SETLKW. The third argument (which we'll call *flockptr*) is a pointer to an *flock* structure.

```
struct flock {
    short l_type;    /* F_RDLCK, F_WRLCK, or F_UNLCK */
    short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
    off_t l_start;   /* offset in bytes, relative to l_whence */
    off_t l_len;     /* length, in bytes; 0 means lock to EOF */
    pid_t l_pid;     /* returned with F_GETLK */
};
```

Generalmente, il F_SETLK o si ottiene oppure no, non è bloccante. Per utilizzare la versione bloccante c'è F_SETLKW. Il comando per sbloccare è “F_UNLCK”.

I lock possono essere di tipo diverso, come si può vedere dalla struttura: più processi possono avere il lock di lettura, ma solo uno può avere quello di scrittura. Se qualcuno ha quello di scrittura nessuno può avere quello di lettura, così come se qualcuno ha quello di lettura non si può ottenere quello di scrittura.

NOTA: se si fa getlock il parametro l_pid diventa un parametro di uscita ed è il pid del processo che ha il lock

NOTA: E' possibile fare l'unlock anche solo di una parte del lock che avevamo.

Apue propone la funzione lock_reg che permette un passaggio di parametri più intuitivo:

```
int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;
    lock.l_type = type;      /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;   /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;        /* #bytes (0 means to EOF) */

    return(fcntl(fd, cmd, &lock));
}
```

e delle function like macro che la utilizzano:

```
#define read_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

Potresti quindi usare le function like macro che usano la funzione lock_reg per evitare di mettere il nome del lock...

Prendiamo ora da moodlis “record locking examples”, in particolare “lock2”.

Si lancia questo programma 2 volte, e fondamentalmente rappresenta la costruzione di un record locking casalingo.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

const char *lock_file = "/tmp/LCK.test2";

int main() {
    int file_desc;
    int tries = 10;

    while (tries--) {
        file_desc = open(lock_file, O_RDWR | O_CREAT | O_EXCL, 0444);
        if (file_desc == -1) {
            printf("%d - Lock already present\n", getpid());
            sleep(3);
        } else {
            /* critical region */
            printf("%d - I have exclusive access\n", getpid());
            sleep(1);
            (void)close(file_desc);
            (void)unlink(lock_file);
            /* non-critical region */
            sleep(2);
        }
    } /* while */
    exit(EXIT_SUCCESS);
}
```

If two processes look into the directory and both see that a certain file is not present, so the resource is available, and now both try to create this file, how to ensure that only one of both will ever succeed? And that's where O_EXCL comes into play. If they both try to create that files with O_EXCL set, this operation will only succeed for one of them and that's the process which now owns the resource lock.

If O_CREAT and O_EXCL are set, *open()* shall fail if the file exists.

Tramite i codici lock3.c e lock4.c invece c'è un vero e proprio meccanismo di record locking.

In lock3.c si crea un file di 100 byte, e vengono presi due lock. In lock4.c invece si cerca di looppare su tutto il file creato da lock3.c cercando di prendere tutti i lock possibili di lunghezza 5.

Per testare poi lock5 bisogna lanciare anche qui lock3.

In questo caso il lock viene fatto con una wait.

NOTA: anche qui si propone il problema dei deadlock. Il kernel comunque in generale fa del suo meglio per rilevare questi problemi e risolverli.

A proposito dei deadlock vediamo l' esempio 14.7

```
#include "apue.h"
#include <fcntl.h>

static void
lockabyte(const char *name, int fd, off_t offset)

{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);
    printf("%s: got the lock, byte %lld\n", name, (long long)offset);
}

int
main(void)
{
    int      fd;
    pid_t   pid;

    /*
     * Create a file and write two bytes to it.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");

    TELL_WAIT();
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {           /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);
    } else {                       /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
$ ./a.out
parent: got the lock, byte 1
child: got the lock, byte 0
parent: writew_lock error: Resource deadlock avoided
child: got the lock, byte 1
```

NOTA: il meccanismo di tell e wait serve a sincronizzarsi per causare il deadlock, altrimenti parent o child prenderebbe entrambi i lock e basta.

La faccenda **I/O multiplexing** è importante ma noi ne faremo solo un cenno.

Quando un processo apre un file descriptor, poi spesso rimane appeso per leggerne il contenuto con una read. Se si vuole evitare di rimanere appesi si potrebbe usare il flag O_NONBLOCK per rendere la chiamata non bloccante.

Nel momento in cui però ci sono molti file aperti, e di ogni tipo, da cui vogliamo leggere, la situazione diventa dispendiosa.

Si creano allora delle rastrelliere di file descriptor (come quelle dei segnali) fd_set in cui si mette un bit a 1 o 0 in base a se si sta aspettando di leggere qualcosa o no. Si viene quindi allora avvisati quando arriva qualcosa da leggere al file.

La comodità è quindi che con una sola istruzione si è in ascolto su tutti i fd.

Memory-Mapped I/O

E' possibile caricare o "mappare" dei file che sono sulla memoria di massa nella memoria del processo.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flag, int fd, off_t off);
```

Dove fd è il file descriptor del file, offset è l'offset dall'inizio del file da mappare, l'indirizzo serve a poter decidere dove mappare esattamente il file nella memoria virtuale (chiedendo al kernel di farlo). Quest'ultimo campo spesso è lasciato a NULL e sarà la funzione a ritornare un puntatore che indica dove è stato mappato il file. Il campo prot indica il tipo di protezione che deve avere la memoria, e può essere uno tra questi:

prot	Description
PROT_READ	Region can be read.
PROT_WRITE	Region can be written.
PROT_EXEC	Region can be executed.
PROT_NONE	Region cannot be accessed.

Figure 14.25 Protection of memory-mapped region

NOTA: ovviamente exec sarà il tipo di protezione più pericoloso, e si può vedere quando con strace si vede il caricamento delle librerie in memoria

I flag invece sono per esempio:

The *flag* argument affects various attributes of the mapped region.

MAP_FIXED The return value must equal *addr*. Use of this flag is discouraged, as it hinders portability. If this flag is not specified and if *addr* is nonzero, then the kernel uses *addr* as a hint of where to place the mapped region, but there is no guarantee that the requested address will be used. Maximum portability is obtained by specifying *addr* as 0.

Support for the **MAP_FIXED** flag is optional on POSIX-conforming systems, but required on XSI-conforming systems.

MAP_SHARED This flag describes the disposition of store operations into the mapped region by this process. This flag specifies that store operations modify the mapped file—that is, a store operation is equivalent to a `write` to the file. Either this flag or the next (**MAP_PRIVATE**), but not both, must be specified.

MAP_PRIVATE This flag says that store operations into the mapped region cause a private copy of the mapped file to be created. All successive references to the mapped region then reference the copy. (One use of this flag is for a debugger that maps the text portion of a program file but allows the user to modify the instructions. Any modifications affect the copy, not the original program file.)

In due parole, **MAP_FIXED** significa che vogliamo per forza l'indirizzo *addr* richiesto, **MAP_SHARED** significa che i cambiamenti fatti nello spazio di memoria si devono riflettere nel file, invece **MAP_PRIVATE** significa che le operazioni di modifica si applicano solo alla copia in memoria del file.

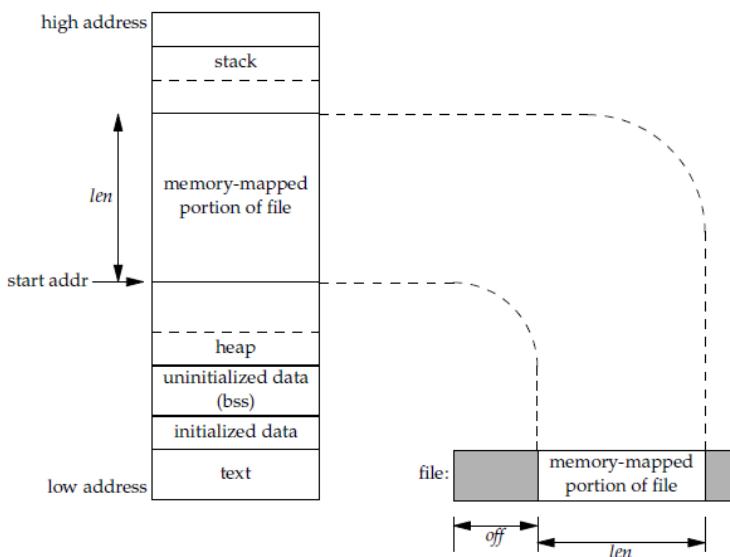


Figure 14.26 Example of a memory-mapped file

NOTA: il mapping funziona in entrambi i versi, ovvero se cambi qualcosa in memoria e c'è il flag shared si riflette nel file, cosi' come se c'è un cambiamento nel file si riflette anche nella memoria.

Vediamo ora l'esempio 14.27

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define COPYINCR (1024*1024*1024) /* 1 GB */

int
main(int argc, char *argv[])
{
    int          fdin, fdout;
    void        *src, *dst;
    size_t      copysz;
    struct stat sbuf;
    off_t       fsz = 0;

    if (argc != 3)
        err_quit("usage: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
        FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[2]);

    if (fstat(fdin, &sbuf) < 0)           /* need size of input file */
        err_sys("fstat error");

    if (ftruncate(fdout, sbuf.st_size) < 0) /* set output file size */
        err_sys("ftruncate error");

    while (fsz < sbuf.st_size) {
        if ((sbuf.st_size - fsz) > COPYINCR)
            copysz = COPYINCR;
        else
            copysz = sbuf.st_size - fsz;

        if ((src = mmap(0, copysz, PROT_READ, MAP_SHARED,
            fdin, fsz)) == MAP_FAILED)
            err_sys("mmap error for input");
        if ((dst = mmap(0, copysz, PROT_READ | PROT_WRITE,
            MAP_SHARED, fdout, fsz)) == MAP_FAILED)
            err_sys("mmap error for output");

        memcpy(dst, src, copysz); /* does the file copy */
        munmap(src, copysz);
        munmap(dst, copysz);
        fsz += copysz;
    }
    exit(0);
}
```

In questo codice si fa la copia di un file in input in quello di output tramite la memoria.

NOTA: copia massimo un giga per volta

Qui è utilizzata un ulteriore funzione, memcpy che copia la memoria di un indirizzo ad un altro indirizzo, e munmap che invece ripulisce la memoria.

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

Vediamo ora “Purposes of flags in mmap”

Visibility of modifications	Mapping type	
	File	Anonymous
Private	Initializing memory from contents of file	Memory allocation
Shared	Memory-mapped I/O; sharing memory between processes (IPC)	Sharing memory between processes (IPC)

Utilizzando il flag Anonymous (o mapanon) in realtà significa che non c'è nessun file. In questo caso al file descriptor si mette -1 per evitare messaggi di errore.

NOTA: se è privato e anonimo si ha l'equivalente di una malloc

Una situazione interessante è quella di shared-anon, che creerà un metodo di condivisione di memoria tra parent e child. Si può infatti creare questo spazio di memoria e poi fare la fork, in questo modo parent e child avranno un vero e proprio spazio di memoria condiviso.

Ora abbiamo un nuovo modo di coordinare processi oltre al TELL_PARENT e TELL_CHILD che erano possibili grazie all'utilizzo di segnali. Da notare che mentre prima era necessario conoscere il pid dei processi, poi abbiamo imparato a farlo con il record locking per il quale è necessario sapere il nome del file.

Da notare che se usi MAP_ANON solo i child prima della exec possono vedere la stessa memoria, altrimenti devi usare il file per poter fare comunicare processi diversi.

Analizziamo ora l'esempio di utilizzo anonimo della memoria mappashared.c

un processo parte e prima di fare una fork fa un mapping con MAP_SHARED, e MAP_ANON. MAP_ANON serve a non usare un file ma semplicemente condividere la memoria, senza MAPSHARED non c'è condivisione, nel caso di un file semplicemente si caricherebbe questo in memoria senza “condividerlo”, tutti i cambiamenti sarebbero in locale.

Tra mac a linux c'è confusione su questo MAP_ANON o MAP_ANONYMOUS

Il file descriptor è -1 nella mmap perchè non c'è fd e la condivisione è 1000byte.

Dopo il mapping si fa una fork, e se sei nel child leggi la locazione di memoria, e poi la legge il parent

insomma vogliamo capire se lo spazio di memoria è esattamente lo stesso

insomma parent cambia valore e il child lo vede cambiato

qui c'è quindi una SHARED MEMORY senza dover passare a un file

PARENT E CHILD POSSONO DIALOGARE TRAMITE UN AREA DI MEMORIA NONOSTANTE SIANO INDIPENDENTI

Analizziamo un esempio con i mutex che utilizzano l'attributo pshared che permette la condivisibilità tra processi.

Scarichiamo da moodle “Example code for pthread_mutexattr_setpshared”

```
int main(int argc, const char* argv[])
{
void* mmap_ptr = mmap (NULL, sizeof (pthread_mutex_t),
PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1, 0)
if (mmap_ptr == MAP_FAILED) {
    perror ("mmap failed");
    return -1;
}

fprintf (stderr, "mmaped at: %p\n", mmap_ptr);

pthread_mutex_t* mutp = (pthread_mutex_t*)mmap_ptr;

// initialize the attribute
pthread_mutexattr_t attr;
pthread_mutexattr_init (&attr);
pthread_mutexattr_setpshared (&attr, PTHREAD_PROCESS_SHARED); // this is

// initialize the mutex
pthread_mutex_init (mutp, &attr);
pthread_mutexattr_destroy (&attr);

// acquire the lock before fork
pthread_mutex_lock (mutp);
fprintf (stderr, "main: took the lock...\n");

pid_t chld = fork ();
if (chld != 0) { // parent
    fprintf (stderr, "main: going to sleep...\n");
    sleep (10);
    fprintf (stderr, "main: unlocking.\n");
    pthread_mutex_unlock (mutp);
    fprintf (stderr, "main: released the lock.\n");
} else { // child
    fprintf (stderr, "child: going to acquire the lock ... \n");
    pthread_mutex_lock (mutp);
    fprintf (stderr, "child: acquired the lock.\n");
    sleep (10);
    pthread_mutex_unlock (mutp);
    fprintf (stderr, "child: released the lock.\n");
}
return 0;
}
```

C'è anche una piccola variante, che semplicemente effettua una copia della memoria di un mutex del padre nello spazio di memoria condiviso. Quindi inizialmente il padre avrà due mutex, uno nella memoria condivisa del figlio.

Pipe

```
#include <unistd.h>
int pipe(int fd[2]);
```

La chiamata pipe prende come argomento un array di due file descriptor che riempirà lei apreendo due file. Questi fd sono l'ingresso e l'uscita di un tubo all'interno del kernel. Se si scrive al file descriptor 1 il messaggio arriva al file descriptor 0.

Ricorda che quando si fa una fork i file descriptor vengono ereditati.

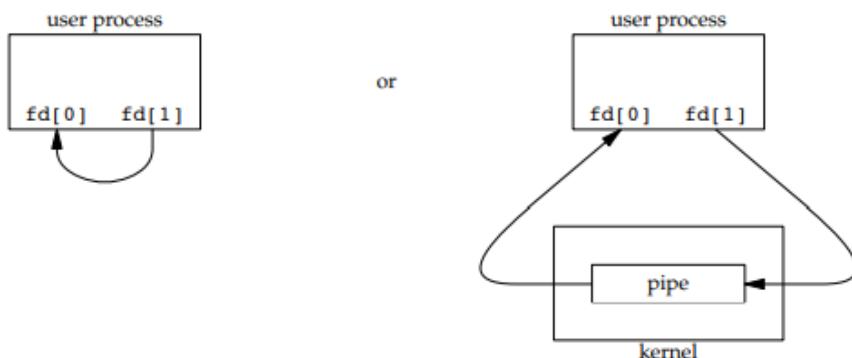


Figure 15.2 Two ways to view a half-duplex pipe

La situazione è interessante perché quando il parent fa una fork, non vengono duplicate le strutture a cui puntano i file descriptor. Dopo la fork, la pipe rimane quindi una.

Quindi ora parent e child possono comunicare tramite questa pipe.

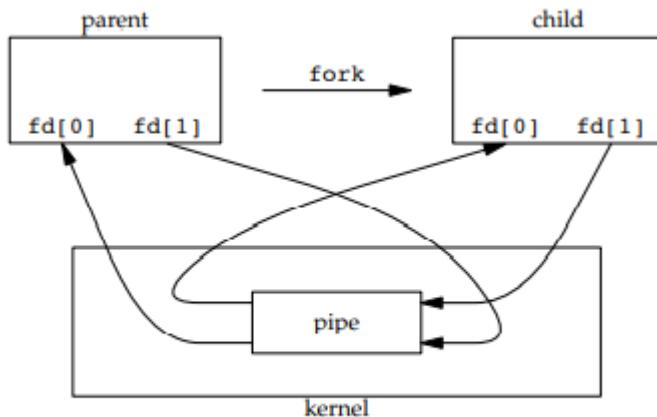


Figure 15.3 Half-duplex pipe after a fork

Al momento la comunicazione è full duplex, ma in genere si usa un single duplex chiudendo al parent la lettura e al child la scrittura. Se si vuole una comunicazione full duplex in genere conviene avere due canali single duplex.

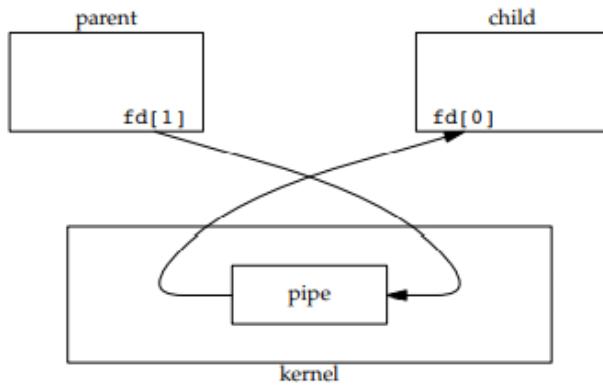


Figure 15.4 Pipe from parent to child

La pipe non è altro che una FIFO, la quale viene anche chiamata named pipe. Questa è sempre una struttura nel kernel, ma ora si accede attraverso un nome nel filesystem. Anche con le fifo, quindi, se un processo scrive da una parte, un altro può leggere da un'altra... il problema è che questi processi devono conoscere il nome della fifo.

NOTA: anche dopo la exec se il close on exec bit non è settato, il nuovo processo legge e scrive dagli stessi file descriptor sulla stessa pipe

I programmi in genere sanno solo di scrivere a 1 e leggere da 0. È il processo che farà la fork e poi la exec in questi programmi, che devono fare le magie.

Al momento però i fd della pipe probabilmente saranno 3 e 4 perché saranno i primi liberi. Entrano allora in gioco le chiamate dup per scambiare i fd.

Quando un capo della pipe è chiuso, si applicano le seguenti regole:

- 1) se leggiamo da una pipe il cui capo di scrittura è stato chiuso, la read ritorna 0 e indica un eof dopo che tutti i dati sono stati letti. Supponiamo si faccia una write di 100 byte, e poi si faccia una close del fd per scrittura. La read intanto legge questi 100 byte e poi darà eof alla read successiva. Se il capo di scrittura non fosse chiuso, la read rimarrebbe appesa.
- 2) se invece il capo di lettura è stato chiuso e noi scriviamo, allora c'è un problema perché si scrive su una pipe su cui non c'è un lettore, e viene generato un segnale SIGPIPE. La write riceve errore -1 e errno diventa EPIPE.

Vediamo poi il programma base 15.5

```
#include "apue.h"

int
main(void)
{
    int      n;
    int      fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) { /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else { /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Segue una tabella che esplicita cosa succede in alcune situazioni limite.

Table 44-2: Semantics of reading n bytes from a pipe or FIFO containing p bytes

O_NONBLOCK enabled?	Data bytes available in pipe or FIFO (p)			
	$p = 0$, write end open	$p = 0$, write end closed	$p < n$	$p \geq n$
No	block	return 0 (EOF)	read p bytes	read n bytes
Yes	fail (EAGAIN)	return 0 (EOF)	read p bytes	read n bytes

The impact of the O_NONBLOCK flag when writing to a pipe or FIFO is made complex by interactions with the PIPE_BUF limit. The *write()* behavior is summarized in Table 44-3.

From “The Linux Programming Interface” p. 918

Table 44-3: Semantics of writing n bytes to a pipe or FIFO

O_NONBLOCK enabled?	Read end open		Read end closed
	$n \leq PIPE_BUF$	$n > PIPE_BUF$	
No	Atomically write n bytes; may block until sufficient data is read for <i>write()</i> to be performed	Write n bytes; may block until sufficient data read for <i>write()</i> to complete; data may be interleaved with writes by other processes	
Yes	If sufficient space is available to immediately write n bytes, then <i>write()</i> succeeds atomically; otherwise, it fails (EAGAIN)	If there is sufficient space to immediately write some bytes, then write between 1 and n bytes (which may be interleaved with data written by other processes); otherwise, <i>write()</i> fails (EAGAIN)	SIGPIPE + EPIPE

Segue ora l'esempio 15.16

```
#include "apue.h"
#include <sys/wait.h>

#define DEF_PAGER    "/bin/more"      /* default pager program */

int
main(int argc, char *argv[])
{
    int      n;
    int      fd[2];
    pid_t   pid;
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE   *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                                /* parent */
        close(fd[0]);          /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);    /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
    }

    exit(0);
} else {                                              /* child */
    close(fd[1]);    /* close write end */
    if (fd[0] != STDIN_FILENO) {
        if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        close(fd[0]);    /* don't need this after dup2 */
    }

    /* get arguments for execl() */
    if ((pager = getenv("PAGER")) == NULL)
        pager = DEF_PAGER;
    if ((argv0 = strrchr(pager, '/')) != NULL)
        argv0++;        /* step past rightmost slash */
    else
        argv0 = pager;  /* no slash in pager */

    if (execl(pager, argv0, (char *)0) < 0)
        err_sys("execl error for %s", pager);
}
exit(0);
}
```

In questo codice stiamo simulando la pipe dalla linea di comando, ma invece di essere bash, è il nostro programma che manda l'output al programma more (da cambiare in less)

C'è poi l'esempio di kerrisk che mostra il quadro completo della situazione pipes/pipe_ls_wc che è uno pseudoshell che fondamentalmente fa ls | wc

Su moodle "pipe call's examples" ci sono anche altri esempi di pipe, in uno dei quali il child non sa il fd della pipe, e lo passa come argomento alla exec.

Vediamo velocemente le chiamate popen e pclose, che abbiamo già assaporato nel programma memory dump.

Popen è imparentata con la system, nel senso che entrambe lanciano comandi di shell dentro un programma. Nessuna di queste funzioni è in realtà consigliata.

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);
>Returns: file pointer if OK, NULL on error
int pclose(FILE *fp);
```

La funzione popen, fa una fork e una exec eseguendo il comando cmdstring e ritornando un puntatore a file. Se type è "r" allora il puntatore è collegato allo stdoutput del comando, se invece è "w", il puntatore è collegato allo standard input di esso.

La funzione pclose invece chiude lo stream input/output, aspetta che termini il comando, e ritorna l'exit status dello shell.

Alcuni comandi, come sort, vogliono una end of file altrimenti continuano a leggere all'infinito; infatti, in questo caso è necessario fare la close. Altri programmi invece danno il risultato man mano.

Vediamo ora un esempio:

```
#include "apue.h"
#include <sys/wait.h>

#define PAGER    "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE   *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

    /* copy argv[1] to pager */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");

    exit(0);
}
```

XSI INTER PROCESS COMMUNICATION

I tre modi che analizzeremo che hanno i processi di scambiarsi messaggi e coordinarsi sono i semafori, le memorie condivise e le code di messaggi. A differenza dei mutex che sono utilizzati tra i thread, o eventualmente tra parent e child, questi meccanismi sono sempre usati tra processi differenti.

L'idea alla base di questi meccanismi è quella della chiave: entrambi i processi conoscono una chiave, che può essere per esempio un numero intero, e sono allora in grado di condividere informazioni, o per esempio un semaforo. Nota però che i processi devono già conoscere la chiave per conto loro.

Queste chiavi devono essere system wide, ovvero il numero è nel kernel e non nel processo, altrimenti non avrebbe senso. Chiunque conosca quel numero, allora può entrare in contatto con i processi che lo utilizzano. Questo meccanismo crea però anche un problema, ovvero più processi potrebbero involontariamente acchiappare lo stesso numero, creando quindi un disastro. Per risolvere questa situazione, si usa la chiamata ftok (file to key), che prendendo il path di un file esistente, calcola l'hash di questo path, e utilizza questo come chiave.

```
key_t ftok(const char *path, int id)
```

Parliamo ora nello specifico dell'interfaccia dei **semafori**, la trattazione di questo argomento si fa riferimento a "A synthesis of the System V semaphore APIFile".

Questa interfaccia è eccessivamente complicata, in quanto offrono molte possibilità addizionali che però spesso non vengono utilizzate.

Questi semafori, invece di essere utilizzati in maniera binaria (passi o non passi) ed essere unici (avere un solo semaforo), sono dei counting semaphores (ovvero c'è un numero di processi che possono "passare" e non solo 1) e sono creati in gruppo.

La chiamata per creare l'array di semafori è la seguente:

```
int semget(key_t key, int num_sems, int  
sem_flags);
```

I parametri passati sono la chiave precedentemente originata, il numero di semafori, e dei flag. I flag più utilizzati sono IPC_CREAT (che crea l'array, altrimenti se ne sta a prendere uno già esistente) e IPC_EXCL (se vogliamo che la chiamata fallisca in caso esista già).

Il valore ritornato è utilizzato per identificare il semaforo in tutte le operazioni che lo coinvolgono.

Per utilizzare il semaforo, la chiamata è semop.

```
int semop(int sem_id, struct sembuf *sem_ops,  
size_t num_sem_ops);
```

Poiché in genere si creano array di semafori, a questa funzione si passano array di operazioni. Nel nostro caso le operazioni sono definite da una struttura sembuf.

```
struct sembuf {  
    short sem_num;  
    short sem_op;  
    short sem_flg;  
}
```

In questa struttura, sem_num è il numero del semaforo su cui fare l'operazione, e sem_op è l'operazione da effettuare (-1 per prendere un lock o +1 per rilasciarlo), e poi possono essere aggiunti dei flag tramite sem_flag.

Un flag molto utile è SEM_UNDO, grazie al quale il kernel può togliere il “lock” di un semaforo da parte di un processo morto in maniera anomala, in modo che non sia occupato il lock a vita.

La funzione di controllo del semaforo è semctl:

```
int semctl(int sem_id, int sem_num, int  
command, ...)
```

Si passa l'id del semaforo, il numero del semaforo dell'array corrispondente a quell'id, e poi il comando da utilizzare. In base al comando utilizzato, il numero di argomenti può variare. L'ipotetico quarto argomento è di tipo union, il quale somiglia a una struttura, ma in base alla variabile avvalorata è di tipo diverso. Quello che fa è allocare lo spazio di memoria per il tipo più grande, e poi inserisce il tipo che vogliamo utilizzare dinamicamente. In questo modo in base al comando utilizzato possiamo avvalorare una variabile diversa di semun che il comando utilizzerà.

```
union semun {  
    int val;  
    struct semid_ds *buf;  
    unsigned short *array;  
}
```

I tipici comandi sono SETVAL che inizializza il semaforo (e si avvalora val di semun) e RMID che elimina il semaforo (non si usa semun).

NOTA: se non si rimuove un segnale, questo rimane nella memoria del kernel.

In genere quindi, prima si crea la chiave, poi si crea il semaforo con la chiave ottenuta, poi si inizializza e infine si può usare.

NOTA: in genere si usa la P per indicare che si prende il lock e la V per lasciarlo

NOTA: le operazioni di attesa dei semafori sono bloccanti.

L'esempio più semplice che vede utilizzare i semafori come dei mutex è il seguente:

Since in most cases what you need is a simple, single and binary, semaphore, the calls become:

Create the key:

```
akey = ftok("/home/yourhome/your_dir/your_file",  
1);
```

Create the semaphore:

```
asem = semget(akey, 1, IPC_CREAT | IPC_EXCL);
```

Initialize the semaphore:

```
sem_union.val = 1;  
semctl(asem, 0, SETVAL, sem_union);
```

Prepare the operation:

```
struct sembuf sem_b;  
sem_b.sem_num = 0;  
sem_b.sem_op = -1; /* P() */  
or  
sem_b.sem_op = 1; /* V() */  
sem_b.sem_flg = SEM_UNDO;
```

Perform the operation:

```
semop(asem, &sem_b, 1);
```

When finished, remove the semaphore:

```
semctl(asem, 0, IPC_RMID, sem_union);
```

Vediamo ora degli esempi presi da moodle in “System V IPC examplesFile”

Analizziamo prima sem1: bisogna lanciare due volte questo programma, una delle quali con un argomento qualsiasi. Nota che viene implementata un interfaccia semplificata tramite le funzioni semaphore_v e semaphore_p, set_semvalue, e del_semvalue.

Nota come sono passati i permessi a sem_id che servono a decidere chi può interagire con il semaforo, in questo caso tutto. Se invece fosse stato 600 solo coloro con l'effective user id del proprietario avrebbero potuto utilizzarlo.

Quello che succede è che ogni processo scrive 2 op_char la volta, il quale per un processo è una O e per l'altro una X. Notiamo infatti che i caratteri O e X sono scritti sempre a 2 a 2.

Siccome il semaforo è kernel wide, l'amministratore di sistema deve avere a sua disposizione dei comandi per gestire la situazioni. Con ipcs si vedono i semafori all'interno del kernel, e con ipcrm -s , passando il numero visto con ipcs, si può rimuovere un semaforo.

Vediamo ora come usare le **memorie condivise** tra processi, tramite gli esempi shm1 e shm2 (di kerrisk).

I processi si trovano a condividere la memoria, ovvero avranno un indirizzo di memoria che gli sembrerà essere loro ma in realtà sarà condiviso tra più processi.

Per creare una parte di memoria condivisa si usa la `shmget`, passandogli la chiave pendentemente creata, la dimensione e i flag.

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

Il valore ritornato da questa funzione è l'id della memoria condivisa, rimane ora da fare l'attachment ad essa:

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

Questa funzione ritorna l'indirizzo su cui è mappata la memoria condivisa (si può cercare di mapparla su `addr` se si vuole, altrimenti si mette 0), e ovviamente poi si fa il typecast al tipo di variabile memorizzata su quella memoria.

NOTA: pure essendo la stessa memoria condivisa, verrà attaccata a due indirizzi diversi

Vediamo ora le **code di messaggi** che rappresentano gli sms tra processi, che utilizzano anche un codice di priorità.

Il meccanismo di creazione della chiave è il solito.

Vediamo gli esempi msg1 e msg2.

NOTA: su macos manca il manuale delle funzioni utilizzate, nonostante siano implementate

Prima, data una chiave, si fa la chiama che crea una coda:

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
```

L'idea poi è che un processo manda messaggi alla coda, e l'altro li legge.

Per leggere i messaggi dalla coda, e rimuoverli, si usa

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Se il tipo di messaggio è zero, allora viene letto il primo messaggio della coda. Se invece è maggiore di 0, si legge il primo messaggio con quel tipo. Nel caso in cui il tipo sia negativo, si legge il primo messaggio con il tipo più piccolo compreso tra 0 e il modulo del tipo negativo.

NOTA: `msgrcv` è bloccante ma si potrebbe usare il flag `IPC_NOWAIT` per evitarlo.

Per inviare un messaggio si usa msgsnd e funziona in maniera analoga alla receive.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
               int msgflg);
```

Description

The **msgsnd()** and **msgrcv()** system calls are used, respectively, to send messages to, and receive messages from, a message queue. The calling process must have write permission on the message queue in order to send a message, and read permission to receive a message.

The *msgp* argument is a pointer to caller-defined structure of the following general form:

```
struct msgbuf {
    long mtype;      /* message type, must be > 0 */
    char mtext[1];   /* message data */
};
```

The *mtext* field is an array (or other structure) whose size is specified by *msgsz*, a nonnegative integer value. Messages of zero length (i.e., no *mtext* field) are permitted. The *mtype* field must have a strictly positive integer value. This value can be used by the receiving process for message selection (see the description of **msgrcv()** below).

msgsnd()

The **msgsnd()** system call appends a copy of the message pointed to by *msgp* to the message queue whose identifier is specified by *msqid*.

Per amministrare la coda invece, si usa:

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

IDEA: potrebbe essere intelligente creare un thread per ogni tipo (priorità) di messaggio, in modo che il programma non si fermi

Sono poi stati creati i semafori dallo standard posix, vedi “A synthesis of the POSIX semaphore APIFile”. C’è una distinzione tra i named semaphores, usati da tutti gli unix, e gli unnamed semaphores, che invece non hanno un nome nel filesystem.

NOTA: mac non ha l’implementazione degli unnamed semaphores.

Nel caso dei named semaphores:

```
#include <semaphore.h>
```

The call to create a POSIX semaphore (the first) or to open an existing one (the second):

```
sem_t *sem_open(const char *, int oflag, mode_t mode, unsigned int  
value);  
sem_t *sem_open(const char *, int oflag);
```

Common flags are `O_CREAT` (to create the semaphore, otherwise you are just opening an existing one) and `O_EXCL` if you want the call to fail when it already exists. The third and the fourth parameters are needed only when creating a semaphore. The third parameter is the mode (same as for files) and the fourth parameter is the initial value of the semaphore (<`SEM_VALUE_MAX`>)

The call returns the identifier used for all other operations, which are essentially seven:

```
int sem_post(sem_t *);  
int sem_wait(sem_t *);  
int sem_trywait(sem_t *);  
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);  
int sem_getvalue(sem_t *sem, int *sval);  
int sem_close(sem_t *);  
int sem_unlink(sem_t *);
```

`sem_getvalue` is not supported by macOS (as of 10.14).

NOTA: `getvalue` è inutile perché sapere il semaforo in un determinato istante non dice nulla considerando che qualcun altro potrebbe prenderlo/lasciarlo esattamente dopo

NOTA: anche la `timedwait` manca su mac

C’è una differenza tra `sem_close` che toglie l’utilizzo del semaforo dal processo, e `sem_unlink` che invece rimuove proprio il semaforo.

Sono utili in questo caso gli esempi di kerrisk psem/, dove sono stati preparati diversi programmi ognuno che fa una operazione diversa. L’utilizzo è il seguente:

```
$ ./psem_create -cx /demo 666  
$ ./psem_wait /demo &  
$ ./psem_getvalue /demo  
$ ./psem_post /demo  
$ ./psem_unlink /demo
```

NOTA IMPORTANTE: i semafori posix hanno una mancanza fondamentale rispetto agli altri semafori, ovvero l’utilizzo del flag UNDO

NOTA: per vedere se ci sono i semafori ls /dev/shm (su linux)

Esiste una versione **POSIX** anche delle **memorie condivise**, che cerca di renderle più aderenti al modello generale di unix. Analizziamo i file dentro la directory pshm di kerrisk. Questi esempi sono strutturati in modo che ci sia un codice per la creazione della memoria condivisa, uno per la scrittura su essa, uno per la lettura e uno per la chiusura.

Prima si chiama il file per creare la memoria condivisa e si usa l'opzione -c.

Per creare la memoria si usa la `shm_open` che restituisce un fd (come con i semafori posix), che permette quindi tutte le comodità del caso come per esempio cambiarne i permessi al volo, oppure la possibilità di usare funzioni come `ftruncate`.

```
fd = shm_open(argv[optind], flags, perms);
```

Anche in questo caso bisogna fare l'attach della memoria ad un particolare indirizzo, e si fa con `mmap` (ovviamente con il flag per la condivisione).

```
addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Da notare che ciò che identifica la memoria è il suo nome, ed è questo che devono sapere i processi per poterla condividere. L'importante è che non ci sia il flag `O_EXCL` quando si fa la `open`, altrimenti non viene fatta nel caso in cui il file sia già esistente.

Per scrivere sulla memoria condivisa, una volta aperta e mappata, semplicemente si può fare per esempio un `memcpy`.

NOTA: essendo un file descriptor, si può anche fare `fstat`

Infine, c'è la cancellazione che si fa semplicemente con `unlink`:

```
shm_unlink(argv[1])
```

NOTA: su mac questo programma dà errore perché a quanto pare non si possono fare due `truncate` ad una memoria condivisa, ma solo uno. Non è comunque un problema perché semplicemente si leggerà una parte limitata della memoria condivisa. Da notare che non c'è bisogno di mettere il terminatore della stringa manualmente.

NETWORKING NETWORKING NETWORKING NETWORKING

References

- **UNIX® Network Programming Volume I, Third Edition: The Sockets Networking API, W. R. Stevens**
<http://www.unpbook.com/>
- **The Open Group Base Specifications Issue 7**
<http://pubs.opengroup.org/onlinepubs/9699919799/>
- **The Linux Programming Interface - A Linux and UNIX® System Programming Handbook, M. Kerrisk**
<http://man7.org.tlpi/>
- **Beej's Guide to Network Programming:**
<http://beej.us/guide/bgnet/>
- **TCP/IP Sockets in C: Practical Guide for Programmers, 2nd Ed,** M. J. Donahoo and K. L. Calvert
<http://cs.ecs.baylor.edu/~donahoo/practical/CSockets2/>
- **GAPIL:** Guida alla Programmazione in Linux
gapil.gnulinux.it/

Dove, la bibbia è il primo di questi.

Conviene sempre confrontare le pagine di manuale di kerrisk (man7.org.tlpi) che sono molto meglio di quelle di sistema.

Poi in italiano c'è GAPIL che ha espanso la parte di networking che non è male.

NOTA: quando si verificano gli standard conviene controllare direttamente la susv4 tanto è

RICORDA: comandi utili nell'ambito del networking sono lsof -i che vede tutte le connessioni di rete attive, e netstat che mostra delle statistiche nell'ambito della rete (mostra anche le code di ricezione, o con -r le routing table della macchina).

Analizziamo ora le interfacce che useremo.

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);

Returns the socket descriptor on success, -1 on error
```

La chiamata `socket` crea un socket e ritorna un identificatore (socket file descriptor). La chiamata prende 3 argomenti:

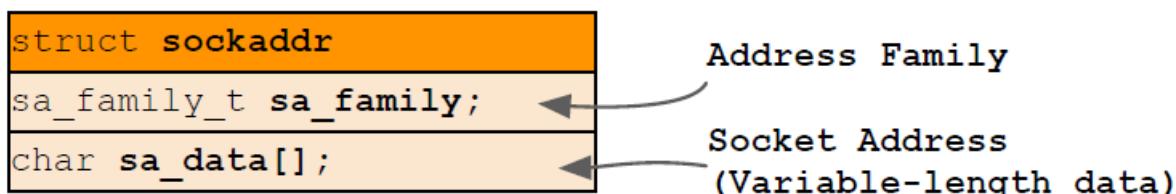
- Il dominio in cui avverà la comunicazione, e si passerà il valore di una costante in base alla situazione.
 - **AF_UNSPEC**: Unspecified
 - **AF_INET**: Internet domain sockets for use with IPv4
 - **AF_INET6**: the IPv6 version
 - **AF_UNIX**: UNIX domain sockets

NOTA: si possono usare i socket anche per comunicare sulla stessa macchina tra processi, infatti sono presenti gli unix socket.

- Il tipo di comunicazione, che può essere DGRAM o STREAM. Datagram è di tipo best effort, invece STREAM crea una connessione ed è affidabile.
Non è detto si debba usare UDP e TCP, infatti l'interfaccia è fatta in modo di essere generica.
- Il terzo campo, protocol, in genere è riempito con uno 0, ed sta a lui a capire che si vuole usare TCP o UDP (in base al tipo di comunicazione)

Per poter avere una comunicazione, è necessaria una quaterna: indirizzo e porta del mittente, e indirizzo e porta del destinatario.

Il collegamento si basa sulla struttura `sockaddr`:



Come vedremo le interfacce sono un po' scomode da usare, perché inizialmente era tutto pensato per l'IPv4, e successivamente ci si è dovuti adattare all'IPv6, cercando di usare però le stesse interfacce.

La struttura `sockaddr` è opaca, nel senso che, come primo campo, prende la famiglia (se si tratta di IPv4: `AF_INET` o IPv6: `AF_INET6`, etc...) e poi dei dati grezzi da specificare.

NOTAZIONE: ipv6 ha 128 bit, e si possono rappresentare con 8 campi da 4 cifre esaecimali. Siccome questi indirizzi sono lunghi, capita che ci siano lunghe sequenze di zeri, che si

possono omettere utilizzando la notazione :: . Non possono esserci più volte presenti perché altrimenti non si saprebbe quante quaterne di zeri sono sostituite.

E3D7:0000:0000:0000:51F4:9BC8:C0A8:6420 -> E3D7::51F4:9BC8:C0A8:6420

Si possono anche mappare gli indirizzi IPv4 sugli IPv6, tramite la seguente notazione:



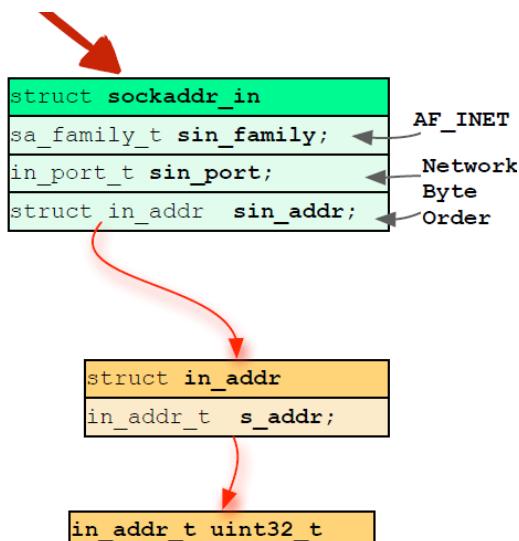
Per esempio: ::ffff:192.0.2.128 corrisponde all'indirizzo IPv4 192.0.2.128

Le prime 1024 porte sono well known e riservate, dalla 1024 alla 49151 sono registrate ma non well known, e le porte da 49152 a 65536 sono quelle effimere che possono essere tranquillamente usate dagli utenti. Per conoscere le assegnazioni delle porte si può leggere il file /etc/services.

Per rappresentare sia i socket IPv4 e IPv6, è necessario utilizzare dei tipi di dato ad'hoc:

sockaddr_in per IPv4 e sock_addr_in6 per IPv6. Questa confusione è dovuta al fatto che si è sempre usata la struttura sockaddr, e in questo modo si possono usare queste nuove strutture tramite un cast a sockaddr, in modo che tutto il codice vecchio continui a funzionare, e si possa ancora fare riferimento alla struttura sockaddr.

Analizziamo la struttura sockaddr_in:



Questa ha come campi la famiglia (AF_INET), la porta, e un puntatore a un'altra struttura che contiene l'indirizzo IP.

Su macos la struttura sockaddr_in è un po' diversa, prende anche la lunghezza della struttura, e ha un campo finale dedicato al padding.

Su MAC OS X:

```
struct sockaddr_in {
    uint8_t         sin_len;
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

Analogamente la struttura sock_addr_in6 specifica subito la famiglia.

struct sockaddr_in6	AF_INET6
sa_family_t sin6_family;	
in_port_t sin6_port;	Network
uint32_t sin6_flowinfo;	Byte
struct in6_addr sin6_addr;	Order
uint32_t sin6_scope_id;	

struct in6_addr	128-bit IPv6
uint8_t s6_addr[16];	address in Network Byte Order

Su mac la struttura prende anche il campo len:

Su MAC OS X in netinet6/in6.h:

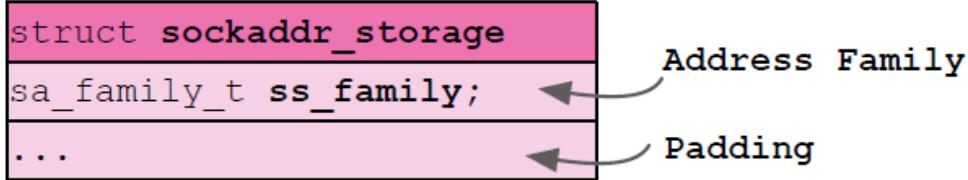
```
struct sockaddr_in6 {
    uint8_t sin6_len; // length of this struct(sa_family_t)
    sa_family_t sin6_family; // AF_INET6
    in_port_t sin6_port; // Transport layer port #
    uint32_t sin6_flowinfo; // IP6 flow information
    struct in6_addr sin6_addr; // IP6 address
    uint32_t sin6_scope_id; // scope zone index
};
```

Su Linux, in /usr/include/linux/in6.h:

```
struct sockaddr_in6 {
    unsigned short int sin6_family; //AF_INET6
    __be16 sin6_port; // Transport Layer port #
    __be32 sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr; // IPv6 address
    __u32 sin6_scope_id; // scope id (new in RFC2553)
};
```

Le interfacce hanno sempre la struttura di tipo sockaddr, a cui viene letto il campo family, e in base a questo si fa il cast di questa struttura a quella passata veramente: sockaddr_in se family è AF_INET o a sockaddr_in6 se invece è AF_INET6.

A queste strutture si aggiunge poi sockaddr_storage:



Questa struttura è sufficientemente grande per contenere sia indirizzi IPv4 che IPv6 e trattarli trasparentemente. Per poter interpretare i dati in essa contenuti si dovrà fare il cast al tipo specifico (sockaddr_in o sockaddr_in6) in base al campo ss_family.

Se si vuole mettere l'informazione di una sockaddr da parte, poiché la dimensione non è nota a priori, invece di sockaddr si usa sockaddr_storage, che ha spazio per tutto.

FORSE l'idea è di sostituire la struttura sockaddr.

NOTA: il network byte order è il big endian, ovvero MSB a sinistra e il LSB a destra (come ragiona l'uomo). Ogni volta che i dati devono essere trasferiti tra due host devono essere prima convertiti in Network Byte Order, e riconvertiti in Host Byte Order al momento della ricezione. Questo è necessario perché non è detto che il processore della macchina legga i valori in codifica big endian.

Per le conversioni sono utilizzate le seguenti funzioni:

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

These functions shall convert 16-bit and 32-bit quantities between network byte order and host byte order.
```

htonl (host to network long) -> prima di spedire qualcosa bisogna convertirlo tramite questa funzione

htons (host to network short) -> utilizzato per i 16 bit delle porte

Chi riceve i valori deve fare la conversione opposta tramite ntohl e ntohs.

NOTA: questa cosa vale solo per il caso di IPv4, perché nel caso di IPv6 sono usati gli array in cui non c'è dubbio sull'ordine.

Ci sono poi delle funzioni che tengono conto della nostra riluttanza a scrivere gli indirizzi con 32 bit, sapendo che preferiamo scriverli in notazione decimale quadripuntata. Le funzioni in questione sono `inet_ntop` (numeric to presentation) e `inet_pton` (presentation to numeric).

```
#include <arpa/inet.h>
const char *inet_ntop(int af, const void *restrict src,
                      char *restrict dst, socklen_t size);

int inet_pton(int af, const char *restrict src, void *restrict dst);
```

NOTA: è necessario indicare se siamo nel campo di IPv4 o IPv6.

Si passa poi la variabile che sarà usata come output e la dimensione del buffer destinazione, che sarà o 16 bit o 46 bit.

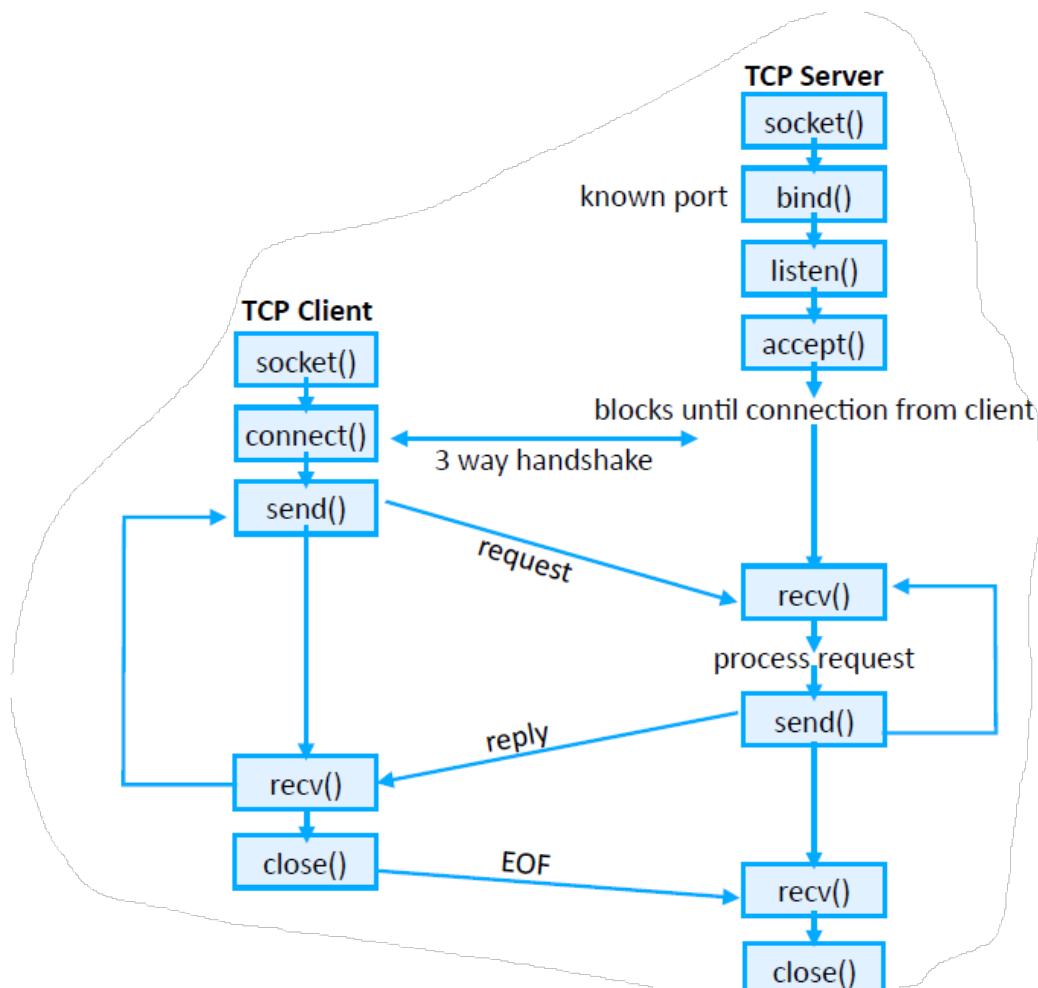
16 -> 3 bit per ogni cifra, $3 \times 4 + 3$ bit per i punti + 1 per il null finale

46 -> perché si considera la possibile notazione che rappresenta un ipv4 in formato ipv6 che arriva a usare 46 bit

La funzione `ntop` ritorna null in caso di errore, o il puntatore al buffer in caso di successo.

La funzione `pton` invece ritorna 1 se la conversione va a buon fine, 0 se la stringa di input non è una rappresentazione valida, e -1 in caso di errore.

Ricorda che la comunicazione tcp avviene con modalità client e server, ovvero c'è un daemon che gira, in attesa di richieste dall'altra parte del mondo.



Il server come prima cosa fa la chiamata socket, poi fa la bind (che lega il socket a un indirizzo ip). Successivamente, si mette in modalità di ascolto tramite la chiamata listen. Ciò però è inutile fino a che non è fatta la chiamata accept, che corrisponde all'aprire la propria porta al mondo.

Quando il server crea una socket e fa il bind e la listen si dice che ha fatto una “passive” open, dopo la connect si ha una active open.

Nota che anche il client deve creare un socket, ma non è necessario che egli faccia una bind: è lui che raggiunge il server e di conseguenza il server conoscerà il suo indirizzo e la sua porta in questo modo, senza che sia necessario che il client li specifichi. La scelta della porta da parte del client sarà automatica, così come l'indirizzo utilizzato dipenderà dall'interfaccia con cui ci si connette al server.

La connect del client corrisponde al bussare alla porta del server.

Dopo il 3 way handshake, si crea un canale di comunicazione full duplex, in cui è possibile fare send e receive in entrambe le direzioni.

NOTA: l'ordine di send e receive non è necessariamente questo, è possibile che appena il

client bussi alla porta del server questo mandi una risposta al client immediatamente, come per esempio con i time server.

Analizziamo la chiamata bind:

```
#include <sys/socket.h>
int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
```

Returns 0 on success, -1 on error.

The <sys/socket.h> header shall define the socklen_t type, which is an integer type of width of at least 32 bits.

Si passa l'identificativo del socket (che fin'ora era solo un numero), la struttura sockaddr (la struttura generale), e infine si da informazione sulla lunghezza della struttura passata (avrà dimensione diversa in base a se è IPv4 o IPv6)

Esiste una costante che permette di fare da server su tutte le interfacce ed è INADDR_ANY, che consiste in tutti 0. Nel caso di ipv4 è necessario usare la funzione htonl, nel caso ipvt6 non è necessario perché come detto ragione sull'array.

Un'altra costante importante è INADDR_LOOPBACK, che corrisponde a 127.0.0.1, ovvero localhost, cioè l'indirizzo locale della macchina. Questo serve a testare il server sulla propria macchina, o per costruire pipe all'interno del sistema e fare comunicare processi.

Segue poi la chiamata listen, che semplicemente prende il socket e il backlog (che corrisponde alla coda massima)

```
#include <sys/socket.h>
int listen(int socket, int backlog);
```

Returns 0 on success, -1 on error.

In genere al parametro backlog si assegna SOMAXCONN che è 128.

La chiamata per "l'apertura della porta" è la accept:

```
#include <sys/socket.h>

int accept(int socket,
           struct sockaddr *restrict address,
           socklen_t *restrict address_len);
```

Upon successful completion, accept() shall return the non-negative file descriptor of the accepted socket. Otherwise, -1 shall be returned and errno set to indicate the error.

Da notare che dopo il socket (necessario perché un processo potrebbe ascoltare su più porte), si passa il parametro address e address_len che sono in realtà un output. Address

(???) è il file descriptor, chiamato connected socket: ogni volta che qualcuno si connette, comunicherà con lui attraverso questo file descriptor, che gode di tutte le proprietà del caso. Ogni accept avrà come risultato un file descriptor diverso.

È quindi a questo file descriptor che si farà la chiamata recv e send, ma ricorda che recv è bloccante.

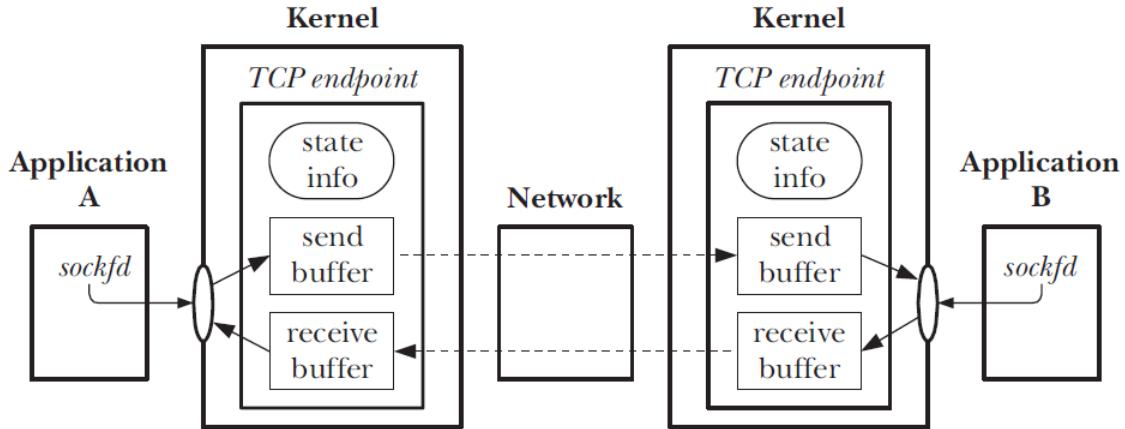
IDEA: un modo per gestire tutte le richieste e i fd, è usare un thread per ciascuno, e fare in modo che ognuno chiuda il proprio file descriptor. Ogni connessione fa una fork, e poi si fa lavorare il child. Un'ulteriore idea per evitare di creare troppi child tutti insieme, è quella del prefork (utilizzata da apache) che prepara i child e quando necessario si assegna al child la richiesta. Si potrebbero anche mischiare child e thread.

Dal punto di vista del client, si crea un socket e si fa la connect e si passa l'indirizzo di destinazione:

```
#include <sys/socket.h>
int connect(int socket, const struct sockaddr *address,
            socklen_t address_len);
Returns 0 on success, -1 on error.
```

NOTA: socket indica il punto di partenza de client, mentre address indica l'indirizzo del server

NOTA: così' come si usano le chiamate send e recv, più specifiche nell'ambito del networking, si potrebbero usare anche le chiamate write e read. L'unica differenza è infatti che le prime chiamate prendono come argomento dei flag.



Quando l'applicazione A fa una send al socket, entrano in gioco dei buffer:

Send buffer è una memoria dove si mette ciò che deve essere spedito, ed è una fifo. Questa fifo è riempita con ciò che vogliamo spedire, ma non viene svuotata finchè non si riceve l'ack. La send infatti può diventare bloccante se non si riceve l'ack.

Il recv buffer ha la stessa logica, viene svuotato solo dopo aver mandato l'ack.

NOTA: su tcp non si parla di limite di pacchetti ma di byte, infatti il number sequence indica il numero del primo byte nel segmento tcp.

Vediamo ora le funzioni send e recv:

```
#include <sys/socket.h>

ssize_t send(int socket, const void *buf, size_t len, int flags);
ssize_t recv(int socket, void *buf, size_t len, int flags);
```

NOTA: Ci sono delle opzioni interessanti che si possono assegnare ai socket, seguono alcune. SO_RCVBUF è un opzione dei socket che permette di fissare la dimensione del receiver buffer. Un'altra opzione è LOW_WATER che permette di decidere quanti byte bisogna accumulare prima di mandarli.

Quali sono invece i flag usati in send e recv? In realtà sono poco usati, e li possiamo vedere sul manuale (nota alcuni flag sono specifici per linux):

Alcuni esempi sono

-msg_dontwait se lo dai alla receive non è più una chiamata bloccante, cosa che noi sapevamo già fare con il flag O_NONBLOCK, ma ATTENZIONE perché in questo caso si riferisce a una lettura in particolare.

-MSG_PEEK: con questo flag, nonostante aver letto i byte, non viene mandato l'ack e quindi rimangono nel buffer di ricezione.

CERCA IMMAGINE CHE MOSTRA I VARI STATI DI UNA CONNESSIONE TCP (closed, established, listen, ... possono essere visti con lsof -i)

PUOI GUARDARE ANCHE IMMAGINE DELLO STEVENS. Questa mostra in ogni stato quali sono i possibili modi per cambiare stato. I percorsi tratteggiati sono quelli dei server, quelli solidi sono quelli del client. Poi c'è proprio la sequenza delle chiamate pacjet exchange for tcp connection from stevens....

Infine c'è la close, che chiude la connessione.

```
#include <unistd.h>

int close(int fd);
Returns 0 on success, -1 on error.
```

Bisogna fare però attenzione al suo utilizzo, perché se c'è un child che ha ancora il fd descriptor aperto, la connessione rimane aperta. C'è un reference count: il kernel conta quanti processi hanno aperto il socket, e finchè non scende a 0 la connessione rimane aperta.

Esiste anche un modo brutale di chiudere la connessione tramite lo shutdown del file descriptor.

NOTA: se ci sono due processi in ascolto sullo stesso socket aperto, i dati sono sempre quelli. Se un processo li legge e manda l'ack, gli altri processi non lo vedono più. E' necessaria quindi della sincronizzazione.

Ci sono poi due chiamate, getsocketname e getpeername:

```
#include <sys/socket.h>
int getsockname(int socket, struct sockaddr *restrict
                local address, socklen_t *restrict address len);
```

Se il client dopo aver fatto la connect vuole sapere il proprio indirizzo ip e la porta con cui ci si è connessi, si usa la chiamata getsockname. Si passa il numero del socket, e tutti gli altri argomenti vengono usati come output. Nota che la stessa funzione potrebbe tornare utile al server, nel caso ci sia un child voglia avere queste informazioni.

```
#include <sys/socket.h>
int getpeername(int socket, struct sockaddr *restrict
                remote address, socklen_t *restrict address len);
```

La chiamata getpeername invece si usa più dal lato server, e serve a sapere il client con cui sta comunicando. In teoria il client si può conoscere dalla accept, ma anche in questo caso il child a cui è assegnata la gestione di questa "richiesta" potrebbe non sapere con chi sta parlando.

NOTA: ricorda che si potrebbe anche fare una accept con parametro NULL perché in quel momento non importa con chi si sta parlando, e poi si usa questa funzione in seguito quando si è interessati.

Vediamo ora qualche esempio, scaricando Networking code examples - v2File da moodlis.

Iniziamo con la directory SocketAddress.

NOTA: sono usati dei tag @ che sono permettono di creare automaticamente la documentazione tramite il software doxygen. Per esempio con @brief si da la descrizione generale del codice, e così via.

La prima funzione, la initSockaddr_in prende i dati in ingresso e costruisce la struttura sockaddr. È usata la funzione memset che azzerà tutto ciò a cui punta il puntatore, perché non si ha certezza di che valore possa avere una variabile nello stack quando viene utilizzata: quando si tira su la tendina, il valore assegnato a un determinato indirizzo di memoria rimane quello, non viene azzerato.

C'è poi la funzione che fa la print, e segue il percorso inverso: prende in input un puntatore alla struttura sockaddr, e mette il valore dell'ip (in formato presentation) in ipv4. In seguito vengono stampati l'ip e la porta.

La stessa procedura è seguita per IPv6.

Procediamo vedendo un semplice esempio di server TCP, la directory 02 tra quelle degli esempi.

NOTA: per sapere se un server è attivo, si usa il comando lsof -i e si controlla se c'è il server sulla porta attesa.

NOTA: non si può spegnere e riaccendere immediatamente un server tcp sulla stessa porta, è necessario del tempo per pulire i numeri di sequenza e le variabili utilizzate per evitare ci siano interferenze.

Per fare una richiesta al dns tramite linea di comando si può usare il comando dig: dig +short hplinux3.unisalento.it

Può essere interessante analizzare il traffico tramite wireshark: ci sono due tipi di filtering, uno che prende tutto ma manda in output solo ciò che si vuole (display filter), e l'altro che cattura direttamente solo ciò che è richiesto (capture filter)

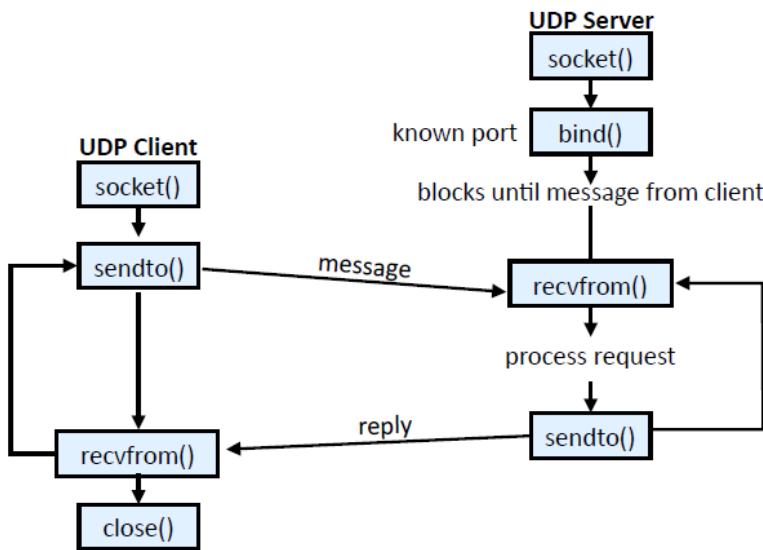
Può essere utile utilizzare il debugger insieme a wireshark per vedere cosa succede passo passo.

NOTA: non c'è un segmento per ogni messaggio, questo perché come abbiamo visto i byte vengono messi in un tubo, ed è il protocollo TCP che decide quando mandarli effettivamente per ottimizzare.

NOTA: quando il client manda la risposta, manda anche l'ack rispetto al messaggio precedente.

NOTA: telnet è l'antenato di ssh, ma se si specifica una porta, avvia una connessione tcp con un server.

Vediamo ora il server UDP.



NOTA: esiste un protocollo chiamato IP RAW, che permette di aggiungere dei dati al protocollo IP. È utile se per esempio si vuole creare un nuovo protocollo di trasporto. C'è un capitolo apposito sullo Stevens.

UDP CONNECT

Esiste un particolare tipo di UDP che è il CONNECTED UDP, che consiste nel “salvare porta e indirizzo ip” in modo da simulare una connessione. Si fa in modo che le due parti si ricordino di sé stessi e rifiutino altre “connessioni”. Funziona come un filtro, `sendto` e `receive` si possono fare solo a un determinato “soggetto”: il socket del client, per esempio, può essere usato solo per inviare a quell’indirizzo e quella porta.

La connect la potrebbero fare sia client che server, ma in genere la fa il client.

NOTA: se è stata fatta la connect, si può usare anche `send` invece di `sendto`

È da notare che in UDP, anche senza la connect, il client riceve sempre sulla stessa porta (da cui manda), anche se logicamente non dovrebbe esserci una connessione fissa.

NOTA: se si fa la connect, e si inviano dati alla porta sbagliata non fallisce la connessione. Questo perché in realtà non c’è nessuna connessione, ma si comporta semplicemente da filtro. L’errore si riceve al momento della `send` se la porta non è raggiungibile. L’errore, per esempio di eventuali `receive`, si legge da `perror`.

NOTA INTERESSANTE: non essendoci una connessione, il client non è detto che debba stare sempre in attesa di risposta dal server sempre sulla stessa porta, ma in realtà lo fa. Cosa succederebbe se si mandassero altri segmenti UDP sulla porta su cui il client attende risposta dal server? In teoria dovrebbe accettarli.

Analizziamo la sendto:

```
#include <sys/socket.h>

ssize_t sendto(int sock, const void *message, size_t len, int flags,
                  const struct sockaddr *dest_addr, socklen_t dest_len);
```

NOTA: si possono mandare messaggi da 0 byte

A differenza di TCP, che aspetta e decide lui quando mandare i datagrammi, UDP li manda immediatamente.

Segue poi la recvfrom:

```
#include <sys/socket.h>
ssize_t recvfrom(int sock, void *restrict message, size_t len,
                   int flags, struct sockaddr *restrict address,
                   socklen_t *restrict address_len);
```

Dove message (puntatore al buffer che riceve il messaggio) e len(lunghezza in byte del buffer) e address sono valori di uscita. Address len deve inizialmente puntatore a un intero che contiene la dimensione inbyte dello storage puntato da address: all'uscita dalla funzione quest'intero conterrà la dimensione richiesta per rappresentare l'indirizzo del connecting socket

NOTA: è bloccante. Inoltre, non fallisce se arriva a destinazione unreachable, rimarrà comunque in attesa.

NOTA: nella recvfrom non si specifica da chi si vuole ricevere... è la sendto che automaticamente sceglie porta e indirizzo che vengono mantenuti

NOTA: basta ricevere un datagramma e si esce dalla receive from, a differenza dal caso TCP in cui si specifica quanto si vuole ricevere.

NOTA: non si vuole mai la frammentazione di un pacchetto, per questo motivo esiste la lunghezza massima corretta che permette di inviare più dati possibile senza che vengano però suddivisi nel percorso. Si sceglie infatti la dimensione minore possibile tra tutte le dimensioni di max transfer unit dei router di passaggio: 512 byte in genere.

NOTA: quando un client manda un messaggio a un server che non è in ascolto su quella porta, si riceve un messaggio icmp. Se il server non si raggiunge proprio, sono i router intermedi a mandare il segnale, altrimenti sarà il server stesso a dire che la porta è unreachable.

Analizziamo l'esempio 04 di UDP basic, iniziando da echo server e echo client.

In pratica, il server prende un messaggio e lo sputa indietro a chi l'ha mandato. In questo esempio è utilizzata la proprietà dei socket, che stabilisce di spedire tutti i byte come vengono mandati, senza aspettare. La disconnessione del socket è platform dependent, la cosa migliore da fare è scollegare la struttura di connessione e fare una connect a una struttura azzerata.

TCP RELIABILITY

Dentro la close del tcp, c'è in realtà un reference count. Per evitare questo meccanismo, si può usare la shutdown che chiude la connessione indipendentemente dal reference count. In questo caso se c'è qualcosa che andava letto, viene letto, e poi si chiude la connessione.

HOSTNAME RESOLUTION

Un record ptr è il record della risoluzione inversa, SE ESISTE, si entra nel db con un indirizzo ip e si esce con un nome.

FQDN: fully qualified domain name, quando è correttamente formato si chiama definisce così

Prima si usavano funzioni che ora non si utilizzano più, come gethostname... Oggi la funzione utilizzata è getaddrinfo().

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict nodename,
                 const char *restrict servname,
                 const struct addrinfo *restrict hints,
                 struct addrinfo **restrict res);
```

I primi due campi sono nodename e servname e si inserisce ciò che si vuole tradurre. Nodename potrebbe essere quindi unisalento.it e service name è il “servizio”, potrebbe quindi essere DNS, http o altri. Il terzo campo è HINTS, perché fissato un nome e un servizio, ci possono essere più soluzioni diverse (per esempio una ipv4 e una ipv6). Hints è allora una struttura addrinfo (che punta a una sockaddr), che si riempie per aiutare la funzione a dare una risposta. Il quarto campo res è il risultato, che è una struttura di tipo addrinfo (e quindi sockaddr) riempita.

NOTA: se si usa questo flag, allora invece di passare service name si passa la porta su cui si ascolta hints.ai_flags |= AI_NUMERICSERV;

In questo modo abbiamo ottenuto più semplicemente la nostra struttura sockaddr e tutte le informazioni necessarie per poter creare un server, o “connettersi” ad esso.

Questo è un puntatore a una linked list di addrinfo, perché la risposta che abbiamo non è detto che sia unica. ** è quindi un puntatore a una lista di strutture.

```
struct addrinfo{
    int           ai_flags      //Input flags
    int           ai_family     //Address family of socket
    int           ai_socktype   //Socket type
    int           ai_protocol   //Protocol of socket
    socklen_t   ai_addrlen   //Length of socket address
    struct sockaddr *ai_addr    //Socket address of socket
    char          *ai_canonname //Canonical name of the hostname
    struct addrinfo *ai_next    //Pointer to next in list
};
```

NOTA: il canonical name (cioè il vero e unico nome DNS), logicamente, è ritornato solo nel primo elemento della lista

I flag si mettono in OR tra loro, e alcuni possono essere;

- AI_PASSIVE, che è un flag che mettono server. I server devono riempire la struttura sockaddr di sé stessi per passarla alla bind. Se nodename è null (nella getaddrinfo), allora l'indirizzo ritornato sarà settato a INADDR_ANY. Se il flag non c'è, l'indirizzo ritornato è usato per una connect (quindi lato client).
Se nodename = NULL e AI_PASSIVE non è settato, l'indirizzo che sarà inserito in result è l'indirizzo di loopback
- AI_CANONNAME: significa che si richiede il canonical name in ai_canonname della struttura addrinfo. Se non c'è un canonical name, è restituita una copia di nonename in questo campo.
- AI_NUMERIC_HOST: se è settato, significa che nodename è una stringa e quindi non ci sarà nessun DNS query poiché si sta già passando l'IP
- AI_NUMERICSERV: evita la risoluzione del nome del servizio, servname sarà ora una stringa che indica la porta
- AI_V4MAPPED che se usato con ai_family = AF_INET6, allora il valore di ritorno sarà un indirizzo IPv4 mappato IPv6

Comn gli ultimi due campi. Si rinuncia al servizio DNS ma si sfrutta semplicemente la funzione per creare la struttura sockaddr velocemente.

L'output sarà una double linked list, che quando non serve più è bene svuotare tramite la funzione freeaddrinfo:

```
#include <netdb.h>
#include <sys/socket.h>

void freeaddrinfo(struct addrinfo * res);
```

Poi c'è la chiamata di risoluzione inversa, ovvero se si vuole avere l'hostname a partire dall'indirizzo che è getnameinfo:

```
#include <netdb.h>
#include <sys/socket.h>

int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char *host, size_t hostlen, char *service, size_t servicelen, int flags);
```

Dove addr punta alla struttura sockaddr passata in input, di dimensione addrlen. Hostname e service saranno delle stringhe di output.

Esiste poi una funzione, `gai_strerror` che interpreta gli errori di `getaddrinfo` e `getnameinfo` e ritorna una stringa che ne descrive l'errore:

```
#include <netdb.h>
#include <sys/socket.h>

const char *gai_strerror(int ecode);
```

Error constant	Description
EAI_ADDRFAMILY	No addresses for <i>host</i> exist in <i>hints.ai_family</i> (not in SUSv3, but defined on most implementations; <code>getaddrinfo()</code> only)
EAI AGAIN	Temporary failure in name resolution (try again later)
EAI_BADFLAGS	An invalid flag was specified in <i>hints.ai_flags</i>
EAI_FAIL	Unrecoverable failure while accessing name server
EAI_FAMILY	Address family specified in <i>hints.ai_family</i> is not supported
EAI_MEMORY	Memory allocation failure
EAI_NODATA	No address associated with <i>host</i> (not in SUSv3, but defined on most implementations; <code>getaddrinfo()</code> only)
EAI_NONAME	Unknown <i>host</i> or <i>service</i> , or both <i>host</i> and <i>service</i> were NULL, or AI_NUMERICSERV specified and <i>service</i> didn't point to numeric string
EAI_OVERFLOW	Argument buffer overflow
EAI_SERVICE	Specified <i>service</i> not supported for <i>hints.ai_socktype</i> (<code>getaddrinfo()</code> only)
EAI_SOCKTYPE	Specified <i>hints.ai_socktype</i> is not supported (<code>getaddrinfo()</code> only)
EAI_SYSTEM	System error returned in <i>errno</i>

vediamo gli esempi, resolve TCP, vediamo TCPconcurrentS.c

NOTA: i pid non sono mai riciclati, se ne prende sempre uno più grande

NOTA: quando chiudi un server TCP, devi aspettare un po di tempo prima di farlo ripartire, perché potrebbero esserci segmenti in volo che si andrebbero a confondere con il nuovo server

Vediamo alcune socket options.

`Setsockopt` e `getsockopt` sono due chiamate usate per ottenere o modificare le opzioni del socket:

```
#include <sys/socket.h>
int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
int getsockopt(int socket, int level, int option_name,
               void *restrict option_value, socklen_t *restrict option_len);
```

Si passa il numero di socket aperto, il “livello” (le opzioni sono divise in livelli), il nome dell’opzione, e un puntatore di tipo di dato generico e la lunghezza dell’opzione. Il tipo di dato passato è generico, perché in base a opzioni diverse possono essere necessari diversi tipi di dato.

Per livello si intende il layer a cui opera: transport, network... Il livello `SOL_SOCKET` è il livello generale, altrimenti ci sono i livelli per gli specifici protocolli: `IPPROTO_IP`, `IPPROTO_IPV6`, etc

Segue una lista delle opzioni più utilizzate:

```
SO_ACCEPTCONN: Socket is accepting connections
SO_BROADCAST: Transmission of broadcast messages is supported
SO_DEBUG: Debugging information is being recorded
SO_DONTROUTE: Bypass normal routing
SO_ERROR: Socket error status
SO_KEEPALIVE: Connections are kept alive with periodic messages
SO_LINGER: Socket lingers on close
SO_OOBINLINE: Out-of-band data is transmitted in line
SO_RCVBUF: Receive buffer size
SO_RCVLOWAT: Receive "low water mark"
SO_RCVTIMEO: Receive timeout
SO_REUSEADDR: Reuse of local addresses is supported
SO_SNDBUF: Send buffer size
SO SNDLOWAT: Send "low water mark"
SO_SNDTIMEO: Send timeout
SO_TYPE: Socket type
```

Tra queste emergono SO_BROADCAST, usata per mandare pacchetti in broadcast, SO_SNDBUF, che determina la dimensione dei buffer di invio (SO_RCVBUF che invece è relativa a quelli in ricezione). SO_KEEP_ALIVE stabilisce che la connessione deve automaticamente essere manenuta viva tramite dei messaggi ogni tot tempo, per evitare che il timer scaduto faccia chiudere la connessione.

SO_REUSEADDR serve a permettere di fare una bind su un socket, anche se lo stesso indirizzo IP è usato su un altro socket.

Come esempio, si può vedere la versione 3 del server che è uguale a quella precedente, ma si usano delle opzioni per i socket (usa su bbedit l'opzione per comparare due finestre).

Comunicazione 1-to-[many | all]-communications

Prendiamo gli esempi dalla direcotry passata dal professore, nella subdirectory multicast.

Iniziamo analizzando MulticastReceiver.c

Vediamo esempio multicast receiver, dove la differenza rispetto ad unicast è la funzione join_group, e la corrispondente leave_group.

Joiningroup tramite la struttura ip_mreq, stabilisce l'indirizzo al quale “ci si vuole iscrivere” per ricevere dati multicast, e l'interfaccia su cui si vuole ricevere. Si setta poi la socket option IP_ADDR_MEMBERSHIP, passando la struttura joinRequest.

Nella corrispondente leave, si passa come opzione IP_DROP_MEMBERSHIP.

Vediamo ora tramite una cattura di pacchetti con wireshark, cosa succede sulla rete. Usiamo receiverM e senderM, filtrano la cattura “igmp”. Si può notare un pacchetto IGMP chiamato membership report per il gruppo specificato (messaggio mandato dall'host al gruppo).

Quando il ricevitore smette di ricevere, fa la leave dal gruppo, si manda un messaggio leave di 224.100.100.100 avente destinazione 224.0.0.2: un indirizzo riservato che indica tutti i router della subnet. In pratica ha detto a tutti i router della subnet che non è più interessato al gruppo.

C'è poi l'altro esempio che è quello della chat multicast che usa UDP.

NOTA: per usare multicast, bisogna essere sulla stessa subnet, quindi in teoria usare la stessa interfaccia.

Il Loop flag indica che si vuole ricevere quello che si è spedito.

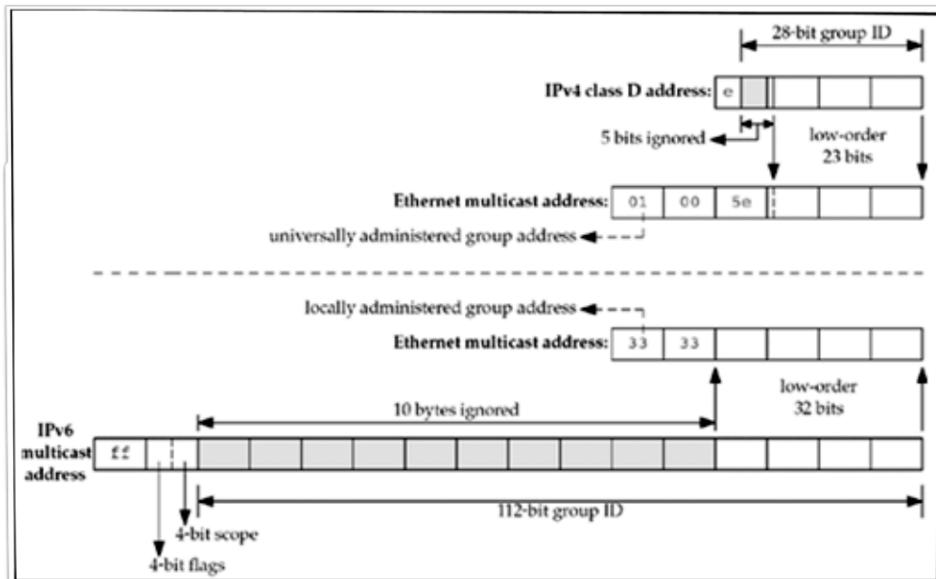
NOTA: in multicast sei sia server che ricevitore, cioe tu quando ti iscrivi: ricevi da tutti e puoi mandare a tutti.

- Among reserved IPv4 multicast addresses the following can be noted:
 - group **224.0.0.1** identifies **all the hosts** in a subnet
 - group **224.0.0.2** identifies **all the routers** in a subnet
 - and, more generally, the range from **224.0.0.0** to **224.0.0.255** is said **link-local** since it is reserved to protocols for management or discovery of *network topology*

● this traffic is never forwarded by a router

L'indirizzo mac di destinazione multicast che appare nei segmenti, deriva dalla seguente regola:

- L'indirizzo IP multicast viene convertito in indirizzo Ethernet multicast:



Alcune opzioni che possono essere utili per il mittente:

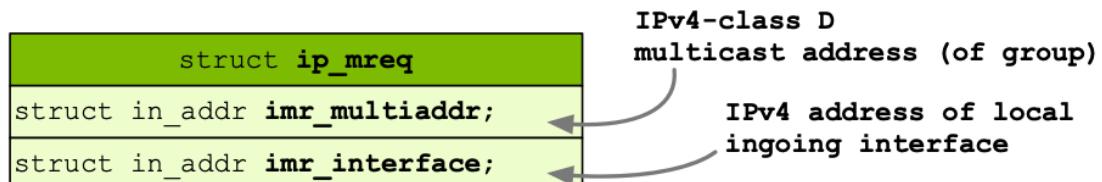
IP_MULTICAST_IF
IP_MULTICAST_TTL
IP_MULTICAST_LOOP

Relative rispettivamente all'interfaccia su cui mandare i pacchetti, il time to leave, e il local loopback (di default attivato, chi manda i pacchetti li riceve anche).

A differenza del broadcasting, in cui è il sender ad abilitare la trasmissione broadcast agendo sul socket UDP, nel caso del multicasting è il receiver che deve effettuare le operazioni necessarie ad abilitare la ricezione sul socket.

Le opzioni dei socket usate sono **IP_ADD_MEMBERSHIP**, e **IP_DROP_MEMBERSHIP**.

Per effettuare il join o il leave ad un gruppo, il receiver deve conoscere l'indirizzo del gruppo (ed eventualmente l'interfaccia) su cui ricevere il flusso multicast. Si utilizza allora una struttura dati specifica:



Vediamo ora la chiamata **select**, che vediamo utilizzata come esempio nel testo di beej a pag 48 (c'è anche il link che porta al download)

La chiamata select prende tre rastrelliere di variabili di tipo fd_set, una per lettura, una per scrittura e una per exception

```
#include <sys/select.h>

int select(int nfds, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict exceptfds,
           struct timeval *restrict timeout);

void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

Dove la FD_ZERO svuota la rastrelliera, FD_SET setta a 1 il bit corrispondente al file descriptor)

Si inserisce il numero massimo di file descriptor in modo da velocizzare la “ricerca” per scoprire su quale file descriptor è stato scritto/letto.

La select è bloccante, si esce solo se in uno dei file descriptor c'è qualcosa da leggere (per esempio). E' possibile aggiungere una struttura temporale che dice quanto aspettare, ma se non si mette, è semplicemente bloccante.

NOTA: per avere un client tcp si può semplicemente usare il comando telnet. Tutti i client scrivono da telnet al server e tutti i client ricevono ciò che il server scrive.

È importante capire sono necessarie due variabili di tipo fd_set chiamate master e read_fds (nel caso si voglia vedere se c'è da leggere). Quando si esce dalla select, il valore di read_fds è cambiato per riflettere il FD da cui si è letto, e si può controllare ciò con FD_ISSET(); quindi, bisogna poi reimporare la rastrelliera del master (se necessario leggere sempre da tutti i file descriptor)

Quando si esce dalla select, bisogna allora controllare tutti i fd fino al massimo previsto prima, e vedere se è nel set iniziale, dove sono stati azzerati tutti tranne quello che ha ricevuto.

Se in particolare è il file descriptor del server, significa che c'è stata una connessione, e bisogna aggiungere il fd a quelli del set, in modo da ascoltare anche da quello.

Vediamo ora come funziona la **gestione del terminale**.

Il terminale in genere funziona in modalità canonica, si scrive qualcosa, si fa invio e lui sa cosa fare. Per esempio la read di un terminale legge in modalità canonica quando si da invio, non specificando quanto bisogna leggere.

C'è un comando stty che dà la configurazione del terminale. Stty -a da tutte le configurazioni. In questa schermata, quando c'è il - significa che la caratteristica non c'è. Per esempio, se facessi stty -echo smetterei di vedere quello che scrivo (questo si fa per esempio negli script quando bisogna inserire una password, per non mostrarla a schermo mentre si digita).

C'è un flag chiamato icanon.

Vediamo dallo stevens come si gestisce questa cosa.

Ci sono due chiamate set e get: con è salvata tutta la configurazione del terminale che vediamo con stty in una variabile, con set invece si passa la struttura con i valori che vogliamo settare (spesso si fa prima una e poi l'altra, aggiungendo alla variabile le impostazioni desiderate).

Le chiamate fondamentali è tcgetattr che prende gli attributi.

La struttura è di tipo termios

VEDI 18.4

Si possono aggiungere delle opzioni in tcsetattr

C'è un piccolo esempio di questa cosa che sta su moodlis si chiama t_raw4 (NOTA USA APUE)

Innanzitutto, ci si cautela dai segnali sigint sigquit e sigtermi con il gestore sigcatch che semplicemente fa una chiamata ttyreset che mette a posto il terminale

Tty_raw cambia la natura del terminale. Come la cambia? Andando nella libreria a cercare la chiamata tty_raw

È nel file tty_modes.c

Questa è la chiamata fondamentale

Prima cosa legge la struttura buf, e si salva lo stato del terminal in buf

Poi si prende quest'astruttura e si lavora su alcuni suoi campi

I local flag sono importanti perché qui stiamo a riga 80 e qui cambia i bit di echo, icanon, iexten e isig

Si fa and bit a bit con la negazione che si passa. Si cambia come sono settati quei bit.

Poi ci sarà bisogno della set

Poi c'è un check che vede se una delle cose che voleva non è successa, rimette le cose a posto

Pace

Poi nel nostro codice, si fa la lettura di un carattere alla volta che finisce dentro c e si legge dallo standard input

Poi si vede il codice scritto dentro questo carattere

A seconda di quello che questo carattere fa si fanno delle cose

Qui si registrano delle coordinate, che si usano per spostare il cursore

Ci sono dei codici che iniziano con \033[numero che fa cambiare cose del terminale

Facendo printf di questi codici fa succedere cose nel terminal

Dopo che si posiziona il cursore nella posizione delle coordinate specificate sopra, si stampa un ragnetto

Usando h k m u quindi si muove

NOTA: SULLA SELECT C' UN PDF CHE SPIEFA COME FUNZIONA

NOTA: CI SONO ANCHE SLIDE SU MULTICAST E BROADCAST