Riassunto da: Enrico Mensa

Sistemi Operativi

Laboratorio C

Il linguaggio C

Il linguaggio C è un linguaggio compilato, imperativo e sequenziale. È piuttosto più "potente" di java in quanto ci permette di lavorare direttamente sulla memoria, ottimizzando così il codice.

La compilazione

Comando	Parametri	Funzionalità
gcc	*nomeFile.c*	crea un file compilato a.out (nome standard).
	nomeFile.c -o *nomeEseguibile*	crea un file compilato "nomeEseguibile" (senza estensione su unix, .exe su windows).

I format

La stampa di variabili nel linguaggio C potrebbe non essere così banale come in java. A seconda di quello che vogliamo stampare, infatti, dovremmo usare "format" diversi. Abbiamo la funzione printf(), che si presenta in questa forma:

```
printf("testo %format1, testo2 %format2", var1, var);
```

A var1 verrà applicato il format1, mentre a var2 verrà applicato il format2. Vediamo quali format ci sono:

Format	Significato	Funzionalità
%d	*digit*	stampa il valore decimale della variabile
%p	*pointer*	stampa il valore in formato indirizzo della variabile
%x	*hexadecimal*	stampa il valore esadecimale della variabile
%o	*octal*	stampa il valore ottale della variabile
%u	*unsigned*	stampa il valore decimale senza segno della variabile
%s	*string*	stampa una stringa
%c	*char*	stampa un char
%f	*float*	stampa un numero reale con la virgola

Questi non sono gli unici, ve ne sono altri ancora.

Struttura tipo di un programma c

Un programma "tipo" scritto in C ha questa forma:

```
#include <stdio.h>
#include <libreria.h>
#define PIGRECO 3.14

int metodo1(int, int *);
void metodo2(char[]);

main(){
      ...
}

metodo1(int a, int *b){
      ...
}
metodo2(char c[]){
      ...
}
```

All'inizio del codice si importano le librerie necessarie, quindi si procede con l'eventuale dichiarazione di costanti (define). Dopo si mettono le intestazioni dei metodi che si intende utilizzare (affinché venga riservato il giusto spazio a quei metodi). Viene quindi scritto il main, e poi il corpo dei metodi precedentemente indicati.

Tipi di dato in C

In C troviamo le seguenti tipologie di dati:
- Vettori (come in java)
- Record (struct)
- Union
- Enum
- Definizioni tramite typedef

Esaminiamoli!

Boolean
Il tipo boolean non esiste: il numero 0 equivale a falso, qualsiasi altra cosa equivale a vero.

I vettori
I vettori sono gestiti in maniera del tutto simile a Java. La dichiarazione di un array si fa così:

```
tipo nomeVettore[lung];
```

Chiaramente è possibile accedere ai vari dati contenuti nell'array tramite la notazione `nomeVettore[n]`.

Il tipo String
Il tipo String, che in Java era "auto"-implementato (anche se era già di per sé un oggetto), in C scompare. Una stringa non è null'altro che un array di char, anche se per convenzione alla fine dei char si inserisce il carattere "\0" (backslash zero): pertanto l'assenza di questo carattere significa che quell'array di char non è una stringa.
Esiste una libreria, (string.h) che offre alcuni metodi per il trattamento delle stringhe, fra cui:
`[supponiamo di aver dichiarato s1 ed s2 come array di char]`

```
strcpy(s1, "stringaVuota"); //copio "stringaVuota" in s1
stren(s1);        //restituisce la lunghezza di s1
strcat(s1, s2);   //copia s2 in fondo a s1
strcmp(s1, s2);   //confronta lessicograficamente s1 con s2 restituendo un
                  numero <, = o > a 0 a seconda che s1 sia minore, uguale o
                  maggiore di s2.
```

Pertanto le stringhe si usano con questi metodi, e non tramite "normali" assegnazioni. Si ricorda che "" è una stringa vuota che contiene solo il carattere \0.

I record
Un tipo di struttura totalmente nuova è il record. Non avendo la possibilità di creare oggetti come accadeva invece in Java, il linguaggio offre l'opportunità di creare un "agglomerato di variabili di tipo diverso", detti strutture. Le variabili delle strutture sarebbero l'equivalente dei campi precedentemente visti in Java. Una struttura si dichiara in questa maniera:

```
struct nomeStr {
      int campo1;
      char campo2[20];

} var1;
```

D'ora in poi la variabile "var1" sarà definita come struttura, cioè avrà due campi, un int e un'array di char. Usando la scrittura `var1.campo1=5;` potremo accedere al suo primo campo e modificarlo.

Volendo dichiarare altre variabili di tipo var1 potremo usare il codice `struct nomeStr var2;` e utilizzarla a partire da quel momento. Ogni variabile struttura avrà le stesse caratteristiche della prima, ma ovviamente sarà allocata in una parte differente della memoria.

Gli union

Un'altra delle novità di C rispetto a Java sono gli "union". Essi fungono come una sorta di "switcher" di campi. Un union viene definito in questo modo:

```
union {
      int campo1;
      float campo2;

} var1;
```

Ora var1 potrà essere usato come var1.campo1 oppure come var2.campo2 . Ovviamente può essere scelto un solo campo (conseguentemente un solo tipo), ecco spiegata la metafora dello "switcher di campi". La differenza rispetto alla structure è palese: soltanto uno dei campi potrà essere "abilitato", mentre gli altri non saranno nemmeno stanziati in memoria. Nel momento della dichiarazione di un union, ad esso viene assegnato una quantità di RAM pari alla dimensione del massimo campo presente nell'union.

I typedef

Ultima novità sono i typedef. Tramite i typedef possiamo definire sostanzialmente un nuovo tipo, come ad esempio:

```
typedef int intero;
typedef int mat_int[10][10];
```

D'ora in poi, durante il corso del programma, sarà possibile dichiarare nuove variabili integer come `intero x;` piuttosto che matrici come `mat_int y;`
Ma l'applicazione forse più efficace è quella di definire un typedef su una struttura, con un codice del tipo:

```
typedef struct {
      int campo1;
      char campo2[20];

} mioTipo;
```

Tramite questo codice, sarà possibile dichiarare variabili di tipo mioTipo che implementeranno direttamente la struttura sopracitata. Sarà quindi legale scrivere ad esempio:

```
mioTipo x;
x.campo1=5;
```

I puntatori

I puntatori sono un tipo dato, una variabile che assume come valore l'indirizzo in memoria di un'altra variabile. Si possono creare puntatori di qualsiasi variabile, purché durante la dichiarazione del puntatore, il tipo sia corrispondente a quello della variabile puntata.
Per poter utilizzare questa funzionalità è necessario comprendere appieno il significato di questi due operatori:
- **&** *(unario)* - fornisce l'indirizzo di una variabile
- ***** *(indiretto)* - fornisce il contenuto della variabile puntata dal puntatore

Vediamo subito una chiara applicazione:

```
int x=5;          //ora la variabile x ha un suo spazio dedicato in memoria, ad un certo indirizzo.
int *p;           //definiamo la variabile *p (puntatore).
int p = &x;       //stiamo assegnando a una variabile puntatore l'indirizzo della cella
                  che contiene il valore di x.

printf("%d", *p);     //stampa 5 (*p significa "il valore della cella puntata da p", ed è 5).
printf("%d", p);      //stampa sottoforma di digit dell'indirizzo della cella x (cioè il contenuto
                      di p). No sense.

printf("%p", *p);     //stampa sottoforma di indirizzo il valore del contenuto di *p (cioè x). (no
                      sense)
printf("%p", p);      //stampa sottoforma di indirizzo il contenuto della p, che essendo un
                      puntatore contiene effettivamente l'indirizzo della cella x. Quindi stampa
                      l'indirizzo della cella x.
```

È necessario porre l'attenzione su due fattori differenti. %d, %p sono "traduzioni" di quello che stiamo scrivendo. mentre *p, x, ecc. hanno contenuti differenti, quindi è importante non fare confusione. Se noi forniamo una variabile puntatore (p), dovremmo usare tendenzialmente %p, mentre per stampare il contenuto di una variabile useremo %d fornendo la forma *p.

Si noti che il tipo di p è lo stesso della variabile x a cui punta.

I puntatori con gli array
Bisogna però fare attenzione agli array, i quali vengono già dichiarati come puntatori. Vediamo questo codice:

```
int vettore[10];
int *p1,*p2;

p1=vettore;
p2=&vettore[5];
```

La variabile vettore è di per se un puntatore, pertanto è possibile usarla "direttamente" per essere copiata nella variabile p1. Ma invece, la cella vettore[5] è una cella "normale", quindi è necessario l'operatore "indirizzo di" (&) per averne l'indirizzo e metterlo nel puntatore p2.

L'aritmetica dei puntatori
Una delle più grandi potenzialità di poter trattare i puntatori come variabili a se stanti, è quella di poterli trattare come veri e propri indici.

Come detto sopra, assegnando una variabile (vett) che è un vettore ad un puntatore p, stiamo implicitamente assegnando a p l'indirizzo della cella vett[0] dell'array. Cioè in sostanza:

```
int vett[10];
int *p;
p=vett;
```

Ora p contiene l'indirizzo della cella vett[0] (quindi *p è il valore della cella a[0]).

Questo apre le porte ad una intera "aritmetica" dei puntatori, infatti se le righe riportate sopra assegnano l'indirizzo di a[0] a p, allora `p=(vett+1);` non potrà che assegnare a *p il valore di a[1] e così via. In un ciclo potremmo addirittura utilizzare il comando p++ per scorrere (letteralmente) un array (facendo molta attenzione a non sforare).

Allocazione dinamica della memoria

Malloc

La funzione malloc(n. di bytes da allocare) restituisce un puntatore di tipo void (void *) ad una zona della memoria contigua grande n.Bytes.
Per farne un utilizzo pratico dovremo eseguire un cast del puntatore, per poterlo assegnare alla variabile del tipo desiderato. Avremo pertanto:

```
int *vett;
vett = (int *) malloc(nByte);
```

Volendo dichiarare un array di 10 int, dovremmo scrivere 4*10 (ogni int occupa 4 byte). Volendo però utilizzare la malloc su strutture o elementi più complicati, può risultare complesso (quindi erroneo) il calcolo manuale dei byte. Si usa pertanto la funzione **sizeof()** che restituisce le dimensioni del tipo passato come parametro. Avremo così:

```
int *vett;
vett = (int *) malloc(sizeof(int)*10); //array di int lungo 10
```

Input/Output (utilizzando stdin e stdout)

Input

Esaminiamo con più attenzione i vari metodi di scrittura

scanf ("%format", &var)
Il metodo scanf legge da tastiera. Tale input viene salvato nella variabile *var* (della quale dobbiamo fornire l'indirizzo) con il formato assegnato tramite il format esplicitato tra virgolette.
Il format, se ha un asterisco frapposto tra % e la lettera (%*s) significa ignora. Vediamo degli esempi:

- `scanf("%d", &x); // legge un intero e lo mette in x`
- `scanf("%d %d", &x, &a); // legge un intero, poi ignora gli spazi, poi legge il secondo intero.`
- `scanf("%d %*s %d", &x, &a); //legge un intero, poi ignora gli spazi, poi ignora una stringa, poi ignora gli spazi, poi legge il secondo intero.`

getchar (void)
Il metodo getchar legge un char da tastiera e lo restituisce (come int, poiché in C i char non sono altro che interi, cioè il corrispondente in codice ASCII del carattere. Ad esempio, 'a' è salvato come 97). Il valore numerico del char è sempre raggiungibile utilizzando i singoli apici. E' importante ricordare comunque che il contenuto della variabile è un intero. EOF in caso di errore.

- `char carattere = getchar(); // legge un char e lo mette in carattere`

gets(char *s)
Il metodo gets legge una stringa da riga di comando, e la assegna al buffer (array di char) s. Attenzione alle dimensioni! s deve essere sufficentemente capiente!

- `gets(parola); // con parola dichiarato come char * parola.`

Output

printf ("%format", var)
Il metodo printf stampa a video le variabili var con il formato esplicitato.

- `printf("Intero: %d", x); // stampa "Intero: valoreDiX"`

putchar (int c)
Intrerpreta l'intero c come carattere e lo stampa a video.

- `putc('a'); // stampa "a".`
- `putc(97); // stampa "a".`

puts (char * s)
Stampa la stringa s (array di char).

- `putc(s); // stampa la stringa s, dichiarata come char * s[N];`

Input/Output (utilizzando FILE *)

Input

I metodi funzionano allo stesso modo dei precedenti, ma hanno come fonte un file anziché la tastiera.

fscanf (FILE * f, "%format", &var) //restituisce EOF quando il file è finito

fgetc(FILE * f) //restituisce il successivo carattere del flusso di stream (f), EOF se errore

fgets(FILE * f, char *s)

Output

fprintf (FILE *f, "%format", var)

fputc (int c, FILE * f) //scrive il carattere c sul flusso di stream (f), EOF se errore

fputs (FILE * f,char * s)

La gestione del FILE

fopen ("path", "mode")
La funzione fopen ci restituisce il puntatore al primo byte di un file, aperto in una modalità tra: a (append), w (write), r (read).

- `FILE * myfile = fopen("doc.txt", "w");`

fseek(FILE * f, long offset, int origin)
Tale funzione imposta la posizione corrente per il file f, le successive operazioni di lettura e scrittura si piazzeranno da quel punto.
Abbiamo tre variabili: SEEK_SET (inizio file), SEEK_CUR (posizione corrente) e SEEK_END (fine del file).
Origin è il punto iniziale da cui calcolare l'offset (espresso in byte).

- `fseek(myfile, sizeof(char), SEEK_CUR); //ci riporta indietro di un carattere rispetto a dove eravamo prima`

La compilazione

Il makefile

Con un makefile si possono definire delle regole di compilazione multiple da eseguire, così da non dover dare ogni comando da terminale tutte le volte. Inoltre il makefile verificherà quale dei file coinvolti nella compilazione è stato modificato rispetto all'ultima volta, e quindi compilerà solo quelli.

Il makefile ha una struttura di questo genere:

```
fileObbiettivo : dipendenza1, dipendenza2, dipendenza3, ...
      comandoDiCompilazione
```

Il makefile non fa altro che controllare se fileObbiettivo esiste. Se non esiste lo crea eseguendo comandoDiCompilazione. Se invece era già esistente, controlla se dipendenza1, dipendenza2 o dipendenza3 sono stati modificati. Se almeno uno di loro è stato modificato allora esegue il comando di compilazione.

Esempio

```
cameriere: cameriere.c ristorante.h
      gcc cameriere.c –o cameriere
```

Come si nota, il comando di compilazione ricrea il file cameriere, che è anche il fileObbiettivo. Questo è un modo sensato di creare il makefile.

Il makefile viene richiamato nella seguente maniera:

```
$terminale : make fileObbiettivo1 fileObbiettivo2
```

indicando quindi quali fileObbiettivo si vuole modificare. Può essere più comodo creare una regola fittizia

all: fileObbiettivo1 fileObbiettivo2 all'interno del makefile, così da poterlo richiamare come:

```
$terminale : make all
```

e compilare automaticamente tutti i fileObbiettivo.

Nota:
Se eseguissimo `$terminale: make`, verrebbe compilata solo la prima regola del makefile.
Pertanto conviene mettere la regola fittizia "all" come prima, così da poter compilare tutto il file usando semplicemente `$terminale: make`.

Sistemi Operativi
Laboratorio Unix

Indice degli argomenti

La shell

La shell è un prompt dei comandi tramite il quale possiamo "ordinare" al sistema operativo cosa fare. È un sistema alternativo agli eventi generabili tramite l'interfaccia grafica.

I comandi

Comando	Parametri	Funzionalità
man	*comando*	mostra il manuale per comando
cd	*dir*	entra dentro a dir (con dir una sottocartella della working directory)
	..	va alla dir superiore (utilizzabile in cascata ../../..)
ps		**process status:** lista dei processi attivi lanciati dall'utente che sono controllati dal terminale.
	aux	mostra tutti i processi (PID è il process identifier).
pwd		**print working directory:** stampa l'absolute path della cartella nella quale ci si trova.
ls		**list:** mostra i nomi di tutte i file/directory contenuti nella cartella.
	-l	si ottengono maggiori informazioni, come ad esempio i permessi attivi su ogni singolo file/directory.
	-d	viene eseguito il comando ls ma sulla working directory, non sul suo contenuto.
mkdir	*nomeCartella*	crea una cartella nomeCartella
rmdir	*nomeCartella*	elimina una cartella nomeCartella
history		mostra tutti i comandi eseguiti nella shell
!n		va al comando n, mostrato nella history
!!		esegue il comando precedente
chmod	*xxx dir*	**Change mod:** modifica i permessi di dir
chgrp	*newGrpName dir*	**Change group:** modifica il gruppo di dir
chown	*newOwnName dir*	**change owner:** modifica il proprietario di dir
who		restituisce quali user sono collegati alla macchina
whoami		restituisce il proprio nome utente
ssh	*username@ip*	permette (in seguito all'inserimento di una password) il login dell'utente "username" sulla macchina avente ip "ip".

I permessi

I permessi di un file o directory possono essere gestiti tramite il comando **chmod xxx dir.**
Prendiamo un output di un ls -l:

drwxr-xr-x 2 enkk staff 68 18 Set 05:34 Applications

Concentriamoci sulla prima parte, e dividiamola in triplette: d rwx, r-x, r-x.
A parte il primo carattere che è “d” se l’oggetto è una directory mentre è “-” se è un file, gli altri tre definiscono rispettivamente i permessi di lettura scrittura ed esecuzione nell’oggetto in questione.
Vi sono tre triplette poiché i permessi valgono rispettivamente per owner, group, others.

Per poter cambiare i permessi tramite il chmod, è possibile utilizzare questo semplice metodo: ogni tripletta è esprimibile come numero binario a tre bit. Ad esempio, se vogliamo assegnare un permesso di sola lettura all’owner, avremo 100 (rwx). Questo numero, in binario, vale 4. Pertanto la prima delle tre x del comando **chmod xxx dir** sarà 4. Le altre due x esprimono invece i permessi per group e others, sempre seguendo lo stesso ragionamento.

Ecco che quindi il comando **chmod 755 esempio** imposterà i permessi: rwxr-xr-x all’oggetto esempio.

Una volta effettuata l’autenticazione tramite il comando **sudo**, si diviene utenti **root** pertanto tutti i permessi vengono ignorati poiché il root è libero di fare qualsiasi cosa.

La fork()

L'operazione di fork permette al processo che sta eseguendo il programma, di generarne un'altro, che da quell'istante eseguirà lo stesso identico codice.
La fork restituisce anche un intero (sia al padre che al figlio), ma assume valori differenti.
Per il padre, la fork restituisce il PID del figlio, per il figlio, la fork restituisce 0.
Inoltre, la fork fa sì che il figlio di erediti tutte le variabili del padre (vengono sdoppiate!). Faremo uso di pid_t, un tipo speciale di interi (necessario per salvare un PID).

Sapendo queste due cose, possiamo già immaginare un codice del genere:

```
pid_t var;

if(var == 0)      //se variabile vale zero siamo in un figlio
 /* Codice del figlio */
else
 /* Codice del padre */
```

Conoscendo però il fatto che in C il valore zero equivale a false, possiamo modificare il codice precedente per renderlo ancora più intuitivo:

```
pid_t padre;

if(padre)    //padre != 0
 /* Codice del padre */
else
 /* Codice del figlio */
```

Creare figli con un for
Utilizzare i for e le fork in maniera combinata può essere piuttosto "pericoloso". Un codice del genere:

```
for (int i = 0; i<5; i++){
      fork();
}
```

genererà ben più dei 5 aspettati figli. Ogni figlio, infatti, eseguirà il codice del padre (il quale avrà ancora (5-i) giri da fare), che quindi produrrà una moltitudine di figli!
Per poter controllare "la nascita" dei figli, si potrebbe invece implementare un codice di questo genere:

```
pid_t padre;

for (int i = 0; i<5; i++){
      padre = fork();
      if(!padre) i=5;
}
```

Questa semplice modifica permetterà ai figli di non creare più altri figli a loro volta.

Altri metodi utili
Lavorando con le fork, potrebbe essere utile utilizzare il metodo **getppid()** (restituisce il PID del padre), ed il metodo **getpid()** (restituisce il proprio pid)

Figli adottivi
Se, durante l'esecuzione, il padre termina prima del figlio, allora il processo figlio viene adottato da init, il processo principale (PID=1). Pertanto il figlio avrà getppid() = 1.

La execl
È chiaro che utilizzare degli if per discernere enormi fette di codice renda il tutto piuttosto illeggibile. Viene pertanto utilizzata la funzione execl("nomeFile", argv[0], argv[1], ... , NULL), la quale non fa altro che eseguire un certo programma nomeFile, passandogli come parametri quel che preferiamo.
Per convenzione, il primo parametro (argv[0]) è sempre il nome del file.
Tutto il codice del figlio (supponendo di volerlo distinguere da quello del padre) può essere inserito tranquillamente nel file nomeFile.
Attenzione! Questo procedimento crea un nuovo stack, etc per il processo chiamato!

I flussi std

In unix vi sono degli operatori particolari utilizzabili nel terminale che ci permettono di modificare i vari standard di input/output. Vediamo prima di tutto quali sono e quali sono gli operatori che ne permettono il ridirezionamento:

Tipo	Nome	Operatore
Input	stdin	<
Output	stdout	> oppure >>
Errore	stderr	2> oppure 2>>

L'utilizzo di una sola parentesi acuta indica la sovrascrizione del file, mentre due parentesi significa accodare il risultato a ciò che è già presente. In ogni caso, se il file non dovesse esistere, verrebbe creato.

Esempio
`ps > result.txt` --> Scrive (sovrascrivendo) il risultato del comando ps dentro al file result.txt

Pipe
Volendo, è possibile utilizzare l'operatore "`|`" per generare un "collegamento" tra gli input e gli output di diversi programmi. Ad esempio `ls | wc` --> prende l'output di ls e lo da come input a wc (il quale conta la quantità di righe).

Background e foreground

Come sappiamo quando un processo viene lanciato da terminale, tale processo si "impadronisce" del terminale. Questa situazione viene detta **esecuzione in foreground**.

Per eseguire un processo in **background** (cioè ignora l'input da tastiera, e non può ricevere dati in ingresso) dobbiamo lanciarlo usando il simbolo &. Pertanto, volendo lanciare firefox in background digiteremo `firefox &`.

Per far andare un processo già attivo in background è sufficiente digitare `Ctrl+Z` e poi eseguire `bg`.
Per far tornare un processo in background in foreground è sufficiente eseguire `fg`. Se avessimo più processi in background, allora dobbiamo eseguire `jobs`, quindi digitare `fg %n` dove n è il numero associato al processo di interesse.

La gestione degli errori

Quando una system call ha esito positivo, restituisce -1. Sarebbe utile poter interpretare quell'errore per un più facile debugging.
In C esiste la libreria **errno.h**, che ci fornisce la variabile globale **errno** ed anche la funzione **perror**.
Vediamo come usarli!

```
#include <errno.h>

main () {
int i;
printf("\nValore iniziale errno = %d\n",errno);

i = execl("questoFileNonEsiste"); //errore se non trova il file

printf("\nRisultato exec = %D, errno = %d\n", i, errno);
perror("errore nella exec ");

}
```

La stampa dell'esecuzione sarà questa, in caso di errore:

```
valore iniziale di errno = 0
risultato exec = -1, errno = 2
errore nella exec: No such file or directory
```

Tramite il comando man errno e man perror è possibile capire cosa significano i codici di errore. Il parametro di perror viene messo in testa al messaggio di errore stesso (no such file or directory, in questo caso).

System call della famiglia wait

Vi sono delle system call apposite per far sì che il padre possa "controllare" il figlio e ottenere informazioni importanti sul suo conto. Sarà necessario importare <sys/wait.h> ed <sys/types.h>
Per esempio, abbiamo:

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

wait(int *status)
Questo comando provoca l'attesa del chiamante fino alla morte di uno dei suoi figli.
Lo status può valere 0 oppure null se non si intende specificare nessun'altra particolarità, altrimenti può valere l'indirizzo di una variabile intera. In tal caso, quella variabile sarà ispezionabile tramite metodi appositi (vedi man) che daranno informazioni su come il figlio è terminato (*es: WIFSIGNALED(status), WIFEXISTED(status), WCOREDUMP(status)*).

waitpid(pid_t pid, int *status, int options)
La versione più generare di wait (int * status). La chiamata equivalente sarebbe waitpid(-1, &status, 0);
Il parametro pid può essere:

- minore di -1: attende un processo figlio il cui process group id sia uguale al valore assoluto del pid
- uguale a -1: attende un processo figlio qualsiasi
- uguale a 0: attende un processo figlio il cui process group id sia uguale a quello del processo chiamante
- maggiore di 0: attendi un figlio il cui PID è uguale a pid (un figlio specifico!)

options è invece un OR bit a bit che permettere di personalizzare le azioni da farsi in seguito alla terminazione del figlio (vedi man).
I processi sono definiti in gruppi per una più facile implementazione. Per il momento basti sapere che un processo figlio, eredita il gruppo del padre.

Processi zombie

Senza wait, i processi figli potrebbero terminare prima del padre, creando così i cosiddetti processi **zombie**. Questo fenomeno è molto grave per due motivi:

1) Il SO non rimuove la entry del processo dalle sue strutture (spazio occupato inutilmente)

2) Il processo padre potrebbe aver bisogno di informazioni prima della terminazione dei suoi figli

Questo codice produce figli zombie:

```
while(1){
      p = fork();
      if(!p) exit(0); //se figlio
      else sleep(2); //se padre
}
```

Il padre produce figli in continuazione, i quali muoiono istantaneamente (e intanto il padre attende).

Evitare la "nascita" degli zombie, è molto semplice. È sufficiente introdurre una wait per ogni processo figlio che è generato (il padre attenderà quindi la sua terminazione).

```
int figli = 0;

while (...) {
   p = fork();

   if (!p) {
      /* codice figlio */
   }
   else {
      figli++;
      /* codice padre */
   }

}

while (figli) {

   wait(0);

   figli--;

}
```

L'ultimo while metterà in wait (la quale verrà interrotta ad ogni terminazione del figlio) il padre.

Intern Process Comunication

Fanno parte dell'inter process comunication:
1) Code di messaggi
2) Semafori (sincronizzazione)
3) Aree di memoria condivisa
Possiamo controllare quali siano allocate sulla nostra macchina col comando da terminale IPCS ed eliminarle usando il comando IPCRM -TIPO ID.

Code di messaggi

Le code di messaggi, seppur allocate da processi, sopravvivono ad essi, cioè esistono anche dopo la morte del processo allocatore.
È pertanto importante disallocarle a fine esecuzione per evitare che la memoria venga intasata.

Le import da fare per la gestione delle code sono le seguenti:

```
#import <sys/types.h>
#import <sys/ipc.h>
#import <sys/msg.h>
```

Operazioni per la gestione delle code di messaggi
1) Allocazione di una coda
2) Accesso ad una coda già allocata
3) Send/Receive
4) Operazioni di controllo (operazioni generiche che ci forniscono meta-dati oltre all'opportunità di disallocare la coda).
Vediamole nel dettaglio.

Allocazione di una coda
Una coda viene allocata tramite il comando:

```
int q = msgget(key_t key, int flag);
```

Key --> è una variabile di tipo key_t (rinomina del tipo intero, come pid_t) che identifica in maniera univoca la coda all'interno del codice, è una sorta di nome della coda;
Flag --> è invece un intero che definisce i diritti di accesso alla coda (read/write) così come già accadeva per il chmod, quindi può assumere per esempio il valore 666 [4+2 per ogni 6] (lettura/scrittura per proprietario, gruppo proprietario e sconosciuti).
return --> ha un valore da interpretare: con -1 si ha un fallimento della syscall, altrimenti viene returnato un n>=0 che definisce l'id della coda all'interno del codice (la variabile q potrebbe essere ad esempio ereditata da processi figli, etc!)

Vi sono due modi differenti di utilizzare la system call `msgget`:

```
A)int q = msgget(IPC_PRIVATE, 0600);
B)int q = msgget(1122, flag);
```

La tecnica A viene utilizzata quando saranno solo i figli ad usare la coda, indi non si specifica una key ma si userà solo la q ereditata. Viene introdotto uno zero davanti alla flag poiché ha altri usi che saranno chiari più avanti.
La tecnica B è il vero e proprio uso di `msgget`, che identifica la coda come 1122 (vedremo nelle send e nelle receive l'utilizzo di questo id!).
Con questa dichiarazione la flag può essere specificata in due modi differenti:

```
B.1) IPC_CREAT|0600
B.2) 1122, 0
```

La dichiarazione B.1 ci porterà ad allocare una nuova coda di messaggi se non dovesse esistere la coda definita dall'id "1122"; se invece la coda è già definita allora viene ritornato il q di quella coda.
La dichiarazione B.2 invece se la coda esiste ci ritorna il suo q, se invece non dovesse esistere ritorna -1.

Invio e ricezione di messaggi sulla coda
Prima di parlare dei due metodi dobbiamo parlare di come sia fatto un messaggio da inviare nella coda.

Come è fatto un messaggio?
```
typedef struct {
      long type;
      * costituzione del messaggio * //ATTENZIONE! Statico!
} m_type;
```

Ogni messaggio ha un tipo (definito a piacere dall'utente, semplicemente discrimina i diversi tipi di messaggio che potremmo desiderare dentro alla coda).

Per effettuare queste due operazioni si utilizzano i metodi:
```
msgsnd(int q, void *m, size_t s, int flag);
msgrvc(int q, void *m, size_t s, long type, int flag);
```

Abbiamo nella **`msgsnd`**:
int q --> l'id della coda
void * m --> il puntatore al messaggio che desideriamo inviare (un puntatore a struttura, come si è visto sopra)
size_t s --> la dimensione del messaggio
ATTENZIONE! Qui si parla del messaggio senza considerare il parametro di convenzione long type. Quindi conviene utilizzare una seconda struttura
```
typedef struct {
      long type;
      m_type mioMessaggio;
}
```
e poi assegnare come size la `sizeof(m_type)`, per evitare equivoci ed errori, dato che la copia del messaggio nella coda viene effettuata byte per byte.
int flag --> definisce il modo con cui il messaggio deve essere mandato. Con flag = 0 si ha un invio sospensivo (il sender attende che un receiver prenda il messaggio) con IPC_NOWAIT invece non si attende.

Abbiamo nella **`msgrvc`**:
int q --> l'id della coda
void * m --> il puntatore al quale intendiamo assegnare il messaggio letto
size_t s --> la dimensione del messaggio da leggere *(vale la stessa questione sulla size di prima!)*
long type --> il valore del tipo del messaggio che vogliamo leggere: la funzione msgrvc prenderà il primo messaggio di quel tipo nella coda
int flag --> definisce il modo con cui il messaggio deve essere letto. Con flag = 0 si ha una ricezione sospensiva (il receiver attende che un sender mandi il messaggio) con IPC_NOWAIT invece non si attende.

Operazioni di controllo
```
msgctl(int q_id, int comando, eventuale_parametro);
```
A seconda del comando dovremo introdurre oppure no un ulteriore parametro (altrimenti verrà ignorato). Il comando più interessante è IPC_RMID che disalloca la coda (non richiede altri parametri).

I semafori

I semafori sono sistemi di sincronizzazione tra i processi.

Le import da fare per la gestione dei semafori sono le seguenti:

```
#import <sys/types.h>
#import <sys/ipc.h>
#import <sys/sem.h>
```

Le operazioni effettuabili sui semafori sono raggruppabili in tre categorie:

1) Allocazione di semafori
2) Operazioni sui semafori
3) Operazioni di controllo (assegnazione, lettura, rimozione di semafori)

L'allocazione di semafori

Non è possibile l'allocazione di una singola "entità" semaforo, bensì solo di array di semafori. Pertanto, il comando di allocazione dovrà avere una variabile nsem che ci definisca la lunghezza dell'array di semafori. I semafori saranno raggiungibili tramite un loro indice i, come accadrebbe con un array qualsiasi. Il comando per l'allocazione è quindi:

```
int s = semget(key_t key, int nsem, int semflag);
```

key --> è una variabile di tipo key_t che identifica in maniera univoca l'array di semafori all'interno del codice, è una sorta di nome dell'array;

nsem --> numero di semafori nell'array

semflag --> è invece un intero che definisce i diritti di accesso alla coda (read/write) così come già accadeva per il chmod, quindi può assumere per esempio il valore 666 [4+2 per ogni 6] (lettura/scrittura per proprietario, gruppo proprietario e sconosciuti). Valgono le regole delle code dei messaggi (per quanto concerne l'allocazione usando IPC_CREAT, ecc.).

return --> ha un valore da interpretare: con -1 si ha un fallimento della syscall, altrimenti viene returnato un n>=0 che definisce l'id dell'array di semafori all'interno del codice.

Operazioni di controllo sui semafori (assegnazione, lettura, rimozione di semafori)

È possibile assegnare un valore, leggere il valore oppure rimuovere un semaforo. Tutte queste operazioni si effettuano tramite il comando:

```
semctl(int sem_id, int semnum, int comando, eventuale_parametro);
```

Come per le code di messaggi, ad un comando differente corrispondono azioni e parametri eventuali differenti, con la differenza che qui abbiamo un parametro semnum indispensabile per identificare un certo semaforo nell'array sem_id. Alcuni dei comandi sono:

1) SETVAL - imposta ad un valore (espresso come intero nel parametro eventuale) il semaforo.
2) SETALL - imposta ad un array di valori (passato come ushort * nel parametro eventuale) tutti i valori dell'array di semafori.
3) GETVAL- restituisce il valore di un semaforo (bisognerà assegnare il metodo a qualche int quindi!).
4) GETALL - imposta l'array di ushort passato come parametro eventuale con i valori del semaforo.
5) IPC_RMID - disalloca l'intero array di semafori.

Il parametro eventuale può quindi assumere ben tre tipi differenti!

Operazioni sui semafori
I semafori vengono utilizzati tramite le famose P() e V().

Abbiamo un system call generale per effettuare operazioni.

```
semop(int sem_id, struct sembuf * sops, size_t nsops);
```
sem_id --> l'id dell'array di semafori
sops --> è un array di operazioni (si veda dopo)
nops --> lunghezza dell'array di operazioni (cioè la quantità di operazioni, in sostanza)

Una singola operazione è definita in questo modo:

```
struct sembuf {
      ushort sem_num;
      short sem_op;
      short sem_flg;
}
```

sem_num --> è l'indice del semaforo sul quale applicare una operazione.
sem_op --> definisce l'operazione da effettuarsi sul semaforo. Può valere:
sem_op > 0) sarebbe la V, incrementa il valore del semaforo di sem_op
sem_op < 0) sarebbe la P, controlla se il valore del semaforo è maggiore o uguale al valore assoluto di sem_op, se sì decrementa il semaforo e prosegue, se no, il processo è in sospensione e il valore del semaforo non viene toccato.
sem_op = 0) se il semaforo ha valore maggiore di zero il processo è in sospensione, nel momento in cui il semaforo assume valore zero, il processo riparte. Ottimo per far partire diversi processi insieme!

Consigli di implementazione

Durante l'utilizzo effettivo di queste system call può essere utile crearsi due piccole funzioni per rendere più leggibile il codice:

P

```
int P(int send_id, int n) {
  struct sembuf op;
  op.sem_num = n;
  op.sem_op = -1;
  op.sem_flg = 0;
  semop(sem_id, &op, 1);
}
```

V

```
int V(int send_id, int n) {
  struct sembuf op;
  op.sem_num = n;
  op.sem_op = 1;
  op.sem_flg = 0;
  semop(sem_id, &op, 1);
}
```

La memoria condivisa

La memoria condivisa funziona in maniera molto simile alle system call viste in precedenza.

Le import da fare per la gestione delle memorie condivise sono le seguenti:

```
#import <sys/types.h>
#import <sys/ipc.h>
#import <sys/shm.h>
```

Le operazioni effettuabili sulle memorie condivise sono raggruppabili in tre categorie:
1) Allocazione della memoria condivisa
2) Allacciamento/sganciamento da una memoria condivisa
3) Operazioni di controllo (rimozione della memoria condivisa)

L'allocazione della memoria condivisa
La memoria condivisa viene allocata tramite il comando:

```
int m_id = shmget(key_t key, size_t size, int shmflag);
```

key --> è una variabile di tipo key_t che identifica in maniera la memoria condivisa all'interno del codice, è una sorta di nome.
shmflag --> è invece un intero che definisce i diritti di accesso alla coda (read/write) così come già accadeva per il chmod, quindi può assumere per esempio il valore 666 [4+2 per ogni 6] (lettura/scrittura per proprietario, gruppo proprietario e sconosciuti). Valgono le regole delle code dei messaggi (per quanto concerne l'allocazione usando IPC_CREAT, ecc.).
size --> dobbiamo dire la quantità di spazio che verrà riservata per la memoria. Si potrebbe dare la sizeof() una struttura, ad esempio.
return --> ha un valore da interpretare: con -1 si ha un fallimento della syscall, altrimenti viene returnato un n>=0 che definisce l'id della memoria condivisa all'interno del codice.

Allacciamento/sganciamento da una memoria condivisa
Dato che noi vogliamo scrivere in una memoria condivisa, non ci basta il suo id, ma necessitiamo invece di un puntatore alla memoria stessa. Viene pertanto effettuata l'operazione di allacciamento:

```
T * sharedMemory = (T *) shmat(int m_id, const void * shmddr, int shmflag);
```

Nota: i parametri shmdrr e shmflag li useremo sempre rispettivamente con NULL e 0.

Questa funzione ritorna il puntatore alla memoria condivisa, ma void. È una sorta di malloc. Sarà quindi necessario effettuare un cast al tipo desiderato (T, nell'esempio sopra).

È altresì possibile staccarsi dalla memoria (non viene disallocata!) tramite il comando:

```
int shmdt(const void * variable);
```

Con l'esempio sopra, scriveremo: `shmdt(sharedMemory);`

Disallocazione di una memoria condivisa
Come al solito, si utilizza il comando:

```
shmctl(int m_id, int comando, eventuale_parametro);
```

con comando IPC_RMID.

I segnali

I segnali sono dei sistemi **asincroni** di comunicazione tra i processi, correlati direttamente ad un evento. Il processo assume un comportamento reattivo: alla ricezione di un segnale risponde con un'azione associata e poi prosegue eseguendo l'istruzione successiva a quella interrotta (*caso particolare: il processo era in attesa, ad esempio da una msgrcv. Vedremo dopo come risolvere questo problema!)*.

La import da fare per la gestione dei segnali è la seguente:
`#import <signal.h>`

Generazione e ricezione dei segnali
Un segnale viene generato quando un evento occorre. Una volta generato, verrà consegnato a uno più processi i quali potranno:
- eseguire un'**azione di default**
- eseguire un'**azione diversa**, se specificato
- **ignorarlo**

Il ricevente è in grado di ottenere informazioni sul segnale ricevuto tramite il comando `siginfo`.

Le tipologie di segnali
I segnali sono in numero finito e predefinito. Ad ognuno di essi è assegnato un valore numerico utilizzabile al posto del nome.
Le azioni di default sono: **exit** (il processo termina), **core** (il processo termina salvandosi una immagine del PCB eventualment consultabile), **stop** (il processo è sospeso), **ignore** (il segnale è ignorato).

Nome	Valore	Default	Evento
SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal instruction
SIGTRAP	5	Core	Trace or breakpnt trap
SIGABRT	6	Core	abort
SIGEMT	7	Core	Emulation trap
SIGFPE	8	Core	Arithmetic exception
SIGKILL	**9**	**Exit**	**Killed**
SIGBUS	10	Core	Bus error
SIGSEGV	11	Core	Segmentation fault
SIGSYS	12	Core	Bad system call
SIGPIPE	13	Exit	Broken pipe
SIGALRM	**14**	**Exit**	**Alarm clock**
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User signal 1
SIGUSR2	17	Exit	User signal 2
SIGCHLD	18	Ignore	Child status changed
SIGPWR	19	Ignore	Power fail/restart
SIGWINCH	20	Ignore	Window size changed

Nome	Valore	Default	Evento
SIGURG	21	Ignore	Urgnt socket condition
SIGPOLL	22	Exit	Pollable event
SIGSTOP	**23**	**Stop**	**Stopped (signal)**
SIGTSTP	24	Stop	Stopped (user)
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input)
SIGTTOU	27	Stop	Stopped (tty output)
SIGVTALRM	28	Exit	Virtual timer expired
SIGPROF	29	Exit	Profiling timer expired
SIGXCPU	30	Core	CPU time limit exc.
SIGXFSZ	31	Core	File size limit exceeded
SIGWAITING	32	Ignore	Concurrency signal
SIGLWP	33	Ignore	Inter-LWP signal
SIGFREEZE	34	Ignore	Check point freeze
SIGTHAW	35	Ignore	Check point thaw
SIGCANCEL	36	Ignore	Cancellation signal
SIGTMIN	*	Exit	1st real time signal
(SIGTMIN+1)	*	Exit	2nd real time signal
(SIGTMAX-1)	*	Exit	2nd-to-last real time sig
SIGTMAX	*	Exit	Last real time signal

Le trap

Un tipo di segnale è la trap. L'evento è causato da un processo, il quale gestisce lo stesso in maniera sincrona (*ad esempio segmentation fault, divisione per zero, etc*).

Esempio

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

La system call crea un timer settandolo a "`seconds`". Questo verrà decrementato ogni secondo (conto alla rovescia). Quando il timer arriva a 0, viene lanciato un segnale **SIGALRM** al processo stesso.

Gli interrupt

Simili ma differenti sono gli interrupt, che sono eventi generati da agenti esterni, come ad esempio potrebbe essere il CTRL+C di un utente. Il processo gestisce un evento segnalato da un altro processo.

La costruzione di un segnale

Il tipo di dato `sigset_t` è usato per costruire insiemi di segnali. Per manipolare insiemi di segnali occorre usare funzioni predefinite, dove signum è il numero (o nome) di uno dei segnali sopra elencati:

```
int sigemptyset(sigset_t *set);
```
costruisce l'insieme vuoto di segnali.

```
int sigfillset(sigset_t *set);
```
costruisce l'insieme completo di tutti i segnali possibili.

```
int sigaddset(sigset_t *set, int signum);
```
aggiunge signum all'insieme (un segnale desiderato).

```
int sigdelset(sigset_t *set, int signum);
```
toglie signum all'insieme dei segnali.

```
int sigismember(const sigset_t *set, int signum);
```
Effettua un test per vedere se signum è parte dell'insieme.

L'invio di un segnale
Un segnale viene inviato tramite le sys call: **kill**, **alarm** oppure tramite comandi come **CTRL-C**, **CRTL-Z**.
La system call kill viene così usata:

```
int kill(pid_t pid, int signum);
```
pid --> pid del processo da killare (se maggiore di zero, altrimenti se = 0 si killano tutti i processi dello stesso gruppo).
sig --> identificatore del segnale (come sopra).

Come catturare un segnale
Un segnale viene trattato come espresso dal valore di default (indicato nella tabella sopra). Volendo però sovrascrivere l'handler, possiamo usare la system call **signal** che non fa altro che modificare l'handler del segnale.

```
sighandler_t signal(int signum, sighandler_t handler);
```
handler: funzione che specifica cosa fare alla prossima occorrenza del segnale in questione.
Può anche valere **`SIG_IGN`** (ignora il segnale) oppure `SIG_DFL` (azione di default).
signum: il segnale per il quale il chiamante cambierà la gestione

ATTENZIONE! SIGKILL e SIGSTOP non possono essere catturati o ignorati!

La funzione handler deve avere un parametro intero (per rispettare così il prototipo) e deve essere void.
Alcuni sistemi operativi utilizzeranno la funzione handler solo per la prima ricezione del segnale. Dopodiché torneranno al default. Sarà quindi necessario a fondo metodo ri-assegnare l'handler.

Interruzioni fastidiose
Come detto sopra, certe interruzioni possono essere molto fastidiose, come ad esempio durante una ricezione sospensiva di un messaggio. A seconda del sistema operativo potremmo avere la sospensione interrotta oppure no.
Dobbiamo quindi sempre verificare il return delle system call di tipo sospensivo o di sincronizzazione (che saranno -1 in caso di errore).

Processi ed accesso ai file

I processi utilizzano quasi sempre files durante la loro esecuzione.

Nella cartella /usr/bin/ troviamo i comandi digitabili da shell.
Tra questi passwd, utilizzato per cambiare la password di sistema. È chiaro che si vorrebbe evitare che chiunque possa accedere al file, leggerlo o addirittura modificarlo. Ma allo stesso tempo l'utente deve poter cambiare la sua password con facilità. Come fare?

Vengono definiti dei permessi di accesso ai file. rw- --- --- sono i diritti assegnati all'eseguibile passwd, il quale è appunto utilizzabile, in sostanza, solo da root. Quando un utente (user123 per esempio) prova ad usare il comando passwd non ha però accesso ai file protetti! Dobbiamo assegnarvi, pertanto, dei diritti di root esclusivi e temporanei.

Questo viene fatto tramite quattro identificatori:

1) utente reale --> esecutore del processo (user123)
2) utente effettivo --> può cambiare. Dipende qual'è l'ultimo elemento della prima tripla dei permessi. Se si ha una **x** (rwx ad esempio) allora utente effettivo = utente reale. Altrimenti, se si ha una **s** (rws ad esempio), l'utente effettivo diviene il proprietario dell'eseguibile (root, nel caso di passwd!)
3) gruppo reale
4) gruppo effettivo

Nota: i gruppi seguono il ragionamento degli utenti.

Quando si tenta l'accesso ad un file, si controlla prima l'utente effettivo e poi quello reale (se ha senso farlo).

Se volessimo assegnare dei diritti del genere ad un file (per l'user), scriveremo:

```
chmod u+s nomeFile
```

mentre per il gruppo:

```
chmod g+s nomeFile
```

Queste operazioni vengono definite come suid (set user id) e sgid (set group id).