Sistemi Operativi

Argomenti

- `$#`: numero di argomenti passati
- `$?`: the exit status of the last operation
- `$*`: tutti gli argomenti passati `arg1 arg2 arg3`
- `$@`: come `$*` ma separa i caratteri quotati
- `$0`: nome del processo
- `$1`: primo argomento passato
- gli argomenti sono sempre considerati delle stringhe (da tenere d'occhio quando devo controllare dei valori NUMERICI negli IF ⇒ `invece di usare >=<! devi usare -gt -lt -eq ecc.`)

Processi

- ps → mostra i processi

per altro vedi processi in background

Operatori di reindirizzamento

```
#operatore <<<
read A B C <<< alfa beta gamma
```

```
echo 1 $A 2 $B 3 $C
> 1 alfa 2 3

#se dentro "" viene considerata come una sola parola
read A B C <<< "alfa beta gamma"
echo 1 $A 2 $B 3 $C
> 1 alfa 2 beta 3 gamma

#operatore <
grep ciao < file.txt #redirezione lo stdin su file

#operatore >
echo "ciao ciao ciao"> file.txt #redirezione lo stdout su file
```

Lettura file

Come leggere più file alla volta

```
exec {First}<first.txt
exec {Second}<second.txt
exec {Third}<third.txt

# -r e -u sono indispensabili per eseguire correttamente l'assegnazione
# alle variabili ; mentre && permette di far aprire il secondo file
# se e solo se il primo si apre bene altrimenti returna un errore

while read -r -u ${First} parolaInFirst \
   && read -r -u ${Second} parolaInSecond \
   && read -r -u ${Third} parolaInThird ;
   do :
    do
      # fai cose con le variabili che ti crei
done
```

Tail

```
tail -n {nRigheDaReturn} {FileDaLeggere}

# tail prende un file e parte a leggerlo dalla sua fine (ne legge una parte) ,
# quindi lo legge al contrario => il contrario di Head

#le rimostra ogni volta che file.txt viene modificato e non termina
tail -f file.txt
```

Head

```
head -n {nRigheDaReturn} {FileDaLeggere}
```

```
# legge le prime n righe del file , se n non è specificato legge le prime righe
```

Chiusura di un File Descriptor aperto in lettura

```
exec {fileDescriptor}>&-
```

wc ⇒ contare n righe , n word, n byte in un file

```
wc file.txt

# out put =>
# nLines nChar nBytes file.txt

wc file.txt --lines

# out put =>
# nLines file.txt
```

Comando read

read prende una stringa in input e istanzia una variabile che ha come valore la stringa

```
read line;
echo ${line}; #prendo in input qualcosa e lo stampo

# con read posso anche fare delle cose di questo tipo =>

while read input ; do
    echo ${input};
done

# questo ciclo è infito e prende in input del contenuto e lo stampa

exec {fileDescriptor}<file.txt
read -u ${fileDescriptor} ciao;
echo ${ciao};

# il flag -u indica a read di leggere da un fileDescriptor , in questa maniera
# (senza un ciclo) read leggerà solo la prima riga del file
```

Comando Sed

```
sed 's/cane/gatto/g'
> io sono un cane
io sono un gatto
```

```
> cane bello
gatto bello
^C

#prende come stdin file.txt e come stdout la console
sed 's/cane/gatto/g' file.txt

#prende come stdin file.txt e come stdout file.txt
sed 's/cane/gatto/g' file.txt > file.txt

#rimuovere la prima occorrenza della parola 'cane' in ogni riga
sed 's/cane//' #se ci fosse stata la 'g' avrebbe cancellato tuttle le occorrenze
```

come si legge: il primo ‘s’ sta per *substitude*

la parola ‘cane’ (tra la prima ‘/’ e la seconda ‘/’) indica la parola da cercare

la parola ‘gatto’ (tra la seconda‘/’ e la terza‘/’)

l’ultima ‘g’ sta per *global* se non ci fosse sostituirebbe solo la prima occorrenza in ogni riga, con g le sostituisce tutte.

al posto di ‘g’ posso mettere un numero e rimuoverò sola la nth occorrenza

Gli si può passare come primo parametro il nome di un file e quello diventerà lo STDIN

Con ‘-i’ modifico direttamente il file dato in input

```
#sed con l'utilizzo di simboli:
#sostituisci qualsiasi carattere all'inizio della riga con niente
sed 's/^.//'

#rimuovo primo e ultimo carattere di ogni riga
sed 's/^.//;s/.$//'

#rimuovi i prima 4 caratteri (uso regex)
sed -r 's/.{4}//'
sed 's/..../' #senza regex
```

Caratteri speciali di ‘sed’

- ‘^’ indica l’inizio di una riga
- ‘$’ indica la fine di una riga
- ‘.’ indica qualsiasi carattere
- ‘;’ esegue due sostituzioni insieme
- con -r posso utilizzare i Regex (non li abbiamo fatti)

RICORDA ⇒ sed non fa modifica ai file quindi quando applico delle sostituzioni oppure elimino delle righe **NON** modifico il file ma alla fine sed applica un filtro al contenuto del file.

Sed fa due cose :

▼ Sostituire parole in un file returnando il suo contenuto

```
sed 's/{parolaDaSostiture}/{parolaCheLaSostiurà}/' {fileToRead}

# questo comando sostiurà la prima occorenza di ogni riga della parolaDaSostiure

# esempio =>

# file.txt =>
# ciao suca
# no perchè ciao

sed 's/ciao/no/' file.txt

# output =>
# no suca
# no perchè no

sed 's/ciao/no/{nOccorenzaDaCambiare}' file.txt

# se nOccorenzaDaCambiare è specificato , sed cambiarà solo
# la nOccorenzaDaCambiare e solo quella , quindi se metti 2
# solo la seconda volta che ciao compare in una riga verrà cambiata

sed 's/ciao/no/{nOccorenzaDaCambiare}g' file.txt

# se dopo nOccorenzaDaCambiare ci metti g sed cambierà tutte le successive volte che
# la parola "ciao" compare nella riga

# esempio =>


# file.txt =>
# ciao suca ciao ciao
# no perchè ciao ciao
# ciao ciao ciao ciao ciao

# !-- risultato della sed --!

# ciao suca no no
# no perchè ciao no
# ciao no no no no

sed '${nRigaDiInizioDellaSostituzione},${nRigaFine} s/ciao/no/' file.txt

# questo comando inizia a sostituire da nRigaDiInizioDellaSostituzione
# fino a nRigaFine comprese

sed -n 's/ciao/no/' file.txt

# il flag -n fa printare solo le righe modificate del file
```

▼ Permette di eliminare righe di un file

```
sed '{nRigaToDelete}d' file.txt

# questo comando permette di eliminare una, ed una sola, del file.txt specificata
# da nRigaToDelete

sed '{start},{end}d' file.txt

# elimina le riga da start a end

sed '$d' file.txt

# $ indica l'ultima riga , quindi questo comando permette di eleminare
# l'ultima riga

sed '/{pattern}/d' file.txt

# questo comando cerca una parola in ogni riga e se lo trova elimina la riga
```

Comando tee

```
#tee prende l'input dallo STDOUT di 'cat' e lo duplica dentro a a.txt e b.txt
cat dati.txt | tee a.txt b.txt
> #il contenuto di dati.txt viene visualizzato anche a video
```

prende lo STDIN e reindirizza lo STDOUT su più risorse

Bash file scripting

if / test

le parentesi graffe dell'if fanno riferimento all'operazione test ⇒

```
if ( [ "codition" ] && [ "condition" ] ) || [ "condition" ] ; then
    # do stuf if true
elif ["condition"] ; then
    # do other stuff
else
    # do someother stuff
fi




test {valore1} >!=< {valore2}

# al posto degli operatori aritmetici >!=< ci possono anche essere dei flag del tipo:
# -eq => equal to ( = )
# -gt => grater than ( > ) -ge => grater equal ( >= )
# -lt => lesser then ( < ) -le => lesser equal ( <= )
```

```
# ...

string="stringa"
while test ${#string} -eq 7; do
    # do stuff
    i=$(( i + 1 ))
    echo "i am in here";
done
```

Come far Uscire (return) il programma

```
exit 1 #oppure un numero casuale != 0

#esempio =>

# se il numero di dati passati allo script è != 1 returna errore
if (( $# != 1 )) ; then
  echo "serve nomefile" ;
  exit 1 ;
fi

# controllo se il file esiste , se non esiste returna errore
if [[ ! -r $1 ]] ; then
  echo "il file $1 non esiste";
  exit 2;
fi
```

Ciclo for

```
# Elenco può essere un comando come ls ./
# che restituiesce un elenco di nomi dei file
# l'importante è che il comando sia contenuto dentro ai `` => `ls ./`

for nomeVariabile in Elenco ; do
    echo ${nomeVariabile}
done

#Oppure così

for (( i=0 ; i <= 10 ; i=i+1 )) ; do
    echo ${i};
done
```

While

```
while [ condizione per ciclare ] ; do
    # do something
done

# esempio =>
```

```
exec ${fileDescriptor} < fileName.txt

# qui leggo tutto il file fino alla sua fine assegnando
# a parola1 la prima parola che incontra nel file
# stesso discorso per parola2 ; mentre per parola3 , gli assegna
# dalla terza parola in poi (se ce ne sono)

while read ${fileDescriptor} parola1 parola2 parola3 ; do
    echo ${parola1};
done
```

Comando Wget

Wget accetta un URL da cui scaricare il file su cui punta URL (lo scarica dove lo script si trova)

```
wget [url];

# con il flag -p scarica tutto il sito sul quale punta l'url
# compresi i file delle immagini presenti sul sito

wget -p [url];
```

Variabile RANDOM

RANDOM **è una variabile d'ambiente** che contiene **un valore randomico con seed**(ciò vuol dire che i numeri sono gli stessi all'inizio di ogni programma) che **cambia in run Time**

```
#se inserisco lo stesso seed la sequenza sarà sempre uguale
#in questo caso 11 è il nostro seed
RANDOM=11
echo $RANDOM #21039
echo $RANDOM #12648
echo $RANDOM #19135
```

Operazioni con le stringhe

Ciclo char per char stringa

```
string=$1;

for (( i=0 ; i < ${#string} ; i=i+1)) ; do
  currentChar=${string:$i:1}; #dentro a current char ho il carattere corrente
```

```
  echo ${currentChar};
done
```

Lunghezza di una stringa

```
Stringa="ma vaffangul";
echo "la stringa è lunga => ${#Stringa} ";
#inserisco un '#' davanti al nome della variabile e dentro una variable substitution
```

Cut

cut permette filtrare una stringa ritornando n bytes

```
cut -b {nCar1},{nCar2},.. "input"

# returna i caratteri( 1 byte = 1 carattere ) nCar1 , nCar2 ecc. per ogni numero
# presente dopo -b

# esempio =>

cut -b 1,2,4 "input"

# output =>

# inu

# per stampare i primi n caratteri di una parola => esempio: i primi 3 caratteri

cut -b -3 "ciao";

# output => cia

sed 's/ciao/no/' file.txt | cut -b 1,2,3,4,5,6

# posso fare anche cose di questo tipo => il risultato di sed viene passato a cut
# e poi stampato a video
```

Grep

```
grep "stringToSearch" "fileToSearch" | echo

# questo codice permette di cercare in ogni riga del file passato una stringa
# a piacimento , e poi di stamparla tramite "| echo "
```

Ottieni la prima parola di una riga

questo comando prende la prima parola (array di carattere separato dal resto della frase da un spazio) di una frase

```
read row;
echo "${row%% *}"

# più in generale =>
${row%%charDivisore*}
```

elimina una stringa da un’altra stringa

```
String="ciao sono un cane"
echo ${String#"ciao"};

# output =>
# sono un cane

# più in generale =>
${String#StringaDaEliminareDaString}
```

rimpiazza dei caratteri in una stringa

```
String="ciao broder bisellone"
echo ${String//"b"/"p"};

# output =>
# ciao proder pisellone

# più in generale =>
${String//charDaSostituire/charCheSostituisce};

# se metto i // => dice alla variable
# espansion di sostituire tutte le occorrenze
# di charDaSostituire con charCheSostituisce
# MENTRE se metto il singolo / => sostituisce
# solo la prima occorenza in tutta la stringa

# sostituisci tutte le occorenze
${String//charDaSostituire/charCheSostituisce};

# sostituisci la prima occorenza
${String/charDaSostituire/charCheSostituisce};
```

Raggruppamento di comandi

```
cat test_string_raggr | (read A B C; echo $A; read D E F; echo $E; read G H I; echo $I )
A
E
I

cat test_string_raggr
A B C
D E F
G H I
L M N

#raggruppa i comandi, con la read legge una riga a comando

cat test_string_raggr | grep A
A B C
```

Separatori di comando e priorità

- Delimitano la fine di un comando
  - `;` `&` e l'andata a capo `newline`
- Connettono due comando
  - `&&` `||` `;` `&` e l'operatore speciale `|`
- Precedenze
  - Gli operatori `&&` e `||` hanno la stessa precedenza.
  - Gli operatori `&` e `;` hanno la stessa precedenza.
  - Gli operatori `&&` e `||` hanno maggior precedenza degli operatori `&` e `;`
  - L'operatore speciale `|` ha la precedenza maggiore di tutti.

processi in background

```
sleep 10 &
> [1] 52
echo $!
> 52

#aspetto 10 secondi
mattia@LAPTOP-3F81T829:~$
> [1]+  Done                    sleep 10
```

`bg` riprendo l'esecuzione in background di un processo in sospeso

`jobs` da una lista con tutti i processi in background o sospesi

`^Z` mette in sospeso un processi

`$$` indica il PID dello script che stai runnando.

`$BASHPID` indica il PID della Bash corrente (dell'istanza della bash corrente). è diverso da `$$` anche se a volte possono dare lo stesso risultato.

`fg %n` porta il process in foreground

`disown –[ar] jobs %indice` sgancia il job dalla shell interattiva che lo ha lanciato
i jobs sono specificati con PID o %indice

```
sleep 101
^Z
[1]+  Stopped                 sleep 101
sleep 102
^Z
[2]+  Stopped                 sleep 102
jobs
[1]-  Stopped                 sleep 101
[2]+  Stopped                 sleep 102
bg #riparte in bakground il processo sospeso più recente
[2]+ sleep 102 &
bg %1 #con % indico quale processo sospeso fare ripartire
[1]+ sleep 101 &
```

Se metto `& ;` nello stesso comando, la bash da errore

```
./comandouno.sh &; ./comandodue.sh & #da errore
./comandouno.sh & ./comandodue.sh & #funziona
```

Comando nohup

```
nohup ./programm uno due tre &
```

lancia in esecuzione un comando sganciandolo dalla shell corrente

anche se chiudo il terminale corrente il processo va avanti

Comando kill

```
kill -SIGTSTP pid (è il segnale lanciato con CTRL Z )
kill -SIGCONT pid (riprende processo sospeso, usato da fg e bg)
kill -SIGINT pid (è il segnale lanciato con CTRL C)
kill -9 pid (funziona sempre e killa di brutto il processo)
```

Comando trap

```
#riceviSIGURS1e2.sh
trap "echo \"ricevuto SIGUSR1 !!! Termino !!!\"; exit 99" SIGUSR1
ricevutoSIGUSR2() {
echo "ricevuto SIGUSR2, continuo";
}
trap ricevutoSIGUSR2 SIGUSR2
while true ; do echo -n "."; sleep 1; done
```

```
./riceviSIGUSR1e2.sh &
CHILDPID=$!
sleep 5 ; echo "mando SIGUSR2"; kill -s SIGUSR2 ${CHILDPID}
sleep 5 ; echo "mando SIGUSR2"; kill -s SIGUSR2 ${CHILDPID}
sleep 5; echo "mando SIGUSR1"; kill -s SIGUSR1 ${CHILDPID}
sleep 4;
```

`trap action lista_di_segnali`

Comando wait

```
(sleep 20; exit ) &
wait $!    #attende terminazione di exit 3
echo "il valore restituito da wait é:"$?
```

Comando killall

```
for (( i=1; $i < 10 ; i=$i+1 )) ; do ./attendiA.exe & ./attendiB.sh & done
#manda il segnale SIGNTERM a tutti i processi che rispondono a questo pattern in RAM
killall -r 'attendi[[:upper:]]*'
#killa i processi creati prima
```

manda SIGNTERM a ‘n’ processi in esecuzione (in memoria RAM)

Oltre a mandare il segnale SIGNTERM con alcuni argomenti si può inviare un altro tipo di segnale (`-s`)

Redirigi l’output sullo standard error

```
echo "ciao" 1>&2
```

1>&2 permette di redirigere l’ouput di echo sullo standard error quindi , di fatto , non viene stampato niente

Find

```
find [Cartella di inizio ricerca] [su che parte del file effettuare la ricerca] [condizione della ricerca]

# esempio

find ./path/to/desktop -name '*.sh' -print

# questo comando find cerca sul desktop e sulle sottocartelle tutti i file che
# hanno come estensione .sh e li printa a schermo (grazie al flag -print) .
# effettuo la ricerca sul nome del file grazie al flag -name .

find ./path/to/desktop -mindepth 2 -maxdepth 2 -name '*.sh' -print

# -mindepth => dice al comando find di iniziare la ricerca dalla n-esima cartella
# in questo esempio find inizierà la ricerca dalla seconda generazione di cartelle
# che trova ( quindi dalle sottodirectory di Desktop )

# -maxdepth => dice fino a quale generazione di cartelle portare la ricerca
# nel caso dell'esempio find cerca i file script solo ed esclusivamente
# nelle sottodirectory di desktop , e non si ferma a loro senza andare dalle
# loro sottodirectory

find ./path/to/desktop -type f

# -type f => indica al find di returnare tutti gli elementi di tipo file che trova
# nel Desktop e nelle sue sottodirectory (per returnare le directory invece di f ci va d)
```

Controlla quale file è il più recente

```
mostRecentFile=0;
for file in `ls`; do
    if [[ ${mostRecentFile} == 0 ]]; then
        mostRecentFile=${file};
    # con questo else if controllo se
    # mostRecentFile è quello modificato
    # più tardi dei due, se si ho un nuovo
    # file più recente
    elif [[ ${mostRecentFile} -ot ${file} ]];
        mostRecentFile=${file};
    fi
done
```

Cenni sui sistemi operativi

Eccezioni ⇒ sono sincrone e sono provocate da una istruzione macchina eseguita dalla CPU che provoca un errore

- TRAP ⇒ l'istruzione che ha generato l'exception viene momentaneamente sospesa , viene eseguita la gestione del TRAP , e poi **l'istruzione che ha generato la TRAP viene ripresa da dove si era fermata.** Esempio ⇒ TRAP per debugging
- FAULT ⇒ l'istruzione che ha generato l'exception viene interrotta , viene eseguita la routine di gestione della fault e alla fine della routine **l'istruzione che aveva generato l'exception viene rieseguita da capo (differenza principale con TRAP).** Esempio ⇒ Page FAULT (la pagina di memoria che contiene l'indirizzo a cui stiamo cercando di accedere è stata swappata su disco, allora la pagina viene caricata in memoria e prima o poi la fault si risolve).
- ABORT ⇒ causata da un errore irrimediabile , l'istruzione abortita ed il processo killato. Esempio ⇒ segmentation fault oppure divisione di un numero per 0

C assembling and build

Variabili globali

Dichiarazione

le variabili globali sono dichiate fuori da ogni funzioni ed è possibile accedervi anche altri file dell'eseguibile

```
//dichiarazione di una variabile globale in un modulo esterno
extern tipo NomeVariabile;
extern int NomeVariabile;
```

```
//dichiarazione della variabile globale nel modulo in cui viene definita la variabile
int NomeVariabile
```

La precedente dichiarazione indica che la variabile `NomeVariabile` non è presente in questo file bensì in un altro file, il Linker dovrà cercarla in tutti i moduli finché non trovare la dichiarazione di `NomeVariabile` senza essere preceduta dalla key world `extern`

Protezione delle variabili globali locali

Se vogliamo dichiarare una variabile globale accessibile solo dal nostro modulo e non da altri moduli bisognerà utilizzare la key world `static`

```
static tipo NomeVariabile
static int NomeVariabile
```

GNU C Compilazione e MakeFile

gcc - opzioni del preprocessore

`-DNOMESIMBOLO=VALORE` definisce un simbolo del preprocessore chiamato NOMESIMBOLO con valore = a VALORE (= VALORE può anche essere omesso)

Esempi di definizione di simboli e di codice compilato in maniera condizionale:

```
#define _POSIX_C_SOURCE 199309L
#if defined _POSIX_C_SOURCE && _POSIX_C_SOURCE >= 199309L
  extern int nanosleep ( const struct timespec *__requested_time,
                         struct timespec *__remaining );
#endif

//_POSIX_C_SOURCE posso crearlo anche dando questo flag al gcc
-D_POSIX_C_SOURCE=199309L
gcc -c -D_POSIX_C_SOURCE=199309L miomodulo.c
```

gcc - opzione del compilatore

solo quelli più utilizzati da noi

- `-c`: compila il modulo specificato senza fare il linking
- `-ansi:` compila secondo lo standard c90
- `-Wpedantic`: segnala codice potenzialmente pericolo o poco leggibile
- `-Wall:` avvisa se ci sono istruzioni sbagliate o potenzialmente pericolose
- `-Werror`: tratta ogni warning come se fossero degli error
- `-S`: traduce il file sorgente in assembly
- `-E`: ordina di effettuare la sola fase di preprocessing, mandando sullo standard output il codice sorgente C generato dal preprocessore.

gcc - opzioni del linker

- `-LPercorsoDirectoryDelleLibrerie`: specifica il percorso relativo o assoluto per raggiungere la directory in cui sono contenute le librerie fornite dall'utente. L'opzione può essere utilizzata più volte per specificare più di un percorso. Es: -L. -L/home/studente/lib
- `-lnomelibreria` specifica di utilizzare la libreria indicata dal nome "ristretto" nomelibreria. Ad esempio, la libreria matematica libm.a viene specificato con -lm mentre una eventuale libreria libpippo.a viene specificato con -lpippo .
    - **Nota bene**: l'opzione -l può essere usata più volte in una stessa riga di comando per indicare tante librerie diverse.

    - **Attenzione**: nelle versioni più recenti del gcc, le opzioni -l devono essere collocate obbligatoriamente alla fine della riga di comando, dopo l'elenco dei moduli da collegare.

- `-Wl,-soname,nomelibreria` (tutto attaccato, e le virgole servono) utilizzato quando si sta creando una libreria dinamica, specifica il nome `nomelibreria` da assegnare alla libreria che si sta creando.
- `-Wl,-rpath,RunTimeLibDir` (tutto attaccato, e le virgole servono) utilizzato quando si sta creando un eseguibile che deve usare una libreria dinamica collocata fuori dalla directory predefinita. In questo modo si indica la directory `RunTimeLibDir` in cui, nel momento della esecuzione (cioé a runtime) dell'eseguibile, si troverà la libreria dinamica da utilizzare. Mnemonicamente, `rpath` significa **Run-Time Path**.

MakeFile

esprime **l'albero delle dipendenze** di un eseguibile

il MakeFile contiene un insieme di regole che:

1. descrivono la **struttura dell'albero** delle dipendenze del progetto;
2. descrivono le **regole di generazione** dei vari **target** di ogni dipendenza del progetto.

Ciascuna make rule :

1. **indica** uno o più **target** a cui la regola si riferisce (target list),
2. **elenca** tutte le **dipendenze** da cui quel **target** dipende (dependency list),
3. ed elenca le **operazioni** (command list) che devono essere svolte quando una delle dipendenze è più recente del target oppure quando il file target non esiste.

Sintassi:

- Il `#` indica una riga commentata.
- Ciascuna dependency list e ciascun comando **termina dove termina la riga**.
- In ciascuna regola, le **dipendenze** sono **separate** tra loro da uno o più **spazi bianchi o tab**.
- Ogni **comando della command list deve iniziare con un carattere TAB**.
- Ciascuna **regola** deve essere **separata dalla successiva** da una **riga vuota**.
- Il Makefile deve **terminare** con una **riga vuota** (o meglio, una andata a capo \n).

```
all : main.exe
riga vuota
main.exe : main.o funzioni.o
TAB gcc -o main.exe main.o funzioni.o
riga vuota
main.o : main.c funzioni.h strutture.h
```

```
TAB gcc -c -ansi -Wpedantic main.c
riga vuota
funzioni.o : funzioni.c strutture.h
TAB gcc -c -ansi -Wpedantic funzioni.c
riga vuota
clean:
TAB -rm main.exe *.o
```

questo `MakeFile` dice che main.exe dipende dai file oggetti `main.o` e `funzioni.o`.

`main.o` dipende dal file C `main.c` e dalle instestazioni dei file `funzioni.h` e `struttre.h`

funzioni.o dipende dal file C `funzioni.c` e dalla intestazioni `strutture.h`

perciò quando creo gli oggetti gli linko un file `.c` e solamente le intestazioni `.h` dei file da cui dipende, sarà poi in fase di compilazione generando il file `main.exe` che `main.c` verrà legato a `funzioni.c`

comando make

Esegue le istruzioni del make file e crea l'eseguibile

```
make #cerca un file chiamato MakeFile nella cartella corrent
make -f nomeMakeFile
make -f nomeMakeFile funzioni.o #posso anche solamente specificare un
#target da compilare e lui eseguira tutti i comandi necessari a generare funzioni.o
```

Best practices

- `all` come prima regola:

  ```
  all: main.exe
  ```

- si possono utilizzare **target fittizi** ad esempio `.clean`, li dichiaro dentro `.PHONY`

  ```
  .PHONY : clean
  clean: NESSUNA DIPENDENZA
  rm *exe *.o *~
  ```

Variabili nel MakeFile

si dichiarano all'inizio del makefile

```
PERCORSO=/usr/local/lib
PERCORSO=`pwd` #ERROR: non posso eseguire comandi all'inizio del MakeFile
```

per usare la **bash** invece della shell sh(la shell sh ha meno cose della bash)

```
  SHELL=/bin/bash #all'inizio del file MakeFile
```

Esempio completo del MakeFile

```
CFLAGS=-ansi -Wpedantic
TARGET=main.exe
OBJECTS=main.o funzioni.o
all : ${TARGET}
riga vuota
main.exe : main.o funzioni.o
TAB gcc -o ${TARGET} main.o funzioni.o
riga vuota
main.o : main.c funzioni.h strutture.h
TAB gcc -c ${CFLAGS} main.c
riga vuota
funzioni.o : funzioni.c strutture.h
TAB gcc -c ${CFLAGS} funzioni.c
riga vuota
.PHONY: clean
riga vuota
clean:
TAB -rm ${TARGET} ${OBJECTS} *~ core
#NEWLINE
```

Dettagli sul MakeFile

- Eseguire il MakeFile non nella cartella corrente, ma su un altra cartella

  ```
  --directory=nomedirectory
  -C nomedirectory
  #con questi file il comando make entra nella cartella data, cerca il MakeFile e lo
  #esegui in quella cartella
  ```

- Utilizzare variabili della shell nel MakeFile: bisogna aggiungere un **$ davanti al nome** della variabile:

  ```
  grep stringa nomefile ; if $$? ; then echo $${PATH} ; else echo $${USER} ; fi
  ```

- Andare a capo anche se il comando deve stare in una riga: inserisco `\` per andare a capo:

  ```
  gcc -c -ansi -Wall \
   main.c -lm
  #se fossi andato a capo senza \, il makefile mi avrebbe dato errore
  ```

```
#IMPORTANTE: il carattere \ deve essere l'ultimo carattere della riga, neanche
#lo spazio è tollerato
```

- **MakeFile ricorsivi**: se ho il progetto dislocato in tante cartelle, posso mettere un makefile in ogni cartella per renderle “indipendenti”.

  Nella directory principale del progetto devo collocare un Makefile in cui il target principale all: ha il compito di ordinare di:

  1. andare in ciascuna directory dove devo svolgere qualche compito;
     1.1. lanciare il make per il Makefile in quella specifica directory.

  Dopo avere eseguito tutti i Makefile nelle sottodirectory del progetto, il makefile nella directory principale dovrà

  1. eseguire il compito di mettere assieme tutti i files generati e creare il risultato finale del progetto. Occorre perciò creare un target diverso dal target principale che esegue il compito di generare il risultato finale.

  Esempio:

```
# NOTARE che nel target principale non metto nessuna dipendenza
# per costringere il make ad eseguire tutti i comandi che specifico sotto
# almeno fino a che non capita un errore in una sottodirectory
# NOTARE anche che PER CIASCUNA RIGA DI COMANDI,
# il make riparte dalla DIRECTORY CORRENTE

all:
 cd 1 ; make     #NB: Potevo anche scrivere solo make -C 1
 cd 2 ; make     #NB: Potevo anche scrivere solo make -C 2
 make main.exe
main.exe: main.o 1/obj1.o 2/obj2.o
 gcc ${CFLAGS} -o main.exe main.o 1/obj1.o 2/obj2.o ${LIBRARIES}
main.o: main.c
 gcc -c ${CFLAGS} main.c
.PHONY: clean
clean:
 - cd 1 ; make clean
 - cd 2 ; make clean
 - rm -f main.o main.exe
```

- **MakeFile ricorsivi**, con ricerca automatica nelle sottodirectory

  Per fare questo è necessario che ogni sottodirectory contenga un makefile

```
# NOTARE che nel target principale non metto nessuna dipendenza
# per costringere il make ad eseguire tutti i comandi che specifico sotto
# almeno fino a che non capita un errore in una sottodirectory
# NOTARE anche che PER CIASCUNA RIGA DI COMANDI,
# il make riparte dalla DIRECTORY CORRENTE

all:
```

```
 for DIRNAME in `find ./ -mindepth 1 -maxdepth 1 -type d -print` ; do if
! make -C $${DIRNAME} ; then exit $$? ; fi ; done
 make main.exe
main.exe: main.o 1/obj1.o 2/obj2.o
 gcc ${CFLAGS} -o main.exe main.o 1/obj1.o 2/obj2.o ${LIBRARIES}
main.o: main.c
 gcc -c ${CFLAGS} main.c
.PHONY: clean
clean:
 for DIRNAME in `find ./ -mindepth 1 -maxdepth 1 -type d -print` ; do if
! make -C $${DIRNAME} clean ; then exit $$? ; fi ; done
 - rm -f main.o main.exe
```

IMPORTANTE: Manca la parte sulle librerie,

> Dispense di SISTEMI OPERATIVI
>
> https://www.cs.unibo.it/~ghini/didattica/sistemioperativi/2_RichiamiLinguaggioANSIC_Makefile.html

I Thread

> https://www.cs.unibo.it/~ghini/didattica/sistemioperativi/5_Introduzione_All_Uso_di_POSIX_Thread.pdf

Libreria

la libreria è pthread

```
gcc –ansi -o thr1 thr1.o –lpthread
```

Metodi dei thread

Creazione di un thread

```
int pthread_create ( pthread_t * thread, pthread_attr_t *attr,
 void* (*start_routine)(void *), void * arg );
```

- crea una thread e lo esegue in parallelo
- Il primo parametro thread è un puntatore ad un identificatore di thread in cui verrà scritto l’identificatore del thread creato.

- Il terzo parametro start_routine è il nome (indirizzo) della procedura da fare eseguire dal thread. Deve avere come unico argomento un puntatore.
- Il secondo parametro seleziona caratteristiche particolari: può essere posto a NULL per ottenere il comportamento di default.
- Il quarto parametro è un puntatore che viene passato come argomento a start_routine

.

Chiusura di un thread

```
void pthread_exit (void *retval);
```

termina l'esecuzione del thread da cui viene chiamata, immagazzina l'indirizzo retval, restituendolo ad un altro thread che attende la sua fine

Identificatore

`pthread_t` è il tipo che rappresenta l'id di un thread

```
pthread_t pthread_self (void);
// restituisce l'identificatore del thread che la chiama.
int pthread_equal ( pthread_t pthread1, pthread_t pthread2 );
// restituisce 1 se i due identificatori di pthread sono uguali
```

TEORIA by Tone

Bash

- **System Calls**: i programmi di sistema e le applicazioni accedono alle librerie di sistema, che offrono delle funzioni (Application Programming Interface), o interfacce, che permettono a linguaggi di alto livello di invocare chiamate di sistema al kernel. Diverse per ogni SO. Esempi di API: Win32 API, POSIX API, Java API
- **Servizi del Sistema Operativo**: viene caricato in RAM, e prende il controllo della macchina fornendo all'utente un'interfaccia e dei servizi (gestione risorse, comunicazione, rilevamento errori, I/O)
- **SO a linea di comando**:
  - **comandi**: ordini la cui implementazione è già contenuta nel file eseguibile della shell

    - **eseguibili**: ordini non built-in. È il nome del file da eseguire.
- **Standard POSIX**: Portable Operating System Interface (standard per system call per linguaggio C, shell, utility, system administration)
    - per utilizzare uno standard diverso in un codice, bisogna usare una `#define _POSIX_SOURCE 1` sopra gli include (1 è relativo al numero della versione dello standard)
- **File System**: l'organizzazione del disco rigido che può contenere file e loro informazioni
    - in Windows: partizioni sono C:, directory hanno solo il nome, in linux non si distinguono
    - in Linux: partizione principale chiamata '/'(anche separatore)
- **Espansioni bash**: la bash legge ogni riga di comando, la interpreta, poi esegue delle operazioni, e modifica il contenuto dei comandi a seconda di ciò che è stato svolto.
    - **History expansion**: utilizzo di comandi già usati
    - **Brace expansion**: la bash cerca di capire se in un comando ci sono delle stringhe, racchiuse tra spazi, a capo o tab, e delimitate da {} non precedute da $ (esempio: `echo a{b,c,d}e = abe ace ade`). Preambolo e postscritto possono mancare, niente spazi bianchi (se non protetti da \ o "". NON PROTEGGERE LE GRAFFE). Sequence expression: `{1..3} = 1 2 3`
    - **Tilde expansion**:
        - ~ = /home/user
        - ~/ = /home/user
        - ~/a = /home/user
        - ~user = /home/user
        - ~user/a = /home/user/a
        - ~nonUnUser = ~nonUnUser
    - **Variable expansion**: riconosce se ci sono delle variabili nel comando ($) e le sostituisce con il loro contenuto
    - **Parameter expansion**: fornire una stringa che è parte o parte modificata del contenuto di una variabile.
    - **Arithmetic expansion**: ((operazione))→risultato dell'operazione (anche assegnamenti)
    - C**ommand substitution**: `` `command` `` oppure `$( command_line )`->output del comando-script
    - W**ord splitting**: la variabile **IFS** contiene i caratteri che fungono da separatori delle parole negli elenchi. Default→ IFS=$' \t\n'.
    - **Pathname expansion**

    - **Quote removal**

- **Utenti e Gruppi**: un utente è un'entità, che può anche rappresentare una persona fisica, caratterizzato da uno username e uno userID. Un gruppo ha un groupname e un groupID. Ciascun file o directory appartiene ad un utente (owner). Ciascun file è associato ad un gruppo.
- **Permessi File**: `drwxrwxrwx`
    - d indica una directory, - un file normale, l se è un link
    - rwx sono lettura, scrittura ed esecuzione, in tre triplette per proprietario, gruppo e tutti gli altri. Nei permessi utente, ci può essere una s a sostituire la x. Significa che è eseguibile coi permessi del proprietario.
- **Accesso remoto SSH**
    - accesso remoto ad una macchina, con comando: `ssh nome_account@nome_host_server`, dove il servizio sshd(demon) è in ascolto sul server
    - i messaggi tra user e server sono criptati
        - Crittografia a chiave pubblica con algoritmo RSA (Rivest Shamir Adelson), chiavi da 1024-2048bit, una privata e una pubblica per nodo. Comunicazione tramite TCP.
    - Operazioni preliminari:
        - `ssh-keygen -t rsa` per creare una coppia di chiavi per RSA, quella privata viene messa nella home directory dell'utente nel file `.ssh/id_rsa` , quella pubblica va messa nella cartella del server `.ssh/authorized_keys` . Si può usare il comando `scp ~/.ssh/id_rsa.pub account_server@nome_host_server:.ssh/authorized_keys`
    - Automatizzazione comandi su server:
        - `ssh -T account_server@nome_host /bin/bash <<PAROLA [comandi bash] PAROLA` si collega al server, lancia la bash, e poi esegue tutto quello compreso fra PAROLA
- **Variabili d'ambiente**
    - $PATH: impostata dal SO ma modificabile, contiene una sequenza di percorsi assoluti nel filesystem di alcune directory in cui sono contenute gli eseguibili. Strutturato così: `percorso:percorso:percorso…` Per modificarlo, aggiungendo un percorso, `PATH=${PATH}:percorso`
    - $HOME: home directory dell'utente
    - $USER: username che ha lanciato la shell
    - $SHELL: percorso interprete attuale
    - $TERM: terminale
- **SubShell**: creata in caso di comandi raggruppati, esecuzione di script o in background. Propria directory corrente, data dal padre, ed eredita le variabili d'ambiente, in copia. Può modificarle,

ma per il padre non cambiano.

  - Esecuzione senza subshell: vedi comandi, si possono modificare le variabili del padre. Usa lo stesso interprete del padre, può generare errori se nel file ne è specificato un altro.
  - `VAR="…" ./script` → VAR è passato come argomento a script. Niente source.

- **Avvio della Shell**
  - non interattiva: esegue script, lanciata con **-c**
  - interattiva NON di login: quella iniziale nel terminale, **lanciata senza niente**
    cerca di eseguire `.bashrc`
  - interattiva di login: come la non di login, ma chiede user e password, lanciata con **-l** o **-login**
    cerca di eseguire `/etc/profile`, poi uno fra `.bash_profile`, `.bash_login` e `.profile` , e poi `.bashrc` . Langia `.bash_logout` quando si chiude.
- **Exit Status**: da 0 a 255
  - 0 è tutto ok
  - 1 errori generali
  - Per espressione aritmetica tra (())
    - 0 se fornisce risultato logico true o risultato intero diverso da 0
    - 1 se fornisce false, risultato intero uguale a 0 o se non è valutabile
- **File Descriptor**: astrazione che permette l'accesso ai file, rappresentato da un integer. Ogni processo ha la sua FD Table, che contiene indirettamente le informazioni sui file utilizzati. Questa contiene un FD per ogni file utilizzato, che punta ad una tabella di sistema che contiene le sue informazioni. Nella directory /proc/ esiste una sottodirectory per ogni processo in esecuzione del processo stesso. `ls /proc/$$/` visualizza il contenuto della directory della shell corrente. In questa cartella è presente una sotto-dir fd, contenente i file aperti di quel processo.
  - fd = open(path, O_RDONLY)
  - read(fd, buffer, MAXSIZE)
  - close(fd)
- **Standard IO**: le shell figlie ereditano una copia della tabella dei file aperti del padre
  - stdin: identificato da costante numerica 0
  - stdout: costante 1
  - stderr: costante 2

  Le shell figlie possono ridirezionare gli stream del padre usando exec. I ridirezionamenti non modificano il valore dei fd che vengono ridiretti. Mettendo in esecuzione si può assegnarli stdin-

out diversi: ***program<nome_file***

- <: ricevere input. Il programma vedrà il contenuto del file come venisse digitato da tastiera. (Ctrl+D per emulare un EOF da tastiera).
  - `program <N fine` per ridirezionare il file sul fd N del programma
- >: sovrascrive file. Stampa l'output sul file invece che a video.
  - &> per mandare anche stderr
  - ***program 2>error >output*** per reindirizzarli contemporaneamente ma separati
  - `program N> file` per ridirezionare il fd N sul file
- >>: scrive su file aggiungendo
  - si può fare anche contemporaneamente: ***program <input >output***
- *program1* | *program2*: manda output di program1 nell'input di program2. I processi sono eseguiti contemporaneamente, quando program1 chiude l'output, program2 vede l'input chiudersi. Se uno dei due program è composto da loop, viene eseguito in una subshell.
- >&: una bash ha due stream, entrambi di input o output, N e M. Con N>&M, ora N punta allo stesso file di M.
- |&: scorciatoia per `2>&1 |`(stderr ridirezionato su stdout, e poi dati entrambi in stdin al processo dopo |)
- <<: Here documents. Passa in input una serie di stringhe comprese tra <<PAROLA e la prima riga che ha all'inizio PAROLA
- <<<: come <<, ma passa solo la stringa che segue

- **Pacchetto COREUTILS**
  - nato dalla fusione di fileutils, shellutils e textutils
  - contiene head, tail, sed, cut, cat, grep e tee
  - possono accettare, dopo le opzioni, come argomenti, i nomi dei file da cui lerre input
  - possono leggere da stdin
- **Processi**
  - processo: più piccola unità di elaborazione completa ed autonoma eseguibile da un computer. Insieme di thread, uno spazio di indirizzamento in memoria e una tabella dei descrittori dei file aperti nel processo stesso.
  - gruppo di processi: può lanciare l'esecuzione di altri processi, solitamente appartengono allo stesso gruppo del padre. Chiamati anche sessioni. Necessari al kernel per raggruppare i processi. Utile per interrompere contemporaneamente tanti processi, ad esempio lanciati da

un certo utente quando fa il logout. Questa funzione è realizzata dal SO lanciando a tutti i processi interessati un segnale detto `SIGHUP("signal hang up")`.

  - terminale di controllo (control tty): processo lanciato per avere un controlling terminal (un terminale da cui è controllato). È dove la shell interattiva esegue. Questa crea una sessione, e la lega al terminale, diventando il leader della sessione. Un terminale è un control tty per una sola sessione.
  - terminale di controllo del gruppo di processi: un processo eredita lo stesso controlling terminal del padre
  - processi in foreground: collegati al terminale dal quale sono stati lanciati per lo stdin, e lo impegnano mutualmente. Un solo processo è in foreground alla volta.
  - processi in background: eseguiti in parallelo, usano una copia dei fd della bash. Per farli continuare anche dopo un eventuale terminazione del terminale, esiste il comando `disown`
  - Jobs di una shell: processi in background o sospesi solo figli di quella shell
  - Job control: permette di portare i processi da fg a bg e viceversa

- **Rischi del wait**
  - processi zombie: un processo figlio termina prima del processo padre, ma mantiene le tabelle pcb (process control block), una descrizione del processo terminato, il suo PID e il risultato. Viene tutto eliminato quando il padre invoca la wait
  - processi orfani: il padre termina senza fare la wait sul figlio, il figlio viene adottato dal processo init, con pid 1, che ogni tanto chiama la wait sui figli facendoli terminare.
- **Precedenze operatori**
  - i fine riga di comando sono: `; & \n`
  - quelli che connettono due comandi sono `&& || ; & |` , | è quello con la precedenza maggiore, poi && e || hanno precedenza più alta di ; e &. Si può modificare queste precedenze tramite raggruppamento con {}. Ricordarsi il ; prima della }

Introduzione Sistemi Operativi

- **Hardware di un computer**
  - CPU: esegue istruzioni, calcoli, gestisce I/O, coordina spostamenti dati
  - BUS: trasferisce dati tra memoria e dispositivi
  - RAM: mantiene programmi quando eseguiti, e i dati che si usano. Volatile.
  - ROM: contiene BIOS, istruzioni per servizi base, utilizzate dal BOOT (accensione del PC)

    - Dischi: mantengono dati e programmi
    - Periferiche I/O
- **Interfacce dei servizi**:
    - L0: Digital logic Level, Hardware
    - L1: Microarchitettura, Firmware
    - L2: Instruction Set Architecture, BIOS, Interfaccia ISA
    - L3: Operating System Machine, fino a qui è binary interface, da qui in poi è software.
        - System Calls
    - L4: Assembly Language
    - L5: Application
- **Servizi di un SO**
    - gestione della protezione
        - protezione della CPU: tiene separati la modalità kernel (che permette di usare tutte le istruzioni ISA, gli indirizzi di memoria, i registri, …) e la modalità utente (limitata)
        - protezione della memoria: la memoria RAM usata dalla parte utente dei processi è organizzata (da indirizzi più bassi a quelli più alti)
            - text: codice eseguibile (istruzioni macchina)
            - data: variabili globali
            - heap: zone di memoria allocate dinamicamente
            - …
            - stack: record di attivazione delle chiamate a funzione
    - gestione della CPU
        - scheduling: quando un processo esegue una chiamata ad interrupt, la CPU entra in kernel mode e aggiugne uno o più stack di sistema al posto di quello del processo, per separare i record di attivazione. In più, per ogni processo c’è uno stack per ciascun livello di privilegio.
    - gestione della memoria
        - indirizzamento (permettere ai processi di accedere alla memoria)
        - segmentazione (dividere logicamente la memoria in base ai diversi utilizzi)
        - protezione
        - memoria virtuale

        - paging
    - gestione di I/O
        - periferiche
        - storage
    - System calls, system library, utilities
- **Supporto ai processi**: i processori moderni permettono di fornire ai livelli superiori un'interfaccia di programmazione denominata "Livello ISA (Instruction Set Architecture)" che mette a disposizione
    - modalità di protezione (kernel-user) e switch tra i modi
    - ALU (Arithmetical and Logic Unit)
    - general pourpose e istruzioni per scambio dati tra registri
    - Registri specializzati:
        - Extended Instruction Pointer (EIP): serve a formare indirizzo di memoria (Program Counter PC) della prossima istruzione da eseguire. Contiene l'offset rispetto all'inizio del segmento di codice.
        - Program Status (PS): Status register
        - Stack Segment Register (SS): inizio dello stack
        - Code Segment Register (CS): inizio sezione text (eseguibile)
        - Data Segment Register (DS): inizio sezione data (variabili globali)
- **Registri**: memorie ad alta velocità nella CPU. 8 registri a 32-bit: EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI. 6 a 16-bit: EFLAGS (Processor status), EIP, CS, SS, DS, ES, FS, GS. + Status Register (non accessibile all'utente)
- **Task**: in un computer sono in esecuzione processi, gestori di interrupt e gestori di eccezioni. Tutti e tre eseguono in un proprio contesto, o task. I task mantengono le informazioni sui segmenti di memoria e sui registri. Lo scheduler decide per quanto tempo eseguire ciascun task. Salvare e ripristinare lo stato di un task permette l'interleaving di più processi. Il livello ISA offre, tra le altre cose, anche istruzioni macchina e registri che consentono al SO l'interleaving, e le system calls in un contesto separato.
- **IA-32** ha un Task register CR3 che punta in RAM al Task State Segment TTS (registri, stack e stato) della task in esecuzione
- **Processi**: istanziamento in memoria di un programma eseguibile. Quando un programma viene lanciato, un loader
    - crea i segmenti di memoria, popolando le tabelle

    - carica in memoria l’eseguibile da text
    - carica la sezione data
    - crea lo stack utente per l’esecuzione delle funzioni
    - crea gli stack di sistema per l’esecuzione delle system calls
    - copia eventuali argomenti
    - ordina alla CPU di eseguire la prima istruzione eseguibile
- **Chiamata a interrupt**: i programmi chiamano Interrupt per ottenere servizi da SO o BIOS. Si usa l’istruzione assembly INT n (n≥0 n-esima posizione nel vettore degli interrupt, che contiene lindirizzo CS:IP della prima istruzione eseguibile della routine di gestione dell’interrupt di indice n. Le routine sono fornite da BIOS e S.O.)
    - Cambio di contesto: è in esecuzione un processo che sta usando lo stack utente. La CPU esegue INT n.
        - L’Instruction Pointer punta all’istruzione successiva ad INT
        - il processore passa in modalità kernel con il livello di privilegio da usare
        - il processore salva i valori dei registri SS, SP, CS, IP nello stack di sistema del processo che ha chiamato l’interrupt
        - il processore carica i registri SS e SP con i valori dello stack di sistema per quel livello di privilegio
        - il processore carica i registri CS e IP con l’indirizzo della routine di gestione dell’interrupt trovato nell’n-esima posizione del vettore di interrupt.
        - Viene eseguita la routine dell’Interrupt nello stack di sistema, che termina con l’istruzione IRET (Interrupt RETurn) che carica i registri IP, CS, SP e SS con i valori salvati sullo stack di sistema, fa tornare il processore in modalità utente e riprende l’esecuzione interrotta
    - Interrupt sincroni: interrupt software, chiamati dalla CPU tramite INT n
    - Interrupt asincroni: interrupt hardware, scatenati da una periferica
    - Interrupt mascherabili: possibile svolgere prima compiti più urgenti
    - Interrupt non mascherabili: non rimandabili
- **System calls**: implementata nella routine di un certo interrupt n, invocata con un’istruzione INT n. Il programma passa i parametri all’interrupt mettendoli in registri (EAX…), chiama l’interrupt, cambia il contesto, esegue su stack del kernel, e ritorna in modalità utente, ripristinando il contesto.

- **Exceptions**: gestite sempre col vettore degli interrupt. Sono sincrone rispetto alle istruzioni eseguite dalla CPU.
  - TRAP: l'istruzione che l'ha causata viene sospesa, ma alla fine della gestione del TRAP viene portata a buon fine (debug)
  - FAULT: l'istruzione viene interrotta, e segue la routine. L'istruzione verrà poi rieseguita dal principio (Page fault)
  - ABORT: errore irrimediabile, istruzione abortita. (Diviso 0, segmentation fault)
- **Ciclo di Neumann**:
  - lettura istruzioni dalla memoria
  - decodifica istruzioni
  - valutazione degli indirizzi
  - lettura operandi dalla memoria
  - esecuzione dell'operazione
  - conservazione risultato
- **Sistemi Multitasking**: più processi presenti in memoria, eseguono concorrentemente contendendosi l'uso della CPU (Time Sharing). Viene gestito dallo scheduler. Lo scheduler alterna i processi nell'uso della CPU. Allo scadere di una time slice (quanti di tempo di esecuzione) il processo corrente viene interrotto e l'esecuzione passa ad un altro. Se un processo deve attendere I/O, la CPU passa ad un altro processo.
- **Sistemi Interrupt Driven**: gran parte delle funzionalità kernel sono eseguite all'interno di qualche routine di gestione di interrupt. Lo scadere di un time slice genera un interrupt.
- **Ciclo di vita dei processi**:
  - New (Start)
    - va a Read (Ammissione, il processo è inserito in cooda)
  - Ready (Wait for CPU)
    - va a Running (Dispatch, messo in esecuzione dallo scheduler)
  - Running (Using CPU)
    - torna a Ready se scade il time slice, generando un Timeout event (Interrupt di tempo)
    - va a Waiting se necessita di I/O (Richiesta)
    - va a Terminated (Conclusione)
  - Waiting (for I/O)
    - poi torna a Ready (Evento che fa terminare l'attesa)

- Terminated (Stop)

Entry Point

- **Inizio esecuzione**
  - **ELF**: formato per eseguibile che prevede una sezione header che contiene un Entry Point Address, il cui valore è l'indirizzo in cui si troverà l'istruzione iniziale da eseguire dopo che il loader l'avrà caricato in memoria. In assembly, la prima istruzione è etichettata con la label _start. Il linker pone l'indirizzo di _start nell'Entry Point Address. Se non trova _start produce un errore.
  - In C il punto di inizio del codice è il main, ma non è la prima istruzione che viene eseguita. Prima il gcc fa alcune operazioni preliminari per far funzionare la libc (imposta vettore interrupt, inizializza lo stack,…) quando effettua l'operazione di linking. Questo codice è contenuto in moduli oggetto. Il gcc, quando effettua il linking, lancia ld passandogli alcuni argomenti che indicano il codice da aggiungere.
  - se il modulo principale del codice è scritto in assembly, il linker non aggiunge niente, perchè è il programmatore a decidere la prima istruzione con _start. Per linkare il codice bisogna usare il linker ld che costruisce l'eseguibile. es: `ld -o print.exe print.o`. Il gcc normalmente aggiunge moduli che contengono già un _start, e provocherebbe un errore "multiple definition of '_start'". Nel gcc c'è anche un salto al main, che però non è definito: "undefined reference to 'main'". Per risolvere questo problema del gcc:
    - linking con gcc senza librerie standard. Aggiungo -nostdlib → `gcc -nostdlib -o print.exe print.o` , ma non posso usare le librerie standard (in teoria non mi servono con l'assembly)
    - faccio partire il codice da main, come se fosse scritto in C, sostituendo
      .glob _start ————————-> .glob main
      _start main:
      poi assemblo il sorgente con as o col gcc che chiama as
      `as -o print.o –gstabs print.s` o
      `gcc -c -g -o print.o print.s`

- **Interrupt**: lanciati con `int n` , n identifica l'interrupt da eseguire. Gli eventuali parametri, in assembly, sono messi in dei registri prima di chiamare l'interrupt, oppure in un'apposita area di memoria, e l'indirizzo di inizio di questa area viene messo in alcuni registri.
- **System call**: un programma in assembly che opera su Intel invoca una system call invocando `int 0x80` , mettendo prima in EAX l'identificatore della system call da chiamare, ed eventuali parametri in altri registri. int 0x80 serve per invocare System calls su processori a 32 bit. Su processori a 64 bit int 0x80 potrebbe non leggere tutto il contenuto dei registri, ma solo i 32 bit

meno significativi. Per esempio, lo stack è collocato nella parte alta della memoria, probabilmente un tentativo di accesso genererà segmentation fault. Questo problema si risolve sotituendo int 0x80 con `syscall` (gli identificatori delle system calls sono diversi).

Linguaggio C, richiami

- **Storia**: definito inizialmente come K&R nel 1978 da Brian Kernighan e Dennis Ritchie, che lo progettarono per scrivere il SO Unix. Nel 1989 ne derivò uno standard internazionale chiamato ANSI C (American National Standard Institute), o C89. L'hanno successivo fu promulgato col nome C90 dall'ISO (International Organisation for Standardisation). Successivamente sono stati pubblicati ISO C99, GNU99…
- **Compilazione**: codice sorgente → PREPROCESSORE → file temporanei → COMPILATORE [→ codice in assembly → ASSEMBLER] + eventuali librerie → object → LINKER → eseguibile. Il passaggio dall'assembly è usato al solo scopo di debug.
- **Compilatori**: cl.exe per sistemi dos-windows, gcc per Unix. Svolgono tre operazioni: preprocessing, compilazione (per generare moduli oggetto), linking (dei vari moduli oggetto e librerie). In Unix sono svolte da programmi diversi (cpp, gcc, ld), attivati da un compilatore. In Windows, le prime due sono fatte dallo stesso programma, e il linker è chiamato dal compilatore. Esiste un utility (make) che permette di eseguire le varie fasi solo per i file recentemente modificati.
- **Variabili globali**
  - extern: definendo una variabile globale con **extern *tipo nome***, il linker sa che non troverà quella variabile in quel file, e la dovrà andare a cercare in qualche altro file. Il file corrente è autorizzato ad usarla.
  - static: visibili solo all'interno del file in cui sono definite. Non è conservata nello stack, ma in una locazione di memoria permanente.

GNU C Compiler e Makefile

- **Opzioni gcc preprocessore**
  - -DNOMESIMBOLO: definisce NOMESIMBOLO senza specificarne un valore
    - =VALORE: assegna un valore (stringa)
    - Feature test macros: esempio: nanosleep() è definita in <time.h>, definito nello stadard POSIX.1b specificato in IEEE Standard 1003.1b-1993. nanosleep() è protetto da una direttiva del preprocessore che controlla l'esistenza di _POSIX_C_SOURCE e abbia valore superiore a 199309L. Per abilitarne la compilazione, si può fare la define di questo

simbolo nel .c, prima di includere time.h, oppure eseguendo gcc specificando -D_POSIX_C_SOURCE=199309L.

  - -D’NOMEMACROCONARGOMENTI=IMPLEMENTAZIONE’ definisce una macro
  - -UNOMESIMBOLO elimina la definizione di NOMESIMBOLO
  - -I*Percorso* indica il percorso relativo o assoluto per la directory degli header files dell’utente. Può essere usata più di una volta

- **Opzioni gcc compilatore**
  - -E: fare solo preprocessing, mandando in stdout il codice sorgente C generato dal preprocessore
  - -S: tradurre il file nel corrispondente assembly
  - -masm=dialect: usato insieme a -S per generare assembly nel dialetto specificato (dialect = intel per Intel, att per AT&T (default))
  - -c: per compilare e non procedere nel linking
  - -o nomefile: per mettere il risultato della fase realizzata nel file nomefile
  - -ansi: compilare secondo lo standard c90
  - —std=STANDARD specifica lo standard da usare (ansi c89 c90 gnu99)
  - -Wpedantic o —pedantic: ordina di avvisare nel caso di programmazione in c90 quando si individuano istruzioni che rappresentano estensioni di C
  - -Wall: avvisa quando trova istruzioni non sbagliate ma potenzialmente pericolose e che rendono il codice meno leggibile
  - -Werror: trattare ogni warning come errore
- **Opzioni gcc linker**
  - -L*Percorso*: indica il percorso relativo o assoluto per la directory degli header files dell’utente. Può essere usata più di una volta
  - -lnomelibreria: alla fine della riga di comando, indica di utilizzare una libreria tramite il nome ristretto: libpippo.a → pippo
  - -Wl,-soname,*nomelibreria*: utilizzato per creare libreria dinamica
  - -Wl,-rpath,*RunTimelibDir*: utilizzato quando si crea un eseguibile per specificare la directory di una libreria dinamica da usare collocata fuori dalla directory predefinita.
- **Makefile**: il file dependency system di Unix è usato per automatizzare il corretto aggiornamento di file con dipendenze. Un file A (target) dipende da un file B se il contenuto di A dipende dal contenuto di B. Se B viene modificato, ne risente anche A. Se io compilo e linko i file, nel caso di una modifica di B, dovrei ricompilare e linkare sia A che B.

- Rigenerazione di un target: se il file non esiste, o è più vecchio di una delle sue dipendenze. Necessaria un'operazione a cascata per rigenerare tutti i file che dipendono dal target.
- un Makefile è un file di testo che contiene le make rules, che descrivono la struttura ad albero delle dipendenze del progetto, e le regole di generazione dei target di ogni dipendenza. Ogni make rule indica uno o più target a cui si riferisce (target list), le dipendenze di ogni target (dipendency list) e le operazioni che devono essere svolte per la rigenerazione (command list)
- Formattazione MAKEFILE: da ricordare che il make costruisce il dependency tree usando come root la prima regola letta, poi vengono aggiunte ricorsivamente le sue dipendenze, e a loro volta si cerca le dipendeze nelle target list. Mediante visita Bottom Up, si parte dalle foglie e si confrontano le date di ultima modifica di figlio e padre. Se il padre è più vecchio, o assente ***(come il mio)*** allora viene eseguita la command_list che ha il padre come target. Mettendo un - davanti ad un comando, anche se quel comando restituisce un errore (facendo in teoria terminare il make), il make continua la sua esecuzione.

```
all: target_da_ottenere
#per buona programmazione scritto in prima riga, ha una lista di comandi vuota
target_list: dependency_list
[TAB] comando
...
[TAB] comando
[riga vuota]
target_list: dependency_list
[TAB] -comando
[newline]

#commento eventuale
```

- è possibile specificare target che non sono file, al solo scopo di eseguire delle azioni. Questi target non hanno dipendenze, e non possono apparire in cima al file, per essere eseguiti solo se chiamati esplicitamente tramite make. Se però nella directory viene creato un file che si chiama come il target fittizio, i comandi non verranno mai eseguiti (si può risolvere esplicitando i target fittizzi con .phony: il make non utilizzerà come target le dipendenze di un file phony). In caso di regole implicite il make tenta comunque di risolvere il target fittizio, sprecando tempo.

```
TARGET FITTIZIO
clean: NO_DIPENDENZE
  rm *exe *.o *~

TARGET FITTIZIO con .phony
.PHONY : clean
clean: NO_DIPENDENZE
  rm *exe *.o *~
```

    - Comando make: per la costruzione dell'albero e l'esecuzione delle command_list, si lancia il comando `make -f nomeMakefile` . Senza -f e parametro, make cerca nella directory corrente un file di nome Makefile e usa quello. make stampa sullo stdout i comandi che esegue, con -n li stampa senza eseguirli. Aggiugnendo un file alla fine, specifico che quel file verrà usato come root. Inserendo "all" alla fine, specifico di usare i target indicati da all.
        - variabili: il Makefile non è uno script. Si possono specificare delle variabili all'inizio del Makefile, per farle utilizzare dal make, ma devono essere costanti. I comandi nel Makefile vengono eseguiti in shell, non in bash, quindi non sono utilizzabili espressioni condizionali con [[ ]], ma solo con [ ], e non esiste -e per echo. Se voglio eseguirli in bash, devo aggiungere all'inizio del Makefile la variabile SHELL=/bin/bash.
- **Dettagli importanti**
    - il make cerca il makefile e lo esegue nella directory corrente. Se si desidera cambiare questa directory, nìbasta aggiungere -C *nomedirectory* o —directory=*nomedirectory* prima di cercare il Makefile
    - se definisco una variabile del make con lo stesso nome di una della shell, nascondo la seconda (con -e oppure —environment-overrides faccio il contrario). Se premetto al nome della variabile un $, posso selezionare entrambe le variabili omonime. Con $$, seleziono solo la variabile shell ($$PATH, anche dentro alla lista dei comandi del Makefile).
    - Ogni riga di comando nella command_list viene eseguita in una shell separata figlia della shell che esegue il make. Se contiene più comandi, ognuno verrà eseguito in una subshell. Una riga di comando termina con l'EOL. Si va a capo con \, come nelle macro
- **Makefile ricorsivi**: più directory separate, ognuna con il suo Makefile. Il Makefile della principale deve avere il target principale all, che deve andare in ciascuna directory e lanciare il suo Makefile, per poi mettere assieme tutti i file generati e creare il risultato finale. In questo modo bisogna elencare ogni sottodirectory. Possiamo automatizzare questo processo:

```
all:
  for DIRNAME in `find ./ -mindepth 1 -maxdepth 1 -type d -print` ;\
 do if ! make -C $${DIRNAME} ; then exit $$? ; fi ; done
  make main.exe
```

Cenni su Librerie

- **Le librerie** sono file che contengono implementazioni, in codice macchina, di alcune funzioni. Solitamente mantenute in directory predefinite, in file il cui nome inizia per "lib". L'estensione varia se la libreria è statica: .a, o se condivisa: .so. Escluso lib e estensione, quello che rimane è il nome ufficiale della libreria.

- **Librerie statiche**: a link-time (quando si genera l'eseguibile), fisicamente accorpate ai moduli e inserite nell'eseguibile generato. Il linker sa dove trovarle (generalmente /usr/lib, altrimenti -L*linktimepath*). L'eseguibile avrà da subito tutto il codice delle librerie caricato in memoria.
- **Librerie condivise**: per ciascuna chiamata di funzione, il linker inserisce nell'eseguibile una parte di codice macchina (stub) e la posizione a run-time (esecuzione dell'eseguibile) su disco della libreria. A run-time, quando sarà invocata la funzione, lo stub cercherà la libreria sul disco, e la caricherà in memoria. (Il linker deve sapere la posizione della libreria sia a link-time che a run-time: a link-time perchè deve verificare che esista, fornisca la funzione richiesta e deve copiare lo stub della libreria, a run-time perchè deve inserire la sua posizione nel codice assieme allo stub). -L*linktimepath* per la posizione delle librerie a link-time, -Wl,-rpath,*runtimepath* a run-time. esempio: `gcc altriflag -o prog.exe prog.o -L/home/studente/temp -Wl,-rpath,/home/studente/lib -lmia` .
  - Il linker cerca sempre di utilizzare librerie condivise. Se non ne trova, usa quelle statiche. Se si vuole usare solo librerie statiche, va specificato al gcc il flag -static
- **Utilities per librerie**: librerie e file binari eseguibili sono file strutturati secondo un formato standard, solitamente elf.
  - **ldd *nome_eseguibile*** o **ldd *percorsolibreria***: stampa quali librerie condivise sono necessari per quell'eseguibile/libreria.
  - **readelf -d *percorsolibreriaoeseguibile***: quali informazioni di linking dinamico sono state inserite nell'eseguibile/libreria.
  - **file *nomefile***: identifica tipo di file, codifica dei caratteri, MIME, metadati, tipo di ELF, il tipo di linkaggio, la build, la versione, il formato (x86-64)
  - **nm *libreria***: eseguibile che elenca tutti i simboli per i quali esistono dei riferimenti nella libreria.
    - *status symbolname*, per ogni valore di status:
      - A→il valore del simbolo è assoluto, e non verrà cambiato da linkaggi futuri
      - B→sezione di dati non inizializzati
      - D→sezione di dati inizializzati
      - T→normal code section (simbolo definito nella libreria)
      - U→simbolo usato ma non definito, è contenuto in un'altra libreria
      - W→simbolo doppiamente definito. È definito in una forma weak, in modo tale da poter essere sostituito da un'altra definizione.
    - nm -D: elenca i simboli per i quali esistono dei riferimenti nella libreria che viene passata come argomento

- **lsof *listing open files***: elenca tutte le librerie dinamiche presenti in memoria
  - lsof | grep mem | grep "\.so"| awk '{print $9}'| sort| uniq

---

Thread e POSIX Thread

non ti senti in colpa a leggere? Cosa penserebbe Lucchi di te? Non sei stato attento?

- **Thread**: singolo flusso di informazioni, all'interno di un processo, che lo scheduler può far eseguire concorrentemente al resto del processo. Per fare ciò, il thread deve possedere un proprio contesto (PID, PC, RS, Stack, Codice, Dati, File Descriptor, Entità IPC, azioni dei segnali, una variabile errno). Tutti i thread di uno stesso processo condividono le risorse del processo (dati globali, tabella dei descrittori di file e CPU) ed eseguono nello stesso spazio utente. Dati globali ed entità del thread sono il suo stato di esecuzione.
- **Pro e contro**
  - Pro: visibilità dei dati globali, più flussi di esecuzione, gestione semplice di eventi asincroni, comunicazioni veloci (stesso spazio di indirizzamento), context switch veloce (viene mantenuta buona parte dell'ambiente)
  - Contro: concorrenza (occorre gestire la mutua esclusione dei dati), il SO deve caricare più funzioni di libreria e system calls contemporaneamente (queste devono essere rientranti (thread safe call)).
- **Operazioni atomiche**: un'operazione indivisibile (nessun'altra operazione con gli stessi dati condivisi può cominciare prima che la prima sia finita). È però possibile interromperla per permettere l'esecuzione di un'operazione più urgente, facendo riprendere l'operazione atomica dall'inizio. Le condizioni iniziali potrebbero essere cambiate nel frattempo, ma il risultato di un'operazione atomica sarà sempre lo stesso. Le istruzioni in C non sono atomiche, gli assegnamenti non sono atomiche (tranne nel caso specifico che la variabile a cui assegnamo una costante ha dimensione minore dell'ampiezza del bus dati), le macro in assembly non sono atomiche, neanche l'istruzione inc lo è (lettura del dato, aumento, salvataggio). La copia di un dato da registro a memoria è atomica, se l'ampiezza del bus è maggiore o uguale alla dimensione del registro.
- **Compare-exchange**: I processori, nel loro livello ISA, mettono a disposizione istruzioni macchina atomiche per eseguire flussi atomici più complessi che si basano sul valore di una variabile di guardia. Queste istruzioni variano da processore a processore, Intel: istruzione cmpxchg e prefisso lock che modifica il comportamento di questa istruzione (in altri processori si parla dicompare-and-swap). `lock cmpxchg` vuole due argomenti espliciti (indirizzo di una variabile intera, indirizzo del registro con il valore da assegnare alla variabile) e uno implicito (registro EAX con il valore da confrontare). Questa istruzione preleva il valore dalla variabile, lo confronta con quello in EAX, se sono uguali assegna alla variabile il valore nel registro. È la verifica in modo atomico del valore di una variabile globale, che indica se una risorsa è occupata o meno.

- **System call e funzioni non rientranti (non thread safe)**: le syscall dovrebbero essere costruite per poter essere utilizzate da più thread contemporaneamente. Alcune non lo sono. Le implementazioni thead per soddisfare gli standard POSIX offrono funzioni thread safe. Tutte le funzioni ANSIC e POSIX.1 sono thread safe (tranne strerror inet_ntoa asctime ctime getlogin rand readdir strtok ttyname gethostXXX getprotoXXX getservXXX). Solitamente il nome delle funzioni rientranti finisce con "_r". char *strerror(int errnum) non è rientrante, ma strerror_r, definito con _POSIX_C_Source ≥ 200112L è rientrante. char *inet_ntoa() non è rientrante: scrive il suo risultato in un buffer visibile al processo e restituisce un puntatore a tale buffer.
- **POSIX Thread**: lo standard IEEE POSIX 1003.1c (1995) specifica l'interfaccia di programmazione (Application Program Interface) dei thread, noti come Pthread (definizioni nel file pthread.h). Le API per Pthread distinguono le funzioni in tre gruppi:
  - thread management: creare, eliminarem attendere
  - Mutexes: funzioni per sincronizzazione semplice chiamata "mutex" (mutua esclusione). Creare e eliminare struttura per mutua esclusione, acquisire e rilasciare tale risorsa
  - Condition variables: per sincronizzazione più complessa. Creare e eliminare struttura per sincronizzazione, attendere e segnalare le modifiche alle variabili.
- **Identificatori Pthread**:
  - pthread_: gestione pthread in generale
  - pthread_attr_: gestione proprietà pthread
  - pthread_mutex_: gestione mutua esclusione
  - pthread_mutexattr_: proprietà delle strutture per mutua esclusione
  - pthread_cond_: gestione variabili di condizione
  - pthread_condattr_: proprietà variabili condizione
  - pthread_key_: dati speciali dei thread
- **pthread.h**: compilabile in dialetto -ansi, occorre definire -D_POSIX_C_SOURCE=1. Per poter usare le funzioni pthread_rwlock_* o strerror_r, bloccate da -ansi, occorre usare al suo posto -std=g99, oppure _POSIX_C_SOURCE≥200112L. In un programma che usa pthread, è bene definire a inizio file `#define _THREAD_SAFE` o (_REENTRANT). In fase di linking, con gcc, usare `-lpthread` alla fine della riga.
- **API per pthread**:
  - int _pthread_create(…): crea un thread e lo mette a disposizione dello scheduler. Si aspetta come attributi (1)un puntatore ad un identificatore di thread, in cui verrà scritto l'identificatore del thread creato (2)caratteristiche particolari, NULL per default (3) nome della procedura da far eseguire (4) puntatore area di memoria allocata dinamicamente che contiene i parametri

da passare al thread (che andrà liberata). Restituisce 0 se ha creato il pthread, ≠0 altrimenti (e identifica l'errore ottenuto).

    - Si potrebbe voler salvare un intero in un puntatore. La dimensione dei pointer dipende da processore e BUS, quindi si usano i tipi intptr_t e uintptr_t (definiti in stdint.h). Per stampare questi valori si usano delle macro contenute in inttypes.h. In questa maniera, se vogliamo passare SOLO un intero ad un thread, al posto di allocare altra memoria, si usa un puntatore con all'interno un intero, al posto di un indirizzo.

  - int pthread_exit(void *retval): termina l'esecuzione del thread che la chiama, immagazzina l'indirizzo retval (che contiene un risultato) e lo può restituire ad un altro thread che attende la sua fine. Main è particolare: anche lui è un thread, se il main termina senza chiamare questa funzione, tutti i thread creati da lui muoiono con lui (come un pthread). Se invece la chiama, i thread continuano a vivere fino alla terminazione (un pthread normale attende la fine dei pthread figli).
  - pthread_t: tipo di dato che contiene l'identificatore univoco di un pthread.
    - pthread_t pthread_self() restituisce l'identificatore del thread che la chiama
    - int pthread_equal (pthread_t p1, pthread_t p2) restituisce 1 se p1 e p2 sono uguali
  - pthread_join(pointer_da_attendere): attendere la fine di un Pthread. Il valore restituito da un Pthread è conservato nello stack del thread, e viene restituito la thread che richiama join. La parte di memoria del thread da attendere rimane in memoria fino ad un join, o fino alla terminazione dell'intero processo. Per liberare la memoria subito dopo la terminazione di un thread, si usano thread detached, che però non possono ritornare risultati.

- **Mutua esclusione**: una variabile mutex (pthread_mutex_t) serve per regolare l'accesso a dati che non possono essere usati da più thread contemporaneamente. Ogni thread deve usare una pthread_mutex_lock per bloccare l'accesso agli altri thread, e una pthread_mutex_unlock per liberarla.
  - Creare e distruggere variabili FAST Mutex
    - inizializzazione statica (in dichiarazione):
      `pthread_mutex mut = PTHREAD_MUTEX_INITIALIZER;`
    - inizializzazione dinamica (dopo la dichiarazione):
      `pthread_mutex_t mut;`
      `pthread_mutex_init(& mut, &attribute);`
    - distruzione
      `pthread_mutex_destroy(&mut);`
  - int pthread_mutex_lock (pthread_mutex_t *mutex);
    se la variabile è libera, la blocca e restituisce 0, altrimenti il codice di errore, sennò sospende

il chiamante e aspetta la liberazione. Se un thread blocca due volte la stessa variabile senza liberarla, il thread si blocca per sempre.

  - int pthread_mutex_unlock (pthread_mutex_t *mutex);
    deve essere chiamata su una mutex bloccata dal chiamante, e restituisce 0, o il codice di errore. Se viene chiamata su una mutex bloccata da un altro processo, il comportamento è impredicibile.
  - int pthread_mutex_trylock (pthread_mutex_t *mutex);
    funziona come la lock, ma se la variabile è già bloccata torna subito EBUSY. Causa busy waiting (si controlla sempre la disposizione di una variabile)

- **condition variables**: consentono di effettuare sincronizzazioni tra pthread mentre questi già detengono la mutua esclusione, bloccandoli e facendoli continuare in base alle condizioni definite dal programmatore.
  - pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex): blocca un thread finchè un altro lo risveglia. Prima deve prendere la mutua esclusione sul secondo argomento. La wait lo blocca e rilascia la mutua esclusione, poi si mette in attesa. Continua solo dopo che un altro thread l'ha risvegliata con pthread_cond_signal(), e riesce a riprendere la mutua esclusione.
  - pthread_cond_signal(pthread_cond_t *cond): controlla se c'è qualche bloccato con wait() sulla condition specificata, e lo sblocca, poi signal termina (anche se non trova thread).
  - pthread_cond_broadcast(): signal a tutti i thread bloccati
    - prima di chiamare la signal o la broadcast, il thread chiamante deve detenere la mutua esclusione sulla mutex specificata nelle wait con lock().
  - Creare e distruggere
    - inizializzazione statica:
      `pthread_cond_t mycond = PTHREAD_COND_INITIALIZER;`
    - inizializzazione dinamica:
      ```
      pthread_cond_t mycond;
      pthread_cond_init(& mycond, &attribute);
      ```
    - distruzione
      `pthread_cond_destroy(&mycond);`

Programmazione concorrente

- i thread vengono eseguiti in interleaving dal SO tramite time slice, che il SO gestisce in maniera non controllabile dai thread.

- **Race condition**: situazione in cui più thread concorrono accedendo a dati condivisi senza controlli di sincronizzazione. Il contenuto finale dei dati condivisi dipende dalla sequenza di esecuzione dei thread (può causare inconsistenza). Si risolvono questi problemi eseguendo le sequenze di istruzioni come se fossero atomiche (senza interleaving)
- **Sezione critica**: segmento di codice in cui un thread ha la necessità di accedere ad una risorsa condivisa avendo la certezza di essere l'unico a farlo (evitando stati di inconsistenza). Bisogna garantire mutua esclusione e liveness (il thread non può rimanere indefinitivamente bloccato, due proprietà: progress, perchè la decisione su quale sarà il primo thread in attesa ad utilizzare una sezione critica non occupata non può essere rimandata indefinitivamente, e bounded waiting, perchè una richiesta d'accesso di un thread può essere negata un numero finito di volte se la critical section è libera). Problemi da evitare:
  - Busy waiting: un thread attende il verificarsi di una condizione continuamente finchè la condizione diventa vera, consumando CPU
  - Deadlock: due o più thread aspettano indefinitamente un evento che può essere causato solo da uno dei thread bloccati
  - Starvation: un thread attende una risorsa che gli viene sottratta continuamente da processi con priorità maggiore
- **Atomicità in senso esteso**: esecuzione di una sezione critica che non può essere interrotta (se non da una porzione di codice che non appartiene a nessuna sezione critica)

Gestione della memoria

- **Binding address**: l'associazione di indirizzi di memoria fisica ai dati e alle istruzioni di un programma. Se durante la compilazione gli indirizzi calcolati saranno sempre gli stessi ad ogni esecuzione del programma (codice assoluto), semplice, veloce e non richiede hardware speciale, ma non funziona con la multiprogrammazione. Se durante il caricamento del programma in memoria, gli indirizzi sono relativi (codice rilocabile). Il loader si preoccupa di aggiornare i riferimenti agli indirizzi. Non richiede hardware particolare, consente la multiprogrammazione, ma richiede una traduzione degli indirizzi. Se durante l'esecuzione, viene fatto dalla memory management unit (MMU):
  - ogni processo è assegnato ad uno spazio di indirizzamento logico, a cui corrisponde un indirizzamento fisico. In un processo ci sono indirizzi logici, che sono riferimenti che la MMU traduce in indirizzi fisici (se il valore del registro di rilocazione è R, lo spazio logico 0-max è tradotto a R-R+max). Il registro limite viene utilizzato per implementare meccanismi di protezione della memoria.
- **Allocazione di memoria**: una delle funzioni principali del gestore di memoria, consiste nel reperire ed assegnare uno spazio di memoria fisica.

- contigua: lo spazio assegnato ad un programma è formato da celle contigue (La MMU gestisce la conversione degli indirizzi in modo coerente all'allocazione)
- non contigua: le aree di memoria sono separate
- statica: un programma mantiene la propria memoria dal caricamento alla terminazione
- dinamica: un programma può essere spostato nella memoria durante l'esecuzione
- a partizione fisse (statica e contigua): la memoria è divisa in partizioni, ogni processo è caricato in una delle partizioni libere che ha dimensione sufficiente a contenerlo. Molto semplice, ma spreca memoria e il parallelismo è limitato dal numero di partizioni. Sistemi monoprogrammati → una partizione.
  - frammentazione interna: spazio sprecato, che è la differenza fra la dimensione del programma allocato e la dimensione della partizione
- a partizioni dinamiche: sempre statica e contigua, ma esistono delle politiche per la scelta delle aree da utilizzare. Quelle non utilizzate sono derivate dalla terminazione dei programmi.
  - First Fit: scorre la lista dei blocchi liberi, fino a trovare il segmento vuoto grande abbastanza da contenere il processo
  - Next Fit: come First Fit, ma parte dall'ultima allocazione (prestazioni peggiori)
  - Best Fit: seleziona il più piccolo fra i blocchi liberi, sufficiente a tenere il processo (più lento, genera più frammentazione di First Fit)
  - Worst Fit: seleziona il blocco più grande (rende difficile l'allocazioni di processi grandi)
  - Miglioramenti: per ottimizzare il costo di allocazione, si può mantenere una lista separata dei blocchi liberi, ed eventualmente ordinarla per dimensione (è richiesta un'unità minima di allocazione)
- Frammentazione esterna: dopo numerose allocazioni e deallocazioni, la memoria è suddivisa in piccole aree
  - Compattazione: se è possibile rilocare i programmi in esecuzione, allora è possibile compattare la memoria (spostare i programmi per riunire le aree inutilizzate). Operazione molto onerosa, e non utilizzabile in sistemi interattivi (i processi dovrebbero fermarsi).
- Paginazione: riduce la frammentazione interna, e minimizza la frammentazione esterna. Necessita di hardware adeguato. Lo spazio di indirizzamento logico di un processo viene suddiviso in un insieme di blocchi di dimensione fissa chiamati pagine. La memoria fisica viene suddivisa in un insieme di blocchi della stessa dimensione delle pagine, chiamati frame. Quando un processo viene allocato in memoria, viene reperito un numero sufficiente di frame per contenere le pagine del processo. La dimensione delle pagine deve essere una potenza di due, a seguito del trade-off (dimensione tabella pagine - frammentazione interna) si sceglie una dimensione, tipicamente 1, 2, 4 KB o 4MB

        - Tabella delle pagine: se conservata in registri del modulo MMU o della CPU sarebbe troppo costosa, ma se conservata in memoria raddoppierebbe gli accessi (risolto con una cache TLB (Translation lookaside buffer) per la tabella).
    - **Segmentazione**: la memoria associata ad un programma è divisa in aree dal punto di vista funzionale. Uno spazio di indirizzamento logico è dato da un insieme di segmenti (aree di memoria di elementi affini logicamente continua). Ogni segmento ha un nome e una lunghezza. Ogni riferimento è dato da <nomesegmento, offest>. La segmentazione sta al programmatore. Tipicamente un segmento è 64KB-1MB. Consente la condivisione di codice e dati.
    - S**egmentazione + paginazione**: ogni segmento è diviso in pagine allocate in frame non necessariamente contigue. La MMU deve avere il supporto per entrambe. Benefici: condivisione, protezione e niente frammentazione esterna.
    - **Memoria virtuale**: tecnica che permette l'esecuzione di processi che non sono completamente in memoria. Permette di eseguire concorrentemente processi che necessitano più memoria di quella disponibile, ma può diminuire le prestazioni se non gestita bene. Le istruzioni e i dati devono essere in memoria, ma non è necessario che l'intero spazio di indirizzamento logico di un processo sia in memoria (i processi non lo usano tutto contemporaneamente).
      Ogni processo ha accesso ad uno spazio di indirizzamento virtuale anche più grande di quello fisico. Gli indirizzi virtuali possono essere mappati in memoria principale o in memoria secondaria. Se bisogna accedere a dati in quest'ultima, questi vengono trasferiti nella prima. Se è piena, si spostano in secondaria i dati meno utili in primaria.
        - Demand paging: si utilizza la paginazione, ma alcune pagine possono essere in memoria secondaria (swap area). Nella tabella delle pagine si utilizza un bit v (valid) che indica se la pagina è in memoria centrale. Se il processore tenta di accedere ad una pagina non in memoria, genera un page fault, e il pager del SO carica la pagina mancante e aggiorna la tabella. Senza il paging demander, si usava lo swap per copiare l'intera area di memoria usata dal processo dalla memoria secondaria alla principale (swap-in) e viceversa (swap-out). La paginazione su richiesta è uno swapdi tipo lazy. Alcuni SO chiamano il pager come swapper. Necessita di un algoritmo per l'allocazione dei frame, e uno per la sostituzione delle pagine.
            - Algoritmo: individua la pagina in memoria secondaria, cerca un frame libero (se non ce ne sono, richiama algoritmo di rimpiazzamento, aggiorna la tabella delle pagine, invalida pagina vittima, se questa è stata variata, scrive la pagina su disco, e aggiorna la tabella dei frame (frame libero)), poi aggiorna la tabella dei frame (frame occupato), legge la pagina da disco, aggiorna la tabella delle pagine e riattiva il processo.

        - È possibile usare un on demand puro, avviando un processo senza pagine in memoria, e caricandole solo quando viene generato un page fault.
        - Bisogna cercare di tenere basso il numero di page fault, o ci sono troppi accessi in memoria. Si può evitare di ricaricare una pagina se questa non è stata modificata, aggiungendo un dirty bit che viene impostato a 1 se la pagina viene modificata.

File System

- astrazione del modo in cui i dati sono allocati e organizzati in un dispositivo di memoria di massa. Il SO offre una visione logica uniforme della memorizzazione delle informazioni sui diversi supporti. Organizzato in una struttura gerarchica di directory, in cui sono contenuti i file.
- **File**: insieme di informazioni correlate e registrate nella memoria secondaria con un nome. Più piccola unità di memoria secondaria assegnabile all'utente. Attributi: nome, tipo, locazione, dimensione, protezione (chi può leggere, scrivere o eseguire), ora, data e identificazione dell'utente (creazione, ultimo utilizzo e ultima modifica). Il sistema mantiene in memoria una tabella dei file aperti, per ridurre le fasi di ricerca. A ciascun file sono associate diverse informazioni: puntatore alla posizione corrente (più puntatori alla posizione corrente se più processi operano sul file), contatore delle aperture (per controllare se sono tutte chiuse, e rimuovere il file dalla tabella), posizione su disco del file.
    - Il SO non si occupa di gestire la struttura interna dei file, per lui sono insiemi ordinati di byte. La loro struttura è delegata alle applicazioni. Si limita così la dimensione del SO, e la flessibilità nel definire nuove strutture. Tutti i SO supportano però almeno il formato dei file eseguibili.
- **Metodi di accesso ad un file**:
    - sequenziale: le informazioni vengono elaborate in ordine (nastri). Il più utilizzato. read: legge la prossima porzione e avanza il puntatore. write: scrive i nuovi dati in coda al file e avanza l'EOF. reset: riposiziona il puntatore all'inizio.
    - diretto: il file è costituito da un insieme ordinato di blocchi a cui si accede direttamente (dischi). Il numero di blocco è assegnato in modo relativo. Le operazioni devono specificare in quale blocco leggere. read(n) legge il blocco n, write(n) scrive il blocco n, seek(n) si posiziona sul blocco n. Si può usare per implementare l'accesso sequenziale.
    - attraverso indice: è associato un indice al file per velocizzare le ricerche (database). L'indice contiene una o più chiavi di ricerca a cui sono associati i puntatori ai blocchi del file.
- **Allocazione**
    - File Control Block: descrittore in cui vengono memorizzati i file, contiene nome, proprietario, dimensioni, permessi di accesso e posizioni dei blocchi. Il descrittore occupa un blocco su disco. Nei sistemi unix si chiama "inode".

- Allocazione contigua: ogni file deve occupare un insieme di blocchi contiguo. L'accesso sequenziale al blocco b+1 seguente all'accesso al blocco b non necessita di spostamento della testina. L'accesso diretto è fatto semplicemente calcolando la posizione assoluta in base alla relativa. Il file è definito dalla posizione iniziale e dalla lunghezza. La difficoltà sta nel reperire una porzione di disco sufficientemente grande da contenere il file (il problema è simile all'allocazione di memoria, di solito si provano first fit e best fit, con deframmentazione per risolvere quella esterna). Per estendere un file allocato in uno spazio di dimensione simile alla sua, spesso occorre copiarlo in una allocazione più ampia. Se si volesse usare già partizioni più grandi, si incorrerebbe in frammentazione interna, non risolvibile.
- Allocazione concatenata: ogni file è costituito da una lista concatenata di blocchi del disco che possono essere distribuiti in qualunque parte del disco stesso. La directory contiene un puntatore al primo blocco. Niente frammentazione esterna perchè ogni blocco è allocato singolarmente, niente frammentazione interna perchè non c'è bisogno di allocare il file, ma l'accesso è più complicato: comporta spostamenti di testine in lettura sequenziale, occorre scorrere il file nell'accesso diretto, e usa molto spazio per i puntatori.
  - Clustering: per risolvere questi problemi, di solito si allocano gruppi (cluster) di blocchi. Meno puntatori, meno spostamenti della testina, più semplice la gestione dei blocchi liberi, ma riappare la frammentazione interna (parte del blocco può non essere utilizzata)
- Allocazione Indicizzata: molti problemi della concatenata possono essere risolti inserendo tutti i puntatori ai blocchi in un'apposita tabella detta index block. Ogni file ha il proprio index block in cui l'i-esimo elemento punta all'i-esimo elemento del file. Accesso diretto senza frammentazione, ma comporta uno spreco di spazio per l'index block. Per la memorizzazione dell'index block:
  - schema concatenato: il blocco indice occupa esattamente un blocco. Se non è sufficiente, il suo ultimo puntatore punta ad un altro blocco.
  - indice multilivello: il blocco indice di primo livello punta ad altri blocchi di secondo livello, etc
  - schema combinato: usato da inode, in base alla dimensioni del file si usano una parte dei puntatori per puntare direttamente ai blocchi del file, e un'altra parte (che cresce con le dimensioni del file) punta a indici multilivello.

Virtualizzazione di sistemi

- **Virtualizzazione**: eseguire uno o più sistemi operativi su un unico pc, in un ambiente che prende il nome di macchina virtuale (VM). Il sistema operativo che esegue la macchina virtuale è detto host, la macchina virtuale è il guest. Il codice della macchina guest è eseguito dal sistema host per simulare una macchina reale al sistema guest. Tramite emulazione di processore, ogni istruzione del sistema guest è tradotto in una sequenza di istruzioni della macchina host.

- **Virtualizzazione HW vs OS**:
  - Hardware: viene simulata un'inetra macchina (livello Hardware), all'utente viene presentata un'interfaccia su cui installare un sistema, quindi una CPU e risorse HW virtuali. I due SO eseguono in modo concorrente sullo stesso HW e possono essere eterogenei. L'hypervisor si occupa di multiplexare l'accessoalle risorse HW, garantire protezione e isolamento fra macchine. Alto overhead per context-switch e di memoria.
  - Container: all'utente viene presentata una partizione del SO corrente, su cui installare ed eseguire applicazioni che rimangono isolate. Un solo kernel e più istanze di user-space (container) che sono indipendenti. Ogni container ha un file system, IPC e network, e fornisce un sistema di gestione delle risorse (CPU, memoria, rete, I/O). Basso overhead per context-switch e di memoria, non può però ospitare SO diversi.
- **Tipi di container**: In sintesi, si crea uno spazio utente isolato, con proprie interfacce di rete, librerie e file, isolando un'applicazione dal resto del SO. Più container possono appoggiarsi ad uno stesso spazio kernel, risparmiando disco e condividendo servizi di base. SI può replicare un container più volte, anche su VM diverse.
  - Docker: si appoggia a Linux, che fornisce un supporto per container detto LXC (Linux Container).
  - Hyper-V: Microsoft, che fornisce unità di isolamento dette Container, ma che in realtà sono macchine virtuali.
  - Container di Windows Server.
- **Docker**: i container Docker sono eseguiti a partire da immagini di container scaricabili dal repository Docker Hub. Dopo che l'immagine è scaricata, la runtime machine di docker crea il container, e la copia dell'immagine viene salvata su un local registry. Per eseguire l'immagine hello-world: `docker run hello-world`
  - Container interattivi: per eseguire un container usando l'ultima immagine disponibile di ubuntu lanciamo `docker run -it –name myubuntu ubuntu` . -i -t servono per interagire con una shell bash all'interno del container. —name myubuntu assegna il nome al container, altrimenti ne genera uno a caso. Per terminare un container, digitare `exit`. Ripetendo la run, si noterà che pur usando la stessa immagine, le eventuali modifiche precedentemente apportate prima della exit non saranno presenti.
  - Installare applicazioni in un container: installare node.js e altre utility

```
apt update
apt install vi wget curl
apt install node.js
```

Configurare poi il server http dinamico editando /root/index.js

```
var http = require('http');
  http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('ciao a tutti\n');
}).listen(8888);
console.log('Server running at http://127.0.0.1:8888/');
```

Poi far partire node.js e verificare che funzioni

```
node index.js &
curl http://127.0.0.1:8888
```

- Salvare i cambiamenti: creare una nuova immagine locale

```
docker commit -m "added node.js" -a "Vic" myubuntu vic/ubuntuvic
per rieseguirlo:
docker run -it --name myubuntu vic/ubuntuvic
```

- Container in background:

```
docker run -it -d --rm -p 80:8888 --name myubuntu2 vic/ubuntuvic node index.js
```

-d per eseguire il container in background, —rm per eliminare il container quando termina, -p collega la porta 80 dell'host alla porta 8888 del container, node è il comando da lanciare non appena il container viene creato e parte.

- **KUBERNETES**: fornire automaticamente scalabilità e resistenza ad applicazioni basate su container docker in esecuzione su cluster di host fisici e virtuali
  - Le applicazioni per kubernetes sono costituite da pods. Un pods è un insieme di container che cooperano come se si trovassero in un unico host fisico e svolgono nel complesso un certo servizio. Un pod può essere replicato in più istanze per distribuire il carico. I pods possono esporre porte di protocollo per ricevere comunicazioni da container di altri pods.
  - Kubernetes organizza degli host (nodi del cluster) fisici o virtuali per poter eseguire e replicare i pods sui nodi, distribuendo così il carico dell'applicazione. L'applicazione decide quanti pods utilizzare e kubernets decide autonomamente su quale nodi allocarli. Un host fa da controllore del cluster (control plane), decide autonomamente in quali nodi eseguire i pods e fa da punto di accesso al cluster. Gli altri host eseguono i pods. Se un pod crasha, kubernetes ne crea immediatamente una nuova istanza.
  - L'applicazione può configurare kubernetes affinchè questo, all'aumentare del carico, svolga:
    - autoscaling verticale: aumento delle risorse ai pods

        - autoscaling orizzontale: più repliche di pods attive
        - cluster autoscaling: più nodi per ridistribuire i pods. Per host fisici occorre un sistema di accensione degli host fisici guidabile via software, per gli host virtuali occore un supporto da parte del cloud fornitore delle VM, per configurare le VM.
    - Nodi Virtuali: si possono creare cluster kubernetes formato da host che siano VM. Queste possono essere create su host fisici o via software su cloud pubblici o privati. Esistono vari provider (Google Cloud, Azure di Microsoft, AWS di Amamzon), oppure si può costruire un cloud privato, partendo da macchine fisiche e gestendole con un insieme di applicativi chiamato Openstack (su Ubuntu). I provider forniscono le API per configurare le VM e di reti virtuali. Le API differiscono da cloud a cloud, Terraform è un software che astrae dai sistemi cloud sottostanti per offrire un insieme di API utilizzabili su tutti i sistemi cloud.
    - Kubernetes-as-a-service: Alcuni sistemi cloud offrono un servizio kubernetes senza che sia necessario creare esplicitamente il cluster. È il provider che costruisce il cluster in base all’applicazione del cliente, al quale vengono fornite le API.
    - Container-as-a-service: far eseguire un container, la cui immagine è fornita dallo sviluppatore, in molteplici istanze su cluster kubernetes, costriuto dal provider. Fornito da Google Container Engine (GKE), Amazon EC2 Container Service (ECS), Azure Container Service (ACS), IONOS Container Cluster.
    - Serverless o Function-as-a-service: alcuni provider offrono la possibilitò di eseguire funzioni implementate dall’utente, anche in più istanze contemporaneamente. Probabilmente il provider incorpora la funzione in un container e la esegue in un cluster. Il provider offre anche delle funzioni di libreria. Fornito da AWS Lambda in Amazon AWS, Azure Functions in Microsoft Azure e Cloud Functions in Google Cloud.