Sistemi Operativi

Leonardo Mengozzi

Titoletti indice link a rispettive sezioni, in alto a sinistra "←Indice" link a pagina Indice.

Contents

1 Bash

Variabili	Descrizione
PATH	V. Ambiente: Modificabile, sequenza di percorsi assoluti, divisi da ":", di directory contenenti eseguibili (lanciabili senza digitare path). Ricerca secondo ordine specificato in PATH, si ferma a primo eseguibile con nome uguale. Eventuale ErrorNotFoutd. Altre variabili d'ambiente: $HOME, $USER, $SHELL, $TERM.
env	Visualizza l'elenco delle variabili d'ambiente.
IFS=$' \t \n'	Contiene caratteri separatori della parole negli elenchi.
/dev/null	File speciale che scarta tutto quello che gli viene scritto.

File speciali	Descrizione
/etc/passwd	Righe sono info ogni utente divise da ":".
/etc/shadow	Righe sono password utente codificate.
/etc/group	Righe sono info ongi gruppo divise da ":".
/usr/bin/passwd	Cambia la pass utente.

Directory	Descrizione
cd *percorso*	Sposta logicamente in una diversa directory, secondo un path asssoluto o relativo.
mkdir *nomeDir*	Crea una nuova directory.
touch *nomeFile.estensione*	Crea un file vuoto nella dyrectory corrente.
rmdir *nomeDir*	Rimuove una directory solo se è vuota.
rm *file*|*dir*	Rimuove una directory vuota o un file. Parametri: • **-r** elimina ricorsivamente sotto cartelle e file. • **-f** non fa chiedere le autorizzazioni di eliminazione.
mv file1 file2|dir	Rinomina file1 in file2 o sposta file1 nella directory specificata.
cp file1 dir	Copia file1 nella directory specificata.
ls [*nomefile*]	Visualizza i files/direcotry contenuti nella directory corrente. Parametri: • **-a** mosta anche file nascosti (anche ., ..). • **-l** mostra più informazioni sui files. • **-h** rende i dati più leggibili. • **-d** fa applicare il comando alla directory e non ai file. • **-R** mostra ricorsivamente contenuto sotto directory. Se specifico un file mi dice se esiste e mi da informazioni solo di lui.
pwd	Visualizza il percorso assoluto, da / fino alla directory corrente.

Controllo Comandi	Descrizione
\	Disabila interpretazione per il carattere successivo, andata a capo, permettendo di stamparlo.
"..."	Delimita un argomento e non fa interpretare nessun comando a eccezzione dell'espansione di variabili ($..) e l'esecuzione di comandi.
'...'	Delimita un argomento e non fa interpretare nessun comando.
$'...'	Espande backslash-escaped direttamente nella stringa espansa in una single-qouted stringa. Backslash-escaped: \a, \e, \f, \r, \v, \', \b, \E, \n, \t, \\, \", \nnn, \xHH, \cn.
pre{s1,...}post	Stringa di testo racchiusa fra separatori (spazio, tab, a capo) con coppia di graffe (non precedute da $) e senza separatori. Le stringhe racchiuse dalle graffe vengono composte con il preambolo (pre) e postscritto (post), che sono opzionali. Alternative: Sono annidabili (quelle più esterne eseguite per prime). Vengono eseguite prima le brace expansions delle variable expansions. • $a_1..a_2$ lettere da a_1 a a_2 nell'alfabeto. • $n_1..n_2$ numeri compresi tra n_1 e n_2.
cmd1 ; ...	Separatore di più comandi, e dei rispettivi argomenti, scritti sulla stessa riga di comando e eseguiti dopo la terminazione del precedente (lista di comandi). L'exit status è quello dell'ultimo comando lanciato. Se racciusi da () eseguiti in una sub-schell.
cmd1 ||*cmd2*	Esegue cmd1 e solo se cmd1 fallisce (exit status≠0) esegue cmd2.
cmd1 && *cmd2*	Esegue cmd1 e solo se cmd1 ha successo (exit status=0) esegue cmd2.
[[...]]	Espressione condizionale, usa && e ||, che restituisce 0=true, altro=false.
*	Sostituito con una qualsiasi sequenza di caratteri anche vuota.
?	Sostituito con un singolo carattere (no spazio vuoto).
[c1c2...]	Sostituito con solo uno dei caratteri specificati in elenco. Alternative: • $a_1..a_2$ lettere da a_1 a a_2 nell'alfabeto. • $n_1..n_2$ numeri compresi tra n_1 e n_2. • **[:digit:]** una cifra. • **[:upper:]** un carattere maiuscolo. • **[:lower:]** un carattere minuscolo. Annidabili.

Scripting	**Descrizione**
echo *testo*	Visualizza a video la sequenza di caratteri passata fino al primo "INVIO". Se passo *"testo"* si disabilità l'interpretazione dei caratteri speciali e andate a capo. • **-e** Stampa i caratteri speciali (\...). • **-n** Non fa andare a capo.
nome=*valore*	Simboli con nome e valore, stringa modificabile, alfanumerici casesensitive. No spazi prima o dopo "=". Sono d'ambiente o ex-novo locali. Solo la bash in cui sono create le variabili le può usare. I programmi lanciati dalla bash hanno una speudocopia della bash.
$variabile	Fa l'espanzione della variabile, ovvero la stostituisce con il suo contenuto.
#variabile	Restituisce il numero di caratteri del contenuto della variabile.
${!variabile}	Fa l'espanzione della variabile che contiene il nome d'unaltra variabile con il valore di quest'ultima (riferimento indiretto). Dalla versione 2 di bash.
export nomevar |nomevar=valore	Variabile d'ambiente. Un shell figlia riceve una copia modificabile che non influenza variabile d'ambiente del padre.
unset *nomevariabile*	Elimina una variabile esistente (vuota o no). Quotare ("...") sempre variabili per evitare errori con variabili vuote o inesistenti.
${nomeVar}	Fa sostituire il nome della variabile con il valore. graffe opzionali se nome variabile seguito da uno spazio.
#...	Commeto.
#!...	Se indicato nella prima riga indica quale interprete deve eseguire lo script. Se non specificato usato quello corrente.
comando1 |*comando2*	pipe (speudo-file temporaneo): collega automaticamente l'output di un comando all'input di un altro. Unidirezionale sinDes.
script.sh c1...	Sono un insieme ordinato di caratteri separati da spazi successivi al nome del programma. Sono immodificabili dopo la sostituzione dei metacaratteri (*, ?,ecc). Riga di comando = nomeProgramma + parametri. Nella riga di comando gli elementi sono indicizzati da 0 (nomeProgramma). • **$#** Contiene il numero di parametri passati. • **$n** Accede all'n-esimo parametro a partire da indice 0. • **$*** Tutti argomenti concatenati e divisi da spazi. • **$@** Vettore di argomenti quotati ("..."). I parametri $* e $@ sono identici se non quotati (concatenazione di argomenti separati da " "). Se quotati $* quota tutti gli argomenti assieme mentre $@ quota singolarmente ogni argomento. $@ è usato per passare parametri a comandi dentro a degli script.
‘comando|./script.sh‘	**Command substitution**. Sostituisce (a run-time) nella riga in cui è specificato il comando o script con l'output (stdout). Comando alternativo $(./script.sh)
$?	Modificato alla terminazione di ogni script, contiene l'*exit status*.

Espressioni aritmetica	Descrizione
((...))	Valuta una stringa come un espressione aritmetica (+,-,*,/,%, (), !, &&, ||) di soli interi. Racchiude un espressione più eventualmente un assegnamento. Si possono usare variabili nell'espressione ($variabile). Exit statuts 0=true, altro=false. Non contiene espressioni condizionali e comandi. Per le operazioni in virgola mobile usare **bc**.
$((...))	Come operatore ((...)) ma è concatenabile con stringhe tramite " ".

Espressioni condizionali	Descrizione
[[...]]	Restituisce exit status 0=true, altro=false. • Non può contenere conmandi, word splitting, brace expansion, pathname expansion. Non esegue assegnamenti a variabile, annidamenti di espressioni condizionali. • Può contenere variable expansion, solo $(()), command substitution (se non genera comandi), process substitution, quote removal. Questi solo per gli operandi. • Può andare a capo. • Si possono usare gli operatori logici !, &&, ||, (). Operatori unari/binari non quotabili ne generaibili da command substitution: • Operazioni sui file: -e (esistenza), -d (cartelle), -f (file), -h (link), -r (leggibile), -s (size>0), -t (fd open e riferisce un terminale), -w (scrivibile), -x (eseguibile), -O (possesso effettivo utente), -G (possesso effettivo gruppo), -L (=h). Confronto date ultima modifica file: f1 -nt f2 (f1 nuovo o f2 inesistente), f1 -ot f2 (f1 veccio o inesistente). • Operatori aritmetici: -eq (==), -ne (!=), -le (<=), -lt (<), -ge (>=), -gt (>). • Operatori lessicografici: ==, =, !=, ¡, ¡=, ¿=, ¿, -z (size==0), -n (size!=0). -o da vero se opzione è abilitata per la shell. Versioni vecchie: [..], test ... no a capo (semmai \), -a, -o, no (). Mettere sempre spazi prima e dopo.

Tilde Expansion	Descrizione
/...	Tilde espansa con il percorso assoluto della home directory dell'effective user. Valido caso con solo e solo /.
userName/...	Tilde e userName espansi con il percorso assoluto della home directory dell'utente specificato. Valido caso con solo **userName/**.

Privilegi	Descrizione
chmod u+x *script.sh*	Modifica permessi file mediante formato numerico: u+x terna 0-7. Ogni numero è la somma dei valori associati hai permessi di r(4), w(2), x/s(1). Ordine: proprietario, gruppo, altri utenti. Può diventare un quartetto aggiungendo per primo l'identificatore numero dei privilegi di *setuid, setgid, sticky bit.*
chgrp ???	Modifica il gruppo di appartenenza di un file.
chown *newOwner nameFile*	Modifica il proprietario (e anche gruppo) di un file.
ls -al *nomeFile.estensione*	Mostra permessi, anche dei file nascosti. Interpretazione: 1°carattere tipo file (- file, d dyrectory, c collegamento seriale, b device a blocchi), 9 caratteri successivi terzine di permessi (r read,w write,x/s execute) per proprietario, gruppo, altri utenti.
whoami	Dice all'utente corrente le sue informazioni.
sudo *comando*	fa eseguire il comando come administratore, può essere chiesta userPass. Solo utenti gruppo sudo (gestito dall'admin) possono usarlo.

Subshell	Descrizione
bash	Crea una shell figlia (interattiva non di login). Eredita dal padre: dir. corrente, copia variabili d'ambiente. Non sono ereditate le variabili locali. Creata in automatico per comandi raggruppati, script, processi in background. I comandi built-in sono eseguiti in shell corrente/padre. • **-c *script.sh*** non interattiva. • **-l |–login** interattiva di login.
var=val comando	Scrivendo le assegnazioni prima dell'esecuzione di un comando si creano delle var. d'ambiente solo per l'imminente subshell. Non saranno ereditate da successive subshell.
. |source *script.sh*	Esegue lo script nella shell corrente. Utile a impostare/modificare variabili shell. Ignorata prima riga opzionale e eseguito con interprete corrente.
exit	Termina bash corrente, elimina l'ambiente e sale alla padre.
exit *exitStatus*	Termina lo script restituendo un valore intero [0-255] per indicarne l'esito di terminazione. 0 indica esecuzione terminata senza errori, qualcosaltro indica un'errore. Viene restituito alla shell esecutrice.
top	Mostra in tempo reale processi in esecuzione e risorse di sistema usate.
ps	Mostra i processi in esecuzione. Con l'opzione *-all* vedo più informazioni (PID, PPID,ecc).
set	Visualizza sia variabili locali che d'ambiente della shell corrente (anche funzioni di shell). I parametri [re]settano dei comportamenti della shell: • **+o *comando*** Disalibilita il comando (tipo history). • **-o *comando*** Abilita il comando (tipo history). • **-a** Successive variabili create/modificate diverrano d'ambiente e ereditate da shell figlie. Per [ri]definire variabili locali usare **export -n *variabile***. • **+a** Successive variabili create/modificate diverrano locali e non ereditabili da shell figlie. (default).

Uso File Descriptor	Descrizione
$$	Variabile con PID della shell corrente. Utile per esplofare /proc/ e fd vari processi.
exec {*n*}*modalità file.estensione*	Apre file e l'associa al fd scento dall'utente (n). Modalità: < (lettura), > (scrittura), >> (aggiungi in coda), <> (lettura e scrittura).
exec {*var*}*modalità file.estensione*	Apre file e l'associa al fd scento dall'os inserito in var.
exec {n}>&-	Chiusura di file con fd=n.
exec {*var*}>&-	Chiusura di file con fd contenuto in var.
program *modalità file*|*prog2 ...*	Ridirezionamento input/output. Modalità: < (strin da file), > (strout sovrascrive file), >> (srtout accoda a file), |(output prog1 è input prog2) Per ridirezionare su uno specifico fd del programma si fa: $N>, N>>, <N$. Quando si omette si sottintende strin, strout, strerr. Nota: Il fine file da tastiera si fa con "Ctrl+D". Si possono fare contemporaneamente i ridirezionamenti I/O. L'ordine di specificazione non conta.
program 0/1/2& > file |&0/1/2	Ridireziona lo strin, strout e strerr del programma nel file indicato sovrascrivendo o in un altro str. Ridirezionare simultanea in file separati: prog *2> fileErr > fileOut*
program N> &M ...	Fa puntare lo stream di N allo stesso stream di M (N e M sono fd). Si possono fare più ridirezionamenti (fd e file, fd e fd) in simultanea. Sono eseguiti da sinistra a destra.
prog1 |& prog2	Ridireziona lo stderr nello stdout di prog1 e lo passa allo strdin di prog2.
costrutto *modalità file*	Applica il ridirezionamento a tutti i comandi contenuti nel costrutto e alla condizione.
program << *word*	Word specificata è delimitatore di fine input da passare al comando. Nell'input la word deve essere a inizio di una riga con solo essa.
program <<< *word*	Ridireziona in input la word. Se si qouta ("...") si possono passare più dati.
(cmd1;...)	Comandi eseguiti in subshell con stdin/out/err condivisi e concatenati. L'exit status è quello dell'ultimo comando eseguito. Permette ridirigere più comandi simultaneamente.

Standard input e output	Descrizione
read *var*	Legge dallo stdin o file una sequenze di caratteri (fino all'INVIO o a capo) e la inserisce nella variabile passata. Ingnora spazi e tabultazioni inizio e fine riga (amenoche si setti IFS=""). Exit status 0=lettura eseguita, >0=eof (var="") o ultima riga senza \n (var="ultimi caratteri"). Necessario controllo con #var. Parametri che considerano spazi e tabulazioni: • **-n numero** Numero massimo di caratteri da leggere. • **-N numero** Numero esattto di caratteri da leggere. Considera \n come semplice carattero. Usa prossime righe se attuale ha caratteri insufficente. Una read parte dalla riga precedente, se non era stata finità di leggere (fino a \n). Il parametro **-u ${*var*}** fa leggere dal file di file descriptor in var.
read *var1 ...*	Alle variabili viene assegnata i-esima parola della riga tranne per l'ultima variabile che riceve la stringa rimanente. Alle variabili in eccesso viene assegnato valore vuoto.

Utilities	Descrizione
head -n *n file*	Mostra le prime n righe del file. Default 10.
tail -n *n file*	Mostra le ultime n righe del file. Default 10.
sed *modlaità file*	Modifica il testo secondo una regola. Modalità: • **'s/originale/nuovo/g'** sostituisce il testo originale (o più testi orinali con [o1, o2, ...]) con quello nuovo. Con l'opzione g si dice di applicarlo per tutte le occorrenze trovate e non solo la prima.
cut *modalità file*	Estrae colonne o intervalli di caratteri dalle righe. Modalità: • **-c S-D** Prende i caratteri dall'indice S all'indice D (inclusi). S o D si possono omettere e si prende dall'inizio o dalla fine fino a S o D. L'indice parte da 1. Posso concatenare più intervalli dividendoli con ",". • **-d '...' -fN** Divide la stringa in parole secondo il carattere delimitatore indicato poi seleziona la parola con lìindice f-ennesimo.
cat *nomeFile.estensione*	Visualizza il contenuto del file.
grep *stringa* [*file*]	Filtra le righe contenenti la stringa passata (che sarà evidenziata), anche se contenuta in altre parole, dal testo che sarà successivamente scritto (termina con "Ctrl+d") o dal file passato opzionalmente. • **-v** Filtra righe senza parola da evidenziare. • **-i** Disabilità il caseSensitive. • **-h o -H** Disabilita la visualizzazione del nome del file dove è avvenuta la ricerca (nel caso si cerci in più file). Spesso usato come secondo comando di |per filtrare output di altri comandi.
tee	...

Vari	Descrizione
./ *comado*	Esegue comando presente nella directory corrente (percorso relativo). Alternativa è inserire il percorso relativo o aggiungere il percorso assoluto alla variabile PATH.
strace *comando*	Dice la sistem-call usata.
clear	Pulisce la CLI. Non modifica variabili create.
which *comando*	Cerca in PATH il comando e se lo trova mostra il path in cui si trova.
lsmod	Elenca tutti i moduli attivi.
modinfo *nomeModulo*	Dice le informazioni sul modulo specificato.
sudo modprobe *nomeModulo*	Carica il modulo specificato.
history	Visualizza comandi, numerati, precedentemente eseguiti, anche da shell precedentemente chiuse. Con !NUMERO lancio il comando corrispondente nell'elenco di history. Con !STRINGA lancio il comando più recente che corrisponde alla stringa.
find *file*	Trova percorsi di file in sotto alberi della memoria. • **-type d** Trova solo directory. • **-tipe f** Trova solo file. • **-maxdepth n** Limità la profondità di ricerca il numero di livelli indicato. • **-mindeph** Indica il livello minino di ricerca. • **-name "..."** Definisce un vincolo sul nome del file da trovare. • **-iname** Disabilità il caseSensitive. • **-exec ... '' ... \;** Istruzioni da fare quando viene trovato un elemento. L'output di find viene messo al posto di ''.
wc	Dice il testo passatogli il numero di righe, parole e caratteri.

Directory `/proc` : info sui processi, con una dir. per ogni processo attivo, create e cancellate continuamente. Directory sulla RAM `/proc/` : contiene i processi in esecuzione e nella sottodirectory `fd` i file descriptor aperti.

Costrutti controllo flusso	Descrizione
for varName in elencoParole ; do listCommand ; done	Dopo il for variabile è usabile.
for ((exp1 ; exp2 ; exp3)) ; do listCommand ; done	Le exp sono espressioni valutate aritmeticamente.
if listA ; then listB ; [elif listC ; then listD ; ...] ... [else listZ ;] fi	...
while list ; do list ; done	...

In tutti i costrutti l'exit value è o quello della lista do comandi o 0 se nessun comando viene eseguito.

Espressioni condizionate su file o variabile: [condizione di un file]. Valutazione di un espressione matematica applicata a variabili d'ambiente: ((istruzini con espressione))

2 Programmazione Concorrente

Inclusioni necessarie:

- **#include <sys/types.h>** Definire tipi di dato speciali usati nelle chiamate di sistema.
- **#include <unistd.h>** Include funzioni e costanti del sistema operativo Linux/Unix.

Multi-processo	Descrizione
fork()	Crea un processo figlio. Condivide codice successivo alla fork e possiede una copia dei dati. Restituisce il pid del figlio al padre ($PID_F > 0$) e 0 al figlio ($PID_F = 0$) o un codice d'errore (¡0).
exit(0)	Termina un processo restituendo lo stato indicato come parametro (0 stato successo). Se padre termina prima del figlio non si possono liberare le risorse.
wait(NULL)	Fa attendere al processo la terminazione del processo figlio. Restituisce il pid del processo terminato.
wait(pidProcesso)	Fa attendere al processo corrente uno specifico processo.
waitpid(pidProcesso, &status, statoAttesso)	Fa aspettare un thread specifico con uno stato specifico.
exec	Sostituisce codice e dati a un processo. Non crea processi figlio.

Un processo possiede: Il proprio PID (gestito dall'os), il PID del processo figlio o 0 e il pid del processo "iziale" (gestito da descrittore di file).

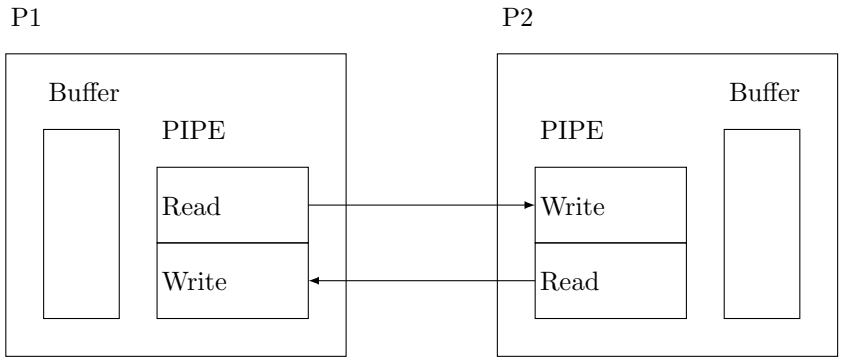
Struttura base programma multi-processo:

```
int main() {
    pid_t pid;

    pid = fork();

    if (pid < 0) { "gestione errore" }
    else if (pid == 0) { "processo figlio" }
    else { "processo padre" }
}
```

Ogni processo tiene referenza di un unico figlio, anche se ne può creare diversi. Quindi dopo ogni fork è buona cosa separare i flussi dei due processi.

PIPE	Descrizione
pipe(varPipe[2] : int) : int	Crea una pipe unidirezionale. Su varPipe[0] si leggerà. Su varPipe[1] si scriverà. Restituisce -1 in caso di errore, 0 se creazione avviene con successo.
close(varPipe[0|1])	Chiude una estremità della pipe.
read(varPipe[0], buffer, bufferSize)	Blocca esecuzione in attesa di dati da leggere.
strcpy(buffer, "...")	Prepara il buffer con il messaggio da inviare.
write(varPipe[1], buffer, sizeBuffer+1)	Scrive nella pipe.

Permette di comunicare fra processi correlati usando sistem-call e i descrittori di file. Il collegamento esiste fino a eliminazione esplicita o del processo.



Ogni processo ha un buffer di caratteri e una pipe (array di due celle) per scrivere e leggere con l'altro processo i dati contentuti nel buffer.

Nota: I processi sono visti come file, per questo le operazioni si chiamano come quelle dei file.

Multi-thread	**Descrizione**
nomeFunzioneThread(arg : void*) : void*	Funzione assegnata da eseguire a un thread. Necessita questa firma specifica per accettare e restituire qualsiasi tipo di dato. Serve eseguire un cast esplicito.
pthread_create(&varThread, &pthreadAttribut, tFunction, &args)	Crea un nuovo thread dentro al processo corrente. I parametri sono: 1. Variabile tipo thread. 2. Puntatore a struttura di attributi del thread. Default è NULL. 3. Funzione che sarà eseguita dal thread. 4. Puntatore a struttura contenente parametri usati dalla funzine.
pthread_join(&varThread, NULL)	Fa attendere processo la fine del thread indicato.

Struttura base programma multi-processo:

```
void *tFunction(void *args) {...}
int main() {
    pthread_t thread;
    ... args = ...;

    ptread_create(&thread, NULL, tFunction, &args);
}
```

Nota: La creazione di un processo è più lenta della creazione di un thread perchè nella prima bisogna creare un intero file descriptor, mentre nella seconda parziale.

2.1 SJF

Calcolo approssimato CPU Bust: $T_{n+1} = \alpha t_n + (1 - \alpha)T_n$.

- t_m tempo n-esimo CPU burst. Storia recente.
- T_n previsione prevista. Storia passata.
- α peso storia recente e passata.

Calcolo Media esponensiale: $T_{n+1} = \sum_{j=0}^{n} \alpha(1-\alpha)^j t_{n-j} + (1-\alpha)^{n+1} T_0$.

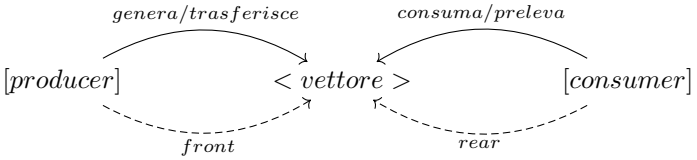
2.2 Produttore e consumatore

$$[producer] \xrightarrow{genera/trasferisce} < variable > \xleftarrow{consuma/preleva} [consumer]$$

```
shared Object buffer;

Semaphore empty =
   new Semaphore(1);

Semaphore full =
   new Semaphore(0);

cobegin

  Producer

//

  Consumer

coend
```

```
process Producer {
  while (true) {
    Object val = produce();
    empty.P();
    buffer = val;
    full.V();
  }
}

process Consumer {
  while (true) {
    full.P();
    Object val = buffer;
    empty.V();
    consume(val);
  }
```

2.3 Buffer limitato



```
Object buffer[SIZE];
int front = 0;
int rear = 0;
Semaphore empty =
  new Semaphore(SIZE);
Semaphore full =
  new Semaphore(0);
```

```
process Producer {
  while (true) {
    Object val = produce();
    empty.P();
    buf[front] = val;
    front = (front + 1) % SIZE;
    full.V();
  }
}
process Consumer {
  while (true) {
    full.P();
    Object val = buf[rear];
    rear = (rear + 1) % SIZE;
    empty.V();
    consume(val);
  }
}
```

2.4 Filosofi orientali a cena

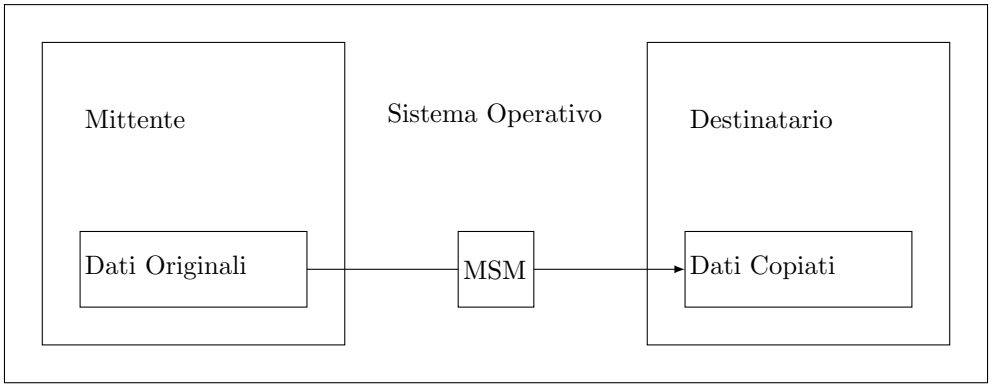
$$pensa \leftarrow\!\!\!\!\!\!\rightarrow [Filosofi] \xrightarrow{mangia} < chopsticks >$$

```
Semaphore chopsticks =
  { new Semaphore(1), ..., new Semaphore(1) };

process Philo[i] {  /* i = 0...4 */
  while (true) {
    think
    chopstick[i].P();
    chopstick[(i+1)%5].P();
    eat
    chopstick[i].V();
    chopstick[(i+1)%5].V();
  }
}
```

Le battecche sono un vetore lungo 5 (0...4) e i filosofi accedono alle battecche i e i+1 in %5.
Rischio di deadklock!!! Esistono varie soluzioni, avere un filosofo mancino è la più semplice:

```
process Philo[0] {
 while (true) {
  think
  chopstick[1].P();
  chopstick[0].P();
  eat
  chopstick[1].V();
  chopstick[0].V();
 }
}
```

2.5 Message Passing



Legenda: p=mittente, m=messaggio, q=destinatario.
Sincrono:

- Mittente attende l'ACK (Acknowledgement) del destinatario.
  `ssend(m,q)`
- Destinatario attende la spedizione di m da p o da un qualsiasi mittente (*). Poi manda l'ACK.
  `m=sreceive(p|*)`

Asincrono:

- Mittente non attende l'ACK del destinatario.
  `asend(m,q)`
- Destinatario attende la spedizione di m da p o da un qualsiasi mittente (*). Non manda l'ACK.
  `m = areceive(p|*)`

Completamente asincrono:

- Mittente non attende l'ACK del destinatario.
  `asend(m,q)`
- Destinatario non attende la spedizione di m da p o da un qualsiasi mittente (*). Non manda l'ACK.
  `m = nb-receive(p|*)`
  Se non si riceve nulla restitusce `null`.

Nella pratica: lo scambio di messaggi avviene tra canali di comunazioni su cui mandare e ricevere. L'istruzione `replay(Thoth)` sblocca il mittente (e non la receive).

2.6 Concorrenza in C

Usata la librerira POSIX **semaphore.h** (API POSIX con semafori, lock mutex e variabili condizionali).
Tipi di semaforo:

- Con nome (**named**) utile a sincronizzare processi non correlati.
- Senza nome (**unnamed**).

Funzioni semafori	**Descrizione**
sem_t *< *variabile* >	Definisce una variabile di tipo puntatore a struttura Semaforo.
sem_ini(&< *varSem* >, 0/altro, ¡valoreIniziale¿)	Crea un semaforo unnamed inizializzato al valaore passato e lo aseegna alla variabile d'indirizzo indicata. Secondo parametro a 0 semaforo condiviso tra thread stesso processo, a $\neq 0$ conviso tra processi.
sem_open("nomeSem", 0_CREAT,0666, ¡valoreIniziale¿)	Crea e restituisce un semaforo named in R/W (3° parametro) inizializzato al valore passato.
sem_wait(&varSem)	Esegue l'operazione P (decrementa).
sem_post(&varSem)	Esegue l'operazione V (incrementa).
sem_destroy(&varSem)	Distrugge un semaforo.

Le CS saranno contenute tra un wait e un post.

3 Esercizi Scheduling

Procedura risolutiva di un esercizio di scheduling data la **durata del quanto di tempo** e una tabella dei processi del tipo:

Processi	Tempo di Arrivo	Tempo d'Esecuzione
...	...	...

Passo 1
Fare una tabella di scheduling come la seguente:

Ready Queue Running Quanto d'Inizio/Fine				
Tempo	0	1	...	n
T.R. P1 ... T.R. PN				

Spiegazione delle righe della tabella:

1. *Ready Queue* specifica per ogni quando i processi pronti per esere eseguiti, tipicamente è FIFO.
2. *Running* indica per ogni quanto di tempo il processo in esecuzione.
3. *Quanto d'Inizio/Fine* evidenzia i quanti d'arrivo (A) e di termine (F) dei processi.
4. *Tempo* sono i quanti del processore. Ogni colonna è un quanto.
5. *Righe T.R.* tempo d'esecuzione rimante per ogni processo a ogni quanto di tempo dalla sua partenza.

Passo 2
Per rispondere alle statistiche richieste fare una tabella come la seguente:

Processi	T. Turnaround	Intervalli di Ready	T. Attesa
...	...	[x1-x2], ...	...
Medie:	...	...	...

Spiegazione delle colonne della tabella:

1. *Turnaround* = T. fine - T. arrivo.
2. *Intervalli di Ready* sono i quanti in cui il processo è stato nella coda di ready (dal quanto in cui vi è entrato al quanto in cui vi è uscito).
3. *T. Attesa* somma degli intervalli di ready.

4 File Descriptors

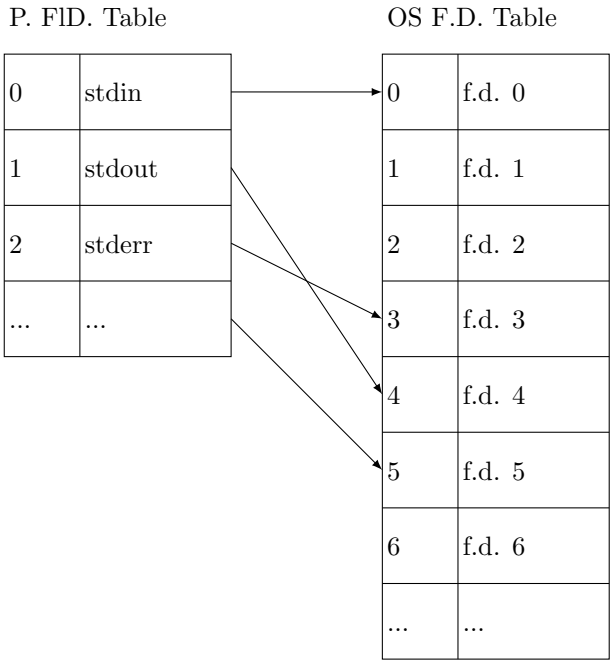
Il file descriptor è un'astrazione (integer, inizio buffer) dell'os, e quindi multi linguaggio, per permettere l'accesso ai file.

Ogni processo ha la **file descriptor table** di record indiretti delle informazioni (file descriptor) sui file aperti dal processo stesso. I record sono indiretti perchè puntano ai record della **tabella di sistema**. La tabella di sistema contiene file descriptor dei file attualmente aperti.

Un processo/shell figlia eredità una copia della f.d. table, successivamente modificabile. Padre e figlio si devono sincronizzate sull'accesso ai file. Ciò permette un modo di comunicazione tra i processi.

Rindizionamenti:

- **Processi figlio**: Padre può cambiare le associazioni fd/stream passate al figlio.
- **Auto-ridirezionamento**: fd viene associato a stream diverso.



In POSIX per ogni processo i file descriptor standard sono: stdin=0, stdout=1, stderr=2.

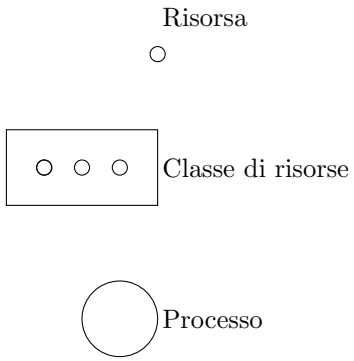
Ricoda che stdin è lo stream input da tastiera e stdout e stderr sono gli stream a video dell'output e dei messaggi d'errore.

Nota: Processi diversi possono avere file descriptor diversi, ma constesso valore, e che referenziano file diversi.

In C per scrivere si usa ¡fprintf(stdout/stderr, "..." [, parametri])¿ e per leggere si usa ¡fgets(buffer, size, stdin)¿, per il fine file usare ¡feof(stdin)¿.

5 Risorse

5.1 Grafo di Holt Generale (classi e processi)

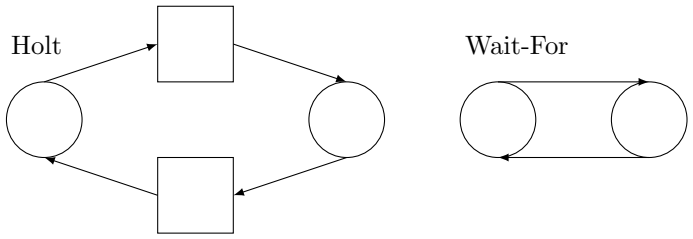


Per gli archi si può segnare la molteplicità, e nelle classi il numero di risorse non assegnate.

5.2 Detaction and recoverty - Caaso 1

Teorema: Siano le risorse ad'accesso mutualmente esclusivo, seriali e non prerilasciabili. Con una risorsa per classe si ha deadlock $\iff$ Holt contiene un ciclo (attesa circolare).

Dimostrazione: In un grafo di Holt si ha attesa circolare se si ha attesa circolare nella rispettiva variante di grafo Wait-For (elimina le risorse collassando gl'archi appropriati).



5.3 Detaction and recoverty - Caaso 2

Teorema: Siano le risorse ad'accesso mutualmente esclusivo, seriali e non prerilasciabili. Con più risorse per classe si ha deadlock $\iff$ Holt non è completamente riducibile (eliminazione tutti archi).

5.4 Detaction and recoverty - Knot

Teorema: Siano le risorse ad'accesso mutualmente esclusivo, seriali e non prerilasciabili. Con una richiesta sospespa per processo si ha deadlock $\iff$ esiste un knot.

5.5 Stato SAFE - algoritmo del banchiere

Sia s una permutazione dei valori $1, \ldots, N$ e $s(i)$ l'i-esima posizione nella sequenza. Si calcola il vettore **avail** cosi:

1. avail[1]=SC
2. avail[1+j]=avail[j]+$p_{s(j)}$, con i=$1, \ldots, N-1$

Uno stato del sistema è safe se vale: $n_{s(j)}$i=avail[j], con j=1,...,N.

Con il banchiere a singola valuta basta ordinare in modo crescente gli n_i per assicurare lo stato SAFE.

Nel caso multi valuta si aggiunge il pedice k della valuta.

Con il banchiere multi valuta ogni vlauta ha un suo ordine.