0. Introduzione

Appunti sistemi operativi di https://github.com/WAPEETY/appunti-SO2

Questi appunti non forniscono in alcun modo un metodo di studio.

Il codice scritto negli esempi non é stato mai compilato, quindi potrebbe contenere errori di sintassi, la maggior parte é stata scritta durante le lezioni e mai revisionata.

Il prof. Focardi (che tiene il corso) **NON** é in nessun modo affiliato alla repo da cui viene questo file e di conseguenza **NON** é responsabile di eventuali errori presenti in esso ~~(molto probabilmente non é nemmeno a conoscenza di questi appunti)~~

0.1 Licenza (CC BY-SA 4.0)

Gli appunti sono rilasciati sotto licenza CC BY-SA 4.0
Ció rende possibile la redistribuzione del seguente materiale, anche con modifiche, gli unici due punti importanti di questa licenza sono:

- Devi dare il giusto credito all'autore, fornire un link alla licenza originale e indicare se sono state apportate modifiche.
- In caso di modifiche o utilizzo (anche indiretto) del materiale, devi distribuire a tua volta lo stesso sotto la medesima licenza.

La copia integrale della licenza puó essere trovata presso:
https://creativecommons.org/licenses/by-sa/4.0/legalcode.

In questo caso specifico per essere in linea con la licenza devo citare anche io a mia volta il Prof Focardi e il sito https://secgroup.unive.it, tutte le immagini che vedrete di seguito (salvo alcune eventuali eccezioni esplicitamente specificate) sono rilasciate sotto la medesima licenza (CC BY-SA 4.0) a cui anche io (autore di questi appunti) mi sono dovuto attenere.
Anche parte dei contenuti testuali potranno essere simili o in alcuni punti uguali a quelli trovati nelle slide o nelle varie pagine del sito raggiungibile a questo link:

https://secgroup.dais.unive.it/teaching/laboratorio-sistemi-operativi/

0.2 Competizione

I processi competono!
cioé cercano di prendersi tutte le risorse

Interferenze:

- se uno si prende tutta la CPU: starvation (attesa indefinita)
- Carico eccessivo

Il SO virtualizza le risorse

0.3 Cooperazione

I processi possono cooperare e questa cooperazione va programmata

0.4 Modelli di comunicazione

i processi possono comunicare in due modi:

- Message Passing
- Shared Memory

0.4.1 Message passing

Processi che si passano messaggi tramite un qualche canale di comunicazione

Primitive di comunicazione (API)

- Send(m)
- recieve(&m)

0.4.1.1 Nominazione

Diretta:

- Send(P,m); //P = a chi mando, m = messaggio
- Recieve (Q, &m); //Q = da chi mi aspetto, &m = riempi con messaggio
  UPDATE: Recieve (&Q, &m); = Q viene riempito con mittente

Indiretta:
Uso un tramite (porta)

P1 <-> [A] <-> P2

se ho bisogno di altre comunicazioni creo altre porte

0.4.2 Sincrono VS Asincrono

- Send sincrona -> bloccante se non c'é la recieve
- Send asincrona -> buffer

1. Creazione di processi

La creazione di un processo richiede alcune operazioni da parte del Sistema Operativo:

- Crea PID
- Alloca Memoria (codice e dati)
- Alloca Risorse (Stdin,...)
- Gestione info nuovo processo
- Creazione PCB (Process Control Block) contenente le informazioni precedenti.

1.1 Processi Unix

Un processo è sempre creato da un altro processo tramite un'opportuna chiamata a sistema. Fa eccezione init (pid = 1) che viene "generato" al momento del boot. *(molto cristiana come cosa...)*

Il processo creante é detto parent (genitore), mentre il processo creato child (figlio). Si genera una struttura di "parentela" ad albero.

D'ora in poi useremo come standard processi UNIX like.

1.2 Relazioni dinamiche

Dopo la creazione di un processo ci sono due possibilitá:

1. il padre dello stesso attende il figlio quindi, ad esempio, una shell attende il comando e di conseguenza i segnali che puó ricevere (CTRL + C) sono riferiti al programma stesso.

Esempio:

```
$ sleep 5  ( ... 5 secondi di attesa ... )
$
```

2. Il processo genitore continua. (basta aggiungere & alla fine)

Esempio:

```
$ sleep 5 &
[1] 20
$ ps
  PID TTY          TIME CMD
   11 pts/0    00:00:00 bash
   20 pts/0    00:00:00 sleep
   21 pts/0    00:00:00 ps
$
```

1.2.1 Visualizzare le relazioni

Per visualizzare il PID del genitore basta passare gli opportuni parametri al programma ps (PPID significa Parent process ID):

```
$ sleep 5 &
[1] 23
$ ps -o pid,ppid,comm
  PID  PPID COMMAND
   11    10 bash
   23    11 sleep
   24    11 ps
$
```

Nota:
Sia sleep che ps sono figli di bash PPID (11)

1.3 Relazioni di contenuto

Due possibilità:

- Il figlio é un duplicato del genitore
- Il figlio esegue un programma differente (ad esempio nei sistemi Windows)

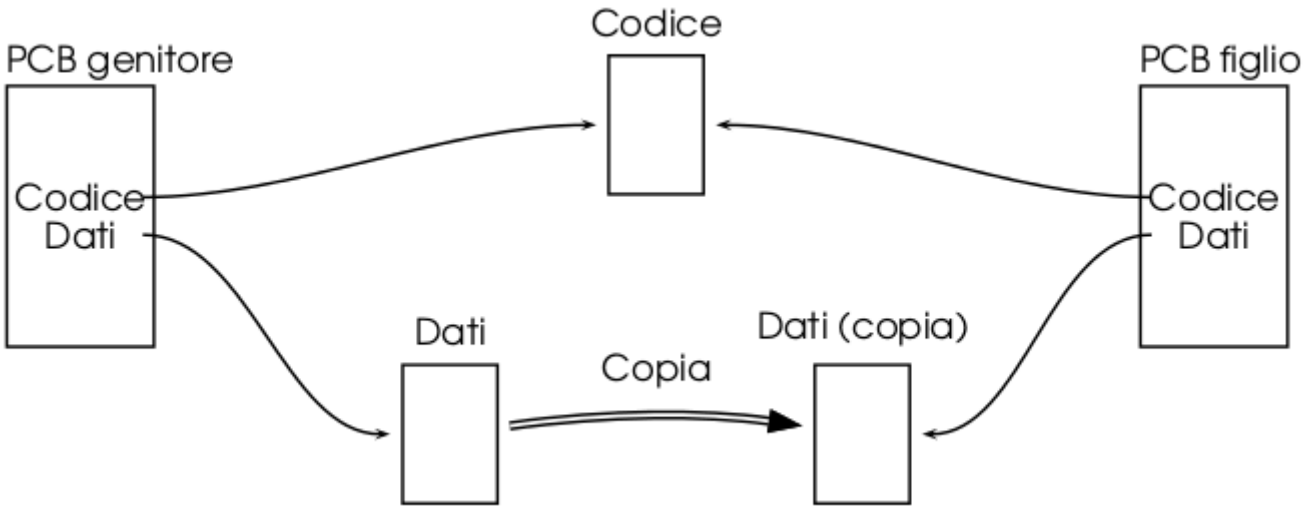
Questo è il comportamento standard ma ovviamente è possibile anche l'altra modalità in entrambi i sistemi.

1.3.1 Fork (System Call)

La chiamata a sistema `fork` permette di creare un processo duplicato del processo genitore.

La fork crea un nuovo processo che:

- condivide l'area codice del processo genitore
- utilizza una copia dell'area dati del processo genitore



1.3.2 Utilizzo della fork

Come fanno i processi a differenziarsi a livello di codice?
Si utilizza il valore di ritorno della funzione `fork()`:

- <0 Errore
- =0 Processo figlio
- >0 Processo genitore: il valore di ritorno è il PID del figlio.

```
pid = fork();
if ( pid < 0 )
    perror("fork error"); // stampa la descrizione dell'errore
else if ( pid == 0 ) {
    // codice figlio
} else {
    // codice genitore, (pid > 0)
}
// codice del genitore e del figlio: da usare con cautela!
```

Esempio:

```
int main() {
    pid_t pid;
    printf("Prima della fork. pid = %d, pid del genitore = %d\n",getpid(),
getppid());

    pid = fork();
    if ( pid < 0 )
        perror("fork error"); // stampa la descrizione dell'errore

    else if (pid == 0) {
        // figlio
```

```
        printf("[Figlio] pid = %d, pid del genitore = %d\n",getpid(),
getppid());

    } else {
        // genitore
        printf("[Genitore] pid = %d, pid del mio genitore = %d\n",getpid(),
getppid());
        printf("[Genitore] Mio figlio ha pid = %d\n",pid);
        sleep(1); // attende 1 secondo
    }

    // entrambi i processi
    printf("PID %d termina.\n", getpid());
}
```

1.3.2 Fallimento della “fork”

Quando fallisce una fork? Quando non è possibile creare un processo e tipicamente questo accade quando non c’è memoria per il processo o per il kernel. Ecco un piccolo test.

```
// fork bomb, usare a proprio rischio e pericolo!
int main() {
    while(1)
        if (fork() < 0)
            perror ("errore fork");
}
```

NOTA: Questo non dovrebbe piú funzionare, infatti ad oggi esistono due parametri, `cgroups` e `ulimit`, che limitano il numero massimo di risorse allocabili evitando si blocchi tutto il sistema.

1.4 Processi orfani

Mettendo una `sleep` subito prima della printf nel figlio lo rendo orfano perché termina il genitore prima di lui:

Come risolvo?

verrá "adottato" da `init` (genitore di tutti i processi di sistema) o da `upstart` nei moderni sistemi Linux:

```
// figlio
sleep(5);
printf("[Figlio] pid = %d, pid genitore = %d\n",getpid(), getppid());
```

NOTA Un processo orfano non viene più terminato da `ctrl-c` (NON RICEVE PIÚ IL SEGNALE DALLA SHELL)

utilizzare ps -o pid,ppid,comm per vedere che il processo è ancora attivo, eventualmente con l'opzione -e.

1.5 Processi zombie

Gli zombie sono processi terminati ma in attesa che il genitore rilevi il loro stato di terminazione. Per osservare la generazione di un processo zombie ci basta porre la sleep prima della printf del processo genitore:

```
// genitore
sleep(5);
printf("[genitore] pid = %d, pid genitore = %d\n",getpid(),getppid());
```

Testare la presenza di zombie con ps -e da un altro terminale oppure lanciando il programma in background tramite &:

```
$ fork &
...
ps -e | grep fork
 5118 pts/3    00:00:00 fork
 5119 pts/3    00:00:00 fork <defunct>
```

Quale é l'output del seguente programma?

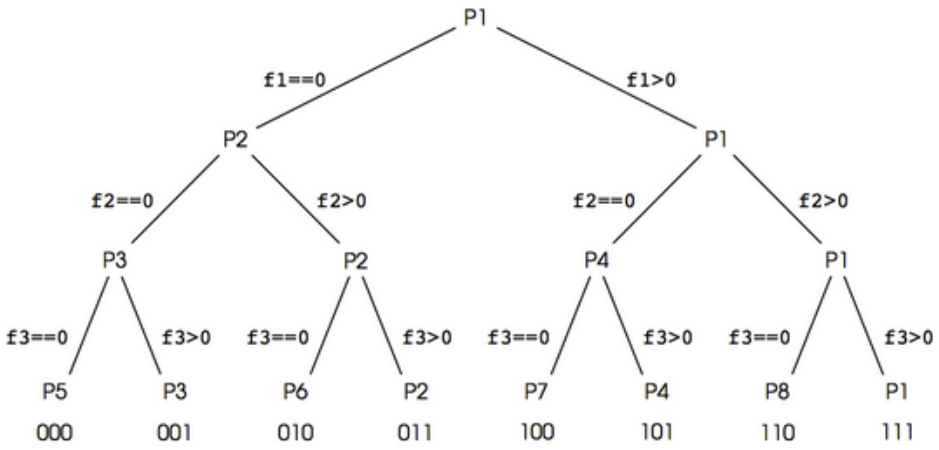
```

int main(){
    pid_t f1, f2, f3;

    f1 = fork();
    f2 = fork();
    f1 = fork();

    printf("%i%i%i ",(f1 > 0),(f2 > 0),(f3 > 0));
}
```

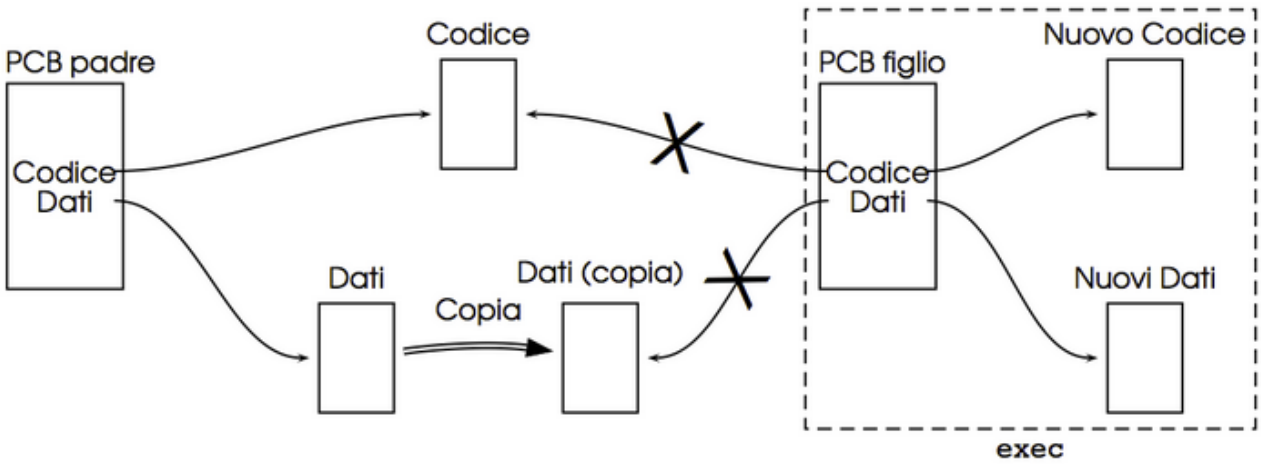
L'output non é deterministico e la printf verrá eseguita 8 volte (2 pow(3))



1.6 System call exec

Per eseguire un programma diverso da quello che fa la fork si usa la `exec`.

Essa sostituisce codice e dati di un processo con quelli di un programma differente.



Dopo che termina la Exec (finisce di eseguire il programma) il processo muore.

La exec ha diverse varianti che si differenziano in base al formato degli argomenti e possono essere riassunte in:

- Lista di argomenti terminata da NULL
  - Array di stringhe argv[] terminato da NULL
  - utilizzo o meno del path della shell
    (presenza della ‘p’ nel nome della exec)

```
execl("/bin/ls", arg0, arg1, ..., NULL);
execlp("ls", arg0, arg1, ..., NULL);
execv("/bin/ls", argv);
execvp("ls", argv);
```

1.6.1 Argomenti

Per convenzione, il primo argomento contiene il nome del file associato al programma da eseguire.

chiamando il seguente programma:

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    int i;
    for(i=0;i<argc;i++) {
        printf("arg %d: %s\n",i,argv[i]);
    }
}
```

tramite

```
$ ./argv prova 1 2 3
```

Avremo il seguente output:

```
$ ./argv prova 1 2 3
arg 0: ./argv
arg 1: prova
arg 2: 1
arg 3: 2
arg 4: 3
```

1.6.2 Return Value

La exec ritorna solamente in caso di errore (valore -1).

In caso di successo il vecchio codice è completamente sostituito dal nuovo e non è più possibile tornare al programma originale.

```
#include <stdio.h>
#include <unistd.h>
int main() {
      printf("provo a eseguire ls\n");

      execl("/bin/ls","/bin/ls","-l",NULL);
      // oppure : execlp("ls","ls","-l",NULL);
            // che tiene in considerazione il PATH

            // scrivo 2 volte ls perché
            //per convenzione il primo arg é il nome
```

```
            // del programma quindi
            // execl("programma da eseguire",
            // "nome programma","arg1",...,"argn"
            //  ,NULL);
            // fondamentale il NULL alla fine

      printf("non scrivo questo! \n");
      // questa printf non viene eseguita, se la exec va a buon fine
}
```

Errore della exec

```
execlp("ls2","ls2","-l",NULL);
```

Cosa accade? exec va in errore e ritorna -1

se invece il parametro non esiste exec comunque carica il programma ma poi sta al programma come rispondere.

Meme
Perché non usa Vim?
Se volete lo uso ma molti di voi lascerebbero l'aula

1.7 Copy On Write

Viene copiata solamente la page-table, e le pagine (quelle contenenti i dati, che dovrebbero essere state copiate) sono invece etichettate come read-only. Un tentativo di scrittura genera un errore che viene gestito dal kernel:

1. copiando al volo (copy-on-write, appunto) la pagina fisica e aggiornando opportunamente la page-table in modo che punti alla nuova copia;
2. impostando la modalità a read-write: da quel momento in poi le due copie sono indipendenti.

Quindi se si fa fork e subito exec nessuna pagina viene effettivamente copiata.

1.8 Sys call exit e wait

- `exit`: termina il processo (già usata negli esempi per i casi di errore);
- `wait`: attende la terminazione di un figlio (se uno dei figli è uno zombie ritorna subito senza bloccarsi).

Sintassi:

- `exit(int stato)`: termina il processo ritornando lo stato al genitore; Si usano le costanti `EXIT_FAILURE` e `EXIT_SUCCESS` che normalmente sono uguali ad 1 e 0 rispettivamente;

- `pid = wait(int &stato)`: ritorna il pid e lo stato del figlio che ha terminato. Si invoca `wait(NULL)` se non interessa lo stato. Se non ci sono figli ritorna -1.

1.8.1 Valore di ritorno della wait

Lo stato ritornato da wait va gestito con opportune macro:

- `WIFEXITED(status)==true` se figlio uscito con una exit
  `WEXITSTATUS(status)` ritorna 1 byte passato dalla exit

```
if (WIFEXITED(status))
    printf("OK: status = %d\n",WEXITSTATUS(status));
```

- `WIFSIGNALED(status)==true` se figlio terminato in maniera anomala.
  `WTERMSIG(status)` ritorna il “segnale” che ha causato la terminazione.

```
    if (WIFSIGNALED(status))
        printf("ANOMALO: status = %d\n",WTERMSIG(status));
```

2. Segnali

Forma molto semplice di comunicazione tra processi.

Tecnicamente sono interruzioni software causate da svariati eventi:

- Generati da terminale. Ad esempio ctrl-c (`SIGINT`);
- Eccezioni dovute ad errori in esecuzione: es. divisione per 0, riferimento “sbagliato” in memoria, ecc…
- Segnali esplicitamente inviati da un processo all’altro;
- Eventi asincroni che vengono notificati ai processi: esempio `SIGALARM`.

Segnali POSIX da `man 7 signal`

```
Signal     Value     Action   Comment
----------------------------------------------------------------------
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard     <== ctrl-C
SIGQUIT       3       Core    Quit from keyboard          <== ctrl-\
SIGILL        4       Core    Illegal Instruction
SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal                 <== kill -9 (da shell)
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no readers
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal          <== kill (da shell)
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated <== gestito da wait()
SIGCONT   19,18,25    Cont    Continue if stopped
SIGSTOP   17,19,23    Stop    Stop process
SIGTSTP   18,20,24    Stop    Stop typed at tty
SIGTTIN   21,21,26    Stop    tty input for background process
SIGTTOU   22,22,27    Stop    tty output for background process

The  signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

2.1 Gestione dei segnali

Ogni segnale ha un comportamento di default descritto di seguito

```
The  entries  in  the  "Action"  column of the tables below specify the
default disposition for each signal, as follows:

Term   Default action is to terminate the process.

Ign    Default action is to ignore the signal.

Core   Default action is to terminate the process and dump core (see core(5)).

Stop   Default action is to stop the process.

Cont   Default action is to continue the process if it is currently stopped.
```

Esempio
`alarm` manda un `SIGALRM` dopo n secondi (serve per dare un timeout). Il default handler termina il programma.

La shell analizza lo stato di uscita del programma e stampa il motivo della terminazione (“alarm clock”).

```c
#include <unistd.h>
int main()
{
    alarm(3);
    while(1){}
}
```

Il programma “punta una sveglia” dopo 3 secondi e attende in un ciclo infinito.

é possibile modificare il comportamento del default handler.
Cosa possiamo fare quando arriva un segnale?

- Ignorarlo
- Gestirlo
- Lasciare il compito al gestore di sistema

2.1.1 Impostare il gestore dei segnali

Tramite la system call `signal` è possibile cambiare il gestore dei segnali. La system call prende come parametri un segnale e una funzione che da quel momento diventerà il nuovo gestore del segnale.

```c
void alarmHandler()
{
    printf("questo me lo gestisco io!\n");
    alarm(3); // ri-setta il timer a 3 secondi
}

int main() {
```

```
    signal(SIGALRM, alarmHandler);
    alarm(3);
    while(1){}
}
```

È possibile **passare alla signal** le costanti `SIG_IGN` o `SIG_DFL` al posto della funzione handler per indicare, rispettivamente:

- che il segnale va ignorato
- che l'handler è quello di default di sistema

Il **valore di ritorno** di signal è

- `SIG_ERR` in caso di errore
- l'handler precedente, in caso di successo

Esempio: Proteggersi da ctrl-c

Se modifichiamo il gestore del segnale `SIGINT` possiamo evitare che un programma venga interrotto tramite `ctrl-c`

```
int main() {
    void (*old)(int);

    old = signal(SIGINT,SIG_IGN);
    printf("Sono protetto!\n");
    sleep(3);

    signal(SIGINT,old);
    printf("Non sono più protetto!\n");
    sleep(3);
}
```

Notare l'uso del valore di ritorno della signal per reimpostare il gestore originale.

2.2 System call `kill`

La chiamata a sistema kill manda un segnale a un processo.

```
int main(){
    pid_t pid1,pid2;
    pid1 = fork();
    if ( pid1 < 0 ) {
        perror("errore fork"); exit(EXIT_FAILURE);
    } else if (pid1 == 0)
```

```
        while(1) { // primo figlio
            printf("%d è vivo !\n",getpid());
            sleep(1); }
    pid2 = fork();
    if ( pid2 < 0 ) {
        perror("errore fork"); exit(EXIT_FAILURE);
    } else if (pid2 == 0)
        while(1) { // secondo figlio
            printf("%d è vivo !\n",getpid());
            sleep(1); }
    // processo genitore
    sleep(2);
    kill(pid1,SIGSTOP); // sospende il primo figlio
    sleep(5);
    kill(pid1,SIGCONT); // risveglia il primo figlio
    sleep(2);
    kill(pid1,SIGINT); // termina il primo figlio
    kill(pid2,SIGINT); // termina il secondo figlio
}
```

2.3 Mascherare i segnali

A volte risulta utile bloccare temporaneamente la ricezione dei segnali per poi riattivarli. Tali segnali non sono ignorati ma solamente 'posticipati'.

`sigprocmask(int action, sigset_t *newmask, sigset_t *oldmask)`

`int action` puó valere:

- `SIG_BLOCK`: l'insieme dei segnali `newmask` viene unito all'insieme dei segnali attualmente bloccati, che sono restituiti in `oldmask`;
- `SIG_UNBLOCK`: l'insieme dei segnali `newmask` viene sottratto dai segnali attualmente bloccati, sempre restituiti in `oldmask`;
- `SIG_SETMASK`: l'insieme dei segnali `newmask` sostituisce quello dei segnali attualmente bloccati (`oldmask`).

2.3.1 Gestione delle maschere

Per gestire gli insiemi di segnali (di tipo sigset_t) si utilizzano:

- `sigemptyset(sigset_t *set)` che inizializza l'insieme `set` all'insieme vuoto;
- `sigaddset(sigset_t *set, int signum)` che aggiunge il segnale `signum` all'insieme `set`;

NOTA:
POSIX non specifica se più occorrenze dello stesso segnale debbano essere memorizzate (accodate) oppure no. Tipicamente se più segnali uguali vengono generati, solamente uno verrà "recapitato" quando il blocco viene tolto.

2.3.2 Mascherare `SIGINT` (ctrl-c)

```c
int main() {
    sigset_t newmask,oldmask;

    sigemptyset(&newmask);                      // insieme vuoto
    sigaddset(&newmask, SIGINT);        // aggiunge SIGINT alla "maschera"
    // setta la nuova maschera e memorizza la vecchia
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
        perror("errore settaggio maschera"); exit(1);
    }
    printf("Sono protetto!\n");
    sleep(3);

    // reimposta la vecchia maschera
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
        perror("errore settaggio maschera"); exit(1);
    }
    printf("Non sono piu' protetto!\n");
    sleep(3);
}
```

Se digitiamo `ctrl-c` mentre il segnale è mascherato esso viene sospeso. Appena la maschera viene rimossa, il segnale è ricevuto dal processo che viene immediatamente interrotto. (quindi non stampa non sono piú protetto se abbiamo premuto ctrl+c)

NOTA:
Non stiamo modificando il gestore del segnale.

2.3.3 Interferenze e funzioni 'safe' POSIX

L'uso della printf nell'handler è rischioso perchè usa dati globali: Se anche il programma interrotto stava facendo I/O i due potrebbero interferire!

Facendo man signal-safety (in sistemi Linux recenti), troviamo la lista di funzioni safe:

```
Function                Notes
abort(3)                Added in POSIX.1-2003
accept(2)
access(2)
aio_error(3)
aio_return(3)
aio_suspend(3)          See notes below
alarm(2)
bind(2)
... (molte altre) ...
```

Usare solo funzioni safe nel handler e assicurarsi che non modifichi variabili globali del programma
oppure
Mascherare i segnali ogni volta che si usa una funzione unsafe nel programma (infattibile in generale)

Esempio: interferenza sulla printf

```c
void alarmHandler();    // gestore
static int i=0;         // contatore globale 'volatile'

int main() {
    signal(SIGALRM, alarmHandler);
    alarm(1);
    while(1){
            printf("prova\n");
    }
}
void alarmHandler()
{
    printf("questo me lo gestisco io %d!\n",i++);
    alarm(1);   // ri-setta il timer a 1 secondo
}
```

NOTA:
eseguire il comando con in coda `| grep io` per evitare di osservare tutte le stampe “prova”.

3. Pipe

Le pipe sono la forma più “antica” di comunicazione tra processi UNIX.
Si possono inviare dati da un lato della pipe e riceverli dal lato opposto.
(non é bidirezionale quindi mandi da un lato e ricevi dall'altro)

Tecnicamente, la pipe è una porta di comunicazione con send asincrona e receive (read) sincrona.

quindi la read é BLOCCANTE

Esistono pipe senza nome e con nome:
senza nome: sono utilizzabili solo da processi con antenati comuni
con nome: il nome è nel filesystem e tutti i processi le possono utilizzare

3.1 Pipe senza nome

Le pipe senza nome sono utilizzate per combinare piú comandi Unix direttamente dalla shell tramite il simbolo “|” (pipe).

Per creare una pipe si utilizza la systemcall
`pipe(int filedes[2])`
che restituisce in filedes due descrittori:
`filedes[0]` per la lettura;
`filedes[1]` per la scrittura;

Notiamo quindi che le pipe sono **half-duplex** (monodirezionali): esistono due distinti descrittori per leggere e scrivere.

Esempio:

```
wapeety@thinkpad: touch nome.txt
wapeety@thinkpad: touch nome2.txt
wapeety@thinkpad: ls
nome.txt
nome2.txt
altro_file
altro_ancora
wapeety@thinkpad: ls | grep nome
nome.txt
nome2.txt
```

il simbolo `|` é la pipe

Esempio:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main() {
        int fd[2];

        pipe(fd); /* crea la pipe */
        if (fork() == 0) { //mi forko
                char *phrase = "prova a inviare questo!";

                close(fd[0]);                           /* chiude in lettura
IMPORANTE! */
                write(
                                        fd[1], //descrittore (file)
                                        phrase, //questo é un pointer di
char
                                        strlen(phrase)+1); /*invia anche
0x00 per far capire che ha finito */
                close(fd[1]); //chiude in scrittura
        } else {
                char message[100];
                memset(message,0,100);
                int bytesread;

                close(fd[1]);  /* chiude in scrittura */
                bytesread = read(fd[0], //descrittore da dove leggere
                                 message, //dove scrivere
                                 99 //99 e non 100 perché devo preservare in
ogni caso l'ultima come terminatore
                                ); //read ritorna i bytes che actually
vengono letti
                printf("ho letto dalla pipe %d bytes: '%s'
\n",bytesread,message);
                close(fd[0]);  /* chiude in lettura */
        }
}
```

3.1.1 Situazioni particolari

Le pipe si comportano “quasi” come dei normali file.
Ci sono casi particolari tipici delle pipe:

- Cosa accade se si fa una read da una pipe che è vuota ed è stata chiusa in scrittura?
  La `read` ritorna 0, corrispondente a un `end-of-file`.
- Cosa accade se si fa una write su una pipe che è stata chiusa in lettura?

Viene generato il segnale `SIGPIPE` che di default termina il processo. Se si ignora o si gestisce il segnale la `write` ritorna un errore e `errno` è settata a `EPIPE`

ESERCIZIO: modificare il programma precedente in modo da osservare i due eventi discussi qui sopra

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main() {
        int fd[2];
                                /* questo serve se ignoro il segnale
                signal(SIGPIPE,SIG_IGN)
                */
        pipe(fd);
        if (fork() == 0) {
                char *phrase = "prova a inviare questo!";
                close(fd[0]);
                write(fd[1],phrase,strlen(phrase)+1);
      /* questo é per la doppia write
            sleep(1);
            write(fd[1],phrase,strlen(phrase)+1);
            */
            close(fd[1]);
        } else {
                char message[100];
                memset(message,0,100);
                int bytesread;

                close(fd[1]);
                bytesread = read(fd[0], message,99);
                printf("ho letto dalla pipe %d bytes: '%s'
\n",bytesread,message);

            /* da qua é per la doppia read
            sleep(1);
            memset(message,0,100);
```

```
            bytesread = read(fd[0],     message,99);
                printf("ho letto dalla pipe %d bytes: '%s'
\n",bytesread,message);
                */
            /* per la doppia write
            printf("questo non lo stampo");
            */
            /* se ignoro il segnale
            perror("")
            */
                close(fd[0]);
        }
}
```

Parte di esempio per simulare pipe di shell
in parole povere l'output (stdout) viene ridirezionato nell'input della pipe (prendendo accesso esclusivo del canale)

```
dup2(fd[1],1);      /* fa si che 1 (stdout) sia una copia di fd[1] */
/* da qui in poi l'output va sulla pipe */
close(fd[1]); /* chiude il descrittore fd[1] cosí da lá in poi é stdout ad
essere l'unico collegato all'entrata della pipe */
```

3.2 Pipe con nome

Le pipe senza nome **non** possono essere utilizzate da processi che non hanno un antenato in comune. Per ovviare a questa limitazione esistono le pipe con nome. Tali pipe possono essere create con il comando mkfifo.

```
$ mkfifo myPipe     <==== (oppure mknod myPipe p)
$ ls -al
totale 36
...
prw-rw-r--    1 wapeety  wapeety         0 mag 23 00:57 myPipe
...
$
```

La pipe esiste nel filesystem e qualsiasi processo con i diritti di accesso al file può utilizzarla.

Esercizio: lettori e scrittori

Consideriamo un processo lettore (destinatario) che accetta, su una pipe con nome, messaggi provenienti da più scrittori (mittenti).
Gli scrittori mandano 3 messaggi e poi terminano.

Quando tutti gli scrittori chiudono la pipe il lettore ottiene 0 come valore di ritorno dalla read ed esce. Lettori e scrittori sono processi distinti lanciati indipendentemente (non necessariamente parenti).

Lettore

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>
#define PNAME "/tmp/aPipe"
int main() {
  int fd;
  char leggi;

  mkfifo(PNAME,0666); // crea la pipe con nome, se esiste gia' non fa nulla (0666 sono i permessi)
  fd = open(PNAME,O_RDONLY); // apre la pipe in lettura
  if ( fd < 0 ) {
    perror("errore apertura pipe");
    exit(1);
  }
  while (read(fd,&leggi,1)) { // legge un carattere alla volta fino a EOF
    if (leggi == '\0'){
      printf("\n"); // a capo dopo ogni stringa
    } else {
      printf("%c",leggi);
    }
  }
  close(fd);      // chiude il descrittore
  unlink(PNAME);  // rimuove la pipe
  return 0;
}
```

Scrittore

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
```

```c
#define PNAME "/tmp/aPipe"
int main() {
  int fd, i, lunghezza;
  char *message;
  mkfifo(PNAME,0666); // crea la pipe con nome, se esiste gia' non fa nulla
  // crea la stringa
  lunghezza = snprintf(NULL,0,"Saluti dal processo %d",getpid());
  message = malloc(lunghezza+1);
  snprintf(message,lunghezza+1,"Saluti dal processo %d",getpid());
  fd = open(PNAME,O_WRONLY); // apre la pipe in scrittura
  if ( fd < 0 ) {
    perror("errore apertura pipe"); exit(1);
  }
  for (i=1; i<=3; i++){   // scrive tre volte la stringa
    write (fd,message,strlen(message)+1); // include terminatore
    sleep(2);
  }
  close(fd);  // chiude il descrittore
  free(message);
  return 0;
}
```

3.2 Atomicità e direzionalità

Le scritture sulle pipe sono atomiche se inferiori alla dimensione `PIPE_BUF` (limits.h), usualmente 4096 bytes.

- Se lanciamo più scrittori le stringhe non si mescolano!
- Il lettore invece legge un carattere alla volta. Provare a lanciare più lettori per osservare interferenze.

L'opzione `O_RDWR` nella open permette di aprire una pipe con nome in lettura e scrittura.

- Come si fa però a evitare che un processo legga ciò che lui stesso ha scritto?
- Tale modalità esiste solo per poter aprire una pipe in lettura/scrittura quando nessuno l'ha ancora aperta in scrittura/lettura. Una open *unilaterale* è **bloccante**.

4. Thread POSIX

Un thread è un'unità di esecuzione all'interno di un processo.

Un processo può avere più thread in esecuzione, che tipicamente condividono le risorse del processo e, in particolare, la memoria.

Lo standard POSIX definisce un insieme di funzioni per la creazione e la sincronizzazione di thread.

4.1 Creazione Thread

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

- `thread`: un puntatore a pthread_t, l'analogo di pid_t. Attenzione che non necessariamente è implementato come un intero;
- `attr`: attributi del nuovo thread. Se non si vogliono modificare gli attributi è sufficiente passare `NULL`;
- `start_routine` il codice da eseguire. È un puntatore a funzione che prende un puntatore a void e restituisce un puntatore a void. Ricordarsi che in C il nome di una funzione è un puntatore alla funzione;
- `arg` eventuali argomenti da passare, NULL se non si intende passare parametri.

4.2 Exit e Join

`pthread_exit(void *retval)`
termina l'esecuzione di un thread restituendo retval. Si noti che quando il processo termina (exit) tutti i suoi thread vengono terminati. Per far terminare un singolo thread si deve usare pthread_exit;

`pthread_join(pthread_t th, void **thread_return)` attende la terminazione del thread th. Se ha successo, ritorna 0 e un puntatore al valore ritornato dal thread. Se non si vuole ricevere il valore di ritorno è sufficiente passare NULL come secondo parametro.

4.3 Detach e self

`pthread_detach(pthread_t th)`
se non si vuole attendere la terminazione di un thread allora si deve eseguire questa funzione che pone th in stato detached: nessun altro thread potrà attendere la sua terminazione con pthread_join e quando terminerà le sue risorse verranno automaticamente rilasciate (evita che diventino thread "zombie").

Si noti che `pthread_detach` non fa sì che il thread rimanga attivo quando il processo termina con exit.

`pthread_t pthread_self()` ritorna il proprio thread id.

ATTENZIONE: questo è l'ID della libreria phread e non l'ID di sistema. Per visualizzare l'ID di sistema si può usare, in Linux, `syscall(SYS_gettid)`.

5. Sezione critica e Semafori

Come giá detto i thread condividono la stessa memoria e per mantenere la coerenza e la "sicurezza" dei dati sono necessari dei meccanismi di sincronizzazione.

La teoria dietro si chiama Sezione Critica.

Definizione:
La Sezione Critica è la parte di codice in cui i processi accedono a dati condivisi.

Consideriamo un generico thread T in cui sia presente una sezione critica:

```
thread T {
  ....
  < ingresso >
  < sezione critica >
  < uscita >
  ....
}
```

Dobbiamo adesso progettare come garantire la mutua esclusione ovvero un codice per regolare l'ingresso e l'uscita dalla sezione critica in modo che solo un processo alla volta possa accedervi.

Questa proprietà deve valere senza fare assunzioni sulla velocità relativa dei processi (temporizzazione), a prescindere quindi da come vengono schedulati.

5.1 Soluzioni software

Proviamo ora ad usare soluzioni software totalmente in user space.

5.1.1 Tentativo 1: lock

Utilizziamo una variabile booleana globale lock inizializzata a false:

```
thread T {
  ....
  while(lock) {}
  lock = true;
  < sezione critica >
  lock = false;
  ....
}
```

Il codice qui sopra cicla a vuoto fin quando lock non é disattivato e questo capiterá solo (teoricamente) se non ci sono thread in sezione critica

Problema:
se due thread entrano contemporaneamente in sezione critica potrebbe accadere che superino il while entrambi, siccome potrebbe capitare:

- T0 legge il while e lo passa
- T0 viene sospeso
- T1 legge il while e lo passa
  e da qui in poi sono entrambi nella sezione critica ma nessuno ha avuto modo di flaggare il lock

In tale caso è come se entrambi acquisissero il lock e non ci fosse piú mutua esclusione.

5.1.2 Tentativo 2: Turno

Utilizziamo una variabile booleana globale turno inizializzata a 1. I thread eseguono codice che dipende dal proprio ID.

```
thread T0 {                          thread T1 {
  ....                                 ....
  while(turno != 0) {}                 while(turno != 1) {}
  < sezione critica >                  < sezione critica >
  turno = 1;                           turno = 0;
  ....                                 ....
}                                     }
```

Il codice cicla a vuoto attendendo il proprio turno, e ció garantisce che ci sia solo un thread alla volta in sezione critica.

Problema:
Anche se é garantita la mutua esclusione uno stesso thread non potrá entrare per due volte di seguito in sezione critica ma dovrá aspettare prima che l'altro ci passi (che potrebbe non accadere mai)

5.1.3 Tentativo 3: pronto

In questa soluzione useremo un array popolato da due booleani che segnano lo stato del thread T0 e T1 rispettivamente e vengono settati entrambi a false all'inizio.

```
thread T0 {                          thread T1 {
  ....                                 ....
  pronto[0] = true;                    pronto[1] = true;
  while(pronto[1]) {}                  while(pronto[0]) {}
  < sezione critica >                  < sezione critica >
  pronto[0] = false;                   pronto[1] = false;
```

```
  ....                                    ....
}                                       }
```

Quando un thread vuole entrare pone a `true` il suo stato di "pronto", successivamente prima di entrare cicla a vuoto aspettando che T1 sia fuori dalla sezione critica.

Problema:
Potrebbe accadere che settino simultaneamente pronto[i]=true e poi si blocchino entrambi sul proprio ciclo while.
Questa situazione di attesa è irrisolvibile e quindi inaccettabile.

5.1.4 Algoritmo di Peterson

Il difetto di pronto è che esiste un caso in cui i thread vanno in stallo.

Per evitare questa situazione irrisolvibile utilizziamo una turnazione, ma solo nel caso problematico: quando entrambi i thread sono pronti. Se uno solo è pronto può tranquillamente accedere alla sezione critica.

```
thread T0 {                                thread T1 {
  ....                                       ....
  pronto[0] = true;                          pronto[1] = true;
  turno=1;                                   turno=0;
  while(pronto[1] && turno != 0) {}          while(pronto[0] && turno != 1)
{}
  < sezione critica >                        < sezione critica >
  pronto[0] = false;                         pronto[1] = false;
  ....                                       ....
}                                           }
```

Quando un thread vuole entrare pone a `true` il suo stato per segnalare all'altro thread la sua volontà di accedere.
Subito dopo cede il turno all'altro thread e cicla a vuoto se l'altro thread è pronto e se non è il proprio turno.

Queste soluzioni permettono di ottenere mutua esclusione tramite un ciclo "a vuoto" che attende l'acquisizione di un lock globale. Si ottengono i così detti spin-lock, ovvero codice che cicla finché non ottiene il lock.

Questo tipo di sincronizzazione non è efficiente per diverse ragioni:

- ciclare a vuoto spreca tempo di CPU inutilmente; (Busy waiting)
- Le istruzioni hardware speciali semplificano la realizzazione degli spin-lock ma la loro realizzazione è complessa. Ad esempio in architetture multi-core è necessario, durante la loro esecuzione, bloccare l'accesso da parte di tutti i core, degradando le prestazioni.

Di fatto gli spin-lock si utilizzano nel codice dei sistemi operativi ma quasi mai a livello applicazione. Il sistema operativo o la jvm, nel caso di java, forniscono meccanismi di sincronizzazione più semplici ed efficaci che possono essere utilizzati per programmare applicazioni multi-threaded.

Per questo esistono i semafori...

5.2 Semafori

Un semaforo altro non è che un contatore intero con una coda di attesa, essi vengono forniti dal sistema operativo.

```
struct semaphore {
   int valore;
   thread *queue;
}
```

Come interpretiamo il valore di un semaforo S?

- S.valore > 0: il semaforo è verde. Il valore indica il numero di accessi consentiti prima che il semaforo diventi rosso;
- S.valore <= 0: il semaforo è rosso. Il valore (tolto il segno) indica il numero di thread in attesa sulla coda del semaforo.

NOTA: *in alcune implementazioni i semafori non assumono mai un valore negativo e un semaforo rosso assume sempre il valore 0, indipendentemente dal numero di thread in attesa.*

(coff coff... MacOS)

5.2.1 Come li uso?

I semafori non vengono mai incrementati e decrementati direttamente ma solo tramite speciali funzioni che gestiscono la sincronizzazione dei thread:

`P(S)` o `wait(S)`:

- decrementa il valore del semaforo S.
  - Se il semaforo era rosso già prima del decremento il thread attende sulla coda associata al semaforo;

`V(S)` o `post(S)`:

- incrementa il valore del semaforo S.
  - Se ci sono thread in attesa sul semaforo, viene sbloccato il primo della coda.

5.2.2 Sezione critica

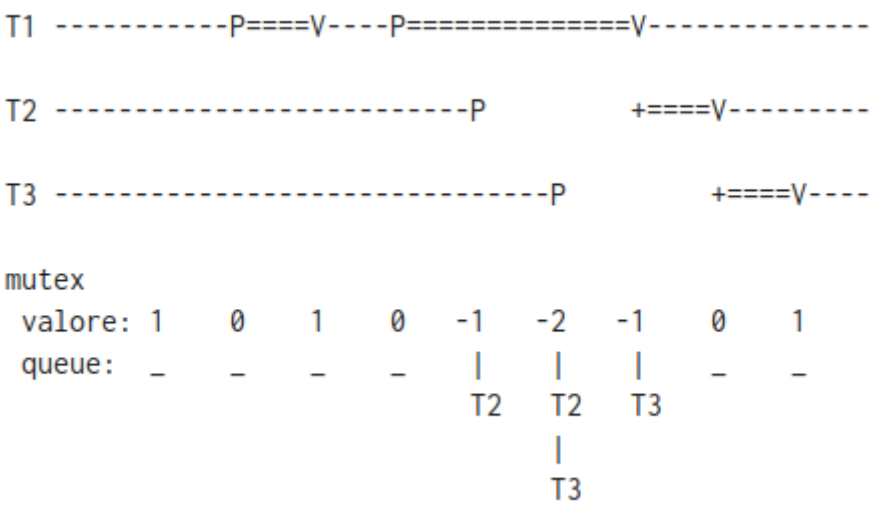
I semafori permettono di realizzare la sezione critica in modo estremamente semplice ed efficace. Il valore del semaforo indica il numero di accessi consentiti prima di diventare rosso.

Nel caso della sezione critica solo un thread alla volta deve accedere, quindi utilizziamo un semaforo mutex che inizializziamo al valore 1.

Per controllare l'accesso è sufficiente eseguire una P(mutex). All'uscita della sezione critica si esegue una V(mutex) per ripristinare il valore del semaforo e sbloccare un thread eventualmente in attesa:

```
thread T {
  ...
  P(mutex);
  < Sezione critica >
  V(mutex);
  ...
}
```

Vediamo un esempio di esecuzione con tre thread in esecuzione su 3 core distinti. Consideriamo un semaforo mutex inizializzato a 1 e per semplicità scriviamo P e V invece che P(mutex) e V(mutex):



Spiegazione:

- abbiamo indicato:
  - con - l'esecuzione fuori sezione critica;
  - con = l'esecuzione in sezione critica;
  - in basso, lo stato del semaforo mutex: valore (inizialmente 1) e coda (inizialmente vuota);
- T1 entra ed esce dalla sezione critica. In basso indichiamo lo stato del semaforo: mutex da 1 (verde) diventa 0 (rosso) e poi torna 1 (verde);
- poi T1 torna in sezione critica ponendo nuovamente a 0 (rosso) il semaforo;

- quando T2 cerca di entrare in sezione critica eseguendo una P(mutex) il semaforo viene decrementato (-1) e T2 va in attesa sulla coda. Lo stesso accade a T3;
- T1 esce dalla sezione critica eseguendo una V(mutex) e sblocca T2, il primo thread in coda portando il semaforo a -1;
- a sua volta T2 esce dalla sezione critica e sblocca T3 che riporterà il semaforo a 1 (verde) uscendo dalla propria sezione critica.

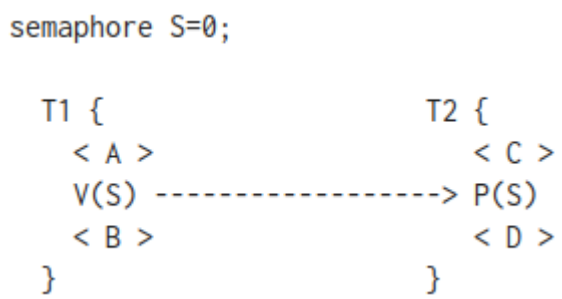
Osserviamo alcuni fatti interessanti:

- per i valori negativi di mutex il valore assoluto corrisponde al numero di thread in coda;
- fuori sezione critica (indicato da -) i thread possono essere eseguiti senza limitazioni, ad esempio i thread sono contemporaneamente in esecuzione all'inizio;
- la sezione critica viene sequenzializzata in modo che non ci sia mai più di un thread in esecuzione. Si può notare infatti che il simbolo = non compare mai contemporaneamente su più thread: viene realizzata la mutua esclusione;
- la sezione critica può andare in parallelo con codice non critico: il simbolo = può andare assieme al simbolo -.

5.2.3 Attendere un altro thread

Esistono altri utilizzi dei semafori, oltre alla realizzazione della sezione critica.

Supponiamo che T2, prima di eseguire la porzione di codice < D > debba attendere che T1 abbia eseguito il codice < A >. Possiamo realizzare questa semplice sincronizzazione con un semaforo S inizializzato a 0 come segue:



Spiegazione: poiché S è inizializzato a 0 (rosso) la P(S) di T2 sarà bloccante e di conseguenza il codice < D > sarà eseguito necessariamente dopo < A >. Quindi, < A > e < C > possono andare in parallelo. Lo stesso vale per < B > e < D >. Anche < B > e < C > possono andare in parallelo in quanto la V(S) non è mai bloccante. Notare che se la V(S) è eseguita prima della P(S) il semaforo diventerà 1 (verde) e la successiva P(S) non sarà bloccante.

Questo semplice esempio ricorda la sincronizzazione che si ottiene nello scambio di messaggi: la V(S) è come una send asincrona mentre la P(S) come una receive sincrona. Abbiamo evidenziato questa analogia con una freccia dalla V(S) alla P(S).

5.2.4 Regolare l’accesso a risorse

Abbiamo visto che un semaforo inizializzato a 1 può essere utilizzato per regolare l’accesso alla sezione critica. In generale, un semaforo iniziato a MAX permette MAX accessi prima di diventare bloccante. Possiamo quindi dare la seguente interpretazione alle operazioni P e V:

- P(S): richiesta di risorsa: se ce ne è almeno una disponibile viene assegnata, altrimenti si attende;
- V(S): rilascio di risorsa: se ci sono thread in attesa il primo viene sbloccato.

Ad esempio se abbiamo 3 stampanti possiamo usare S inizializzato a 3 e quando i primi 3 thread eseguiranno P(S) per allocarsi una stampante, il semaforo diventerà rosso bloccando eventuali altri thread che desiderino stampare. Finita la stampa, l’esecuzione di V(S) sbloccherà il primo thread in attesa.

5.5 Produttore e consumatore con semafori

Proviamo ora a risolvere il problema del produttore-consumatore usando i semafori. Abbiamo due sincronizzazioni da realizzare: quando il buffer è pieno il produttore deve attendere, per evitare di sovrascrivere; quando il buffer è vuoto il consumatore deve attendere, per evitare di leggere celle non ancora scritte.

La soluzione è di usare due semafori distinti che regolano l’accesso a risorse: vuote inizializzato al numero MAX di celle inizialmente vuote e piene inizializzato a 0 in quanto inizialmente non abbiamo celle piene.

5.6 Ecco la soluzione per un produttore e un consumatore:

```
semaphore piene=0, vuote=MAX;

Produttore {
  while(1) {
    < produce d >
    P(vuote); // richiede una cella vuota
    buffer[inserisci] = d;
    inserisci = (inserisci+1) % MAX;
    V(piene); // rilascia una cella piena
  }
}

Consumatore {
  while(1) {
    P(piene); // richiede una cella piena
    d = buffer[preleva];
```

```
    preleva = (preleva+1) % MAX:
    V(vuote); // rilascia una cella vuota
    < consuma d >
  }
}
```

5.7 Tanti produttori e consumatori

Cosa accade se abbiamo tanti produttori e tanti consumatori?

Potrebbero esserci interferenze in scrittura e lettura sul buffer. Ad esempio due produttori potrebbero scrivere sulla stessa cella buffer[inserisci], sovrascrivendosi l'uno con l'altro, e poi entrambi incrementare inserisci. Oppure potrebbero interferire sull'incremento di inserisci, come abbiamo discusso nelle lezioni precedenti.

La soluzione è di proteggere il codice che aggiorna variabili condivise con una sezione critica. Ecco la soluzione con tanti produttori e tanti consumatori utilizzando un semaforo mutex inizializzato a 1:

```
semaphore piene=0, vuote=MAX, mutex=1;

Produttore {
  while(1) {
    < produce d >
    P(vuote); // richiede una cella vuota
    P(mutex); // entra in sezione critica
    buffer[inserisci] = d;
    inserisci = (inserisci+1) % MAX;
    V(mutex); // esce dalla sezione critica
    V(piene); // rilascia una cella piena (l'ha appena creata)
  }
}

Consumatore() {
  while( {
    P(piene); // richiede una cella piena
    P(mutex); // entra in sezione critica
    d = buffer[preleva];
    preleva = (preleva+1) % MAX:
    V(mutex); // esce dalla sezione critica
    V(vuote); // rilascia una cella vuota (ha giá letto)
    < consuma d >
  }
}
```

Notare che se scambiamo P(piene) o P(vuote) con P(mutex) il primo thread che entra acquisisce il mutex e se si blocca in sezione critica potrebbe creare uno stallo. Ad esempio, se venisse schedulato per primo un consumatore, esso si bloccherebbe su P(piene) bloccando, a sua volta, tutti gli altri thread sul mutex indefinitamente. Quando si inseriscono sezioni critiche bisogna quindi fare attenzione che non ci siano semafori bloccanti al loro interno.

6. Semafori POSIX

I semafori POSIX sono semafori contatori che permettono di gestire la sincronizzazione dei thread POSIX.

Esistono altri meccanismi che, per mancanza di tempo, menzionano solamente:

- Pthread Mutex: sono semafori binari utilizzabili per realizzare la sezione critica
- Pthread Condition: vengono utilizzate per ‘simulare’ il costrutto dei monitor che vedremo invece nel linguaggio Java.

6.1 sem_init, sem_wait e sem_post

- `sem_t sem_name`
  dichiara una variabile di tipo semaforo;
- `int sem_init(sem_t *sem, int pshared, unsigned int value)`
  inizializza il semaforo sem al valore value. la variabile pshared indica se il semaforo è condiviso tra thread (true) o processi (false);
- `int sem_wait(sem_t *sem)`
  esegue una P(sem);
- `int sem_post(sem_t *sem)`
  esegue una V(sem);

6.2 sem_getvalue e sem_destroy

- `int sem_getvalue(sem_t *sem, int *val)`
  Legge il valore del semaforo e lo copia in val;

ATTENZIONE: in alcune implementazioni il semaforo rosso è 0, in altre è negativo (e indica il numero di processi in attesa);

- `sem_destroy(sem_t *sem)`

Elimina il semaforo. Da NON usare se ci sono processi in attesa sul semaforo (comportamento non specificato).

NOTA: su MacOS si devono usare i semafori con nome (perché evidentemente pagare 2k di pc non é abbastanza per implementare tutto)

7. Monitor

I semafori sono estremamente potenti e flessibili ma presentano alcune difficoltà:

- La sincronizzazione è gestita in modo decentralizzato dai singoli thread. Se il programmatore non organizza bene le funzioni di sincronizzazione la manutenzione del codice può risultare difficoltosa
- La gestione di mutex e sincronizzazione diventa complessa quando i thread devono attendere per condizioni non facilmente codificabili tramite un singolo semaforo. Ad esempio, nei lettori-scrittori abbiamo visto che viene eseguita un'attesa dentro una sezione critica.

I Monitor sono un costrutto linguistico di Tony Hoare (che li ha inventati nel '74) che permette di risolvere i due problemi discussi sopra.

Da un lato costringe il programmatore a centralizzare tutta la sincronizzazione in un un unico punto del programma (il monitor); dall'altro permette di gestire in modo molto efficace la mutua esclusione, soprattutto nel caso in cui un thread debba attendere su condizione complesse.

I Monitor ricordano le classi della programmazione ad oggetti, in cui procedure e dati sono incapsulati in un'unità modulare: i thread non possono accedere direttamente ai dati ma devono utilizzare le procedure del Monitor.

7.1 cosa ci danno i monitor?

1. Mutua esclusione su tutte le funzioni
2. variabili speciali ***condition*** con due operazioni:
    - `wait`: il thread attende sulla coda relativa alla condition
    - `signal` / `notify`: sblocca il primo thread in attesa sulla coda della condition (se é vuota non fa nulla)

NOTA: Le condition sono simili ai semafori ma non c'è un valore: sono come semafori sempre rossi in quanto la wait è sempre bloccante e la signal o la notify non fa nulla se non ci sono thread in coda.

7.2 Differenza tra signal e notify

Quando viene eseguita una signal o notify e viene sbloccato un thread è necessario gestire la presenza simultanea dei due thread nel Monitor. Ci sono due possibilità:

- signal: Il thread che viene sbloccato dalla signal va subito in esecuzione nel Monitor mentre il thread che ha eseguito la signal attende, su una coda prioritaria, che il thread sbloccato esca dal Monitor;

- notify: Il thread che viene sbloccato dalla notify si mette in coda per riaccedere al Monitor mentre il thread che ha eseguito la notify prosegue la sua esecuzione.

La differenza fondamentale è che con la signal siamo sicuri che il thread sbloccato verrà eseguito immediatamente mentre con la notify verrà eseguito in base allo scheduler. Questo significa che, con la notify, tra lo sbloccaggio e l'effettiva esecuzione potrebbero essere eseguiti altri thread cambiando eventualmente lo stato del Monitor.

Esempio: Produttore-Consumatore

```
Monitor pc {
  dato buffer[MAX];
  int contatore=0, inserisci=0, preleva=0;
  condition piene,vuote;
  void riempi(dato d) {
    if (contatore == MAX)
      vuote.wait(); // buffer pieno attendo
    /* scrivi sul buffer */
    buffer[inserisci] = d;
    inserisci=(inserisci+1)%MAX // Il buffer è circolare
    contatore++; // aggiorno il contatore
    piene.signal();
  }
  dato svuota() {
    if (contatore == 0)
      piene.wait(); // buffer vuoto attendo
    /* leggi dal buffer */
    d = buffer[preleva];
    preleva=(preleva+1)%MAX // Il buffer è circolare
    contatore--; // aggiorno il contatore
    vuote.signal();
    /* consuma d */
  }
}
```

8. Thread e monitor in java

I thread in Java vengono creati estendendo la classe Thread e definendo il metodo run (il codice che il thread eseguirà).

Per eseguire il nuovo thread è sufficiente invocare il metodo start.

```
public class CreaThread extends Thread {
    public void run() {
        System.out.println("Saluti dal thread " + this.getName());
    }

    public static void main(String args[]) {
        CreaThread t = new CreaThread();
        t.start(); //nota che chiama start e non run
    }
}
```

Se c'è esigenza di estendere altre classi, i thread possono anche essere creati come oggetti della classe Thread, al cui costruttore viene passato un oggetto che implementa l'interfaccia Runnable, che richiede l'implementazione del metodo run.

```
public class CreaThread2 implements Runnable {
    public void run() {
        System.out.println("Saluti dal thread "
                           + Thread.currentThread().getName());
    }

    public static void main(String args[]) {
        CreaThread2 r = new CreaThread2();
        Thread t = new Thread(r);
        t.start();
    }
}
```

8.1 Addormentare un Thread

Come per i thead POSIX, possiamo anche "addormentare" un thread Java, in questo caso il tempo è dato in millisecondi ed è necessario gestire l'eccezione InterruptedException sollevata da sleep nel caso il thread t venga interrotto ad esempio tramite t.interrupt().

```
public void run()  {
    try {
        sleep(1000); // attende un secondo
    } catch(InterruptedException e) {
        System.out.println("["+getName()+"]"+" mi hanno interrotto!");
        return;
    }
```

8.2 I Monitor di Java

In Java è implementata una forma semplificata dei monitor con le seguenti caratteristiche:

- Ogni oggetto ha un *mutex* implicito utilizzato per garantire mutua esclusione sui metodi;
- I metodi sono eseguiti in mutua esclusione solo se dichiarati `synchronized`;
- Ogni oggetto ha un'unica condition implicita sulla quale si possono effettuare le operazioni standard `wait()`, `notify()`, `notifyAll()`;
- se il metodo è statico allora il *mutex* è a livello di classe invece che di oggetto;

8.2.1 Sincronizzare porzioni di codice

è inoltre possibile sincronizzare parti di codice di un metodo non synchronized nel seguente modo:

```
synchronized(this) {
            contatore = contatore+1
}
```

`this` indica a quale oggetto fare riferimento per ottenere il lock implicito.
In questo caso è l'oggetto stesso ma è possibile indicare oggetti differenti e quindi sincronizzarsi utilizzando il mutex di tali oggetti.

NOTA: la sincronizzazione è "rientrante": è possibile chiamare un metodo `synchronized` da un altro `synchronized` senza problemi.

Uno zucchero sintattico per avere tutta una funzione in mutua esclusione é il seguente:

```
synchronized foo(){
    System.out.println("this whole function is synchronized");
    ...
}
```

9. Stallo

Stallo o **deadlock**: Un insieme S di processi o thread è in stallo se ognuno attende un evento che può essere causato solo da processi o thread appartenenti ad S.

in sostanza si aspettano a vicenda. (é uno stallo alla messicana!)

esempio scemo:
A: *"Dimmi quanto venivi pagato nell'azienda precedente e ti assumo."*
B: *"Assumimi e ti dico quanto mi pagavano."*

9.1 Condizioni necessarie per lo stallo

Per quanto riguarda l'allocazione delle risorse, lo stallo avviene solo sotto alcune condizioni:

- **Mutua esclusione.** - Altrimenti se una risorsa può essere utilizzata contemporaneamente da più processi o thread non è necessario attendere quindi non si forma una situazione di stallo. Ad esempio la lettura simultanea di dati condivisi non richiede mutua esclusione.
- **Possesso e attesa.** - Altrimenti se i processi o thread non allocano mai le risorse in modo incrementale, acquisendo una risorsa (possesso) e poi chiedendone altre in seguito (attesa) allora non può succedere che si formi un attesa circolare.
- **Assenza di preemption.** - Se il sistema può far rilasciare in modo forzato le risorse a thread o processi, allora è possibile risolvere situazioni di stallo tramite preemption.
- **Attesa circolare.** - Per definizione, lo stallo avviene quando c'è una attesa circolare che fa sì che nessun processo o thread uscirà mai dallo stato di attesa.

9.2 Gestione delle situazioni di stallo

Lo stallo può essere gestito in vari modi a seconda di se e quando si evidenziano (potenziali) situazioni si attesa circolare.

- Prevenzione
- Controllo:
- Riconoscimento
- Nessuna azione

9.2.1 Prevenzione dello stallo

Per prevenire lo stallo di fatto dobbiamo negare una delle condizioni necessarie discusse sopra.

9.2.1.1 Negare la mutua esclusione

Il fatto che l’accesso a una risorsa sia in mutua esclusione dipende dalla risorsa stessa e dal tipo di accesso, quindi non possiamo evitare la mutua esclusione per prevenire lo stallo, quando essa è necessaria.

9.2.1.2 Evitare il possesso e attesa

Possiamo evitare possesso e attesa allocando tutte le risorse assieme.

Attenzione:
non è però sempre possibile perché non è detto che si conoscano tutte le risorse necessarie a priori.

Attenzione:
Inoltre allocare tutte le risorse all’inizio può essere inefficiente in quanto priva gli altri thread o processi di tali risorse e puó quindi causare starvation.

Infatti un thread potrebbe attendere per sempre che TUTTE le risorse a lui necessarie siano disponibili contemporaneamente e magari queste vengono alternativamente allocate e utilizzate da altri thread.

9.2.1.3 Preemption

Togliere "forzatamente una risorsa ad un processo".

L'applicabilitá dipende dal tipo di risorsa siccome sono risorse per loro natura preemptable e altre no. In linea generale, se la risorsa può recuperare successivamente il proprio stato e ripristinarlo a quando era stata tolta allora può essere preemptable (un esempio perfetto é la CPU).

9.2.1.4 Prevenire l’attesa circolare: l’allocazione gerarchica

È possibile evitare l’attesa circolare con opportune strategie di allocazione delle risorse. Un esempio interessante è l’allocazione gerarchica:

Le varie risorse sono ordindate in insiemi ordinati fra loro e il possesso e attesa è ammesso ma solo seguendo l’ordinamento (stretto) degli insiemi Ri.

Questa politica di allocazione previene l’attesa circolare.

Esempio:

R -> P indica che P possiede almeno una risorsa di tipo R
P -> R indica che P sta chiedendo una risorsa di tipo R

Supponiamo per assurdo che ci sia un assegnamento di risorse che crea un attesa circolare. Avremo quindi

R1 -> P1 -> R2 -> P2 … Rw -> Pw -> R1

l’allocazione gerarchica richiede che R1 < R2 < … < Rw < R1 che implica l’assurdo R1 < R1 .
Abbiamo quindi dimostrato che l’allocazione gerarchica previene l’attesa circolare e di conseguenza lo stallo.

Cosa accade se un processo viola la politica di allocazione gerarchica? La richiesta di una risorsa che non rispetta la gerarchia (che non è quindi strettamente maggiore delle risorse attualmente possedute dal processo) viene rifiutata e ritorna un apposito codice di errore. Il processo dovrà gestire l’errore.

10. Controllo dello stallo

Sicuramente il miglior modo per evitarere lo stallo é negare almeno una delle condizioni necessarie

Per poter controllare ed evitare la formazione degli stalli a run-time l'idea è di consentire l'assegnamento delle risorse solo nel caso in cui il sistema possa garantire che tale assegnamento non porterà a una situazione di stallo.
(che ricordo essere **Mutua esclusione**, **possesso e attesa**, **assenza di preemption**, **attesa circolare**).
Tramite tali strategie, lo stallo non può formarsi.

Vediamo adesso come poter controllare lo stallo a runtime, possiamo avere come idea di base quella di **consentire l'assegnamento delle risorse solo nel caso in cui il sistema possa garantire che tale assegnamento non porterà a una situazione di stallo**.

10.1 Grafo di assegnazione

Tramite un grafo che tiene traccia delle varie assegnazioni é possibile gestire processi e risorse. Esso è costituito da 2 tipi di nodi:

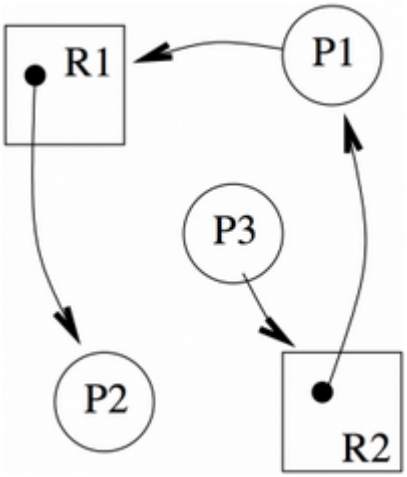
- processi/thread (P)
- risorse (R)
  Esistono tre tipi di arco che connettono un processo/thread a una risorsa:

P ← R : La risorsa R è assegnata al processo P

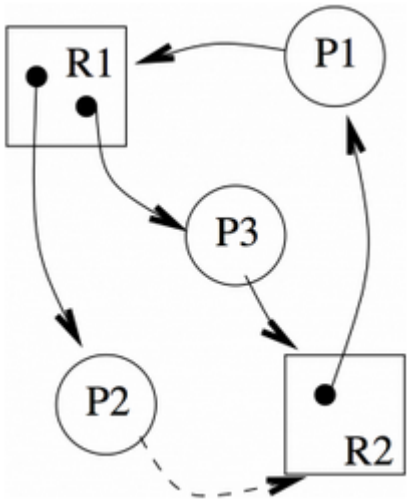
P → R : Il processo P chiede la risorsa R

P ⇢ R : Il processo P potrà chiedere in futuro la risorsa R

Una risorsa può offrire una o più istanze. La molteplicità viene rappresentata indicando all'interno del nodo un • per ogni istanza presente. Ad esempio:



- Le risorse R1 e R2 hanno ognuna una singola istanza
- La risorsa R1 è assegnata a P2 e la risorsa R2 è assegnata a P1
- Il processo P1 chiede la risorsa R1
- Il processo P3 chiede la risorsa R2



in aggiunta rispetto all'esempio precedente, abbiamo:

- Una ulteriore risorsa R1 assegnata a P3
- Una possibile richiesta futura da parte di P2 della risorsa R2. (Questa richiesta non è stata effettuata ma il grafo indica la possibilità che tale richiesta avvenga in futuro.)

10.1.1 Grafo sicuro

Definizione:
Un grafo di assegnazione è sicuro se è privo di stallo anche considerando le richieste future.

Come si fa a decidere se, dato un grafo, esiste una situazione di stallo?

1. Se c'è una sola istanza per risorsa: c'é stallo sse esiste un ciclo nel grafo;
2. Se ci sono più istanze per risorsa (caso generale): lo stallo implica l'esistenza di un ciclo nel grafo;

Quindi nel primo caso è sufficiente verificare se il grafo è ciclico.
Nel secondo caso, invece, serve un algoritmo diverso.

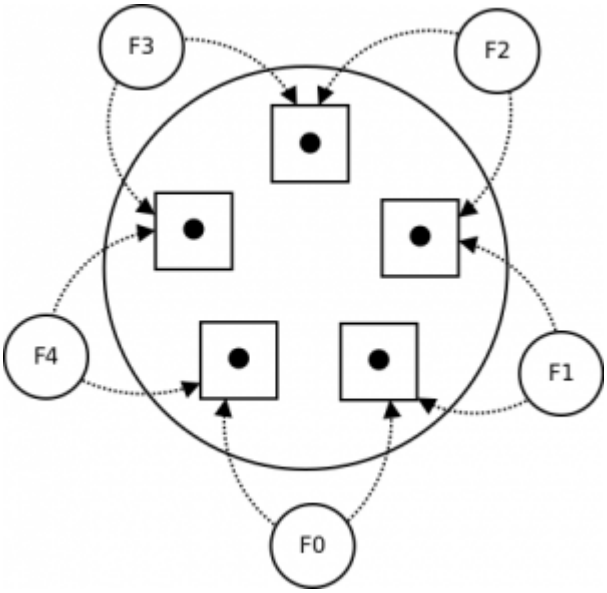
10.2 Algoritmo con grafo di assegnazione

Nel caso con una sola istanza per risorsa, l'individuazione di uno stallo equivale alla ricerca di un ciclo nel grafo.

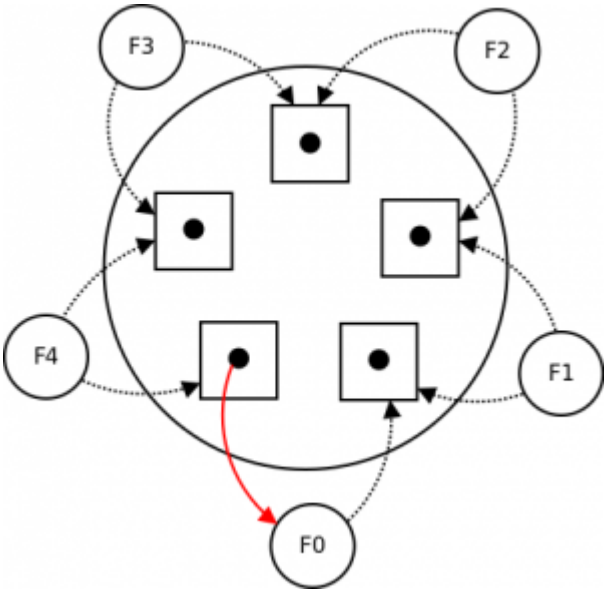
Algoritmo
Per ogni richiesta di risorsa, se la risorsa non è disponibile il processo attende, altrimenti:

1. Simula l'assegnamento della risorsa al processo
2. Verifica se il grafo contiene uno stallo (in questo caso se è presente un ciclo), considerando anche tutte le richieste future:
    - In caso di stallo: il processo viene messo in attesa della risorsa
    - Altrimenti: la risorsa viene assegnata e la modifica sul grafo viene confermata

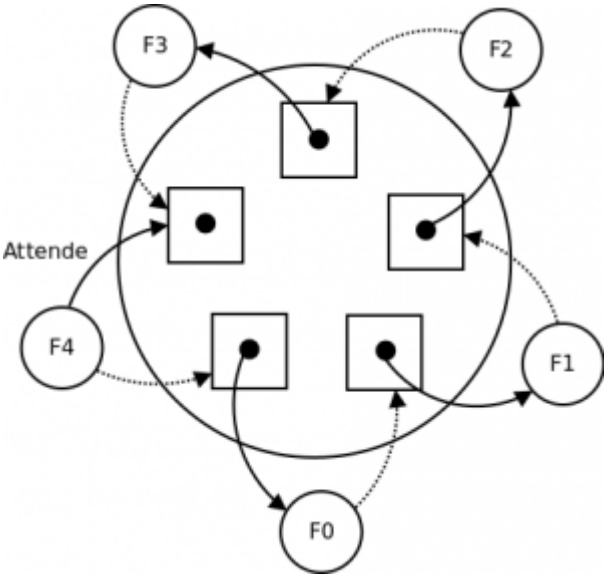
Portando il classico esmpio dei filosofi a cena, simuliamo ogni volta che un filosofo chiede l'assegnamento di dare la risorsa e vediamo come va:



Sopra abbiamo tutti gli assegnamenti "potenziali"



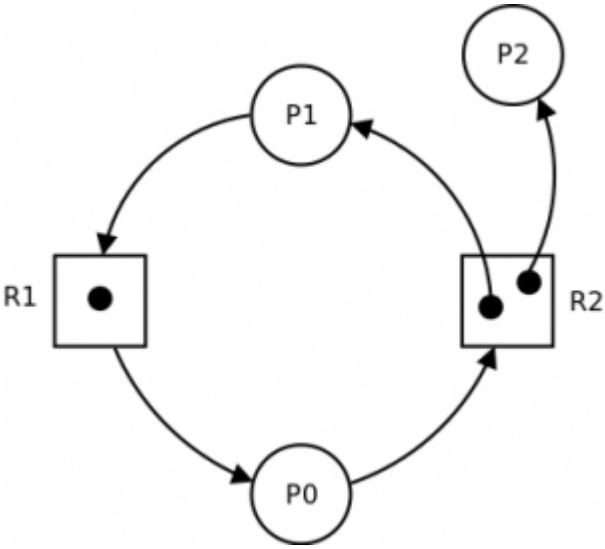
Simuliamo di dare al filosofo F0 la bacchetta alla sua sinista



Andando avanti avremo che se tutti prenderanno la bacchetta sinistra non avremo problemi... ma se il filosofo F4 chiedesse la bacchetta destra avremmo un ciclo di risorse!

10.3 Algoritmo del banchiere

Nel caso in cui ci sono diverse istanze per ogni risorsa non é possibile utilizzare la presenza di cicli come criterio per individuare potenziali stalli siccome una stessa risorsa puó essere assegnata contemporaneamente N volte.



prendendo questo esempio P2 può rilasciare la risorsa R2 che può essere assegnata a P0 che a sua volta libererà R1 per P1. Intuitivamente i processi possono “terminare” nella sequenza <P2,P0,P1>. In questo caso diciamo che tale sequenza è una sequenza “sicura” o di “terminazione”.

Definizione:
Una sequenza <P1,P2,…, PN> di processi è una sequenza sicura se ogni processo P<n> nella sequenza può ottenere tutte le risorse che necessita (incluse quelle future) da quelle disponibili inizialmente più quelle possedute dai processi che lo precedono nella sequenza cioè dai processi P<n-x> tali che <n-x> < i.

Intuitivamente una sequenza sicura è una sequenza che permette ai processi di ottenere tutte le risorse che necessitano e quindi di rilasciarle a vantaggio dei processi successivi.

Per computare se esiste almeno una sequenza sicura si può utilizzare il seguente algoritmo:

1. Cerca un processo che possa ottenere tutte le risorse necessarie (incluse quelle future) da quelle disponibili. Se tale processo non c’è l’algoritmo fallisce (non esiste una sequenza sicura)
2. Rilascia tutte le risorse possedute dal processo, aggiungilo alla sequenza sicura e toglilo dal grafo
3. Se ci sono ancora processi nel grafo vai al punto 1, altrimenti dai in output la sequenza sicura

L’algoritmo del banchiere è identico all’algoritmo con grafo di assegnazione ma utilizza la non esistenza di una sequenza sicura per individuare potenziali stalli:

Per ogni richiesta di risorsa, se la risorsa non è disponibile il processo attende, altrimenti:

1. Simula, sul grafo, l’assegnamento della risorsa al processo
2. Verifica se il grafo contiene uno stallo (se non esiste una sequenza sicura), considerando anche tutte le richieste future.
    - In caso di stallo: il processo viene messo in attesa della risorsa
    - Altrimenti: la risorsa viene assegnata e la modifica sul grafo viene confermata