

Lezione del 14/10/2020 – Comandi della shell

La shell rappresenta l'interfaccia con cui l'utente interagisce attraverso linea di comando. Prende il nome dal fatto che indica "lo strato piu' esterno del computer". Per usare la shell, prima bisogna mettere il predicato e poi i complementi oggetti.

Es: `ls /`, con questa riga indico che "lista gli elementi della radice", e appunto come si nota abbiamo messo prima il predicato e poi il complemento oggetto. I comandi della shell sono dei programmi normalissimi. Posso usare `ls` anche per fare delle ricerche nei nomi dei file (pattern matching), usando i caratteri "???" oppure usando gli "*" per i caratteri singoli (wildcard). Le wildcard inoltre permettono anche di verificare la presenza di un range di caratteri, es: `ls ../*[0-9]` mostra i contenuti delle cartelle avente una cifra come suffisso nel nome (se volessi il prefisso, mi basta mettere l'asterisco a destra).

Alcuni comandi:

<code>echo \$PATH</code>	Con questo comando mostro le cartelle in cui sono presenti gli eseguibili dei comandi della shell. PATH infatti e' una variabile globale della shell in cui sono presenti esattamente questi.
<code>bin/sleep/</code>	Comando sleep, solo preso facendo riferimento alla cartella in cui si trova effettivamente.
<code>ls -l /bin</code>	Lista tutti i file in /bin mostrando anche i dettagli (-l sta per long) Se al posto di -l metto -dl, allora mostra le info della directory.

In Unix esiste il concetto di gruppo di proprietari di file. Con `ls -l` appunto possiamo vedere nella prima colonna i permessi che un utente ha (la riga -rwxr-xr-x), nella terza colonna vediamo invece l'utente proprietario e nella quarta il gruppo degli utenti proprietari. Nella seconda colonna vediamo il numero di "nomi" con il quale posso fare riferimento a un dato file (hard links). Quinta riga indica la dimensione, sesta la data dell'ultima modifica.

Un comando molto importante e' il comando `man`, che fa da "manuale" di linux. Basta scrivere man e specificare un comando per capire cosa fanno questo e i suoi parametri. Possiamo anche fare `man man`, ed avremo l'intera guida del funzionamento di man con la descrizione dei suoi capitoli (il capitolo 2 del man e' dedicato alle SYSTEM CALL). Il comando `$ man [nome capitolo] intro` mostra l'introduzione al capitolo del man, descrivendo di cosa parla. Il capitolo 4 e' dedicato ai file speciali: Unix infatti permette di usare dei file speciale per accedere ad esempio ai dispositivi (es: stampanti), ma che possono essere usati anche per altro (questi file speciali hanno una lettera "c" davanti alla descrizione dei permessi quando si usa il comando `ls -l`).

Comandi:

<code>cat [arg]</code>	Copia i dati in input nell'output (es. file o stringa). Se gli argomenti non vengono specificati, semplicemente copia le stringhe di caratteri scritte.
<code>groups</code>	Mostra i gruppi a cui lo user appartiene.
<code>whoami</code>	Mostra il nome dello user.
<code>stat [arg]</code>	Fa la stessa cosa di <code>ls -l</code> per un file, ma mostrando piu' dettagli.

Con CTRL+D possiamo terminare la fase di input dal terminale, e tornare a scrivere i comandi. Con TAB, la shell ci autocompleta il comando. WARNING: se avessi fatto il comando `sudo cat < /dev/urandom > /dev/vda`, avrei distrutto il sistema. Il disco si sarebbe riempito di dati random! Per questo motivo usare sudo (super user do, da i permessi massimi all'operazione che si vuole svolgere) e' sconsigliato, soprattutto quando si lavora su dati sensibili. (Il prof ha parlato di GTA

qui, letteralmente di Grand Theft Auto, e solo perche' su linux ci sono giochi, come mostra il capitolo 6 del man. Purtroppo, linux non viene distribuito con una copia di GTA installata.)

Cartelle nella root:

Nella cartella `/etc`, abbiamo i file di configurazione.

Il comando `cat /etc/passwd` mostra tutti gli utenti presenti nella macchina. Alcuni sono falsi e servono unicamente per scrivere i permessi sui file. Un tempo questo comando ora mostrava anche la password dell'utente, ma in forma criptata; nonostante cio', questa sostituita da una "x" nelle nuove versioni di linux.

Nella cartella `/bin`, abbiamo l'elenco di alcuni dei comandi.

Nella cartella `/lib`, abbiamo le librerie del sistema. Tuttavia, non ci vengono mostrate tutte le librerie se scriviamo `ls /lib`, questo perche' debian (il prof usa debian btw) supporta numerosissime architetture cpu (es. x86_64, x86, PowerPC, ARMv7) diverse, quindi mostrarle tutte sarebbe un delirio. In linux, le librerie dinamiche presentano il suffisso ".so" (che sta per shared object), e quelle statiche invece presentano il suffisso ".a".

Nella cartella `/var`, abbiamo informazioni come i log di sistema, la cache, l'elenco dei pacchetti e gli SPOOL. Questi ultimi sono archivi temporanei che servono per gestire sequenze di eventi. Lo spool piu' famoso e' quello della posta elettronica.

Nella cartella `/usr`, abbiamo sbin, src, lib, include e games. Qui, la cartella `/local`, il suo contenuto e' gestito dalla distribuzione di linux. La cartella `/include` contiene i file .h, ovvero le librerie c e c++. (N.B. una distro di linux e' una antologia di strumenti software gestiti in maniera armoniosa. Le distro di linux NON SONO sistemi operativi, bensì sono posti sopra un sistema operativo (in questo caso linux). Quindi, meglio non dire che debian, ubuntu etc sono sistemi operativi).

Col comando `cat hw.c > hw2.c`, ho ridirezionato l'input nell'output, cioe' ho copiato il contenuto di "hw.c" dentro ad "hw2.c". Posso anche usare il minore per fare l'operazione inversa (DA CONTROLLARE). I simboli maggiore e minore ci permettono di fare esattamente questo, ovvero di ridirezionare per l'output di un comando dentro a un file, permettendo di scrivere dentro di esso.

<code>grep [arg]</code>	Ricerca l'arg nell'input stream.
<code>sed [str arg]</code>	Prende in input una stringa e in base a quella sostituisce i pattern nell'input stream attraverso pattern matching.
<code>diff [arg] [arg2]</code>	Mostra le differenze fra il primo e il secondo file.
<code>tee [arg]</code>	Legge dallo stdin e copia nello stdout
<code>touch [arg]</code>	Crea il file [arg].
<code>ln [nomefile] [nomelink]</code>	Crea un hard link a un certo file (simile a collegamenti in Windows).
<code>type [espressione]</code>	Mostra il tipo dell'espressione che sto usando (e' un comando linux, e' un file etc...)

Gli errori e le eccezioni che vengono lanciate sul terminale vengono mostrati scrivendo nello stderr (standard error), che e' appunto una specie di stdout riservato agli errori. L'utilita' di questo sta nel fatto che se per esempio dovessimo ridirezionare l'output in un file, gli errori non vengano "persi" o confusi dentro all'output.

stdin, stdout e stderr sono detti **standard streams**.

Lezione del 16/10/2020 – Permessi dei file (e comandi pt. 2) ed inizio programmazione in C

Possiamo aggiungere nei comandi anche un modo per mettere una stringa con un comando, delimitato dagli apici (o accenti strani come li chiamo io).

input	output
<code>echo "Questa directory contiene `ls -a`"</code>	Questa directory contiene .
<code>echo "Questa directory contiene \$(ls -a)"</code>	..

Permessi:

-rw-r--r-- indica che io ho permessi di lettura e scrittura, il mio gruppo ha permessi di lettura e gli altri utenti pure hanno solo permessi di lettura. Come faccio quindi a cambiare i **permessi** di un file? Uso il comando **chmod**! Con questo comando possiamo cambiare i permessi di un file:

chmod [ugoa] [+ -] [rwx] [file] – cambia i permessi di un file in base all'utente (u), gruppo (g), other (o, utenti fuori dal gruppo e dall'utente stesso) o tutti (a), aggiungendo o togliendo (+ o -) la possibilità di eseguire (x), scrivere (w) o leggere (r) tale file. Es: **chmod g+w [file]** → il gruppo a cui l'utente appartiene può ora scrivere sul file. Al posto della notazione g+w, posso anche usare una notazione basata sui valori binari. Ovvero, qualcosa come **chmod 755 [file]** mi direbbe che devo dare i permessi di rwx all'utente corrente (111 = 7), dare solo i permessi di rx al gruppo (101 = 5) e stessa cosa per l'other.

In realtà però ci sono altri permessi fuori dal rwx, infatti se andiamo a controllare ad esempio il file **/usr/bin/passw** noteremo che abbiamo anche il permesso s (che sta per setuserid), per dire che appunto solo l'utente "root" può accedervi.

Possiamo creare anche dei file di testo eseguibili dalla shell, semplicemente aggiungendo la locuzione **#!/bin/bash** e scrivendo subito i comandi che il file dovrà eseguire. Per chiamarlo, basterà usare il comando **./[file]**. N.B. questi comandi nel file però avranno come input il file stesso!

Inizio Programmazione in C

<code>int i = (3 + 2), 10;</code>	Questo codice da come risultato i=10, la parte prima della virgola ignorata.
<code>42;</code>	Questo riga di codice è ammissibile, ma non fa nulla

Per scrivere un programma che prende argomenti in input da linea di comando, devo aggiungere nel main anche i parametri **int argc** e **char* argv[]** (il primo parametro indica la lunghezza dell'array **argv[]**, e la seconda indica l'array di stringhe che si passano come input dei comandi). Per fare riferimento a uno dei caratteri delle stringhe date come argomento, bisogna usare l'espressione **argv[numero_stringa][numero_carattere]**.

Per compilare un programma in c, bisogna usare il comando **gcc -o file file.c**, e eseguirlo con **./file**. Inoltre, ogni volta che non si usano delle librerie standard di c bisogna aggiungere il flag **-l[nome_libreria_senza_h]** alla fine del comando.

Se abbiamo una stringa molto lunga, possiamo assegnarla ad una variabile separandola in due parti più piccole, aggiungendo uno spazio fra le due stringhe.

#ifndef FILE_H, espressione del file c che serve ad evitare la doppia inclusione del file nel progetto/programma.

`#define STAMPA(X) printf("%d\n", (X))`, questa viene detta macro, ed essenzialmente la funzione `stampa(X)` viene sostituita con `printf.etc..` ed la variabile `X` viene sostituita con la variabile `X` in `printf`.

Posso anche giocare con i nomi delle variabili nelle macro:

`#define STAMPA(X) printf("var = " #X "%d\n", (X))`, vuol dire letteralmente "trasforma il nome della variabile in stringa e mettilo lì".

`#define STAMPA(F, X) F ## f("var = " #X "%d\n", (X))`, aggiunge il valore (si esatto, il valore, non il contenuto. Quindi tu puoi scrivere una stringa senza le virgolette come parametro `F` e non solo te lo prende giusto, te lo sostituisce pure all'`F` nella macro!) di `F` concatenandolo al resto della macro, es: `int i = 0; STAMPA(print, i)` viene tradotto con `printf(etc...) [HA AGGIUNTO LA f AL print]`

Tipo `union`, ha la stessa sintassi e uso della `struct`, ma i campi sono SOVRAPPOSTI, ovvero la loro memoria comincia allo stesso indirizzo, dunque condividono le stesse celle. E' quindi un modo di ottimizzare la memoria.

Esempio 1:

```
union un{
    uint32_t i32;
    uint8_t i8[4];
}

int main(){
    union un u;
    u.i32 = 1040;
    uint8_t i;
    for(int i=0; i<4; i++){
        printf("%d ", u.i8[i]);
    }
    return 0;
}
```

Il codice ritorna 16 4 0 0.
Quella union e' come scrivere 1040 in int a 32 bit, quello l'ha letto come un 4 byte. 16 4 0 0 sono quindi i "byte", e 16 e' il meno significativo (i byte sono disposti al contrario).

Esempio 2:

```
#include <stdio.h>
struct oggetto{
    int tipo;
    union{
        struct {
            char* marca;
            int cilindrata;
            struct {
                int eta;
                float peso;
            };
        };
    };
};

void f(struct oggetto *ogg){}
enum tipo{AUTO, UMANO};

int main(){
    struct oggetto o = {.tipo = AUTO, .marca = "FCA", .cilindrata = 1000 };
    struct oggetto o2 = {.tipo = UMANO, .eta = 20, .peso = 70.5};
    f(&(struct oggetto){.tipo = AUTO, .marca = "FCA", .cilindrata = 1000 });
    //abbiamo fatto casting a puntatore per struct oggetto
    return 0;
}
```

Per C, 0 e' sempre falso e 1 e' sempre vero, per qualsiasi struttura dati. Talvolta nel codice troviamo istruzioni come `!!x`, che utile per convertire una variabile in un valore di verita' (booleano) 0 o 1.

Lezione del (21/10/2021) - Inizio Programmazione Concorrente

Unix non e' un sistema operativo, ma e' uno standard di un sistema operativo. Infatti, si dice UNIX-like qualsiasi sistema operativo che soddisfa le regole di POSIX IEEE 1003, in particolare 1003.1 sono le system call. Per creare la **simultaneita'**, POSIX ci dice di usare i **pthread**. In inglese con thread si traduce con "filo", quindi e' come si facesse un programma con piu' fili.

Usando le librerie `pthread.h` e `unistd.h` abbiamo a disposizione le funzioni e le strutture per permettere di costruire i thread e la simultaneita' dei processi.

A seguire, una introduzione di alcune delle funzioni e delle strutture necessarie per descrivere i thread.

<code>pthread_t nome_t;</code>	Struttura dati usata per descrivere e dichiarare un thread.
<code>pthread_create(&nome_t, NULL, f_to_execute, NULL);</code>	Crea ed un thread, assegnandoli una funzione che questo dovra' eseguire.
<code>pthread_join(nome_t, NULL);</code>	Segna quando eseguire il thread / attiva il thread. Anche senza questa funzione, il primo thread creato svolge comunque una piccola parte del suo codice, ma poi si ferma appena puo'.

Cosa succede se facciamo una modifica concorrente ad una variabile? Facciamo per esempio 1000 di somme e sottrazioni con due funzioni p1 e p2 ad una variabile `sum = 0`. Noteremo che si otterranno dei valori che sono praticamente **casuali**. Questo avviene perche' succede che il perche' mentre il processo p1 sta modificando la variabile (in una di quelle 1000 somme), al contempo anche il processo p1 sta eseguendo la somma (proprio perche' e' una modifica concorrente), dunque p2 potrebbe leggere un valore che non e' ancora stato modificato. Infatti, per ogni operazione di somma in p1, vengono eseguite in realta' 3 operazioni:

- 1) Viene prelevato il valore di `sum` e messo nel registro `%eax`.
- 2) Viene aggiunto 1 a `%eax`.
- 3) Viene prelevato il valore di `%eax` e messo in `sum`.

Questo ci porta a parlare delle bestie nere della programmazione concorrente:

- **Race condition** (condizioni di gara) – si ha quando il risultato non dipende piu' dal programma scritto, ma dipende da come si sono succedute le istruzioni di esecuzione del programma, e quindi da come si sono correlati i vari componenti simultanei. Piu' avanti troveremo il modo di risolvere questo problema.
- **Deadlock** – prendiamo questo esempio: se un thread t1 vuole prendere il controllo univoco di una certa istruzione A (**Mutua Esclusione**) per risolvere il problema A, pero' ha anche bisogno di un'altra istruzione B per risolvere il problema B, del quale pero' un altro thread t2 ha il controllo univoco e intanto t2 ricerca l'istruzione A, il sistema si blocca e va appunto in deadlock, perche' nessuno dei due thread potra' proseguire finche' t1 non avra' perso il controllo di A (il che si avra' solo se prendera' il controllo di B) e cosi' via. In poche parole, due o piu' programmi aspettano il loro turno senza mai proseguire e bloccandosi a vicenda.

Per descrivere i codici dei programmi concorrenti, il prof fa uso di dei modelli astratti: non usa codice vero, ma uno pseudo-pseudo-codice che permette di esprimere facilmente il significato di un programma.

Il concetto di **processo** cambia con la programmazione concorrente: normalmente un processo e' l'attivit  risultante dall'esecuzione di un programma. Ma in questo modo si associa con processo al singolo 'filo esecutivo' (prima fa una cosa, poi un'altra etc..), che pero' non e' cosi'; nell'atto pratico, il

processo diventa esclusivamente l'intestatario delle risorse per poter eseguire un programma. Il programma infatti ha 1 o piu' fili esecutivi, che appartengono al processo, dunque non e' piu' sequenziale, ma ha piu' parti contemporanee e simultanee, e la storia esecutiva non e' piu' univoca, ma e' data dal punto e dallo stato di ogni filo in ogni istante. (**META** – nonostante cio', il prof alcune volte intercambia le due definizioni, e' quindi necessario prestare molta attenzione).

Nonostante cio', comunque nei sistemi formali (o modelli) con processo intendo il singolo filo esecutivo (anche se appunto non e' cosi' come visto prima).

Posso distinguere due tipi di sistemi formali diversi: sistemi a **memoria privata** e sistemi a **memoria condivisa**. Nel primo caso ogni processo ha la sua memoria privata, mentre nel secondo piu' processi condividono la stessa memoria.

[DISCLAIMER- ci sono due cose da tenere a mente che davoli dira' nelle lezioni successive:

- Noi trattiamo in particolare di sistemi a programmazione concorrente a memoria condivisa.
- I thread vengono usati a livello utente, mentre la fork() e' al livello kernel! Infatti, siccome noi principalmente non usiamo i thread, dovremo preoccuparci di gestire la memoria condivisa (a livello kernel infatti non c'e' protezione di memoria). Chi usa i thread (ovvero i processi a livello utente) solitamente non ha questo problema, in quanto i thread sono a memoria privata e possono passarsi le informazioni solomandente spedendosele.
- Dopo aver visto le lezioni del secondo semestre, posso confermare che davoli se ne frega altamente del significato di thread e di processi. A volte li intercambia, a volte dice che il thread e' il contenitore di un processo, altre ancora che il processo e' un contenitore di thread. Porca vacca.

]

Abbiamo un'assioma fondamentale, senza il quale tutta la teoria non sta in piedi. Questo si chiama assioma di **finite progress**, che dice che i processi avanzano sempre, quindi dobbiamo mettere come vincolo che tutte le istruzioni abbiano lunghezza finita e non infinita, altrimenti il deadlock e' assicurato.

Oltre a cio', dobbiamo riconoscere la differenza tra parallelismo reale (nei sistemi multicore, che hanno piu' processi che lavorano in parallelo, SMP) e parallelismo apparente (come appunto succedeva col cuoco che cambiava uniforme, e che quindi e' un parallelismo che avviene in una singola CPU). Il meccanismo che permette il parallelismo apparente e' detto **interleaving**, e consiste appunto nell'alternarsi compiti diversi.

Con **atomo** intendiamo istruzioni che semplicemente o vengono svolti o no, e che quindi non sono ulteriormente separabili. Nell'esempio con p1 e p2, l'intera funzione p1 come abbiamo visto non era un atomo, siccome infatti l'esecuzione del for a un certo punto e' stata interrotta dal passaggio al secondo thread. L'interruzione dell'esecuzione di una funzione da parte di un thread e' causata dalla presenza di un **interrupt**, che puo' essere un interrupt time (che ci dice che il processo ha "finito il suo tempo" e deve passare all'altro) oppure un interrupt generato da un device che fa prendere il controllo al kernel. Ad ogni ciclo di clock il processore controlla se c'e' un interrupt. Il kernel decide quale dei thread deve prendere il controllo (quindi fare un thread per ogni processo non mi basta).

Normalmente, le istruzioni assembler sono atomiche, tuttavia molti processori oggi contengono degli Instruction Set con delle macrofunzioni che si espandono in piu' righe di assembly. Ad esempio, nelle macchine RISC l'assegnamento di un numero abbastanza grande richiede due istruzioni, una per la bassa e l'altra per quella alta.

Anche se il problema dell'atomicita' vera e propria non c'e' nelle macchine multicore, c'e' comunque il problema della condivisione del BUS. L'arbitraggio cosi' verra deciso un po' prima e un po' dopo rispetto all'accesso vero e proprio alla memoria.

Il risultato e' che, in questo modo, la teoria della programmazione concorrente **non cambia** per i due parallelismi.

Il problema che rimane però è decidere nel modello quali istruzioni sono atomiche e quali no. Consideriamo atomiche l'assegnamento di una costante, il resto no. Quindi, anche incremento, operazioni logiche etc non le considereremo atomiche.

L'altra cosa sulla quale dobbiamo accordarci nei nostri modelli è come spieghiamo/indichiamo i programmi concorrenti (essenzialmente la notazione). L'idea è di usare la keyword process, e con questa assicuriamo anche che i nostri processi sono sincronizzati.

Ci sono due proprietà che ogni programma concorrente deve rispettare:

- Proprietà di **liveness**: “Qualcosa di buono prima o poi accade”, e alla base di questa proprietà sta il fatto che il programma prima o poi DEVE finire, e che quindi la sua esecuzione non deve loopare. La deadlock e la starvation sono i risultati di non aver rispettato questa proprietà’.
- Proprietà di **safety**: “Non deve accadere niente di cattivo”, e alla base di questa proprietà ci deve essere che il programma DEVE ritornare il risultato voluto e corretto.

Un programma che procede all'infinito rispetta la proprietà di safety, perché il risultato non viene dato (quindi assumo che potrebbe dare il risultato che voglio, usando l'approccio alla logica intuizionista), ma non quella di liveness.

Nella programmazione concorrente (ed anche distribuita, ovvero in cui ho proprio più sistemi), c'è il cosiddetto problema del consenso (**byzantine problem**), in cui abbiamo più processi che comunicano fra loro e devono arrivare ad accordarsi su un valore, e devono farlo nonostante ci possano essere alcuni di questi processi che possono guastarsi. Anche qui quindi, tutti i processi dovranno decidere un valore (bisogna rispettare liveness), e quello risultante dovrà essere fra quelli proposti. Ma in questo modo potrei scegliere anche risultati che sono banali, quindi devo aggiungere una terza proprietà: ogni processo che non sia guasto dovrà decidere il valore (infatti tutti i processi potrebbero andare in loop, oppure un processo guasto potrebbe andarci).

Quindi, ora non ci resta che trovare un modo per far sì che quelle 3 istruzioni assembly di prima vengano fatte tutte o nessuna, in modo da creare atomicità quasi. In poche parole, vogliamo fare in modo che le risorse siano riservate al thread t1 quando viene eseguita una certa istruzione importante, in modo così che questa istruzione sia riservata a t1 e basta. Questo significa che vogliamo permettere la **mutua esclusione**. Le istruzioni che richiedono o in cui vogliamo implementare questo meccanismo prendono il nome di **sezione critica**, e queste istruzioni o verranno eseguite tutte oppure nessuna.

Le caratteristiche che dovrà avere la sezione critica sono:

- **Mutua esclusione**, ovvero che l'accesso alle istruzioni sia esclusivo. In questo modo, solo un processo (un thread) alla volta esegue quel codice.
- **Non ci sia deadlock**, infatti i processi potrebbero tentare di accedere alla sezione critica e bloccarsi a vicenda, e questo non ci piace.
- **Non ci sia starvation**, ovvero un processo che vuole entrare nella sez. critica, prima o poi dovrà riuscirci. Non deve proseguire solo il processo più veloce o con priorità più alta.
- **No attese non necessarie**. Voglio che se nessun altro abbia bisogno di accedere alla sezione critica, io vi ci possa accedere ASAP. In poche parole, se io voglio accedere alla sezione critica e nessuno ne fa domanda, devo accedervi SUBITO, senza aspettare ritardi inutili.

La prima serve per garantire la proprietà di safety, le altre tre quella di liveness.

L'**algoritmo di Dekker** ci permette di realizzare sezioni critiche nel nostro modello avente solo l'assegnamento come istruzione atomica. Nell'articolo di Dijkstra in cui illustra il funzionamento dell'algoritmo di Dekker, l'autore procede a spiegarlo facendo 4 esempi successivi contenenti errori e come risolverli; noi useremo lo stesso approccio per spiegarlo. (paper originale qui:

<https://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html>)

N.B: dal commento `//codice non critico` a quello `//codice critico` dobbiamo immaginarci che ci siano due funzioni `ccenter()` e `csexit()` rispettivamente subito sotto a ciascun commento, che indicano che si sta entrando ed uscendo in una sezione critica.

<pre> turn = p p: process while(true) //CODICE NON CRITICO while(turn == Q) pass //CODICE CRITICO turn = Q Q: process while(true) //CODICE NON CRITICO while(turn == p) pass //CODICE CRITICO turn = p </pre> <p>Questa e' una specie di soluzione a turni, quando p arriva che e' il turno di Q, attende, e la stessa cosa vale per Q.</p> <p>In questo caso, il programma e' funzionante, ma la terza proprieta' non viene rispettata. Infatti abbiamo il fattore dell'attesa in caso il turno sia occupato.</p>	<pre> inP = false inQ = false p: process while(true) //codice non critico while(inQ) pass inP = true //codice critico inP = false Q: process while(true) //codice non critico while(inP) pass inQ = true //codice critico inQ = false </pre> <p>Questo approccio non e' corretto. Mettiamo che p e Q arrivino contemporaneamente al secondo while: succedera' che entrambi svolgeranno la parte critica (provocando errori) e in piu' entreranno in deadlock, siccome entrambi i booleani sono settati a true.</p>
<pre> inP = false inQ = false p: process while(true) //codice non critico inP = true while(inQ) inP = false pass //attende un po inP = true //codice critico inP = false Q: process while(true) //codice non critico inQ = true while(inP) inQ = false pass //attende un po inQ = true //codice critico inQ = false </pre> <p>Questo codice genera starvation, siccome i due processi potrebbero procedere all'unisono.</p>	<pre> needP = false needQ = false turn = P p: process while(true) //codice non critico needP=true while(needQ) needP = false while (turn == Q) pass needP = true //codice critico turn = Q needP = false Q: process while(true) //codice non critico needQ = true while(needP) needQ=false while (turn == P) pass needQ = true //codice critico turn = P needQ = false </pre> <p>Questo codice svolge l'algoritmo correttamente.</p>

- mutex: dimostrazione per assurdo: supponiamo entrambi siano in codice critico, siano riusciti ad entrarci => sia needP che needQ sono veri. Supponiamo P sia entrato per primo, turno doveva essere P ma perchè Q entri turno deve essere Q. Assurdo.

- no deadlock: nel caso ci fosse, tutte e due i processi sarebbero bloccati al primo while perchè avvenga deadlock servirebbe che entrambi i processi rimangano bloccati in uno dei due while all'inizio, ma perchè siano nel loop occorre che .sia needP che needQ siano veri contemporaneamente, ma è assurdo e uno dei due esce.

- no starvation: mettiamo che p voglia entrare: nel peggiore dei casi Q dà il turno a P quando esce, quindi basta aspettare un turno, e non può esserci starvation

L'algoritmo di Dekker crea una sezione critica, risolvendo così il problema dei ritardi inutili, della mutex, della deadlock e della starvation.

Lezione del (23/10/2020) – Introduzione alle SysCall

Una **System call** (trad. chiamata di sistema) è una primitiva del sistema operativo (in particolare kernel) che consente accesso alle funzionalità da esso offerte. L'invocazione di una syscall è realizzata mediante **trap**, che consente l'esecuzione di codice kernel e il passaggio dallo user-mode al kernel-mode non appena ci si imbatte su di essa.

Le trap sono particolari tipi di interrupt software, ovvero funzioni in grado di fermare il processo in esecuzione. La differenza sostanziale tra interrupt hardware e software è che i primi (causati da dispositivi hw esterni al processore) possono arrivare in maniera asincrona rispetto al clock della CPU. Tutte le syscall restituiscono un valore intero. Questo può anche essere rappresentato come `size_t` o altri tipi particolari.

Un processo quando è in modalità user può fare solo operazioni aritmetico-logiche e salti utilizzando unicamente la memoria assegnata.

Il sistema operativo non è un collegamento fra l'utente e l'elaboratore, infatti se noi colleghiamo una persona a un pc prende la scossa e si fulmina. Gli utenti degli elaboratori sono gli utilizzatori ultimi dei servizi, ma i veri utenti dei sistemi operativi sono i processi, non gli umani.

Cerchiamo di fare un catalogo di syscall in cui sono necessari:

- Accesso ai device
- Gestione dei processi
- Esecuzione dei programmi
- Gestione memoria
- Accesso ai file system e ai file
- Gestione degli utenti
- IPC (Inter Process Communication), ovvero la comunicazione fra processi e la loro sincronizzazione
- Debug e profiling
- Gestione tempo e gestione eventi.
- Networking

Gestione Utenti: se abbiamo un s.o. come UNIX che è molti utenti occorrerà una syscall per la gestione degli utenti. Il login iniziale fa uso di systemcall: il processo che chiede login e che chiede la password (sono infatti due programmi dello stesso processo) è in esecuzione come root, quindi può accedere a tutti i file del sistema. Una volta che ha scritto la password correttamente, il processo deve lanciare una shell che sia dell'utente, quindi in quel momento il processo deve cambiare proprietario e per farlo il processo deve interrogare il kernel e il kernel vede che un processo che ha permessi di root e quindi chiede "bene ora voglio essere un processo di non più di un root, ma di un dato utente" e se le credenziali sono giuste, il kernel gli cambia proprietario. In questo modo, il kernel controlla chi può fare cosa.

Networking: anche quando si usa ssh oppure si apre il browser fa uso di syscall, perché il processo non può avviare da solo una sessione TCP, bensì dovrà chiederla al kernel.

Gestione eventi: permette infatti per esempio di vedere se un utente sta scrivendo in una casella di testo o in un'altra. Infatti, se non ci fosse la gestione eventi il programma deve controllare la gestione di input in una delle due caselle. Con gli eventi, invece, ho una segnalazione di quando e dove (in quale delle due caselle) succede effettivamente qualcosa.

Gestione tempo: Anche solo richiedere a un sistema l'ora o aspettare 100 ms, lo deve chiedere al kernel. (Nei microcontrollori posso anche non interrogare il kernel siccome non ho processi, quindi mi basterebbe fare dei cicli e so esattamente quanto ci mette).

Debug: il comando `strace [programma]` mostra tutte le syscall di cui un processo fa uso, e per fare cio' usa anche un lui una syscall, e questo ci serve per debuggare appunto. Stessa cosa per il profiling (sinonimo di benchmarking), per avere tutte le informazioni sul consumo di memoria etc le ha attraverso syscalls.

Il networking e' l'unico campo in linux che viola la logica del file system, ovvero e' l'unico insieme di device che non viene gestito come se fosse un file. In questo modo si migliorano le performance.

Il prof fornisce lui stesso un catalogo delle syscall al sito [http://so.v2.cs.unibo.it/wiki/index.php?title=Il %27%27catalogo%27%27 delle System Call](http://so.v2.cs.unibo.it/wiki/index.php?title=Il_%27%27catalogo%27%27_delle_System_Call) . IMO e' abbastanza utile.

Due delle syscall piu' importanti sono `fork()` ed `exec()`.

La syscall **fork()** permette di creare un nuovo processo figlio del processo dal quale e' stato chiamato. Il figlio di questo processo sara' uguale al "padre" che l'ha creato, essenzialmente sara' una sua copia. Abbiamo visto nella lezione precedente che i processi sono solo dei "titolari di risorse", e che l'esecuzione in se' viene fatta in uno o piu' thread. Quindi in quest'ottica, l'idea di processo e di esecuzione sono separati. Non esistono altre syscall capaci di creare processi. La funzione ritorna 0 se il programma e' stato eseguito dal figlio, un numero positivo (id del processo creato) se invece il programma e' stato eseguito dal padre e -1 se ha fallito. Il processo figlio continuera' l'esecuzione del programma dal punto in cui avviene la `fork()`. Inoltre, le variabili di padre e figlio sono private e separate fra loro.

Esempio 1

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    if(fork())
        printf("vero %d\n", getpid());
    else
        printf("falso %d\n", getpid());
}
```

Il programma stampa

"vero 18218
falso 18219"

Sembra quasi infatti che vengano eseguiti entrambi i casi dell'if... ma cio' avviene perche appunto ho due processi diversi. Infatti, al processo padre da' l'identificazione del figlio, e al processo figlio ritorna 0. Quindi, vero e' il processo del padre, e falso e' il processo del figlio (anche se il suo pid e' piu piccolo).

Esempio 2

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]){
    pid_t pid;
    if((pid = fork()) != 0)
        printf("vero %d %d %d\n", pid, getpid(), getppid());
    else
        printf("falso %d %d %d\n", pid, getpid(), getppid());
}
```

Usiamo questa sintassi per evitare warning, senno' il compilatore penserebbe che volevamo fare un confronto ma che abbiamo sbagliato

Il programma stampa

"vero 18305 18304 21803"
falso 0 18305 18304"

21803 e' l'id del processo della shell corrente, che appunto e' quello che ha eseguito questo codice. Per stampare l'id del processo della shell corrente, basta usare il comando "`echo $$`", che in questo caso ritornerebbe 21803

`_exit(0)` e' la syscall che permette di uscire dal programma una volta che il codice e' stato eseguito, terminando cosi' il processo. La funzione `main()` inoltre non e' la very entry point alla funzione, bensì e' `start()`, che chiama `main()`, e quando `main()` termina chiama una `exit`.

Un processo genitore puo' lanciare un processo figlio ed aspettare che questo termini. Per fare cio', si possono usare piu' chiamate, ad esempio esiste la `waitkid()`, (queste cose si possono guardare il capitolo `man 2 wait`). Per attendere la terminazione di un processo figlio si utilizza la sys call `wait` (`waitpid()` aspetta un processo con un certo id). Il primo processo figlio che termina fa svegliare il

padre dalla wait. La waitpid permette di aspettare singoli processi, e l'argomento -1 fa aspettare tutti i processi.

La funzione `_exit([numero_di_ritorno])` permette anche di settare un valore di ritorno, che e' visibile nella shell tramite il comando `echo $?`. Negli script bash, il valore di ritorno e' un po' antintuitivo, perche' mentre normalmente la *return 0* indica i processi/programmi che sono stati conclusi correttamente, per bash tutti i valori uguali a 0 indicano vero, mentre tutti gli altri falso. Il valore di ritorno viene consegnato al processo padre.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char *argv[]){
    pid_t pid;
    if((pid = fork()) != 0)
    {
        int status;
        printf("genitore %d %d %d\n", pid, getpid(), getppid());
        waitpid(pid, &status, 0);
        printf("status %d %x\n", status, status);
        printf("exit status %d\n", WEXITSTATUS(status));
        _exit(0);
    } else
    {
        printf("figlio %d %d %d\n", pid, getpid(), getppid());
        sleep(1);
        _exit(44);
    }
}
```

Con l'aggiunta di questa riga,
il valore in hexadecimal
dell'exit status viene
convertito in decimale

Il codice ritorna, oltre alle prime due righe del processo precedente, anche la riga "status 11264 2c00".

Notare come in hexa 42 e' solo 2c, dunque perche' e' stato moltiplicato per 256? La risposta e' che status non mostra solo il valore di ritorno, ma contiene tante altre informazioni, per esempio ci dice se' finito correttamente oppure a causa di un errore etc

Tra le varie funzioni **exec()**, `execve([filename], argv[], vettore di stringhe ambiente[])` permette di eseguire un nuovo programma SENZA lanciare un nuovo processo, quindi essenzialmente lo stesso processo iniziale cambia programma.

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char* argv[])
{
    char *newargv[] = {"argv[0] di showpid", NULL};
    printf("io sono testexecve %d\n", getpid());
    execve("./showpid", newargv, NULL);
}
```

Nome del programma
compilato che vogliamo
eseguire

Il programma prima stampa "io sono testexecve 190708", ovvero l'id del processo, e poi stampa il contenuto del secondo programma, ovvero "argv[0] di showpid 190708".

Se volessi lanciare l'esecuzione di un nuovo programma con un nuovo processo, devo combinare sia fork() sia exec().

Esempio:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    pid_t pid;
    if((pid = fork()) != 0)
    {
        int status;
        printf("genitore %d %d %d\n", pid, getpid(), getppid());
        waitpid(pid, &status, 0);
        printf("status %d %x\n", status, status);
        printf("exit status %d\n", WEXITSTATUS(status));
        _exit(0);
    }
    else
    {
        char *newargv[] = {"argv[0] di showpid", NULL};
        printf("io sono procfiglio %d\n", getpid());
        execve("./showpid", newargv, NULL);
        _exit(-1);
    }
```

Questo `_exit(-1)` in teoria non viene eseguito se tutto e' corretto.

Il processo figlio svolge l'`execve`, siccome e' nel ramo `else`.

Abbiamo tantissime versioni diverse di funzioni `exec()`, e a seconda del suffisso si hanno delle funzioni diverse.

Le `syscall`, come abbiamo visto prima, permettono anche un certo grado di gestione degli errori. Le `syscall` ritornano sempre un intero, normalmente e' 0, ma possono anche ritornare un file descriptor (ovvero un intero usato per identificare un file aperto da un processo) rappresentato con un intero ≥ 0 . Se invece ritornano -1, o un numero negativo in generale, abbiamo avuto un errore. Ma rimane comunque un problema: come facciamo a capire di che errore si tratta? Per questo, esiste una pseudo variabile globale detta **errno**, che sta per error number, che con un codice mi dice qual'e' l'errore riscontrato. Perche' non e' una variabile veramente globale? Perche' i moderni sistemi consentono di fare appunto multi-threading, quindi appunto se piu' thread accedessero contemporaneamente a questa variabile ci sarebbero problemi, percio' e' globale unicamente a livello di thread. Per vedere il a cosa corrisponde un dato codice d'errore, basta andare a vedere il file `/usr/include/asm-generic/errno-base.h`. `strerror([int errno])` e `perror([char* prefisso])` permettono di associare al numero di errore a cosa corrisponde, queste sono semplici funzioni di libreria e non `syscall`.

Abbiamo detto che il processo figlio manda il valore di ritorno al processo padre. Ci saranno quindi due casi da esaminare:

1. Il processo padre potrebbe disinteressarsi del processo figlio, ovvero lancia il processo figlio e poi non fa la `wait` per aspettare il valore di ritorno.
2. Il processo padre lancia il processo figlio e termina prima il processo padre, e quindi a chi verra' mandato il valore di ritorno?

I processi padre, come in natura, sono responsabili dei figli, quindi se hanno generato un processo figlio DEVONO aspettare che il processo figlio ritorni il segnale di ritorno. Cosa succede se uno non lo fa? Il processo figlio pero' non puo' liberare tutte le risorse, perche' deve tenere il valore di ritorno da dare al processo padre.

Esaminiamo il caso 1: il processo figlio viene etichettato con stato Z, ovvero con stato zombie, cioè il processo è nato, ma non potrà completare il percorso (ovvero non morirà) fin quando il genitore non farà la wait per ricevere il valore di ritorno. Questo succede spesso nei server web (tipo APACHE) in cui essenzialmente viene fatto un `fork()` ogni volta che si vuole risolvere una richiesta

Quindi, un **processo zombie** è un processo che sta cercando di terminare, ma che non può siccome il genitore non ha ancora letto/ricevuto lo stato di ritorno. In questo caso infatti, il kernel potrà liberare le risorse del processo, ma appunto non potrà rimuovere la sua voce dalla tabella dei processi, siccome lì è contenuto lo stato di ritorno del processo, che deve poter essere letto dal processo genitore tramite `wait()`.

N.B.: per cercare un processo fra i processi in esecuzione, basta usare il comando `ps ax | grep [nome_processo]`.

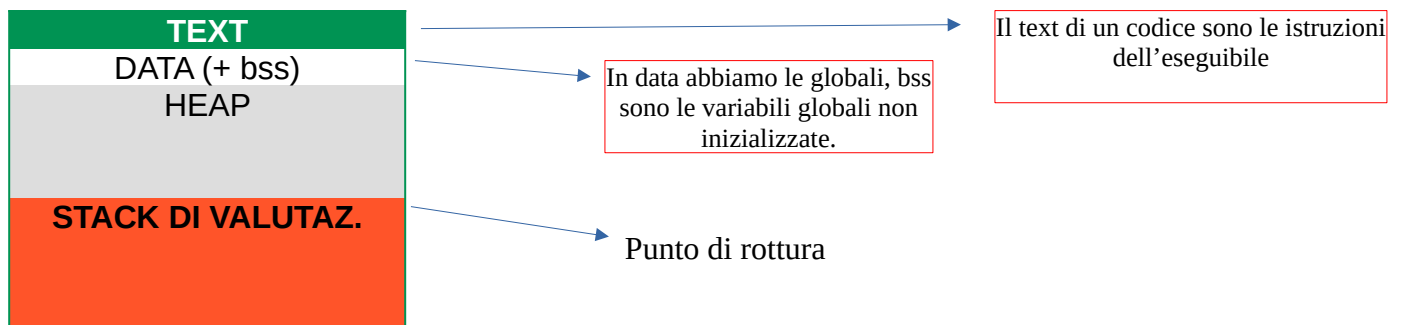
Passiamo invece al caso 2: normalmente, un processo orfano manda l'exit status al processo 1, ovvero init, che fa la wait per tutti i processi che terminano e che sono orfani, li adotta praticamente.

Questo è anche il modo in cui i **processi demoni** (daemon process) vengono creati: quando lanciamo un programma che deve rimanere attivo virtualmente per sempre perché è un demone che deve fare dei servizi, non deve essere un processo che deve essere figlio della shell (perché se non quando termino la shell termina pure lui), bensì deve venire lanciato attraverso il processo di demonizzazione che consiste nel creare un processo figlio che poi lancerà il processo che vogliamo demonizzare, che quindi diventerà un processo nipote. In questo modo, possiamo staccare il processo demone dal padre e poi cessare il processo padre del demone direttamente, e il processo demone vivrà la sua vita staccato dalla shell, ed attaccato solamente ad init.

Prima abbiamo accennato l'**environment**, ma che cos'è effettivamente? L'environment è un vettore di stringhe, e ciascuna stringa è composta da un identificatore e un valore (nel formato [IDENTIFICATORE]=[VALORE]) e possono essere usate dai programmi per avere particolari informazioni. In `execve`, tutte le stringhe dell'env del programma chiamante vengono passate al chiamato se si mette il parametro a null, altrimenti solo quelle che specifichiamo.

Per creare nuove variabili env in una shell, basta definirle col formato precedente definito sopra, senza chiamare nessun comando. Se vogliamo fare in modo che tale variabile venga copiata anche nelle shell lanciate dopo, allora bisogna precedere la definizione della variabile col comando `export`. Inoltre, per mostrare una variabile env specifica basta usare il comando `echo ${nome_var}`. (DA NOTARE: Le variabili dell'env non sono globali, semplicemente vengono copiate sempre tra una shell e una nuova che è stata lanciata).

Le syscall inoltre permettono quindi la gestione della memoria. In realtà, esiste solo una syscall che permette di fare ciò, ed è `brk()` e solitamente non viene usata dagli utenti.



Text e Data, nel momento in cui un programma viene caricato in memoria, vengono copiati dall'eseguibile.

La quantità di memoria a disposizione del processo dipende dal punto di rottura che ci dice dove finisce lo heap, quindi lo heap serve per poter avere delle variabili allocate dinamicamente. Tutta l'allocazione dinamica però non è fatta dal kernel, ma da un allocatore dinamico che è in libreria (es.

malloc, calloc e free in C). Se l'heap ha bisogno di allocare piu' dati, allora semplicemente viene chiesto al kernel di "dare piu' spazio" all'heap che abbassera' il punto di rottura.

brk(), che non si usa mai indirettamente, fa esattamente questo.

Lezione del (28/10/2020) – Dekker, Peterson, TS e semafori

Perche' si studia programmazione concorrente nei corsi di SO? Perche' l'astrazione di concorrenza, anche di grado superiore al numero di processori, viene data dai S.O.; in piu' i sistemi operativi che permettono cio' devono dare anche il supporto utente per permettere di farne effettivamente uso.

Esistono 3 gradi di concorrenza:

- Processi che sono ignari l'uno dell'altro ma che appartengono allo stesso sistema. Allora bene o male, anche se non sono stati scritti con logica concorrente, esiste il sistema operativo che li deve far convivere con gli altri, e quindi in realta' c'e' concorrenza anche in quel caso. Anzi, il termine concorrenza qui calza perfettamente, siccome i processi di questo tipo competono nell'accesso delle risorse.
- Programmi indirettamente correlati: per esempio, anche se sono sequenziali iterativi, hanno delle risorse comuni. Ad esempio, il comando che usavamo per vedere a che gruppo noi apparteniamo avevano una pipe, quindi l'output di quel programma diventa input di un altro, e quindi anche se sono sequenziali iterativi hanno una forma di concorrenza.
- Il grado massimo di concorrenza, processi che creano thread in base alle necessita': un esempio puo' essere il browser, che ogni volta che apre una finestra o un tab, crea un altro thread che gestisce quello.

Nella lezione della settimana scorsa abbiamo l'algoritmo di Dekker in pseudocodice, in questa lezione invece lo vedremo in azione in un programma compilato vero:

Dekker1.c :

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
#include<stdlib.h>
```

Funzione che aspetta una data
quantita' di tempo in
millisecondi.

```
#define CRITICAL_CODE(X, MIN) usleep(random() % X + MIN)
#define NON_CRITICAL_CODE(X, MIN) usleep(random() % X + MIN)
#define DELAY(X) usleep(random() % X)
```

```
#define MAX 10
#define P 0
#define Q 1
```

```
volatile int turn;
```

Volatile dice al compilatore che il valore potrebbe
cambiare in qualsiasi momento. E' una keyword
usata di solito proprio per le variabili globali che
possono subire accessi concorrenti.

```
void *p(void *arg)
{
```

```
    int i;
    for (i = 0; i < MAX; i++)
    {
        NON_CRITICAL_CODE(1000, 0);
        /*mutex in*/
        while (turn == Q)
            ;
        printf("P in\n");
        CRITICAL_CODE(1000, 0);
        printf("P out\n");
        /*mutex out*/
        turn = Q;
    }
```

Se e' il turno di Q, non
fare nulla.


```

}

void *q(void *arg)
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        NON_CRITICAL_CODE(1000, 0);
        /*mutex in*/
        while (turn == P)
            ;
        printf("Q in\n");
        CRITICAL_CODE(1000, 0);
        printf("Q out\n");
        /*mutex out*/
        turn = P;
    }
}

int main()
{
    srand(time(NULL));
    pthread_t pp;
    pthread_t pq;
    pthread_create(&pp, NULL, p, NULL);
    pthread_create(&pq, NULL, q, NULL);
    pthread_join(pp, NULL);
    pthread_join(pq, NULL);
}

```

Dekker2.c :

/*define e include invariati da dekker1.c*/

```

volatile int inp;
volatile int inq;

void *p(void *arg)
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        NON_CRITICAL_CODE(1000, 0);
        /*mutex in*/
        while (inq)
            ;
        inp = 1;
        printf("P in\n");
        CRITICAL_CODE(200000, 0);
        printf("P out\n");
        /*mutex out*/
        inp = 0;
    }
}

void *q(void *arg)
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        NON_CRITICAL_CODE(1000, 0);
        /*mutex in*/
        while (inp)

```

Il programma stampa un successione di

"P in
P out
Q in
Q out"

E il codice di ogni processo P e Q, per come l'abbiamo definito noi, ha una durata randomica.

Questa esecuzione quindi sembra coerente, cio' non vuol dire che funzioni, ma non e' vero che non funzioni. Il problema di questo codice e' appunto che se P ci mettesse almeno 1 secondo un secondo per completare l'esecuzione, Q sarebbe OBBLIGATO ad aspettare P, perche' appunto devono andare a turni.

Anche qui, il codice sembra funzionare, e senza ritardi causati da P o da Q.

Se pero' ho un delay dentro la sezione critica (e non fuori come facevamo prima) la mutua esclusione viene ROTTA, e questo e' un bel problema perche' entrambi accedono alla critical section.

```

        ;
        inq = 1;
        printf("Q in\n");
        CRITICAL_CODE(200000, 0);
        printf("Q out\n");
        /*mutex out*/
        inq = 0;
    }
}

int main()
{
    /*invariato da dekker1.c*/
}

```

Dekker3.c

/*define, include, main invariati da dekker1.c*/

```

volatile int needq;
volatile int needp;

```

Indicano quali processi hanno bisogno della sezione critica

```

void *p(void *arg)
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        NON_CRITICAL_CODE(1000, 0);
        /*mutex in*/
        needp = 1;
        //DELAY(10000);
        while (needq)
            ;
        printf("P in\n");
        CRITICAL_CODE(200000, 0);
        printf("P out\n");
        /*mutex out*/
        needp = 0;
    }
}

```

Anche qui, il codice sembra funzionare,, risolvendo parte del problema iniziale. Tuttavia, se si aggiungono i due delay (che ho commentato) prima dei while in p e q, noteremo che i nostri processi andranno in DEADLOCK, infatti non proseguiranno e semplicemente a un certo rimarranno entrambi bloccati.

```

void *q(void *arg)
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        NON_CRITICAL_CODE(1000, 0);
        /*mutex in*/
        needq = 1;
        //DELAY(10000);
        while (needp)
            ;
        printf("Q in\n");
        CRITICAL_CODE(200000, 0);
        printf("Q out\n");
        /*mutex out*/
        needq = 0;
    }
}

```

Se aggiungiamo questo delay, ci sara' deadlock.

Dekker4.c

/*define, include, main e variabili globali invariati da dekker1.c*/

```
void *p(void *arg)
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        NON_CRITICAL_CODE(1000, 0);
        /*mutex in*/
        needp = 1;
        while (needq)
        {
            needp = 0;
            DELAY(100000);
            needp = 1;
        }
        printf("P in\n");
        CRITICAL_CODE(200000, 0);
        printf("P out\n");
        /*mutex out*/
        needp = 0;
    }
}
```


Per evitare la deadlock, appena scopro che l'altro ha bisogno della critical section, rinuncio e cedo il passo.

In questo codice non avviene deadlock, ma c'è un altro problema comunque molto grande: se i due processi continuano all'unisono, infatti, abbiamo STARVATION, ovvero in pratica se abbiamo un delay altissimo in P (di un milione di secondi tipo) allora Q non andrà mai avanti.

```
void *q(void *arg)
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        NON_CRITICAL_CODE(1000, 0);
        /*mutex in*/
        needq = 1;
        while (needp)
        {
            needq = 0;
            DELAY(100000);
            needq = 1;
        }
        printf("Q in\n");
        CRITICAL_CODE(200000, 0);
        printf("Q out\n");
        /*mutex out*/
        needq = 0;
    }
}
```

DekkerFinale.c

/*variabili globali, include e define UGUALI a dekker4.c*/

volatile int last;  Indica qual e' l'ultimo processo ad aver lasciato la sezione critica.

```
void *p(void *arg)
{
```

```
    int i;
    for (i = 0; i < MAX; i++) {
        //usleep(random() % 1000);
        needp=1;
        while(needq) {
            needp=0;
            while (last == Q)
                usleep(random() % 1000);
            needp=1;
        }
        printf("P IN\n");
        usleep(random() % 200000);
        printf("P OUT\n");
        needp=0;
        last=Q;
    }
}
```

Aggiunto un nuovo while che aspetta nel caso il turno non sia il suo...

Continua a ciclare se l'ultimo e' stato P.

```
void *q(void *arg) {
    int i;
    for (i = 0; i < MAX; i++)
    {
        usleep(random() % 1000);
        needq=1;
        while(needp)
        {
            needq=0;
            while (last == P)
                usleep(random() % 1000);
            needq=1;
        }
        printf("Q IN\n");
        usleep(random() % 200000);
        printf("Q OUT\n");
        needq=0;
        last=P;
    }
}
```

L'algoritmo di Dekker finale elimina la contesa specificando di chi e' il turno. Infatti, nella versione precedente, se l'altro processo ne aveva bisogno aspettava un periodo di tempo arbitrario, e quindi non era certo di quando effettivamente il programma potesse procedere... ci potevano essere programmi "fortunati" che facevano l'interrogazione al momento giusto. Se invece qui succede il caso che entrambi vogliono accedere alla sezione critica, se la giocano a turni, e quindi siamo certi che prima o poi entrambi andranno avanti... Se un processo e' in uscita dalla sezione critica, fa due cose: lascia il need e dichiara che il turno e' dell'altro, in modo che se tutte e due vogliono accedere alla sezione critica va avanti quello che non e' stato l'ultimo a lasciarla.

Quindi: Se io ho bisogno della sezione critica e vedo che anche l'altro processo ha bisogno della sezione critica, allora dico di non averne piu' bisogno e aspetto che l'ultimo ad entrare sia stato quell'altro, e allora vado avanti io.

Nonostante il codice funzioni, al tempo i ricercatori si fecero due domande fondamentali:

- Posso eliminare il doppio ciclo?
- Posso estendere questo algoritmo a piu' di due processi?

Dunque nasce l'**algoritmo di Perterson**, che risolve esattamente questi problemi.

Peterson_theory.c

/*variabili, define e include uguali a dekkerfinale.c, inoltre ho messo solo uno dei processi perche' bisogna salvare gli alberi. Il processo Q e' simmetrico a P.*/

```
void *p(void *arg)
{
    int i;
    for (;;) → Uguale a while(true)
    {
        //NC
        needp = 1;
        last = P;
        while(needq && last == P)
        ;
        //CRIT
        needp = 0;
    }
}
```

→ P entra se Q non ha bisogno e se P e' il primo a richiederne l'accesso...

Notiamo subito che il codice e' molto piu' conciso, pero' e' molto piu' tricky.

La cosa ingegnosa e' mettere last in cima, e NON alla fine come in Dekker, e tutte le volte che dico che ne ho bisogno, sono anche l'ultimo ad averne bisogno. In questo modo, se tutti e due i processi ne hanno bisogno, il last viene conteso, e l'ultimo che arriva a scrivere last e' quello che aspetta, perche' needq rimane acceso, ma last sono io.

L'altro (q) consentira' al primo di accedere quando esce dalla sezione critica e mettera' needq = 0, e falsifica cosi' il primo fattore di quell'and e quindi p entra.

(**METANOTA**: il prof poi ha mostrato una versione multiprocesso dell'algoritmo di Peterson). Il meccanismo che usa Peterson e' quello di creare una sorta di torneo fra processi, che combattono a due a due e devono vincere la competizione n volte. La variabile stage infatti indica lo stadio di avanzamento, e per vincere ciascuno stadio un processo non deve essere l'ultimo a settare last.

Esaminando questi due algoritmi abbiamo due risultati:

1. Lavorare con la sezione critica si puo' fare...
2. ... ma non e' la maniera giusta per farlo. Infatti, questo metodo non e' esattamente user-friendly.

Abbiamo bisogno di avere nelle mani costrutti piu' semplici con cui lavorare. Questi costrutti dovranno avere varie proprieta': devono essere comodi da usare, devono consentire di risolvere il problema e infine devono essere implementabili.

Nel fare cio', l'hardware ci puo' dare una mano per creare queste sezioni critiche? Potremmo usare le **maschere di interrupt**: ogni processore ha una maschera di bit, e se il bit e' mascherato, vuol dire che quell'interrupt, anche se viene generato, in quel momento non viene recepito. Conseguentemente, cio' che accade e' che la sequenza di programmi in esecuzione procedera' nell'ordine seguito fino a quel momento. In questo modo, si potrebbero implementare i semafori a livello kernel, in modo che i processi utente possano chiedere al kernel un accesso alla sezione critica. Questa potrebbe essere una soluzione, ma ha due problemi: 1) funziona solo su processi monoprocessori, perche' se uso un sistema multiprocessore ogni processore fa uso di un diverso mascheramento di interrupt. 2) questo strumento non lo possiamo lasciare nelle mani di un processo utente, perche' se questo potesse disabilitare gli interrupt, se poi non li riattiva l'elaboratore si blocca.

Per i sistemi monoprocessore abbiamo una maniera per fare in modo che le cose pericolose possano essere fatte dai processi utente senza creare danni al sistema. Inoltre, sarebbe anche pericoloso che un processo possa andare a scrivere su disco dove vuole... e per impedire che il processo faccia questo possiamo usare le syscall. Quindi, per risolvere questo problema facciamo in modo che il mascheramento non venga fatto direttamente dal processo utente, bensì attraverso una syscall.

Tuttavia, il primo problema non e' comunque risolto.

Dunque, col tempo, ci si e' chiesti se e' possibile aggiungere delle istruzioni nell'Instruction Set dei processi che ci permettano di creare delle sezioni critiche (ovviamente questa istruzione dovra' essere atomica). L'idea e' stata quella di creare delle istruzioni che risolvono il problema, quella che viene progettata normalmente e' detta **Test and Set**.

La funzione *test and set* e' fatta in questo modo:

TS(x, y) < x = y; y = 1 >

→ Il passaggio da x a y viene fatto per riferimento

(Nota sulla sintassi: con le parentesi angolose indichiamo che le istruzioni che ci sono dentro sono atomiche). La TS è stata inserita anche nei sistemi RISC in cui ogni istruzione viene eseguita in un singolo ciclo di clock, anche se è stato complicato siccome la TS è composta da due istruzioni e c'è stato bisogno di usare un LL/SC. (Lo swap atomico è un'altra tipologia di TS).

Questa funzione può essere usata quasi sempre come mattoncino per la creazione di sezioni critiche, e viene usata in questo modo:

```
global busy = 0;
```

Indica se la critical section è libera

```
Pi: process i = 1...N
    local localcopy;
    while (true)
        //non critical code
        //mutex in
        do
        {
            TS(localcopy, busy);
        } while (localcopy == 1)
        //critical code
        mutex_out();
```

La ts assegna alla variabile locale quella della variabile globale

Senza la TS, il problema più grande era che potevano esserci mille processi che chiedevano se la critical section fosse libera, e quindi tutti accedono contemporaneamente.

La TS dice subito che la zona è occupata ($y = 1$), ma prima di fare ciò copia il valore della variabile globale nella variabile locale. In questo modo, solo il primo che farà la TS in maniera atomica e che avrà la critical section libera avrà nella locale 0.

Questa è quasi una critical section... questo perché, nonostante le proprietà di no deadlock, mutua esclusione e no attese non necessarie siano rispettate, la starvation è possibile (anche se statisticamente difficile).

Esaminiamo il caso in cui questo può accadere: poniamo che ci siano due processi $p_{fortunato}$ e $p_{sfortunato}$. Il $p_{fortunato}$ arriva, prende la critical section ed entra, quello $p_{sfortunato}$ arriva e si mette a ciclare perché la critical section è occupata. Il primo processo, che è velocissimo, esce dalla critical section e torna al punto iniziale prima che l'altro faccia qualsiasi cosa, riprendendosi nuovamente la TS prima che il $p_{sfortunato}$ possa entrarci. Questa cosa succede ancora tantissime volte, e questo perché $p_{sfortunato}$ controlla sempre la TS al momento sbagliato.

Detto ciò, nonostante non risolva effettivamente la critical section, è usato tantissimo nei sistemi operativi reali. Esistono infatti modi per mitigare l'effetto di questo difetto, infatti basta fare in modo che le attese della critical section siano brevi (tra l'altro sarebbe anche inefficiente utilizzare la TS per le attese lunghe). Si parla infatti di **busy wait**, perché in questo tempo il processore (o i core) è al 100% del suo utilizzo senza fare effettivamente nulla di utile.

Questo meccanismo di andare a controllare continuamente quando la critical section è bloccata è detto **spinlock**, in cui il thread rimane attivo (perché sta controllando se la sezione è disponibile) ma senza fare niente di veramente utile.

La TS è stata inserita anche nei sistemi RISC in cui ogni istruzione viene eseguita in un singolo ciclo di clock, anche se è stato complicato siccome la TS è composta da due istruzioni e c'è stato bisogno di usare un LL/SC.

--(il prof dopo questo ha spiegato la risoluzione di un esercizio che reputo utile ma che non sono riuscito a trasporre in versione scritta, a parte per queste righe:).

Per risolvere alcuni problemi, si può capire come effettivamente funzionano vedendo i casi in cui è assolutamente sbagliato. Per questo tipo di problema abbiamo una struttura globale che deve rappresentare l'idea di critical section occupata o libera, e c'è bisogno di una struttura locale che ci può far capire se prima la critical section era libera o occupata. E occorre che per effetto della nostra funzione la variabile globale raggiunga sempre lo stato di occupato. Se nel protocollo d'ingresso (dopo la `mutex_in`) si accede alla variabile globale per più di una volta è sicuramente sbagliato.

--(fine sezione esercizi)

Risolvere il problema del busy wait è molto complesso, e in più bisogna contare che molte volte avere una critical section non è la risposta. Per ragionare sulla programmazione concorrente, proviamo a lavorare con problemi campione:

Il problema più semplice della programmazione concorrente è il problema del produttore-consumatore. Abbiamo due processi: uno detto produttore che produce dati e uno detto consumatore che li consuma in modo coordinato, senza consumarne più del dovuto e senza consumare sempre lo stesso dato (il buffer è in sola lettura da questo processo). Per comunicare, questi due processi usano un buffer condiviso che contiene l'elemento: il produttore mette nel buffer un dato e il consumatore lo consuma. Se si cerca di mettere un dato nel buffer mentre un questo è pieno, allora il dato viene perso. Viceversa, il consumatore se è troppo veloce a consumare può leggere uno spazio vuoto o consumare più volte lo stesso oggetto. Il programma viene scritto in questo modo:

```
producer : process
    while (true)
        x = produce();
        buf = x;

consumer : process
    while (true)
        x = buf;
        consume(x);
```

Questo codice spiega il problema, ma senza risolverlo effettivamente. Infatti, tutti i problemi descritti prima si possono verificare tranquillamente. Inoltre, la mutua esclusione non basta per risolvere questo problema. Abbiamo bisogno di costrutti e sintassi più complete.

Queste strutture e sintassi più complete dovranno risolvere questo problema e quello della sezione critica.

Per risolvere questi problemi allora creiamo due primitive P (iniziale della parola olandese *passeren*, ovvero passa/attendi) e V (iniziale della parola olandese *verhogen*, ovvero incrementa) che saranno presenti in una classe chiamata **semaphore** (semaforo).

In questa classe semaforo avremo un costruttore in cui vengono passati i parametri iniziali, e due funzioni P e V che non ritornano nulla e non prendono alcun parametro. Però c'è una regola: se nP è il numero delle operazioni P completate io l'ho invariante:

$$nP \leq nV + \text{init}$$

Init è il parametro passato al costruttore

Se la P tende a portare il valore del semaforo a un valore negativo, ci si blocca, perché l'invariante così non sarebbe confermata.

Ecco un esempio:

```
semaphore cs(1)

process pi i = 1...N
    while(true)
        //NC
        cs.P()
        //C
        cs.V()
```

Con la P inizializzata in questo modo, non posso chiamare la P più di una volta senza chiamare la V.

In questo modo, tutti i processi che vorranno eseguire la P, potranno chiamarla ma non potranno completarla, perché senno' il semaforo non sarebbe più valido. Il primo che farà la V consentirà ad uno degli altri processi che ha chiesto la P ma che non l'ha ancora ottenuta di ottenerla.

Quindi usiamo il semaforo come se fosse una critical section, e questo è anche più facile da usare di una TS. Il semaforo però ci dà anche qualcosa in più: il numero di init può essere visto come il numero di risorse a disposizione, e ogni volta che facciamo la P prendiamo una risorsa, e tutte le volte che facciamo la V ne liberiamo un'altra.

Inoltre, se avessimo messo il valore 0 come parametro iniziale a posto di 1, possiamo usarlo come strumento di sincronizzazione, perché se è 0, se uno fa la P aspetta, e si sbloccherà solo quando qualcun'altro gli farà la V.

Proviamo ora a scrivere il problema del produttore-consumatore usando i semafori:

```
semaphore ok2write(1)
semaphore ok2read(0)

producer: process
    while(true)
        x= produce() // non ci interessa come è fatta
        ok2write.P()
        buf = x
        ok2read.V() //abilita ad andare avanti il consumatore

consumer: process
    while (true){
        ok2read.P() //se il consumatore arriva prima si deve fermare
        x = buf
        ok2write.V()
        consume(x)
```

Consideriamo ora un caso anomalo, in cui il produttore e' molto veloce e il consumatore e' lento: il produttore arriva, genera un elemento con `ok2write.P()`, scrive l'elemento e fa un `ok2read.V()` in modo da generare un elemento in `ok2read`, poi torna a capo, cerca di fare `ok2write.P()`, ma non riesce siccome il semaforo e' a 0. Dunque, ricomincerà solo quando il consumatore avrà eseguito `ok2write.V()`.

Il semaforo e' utilissimo per scrivere programmi concorrenti, tuttavia per ora non abbiamo effettivamente implementato tale struttura ma abbiamo una rappresentazione schematica, e nella lezione dopo si vedrà come farlo.

Alcune considerazioni fatte a fine lezione:

Oltre al deadlock, esiste anche il **livelock**, che si ha quando piu' processi non avanzano anche se pero' fanno qualcosa (la CPU quindi non è allo 0%, bensì solitamente al 100%), contrariamente alla deadlock in cui abbiamo i processori fermi e bloccati e quindi non abbiamo carico di CPU.

Quando abbiamo parlato di liveness, abbiamo parlato di starvation (che appunto normalmente e' un problema di liveness) a volte viene classificato come un problema di safety, perche' infondo bloccarsi a vicenda e' qualcosa di male.

La speranza massima in ambito della concorrenza sarebbe di far gestire la concorrenza da compilatori che fanno tutto da soli. Ma non potrà mai essere così, e questo esempio ci dice perche':

$y = w = 0$

```
p: process
    while (true)
        y *= 3
        w *= 3

q: process
    while(true)
        y += 2
        w += 2
```

Il programmatore potrebbe voler garantire l'invariante $v == w$.
Il compilatore dovrebbe decidere qual è la compilazione corretta e qual è quella sbagliata di questo codice, ma possono esserci punti di vista differenti che non prescindono dall'invariante. Il compilatore non può scegliere da solo, bisogna vedere qual è il significato della concorrenza in un determinato caso.
Bisogna pensare in modo concorrente.

Lezione del (30/10/2020) – File System di Linux

Introduzione:

Abbiamo visto nella lezione della settimana scorsa che possiamo creare attraverso una syscall un processo figlio che e' ad immagine e somiglianza del processo padre tramite la syscall `fork()`. Inoltre abbiamo anche visto che in UNIX il concetto di creare un processo ed eseguire un programma sono due concetti separati. Infatti, posso creare un processo senza eseguire un programma ed eseguire un programma senza creare un processo. L'operazione di creare un processo corrisponde allo scoppio logico del processo, inoltre nel processo figlio le variabili (dell'environment) del processo padre vengono copiate, e se cambiamo un valore di variabile nel processo padre cambia anche nel processo figlio (questo se usiamo il comando `export`).

Invece l'esecuzione di un programma non corrisponde alla creazione di un processo, infatti quando usiamo l'`execve()` eseguiamo un altro codice, ma il processo rimane sempre quello.

`_exit()` e' la syscall che ci permette di uscire da un programma, e viene invocata sempre, anche quando non lo sappiamo. Infatti il main viene invocato dalla funzione `start()`, e una volta che si esce dal main si invoca la syscall `_exit()`.

File system:

Andiamo a vedere le syscall per la gestione del file system. Il file system e' quella astrazione data dal sistema operativo di un archivio con cartelle, documenti e pratiche che consente di avere una visione comoda per accedere alla memoria di massa, sottoforma di gerarchia. Un tipo di filesystem e', per esempio, ext2.

Molti dei comandi della shell di linux hanno delle syscall corrispondenti che fanno esattamente la stessa cosa del comando. Alcuni di questi sono `chdir()`, `mkdir()` etc. Ogni processo ha la sua working directory. Se vogliamo sapere in che directory siamo, basta usare la syscall `getcwd()`. Nonostante cio', esiste anche la funzione di libreria `getcwd()`, che ritorna un char al posto di un int. Infatti, un buon modo per riconoscere una syscall da una funzione di libreria e' vedere se ritorna un intero (che corrisponde all'errore). `chdir()` e' un po' anacronistico, siccome la working directory rappresenta uno stato del processo. Dunque, se il programma e' multithread, potrebbero danneggiarsi a vicenda piu' thread che vogliono accedere a due file diversi. In questa lezione vedremo come gestire queste eccezioni.

La syscall `mkdir()` prende 2 parametri: il path della cartella che si vuole creare e i permessi che quel file deve avere (sottoforma di numero, come abbiamo visto nella lezione del 16/10/2020, con uno 0 davanti per indicare che vogliamo passare la sua traduzione in binario).

Se proviamo a mettere come permesso 0777 (che e' una codifica dei permessi valida indicante che il file ha tutti i permessi per tutti), noteremo che il nostro programma lanciato dalla shell non riuscirà a creare una cartella con questo valore di permessi. Questo perche' la shell da noi avviata ha una `umask` (che una proprieta' del processo) che e' 022; siccome magari sto usando un programma che non ho scritto io e che e' stato creato per dare diritti d'accesso molto ampi, io posso settare l'`umask` per dire "quanto sono cattivo in questo momento", ovvero quali permessi effettivamente posso dare a un certo file. L'`umask` a 022, la regola e' che quando si apre un file o si cambiano i permessi a un file, quei bit che sono accesi nella `umask` vengono spenti quando si crea un file. L'`umask` a 022 e' inoltre la protezione standard, che dice che permette l'accesso in lettura per tutti gli utenti, ma non la rx per i membri del gruppo e tutti gli altri.

Se per caso io voglio cambiare l'`umask`, posso usare il comando `umask` o la omonima syscall.

In UNIX, i nome e le proprieta' dei file sono entita' differenti. Fra le proprieta' dei file infatti non c'e' il nome, e la directory altro non e' che un elenco di nomi che indicano file, come se fosse un elenco di puntatori (in realta' sarebbe un elenco di indici). Se andiamo a vedere come e' memorizzata una directory, noto che ho una directory "." (che e' l'elemento 684751 nel pc del prof), un'altra ".." (elemento 678525), file "a.out" (el. 684733) etc... quindi nella cartella ho essenzialmente un insieme di coppie composte da *nome* e *numero di file*. Data questa astrazione, si puo' pensare che in directory

diverse esistano nomi diversi che indicano lo stesso file (basta fare in modo che abbiamo lo stesso numero di file). Questa proprieta' si chiama **hard link** (o link fisico). In questo non ci ritroviamo due file, bensi' due nomi diversi per lo stesso file. Per creare un hard link si usa, come abbiamo visto, il comando `ln` che corrisponde alla syscall `link()`. Col comando `ls -la` possiamo vedere l'elenco dei nomi dei file con associato il loro numero.

Per cancellare un file, non esiste la syscall `remove` o `delete`, bensì devo usare la syscall `unlink()`, che toglie il nome di un file e possibilmente anche il file stesso a cui si riferisce (questo succede in modo automatico solamente se non ho più nomi che si riferiscono a quel file e se non è in uso da un processo al momento dell'esecuzione del comando).

Oltre ai link fisici, o anche i **link simbolici**. Un link simbolico è semplicemente un rimando: se io apro il link, viene visualizzato direttamente il file a cui punta. Se io cancello il file di un link simbolico, semplicemente quel link non punterà più a niente, e se accederò a quel link, anche se il link in sé esiste, avrò un errore di file non trovato. Per creare un link simbolico, posso usare il comando `ln -s` oppure la syscall `symlink()`. Il link simbolico permette anche di fare rimandi a file inesistenti (anche se sarà un rimando che non porterà a nulla), ma la stessa cosa non posso farla con un link fisico. Inoltre, posso fare un link simbolico anche fra file system differenti. La syscall `readlink()` permettere di vedere quali siano i link simbolici di un file.

Esiste la syscall `rename()`, che cambia il nome di un file. La stessa cosa però si poteva fare con `link()`, creando un nuovo hard link al file iniziale e dandogli un nome diverso, e poi con `unlink()`, cancellando il vecchio nome del file. I motivi per cui la syscall `rename` esiste nonostante ci sia già la syscall `link` sono:

- Atomicità: vogliamo che in ogni momento o ci sia il vecchio nome o il nuovo, non entrambi. La `rename` fa il suo compito in un colpo solo.
- Problemi di accesso ai file system: il file system può essere realizzato in maniera molto diversa... per esempio, una chiavetta USB nuova non formattata ha solitamente un filesystem di tipo FAT, che non è capace di fare i link fisici! Ma la `rename` risolve questo problema.

Se facciamo il comando `stat`, possiamo vedere moltissime informazioni sul file, tra cui numero del file (detto **inode** number, che sta per "information node", ovvero l'indice del nodo che contiene le informazioni del file come dimensioni, data d'accesso, data di modifica etc...) e il numero di link al file. Se io queste informazioni le voglio ottenere tramite un programma, uso la syscall `stat()`, che ritorna uno struct contenente tutte queste info.

Molte syscall sono disponibili nella forma `lstat()` e `fstat()`. La differenza sta che la prima vede i link simbolici come pacchi, quindi essenzialmente considera il link simbolico come un file, senza buttarsi direttamente sul file a cui il link punta.

Esempio: avendo un ipotetico file "due" contenente la stringa "ciao\n" (quindi lungo 5 byte), e un altro file "tre" che è symbolic link di "due", la lunghezza fornita dal comando `lstat` è di 3 byte siccome il contenuto del file è la stringa "due", ovvero il nome del file a cui punta.

La seconda invece serve per i file aperti in quel momento. Se abbiamo i file aperti in quel momento, infatti, abbiamo un file descriptor in memoria (ogni file quando viene aperto ritorna un file descriptor), quindi possiamo direttamente vedere le stat del file attraverso il file descriptor e senza andare a vedere il path del file. `fstat()` infatti prende un int che indica il file descriptor.

`chown()` e `chmod()`, come i comandi corrispondenti, servono per cambiare l'ownership di un file e i permessi di accesso rispettivamente.

`truncate()`, tronca un file, nel senso che se mettiamo come parametro `length` a 0, abbiamo lo stesso file di prima ma senza contenuto, oppure se mettiamo un numero `n` avremo lo stesso file ma con solo i primi `n` byte. Questo comando però può essere anche usato per estendere i file, aumentandone così le dimensioni e aggiungendo byte settati a 0. Tuttavia, la `truncate` non alloca sempre tutto lo spazio, semplicemente perché i file di UNIX possono avere dei buchi, ovvero possono esserci aree

(parliamo di blocchi) del file non allocati, in tal caso vengono letti come se fossero completamente pieni di zeri e per allocarli semplicemente devo scrivere in queste zone.

`mount()` fa una operazione che si trova unicamente nei sistemi operativi UNIX, ovvero monta un filesystem: UNIX, al contrario di molti altri s.o., ha un'unica gerarchia di file. Se quindi ho piu' dischi e unita' di massa (es. chiavetta usb) devo "innestare" nell'albero principale del nostro filesystem il sottoalbero che rappresenta l'unita' di massa. Questa operazione puo' essere fatta solo da root. Inoltre, questa operazione va fatta in una directory vuota, altrimenti il contenuto sottostante sarebbe inaccessibile (ritornera' accessibile solo dopo aver rimosso ("unmounting") il device). Il comando `mount` fa esattamente la stessa cosa.

Il comando `su [utente]` serve per cambiare utente (se non si specifica il nome utente si diventa root) prendendo anche il vecchio enviroment dell'utente precedente.

Il comando `su -` serve per far assumere l'identita' di un certo utente (in questo caso root siccome non c'e' un nome utente a seguire la linetta) resettando gran parte delle variabili dell'enviroment.

Il comando `sudo` permette di fare una operazione come root.

`getdents()` e' l'equivalente in formato syscall del comando `ls` (`getdents` sta per "get directory entries"). Tuttavia, questa syscall e' pesantissima da fare siccome si un buffer in cui vengono messi tante entries di lunghezza variabile, ognuna contenente uno struct molto lungom, dunque .e' altamente sconsigliata (anche dal man ufficiale).

Per ottenere l'elenco dei file di una directory allora si fa uso di una libreria C chiamata `dirent.h`, e si chiamare la fuzioni `opendir()`, `readdir()` e `closedir()`.

Per aprire un file, si usa la syscall `open([file], [flags])`, con il quale posso anche aprire le directory (usando i flags `0_DIRECTORY | 0_RDONLY`), e che ritorna un file descriptor sottoforma di int. Possiamo anche creare un file nuovo a seconda dei flag. I flag disponibili sono visibili guardando il `man`. Ogni che un file si apre, va chiuso usando la funzione `close()`.

Torniamo a `getdents()`, questa funzione fa cosi' cagare che in pratica non esiste nelle librerie in C delle syscall. Ma noi siamo superiori a questo. Infatti, devi sapere, mio caro lettore, che possiamo chiamare anche le syscall che non sono presenti in libreria attraverso la fighissima funzione `syscall()` della libreria `sys/syscall.h`. Per chiamare `getdents()`, semplicemente facciamo:

```
syscall(__NR_getdents, d, (struct linux_dirent *) buf, DIRBUFSIZE)
```

Syscall vera e propria

Parametri da passare alla syscall, precisamete il filedescriptor della cartella, il buffer che serve al `getdents` e la sua dimensione.

Esporiamo ora l'uso della funzione `read()` e `open()`.

Per creare un file con contenuti random di 100 MB, uso il comando `dd if=/dev/urandom of=files bs=1M count 100`. Il comando `dd` in se' permette di copiare dei file velocemente. Per misurare il tempo di esecuzione di un comando, uso il comando `time [comando]`.

Usando la syscall `read()`, possiamo leggere un file attraverso un file descriptor, ma per fare cio' fara' uso di un buffer. Maggiore sara' questo buffer e maggiore sara' l'efficenza del programma, tuttavia non possiamo nemmeno farlo troppo grande. Dunque, un modo per trovare la funzione del buffer ottimale e' usare `buf.st_blksize`, che mi dice la lunghezza del buffer consigliato (questo dopo aver usato `fstat()`)

Ecco il codice d'esempio:

```
#define CPBUFSIZE
```

```
int main(int argc, char*argv[])
{
    uint8_t buff[CPBUFSIZE]
    int fdin, fdout;
    ssize_t len
    struct stat, sbuf
    fdin = open(argv[1], O_RDONLY);
    if(fdin == -1) error("open in");
    fdout = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0777 );
    if(fdout == -1) error("open out");

    int rv = fstat(fdout, &sbuf)
    if(rv == -1) error("fstat")
    blksize_t blksize = sbuf.st_blksize;
    uint8_t buf[blksize];

    while((len = read(fdin, sbuf, CPBUFSIZE )) > 0)
        write(fdout, buf, len)

    close(fdin);
    close(fdout);
}
```

Crea il file dandogli i
permessi massimi e lo apre
in sola scrittura.

Ottimizzazione
dell'uso del buffer.

Quando l'assegnamento da errore,
ho finito di leggere il file

Possiamo inoltre spostarci da un punto di un file ad un altro usando la syscall `lseek()`.

Torniamo ora a parlare dei file descriptor. In pratica, per ogni processo esiste una **tabella dei file descriptor attivi** associato a tale processo, che tiene traccia di quali file sono aperti dal processo.

Normalmente, un programma UNIX quando viene attivato un processo riceve 3 file di testo, che sono stdin, stdout e stderr, quindi per ogni processo ci saranno almeno 3 elementi nella tabella. Possiamo pensare alla tabella dei descrittori di file come un vettore. Ma che cosa e' associato a questo vettore? Nella tabella dei file aperti abbiamo due informazioni importanti per ciascun file: l'informazione piu' importante e' l'*offset*, ovvero il punto all'interno del file corrente, dopodiche' abbiamo il *vnode*, che e' una copia estesa dell'inode (che contiene tutte le informazione del file). Ora, puo' succedere che due processi aprano indipendentemente lo stesso file. In questo caso, se c'e' un file aperto da piu' processi, il *vnode* e' unico. L'*offset* invece e' diverso da processo a processo. Tuttavia, posso avere dei casi in cui due file dello stesso processo o due processi per lo stesso file possano avere l'offset in comune.

Esempio:

```
int main(int argc, char*argv[])
{
    int fdout;
    int status;
    fdout = open(argv[2], 0_WRONLY | 0_CREAT | 0_TRUNC, 0777 );
    if(fdout == -1) error("open out");

    /*codice aggiunto dopo
    dup2(fdout,STDOUT_FILENO);
    */

    switch(fork())
    {
        case 0:
            write(fdout, "CIAO\n", 5);
            exit(0);
        default:
            write(fdout, "MARE\n", 5);
            wait(&status);
        case -1:
            exit(0);
    }
    close(fdout);
}
```

Duplica il file descriptor ma in modo volontario... non per quello che abbiamo detto nell'osservazione qui a fianco.

In questo caso, duplico lo stdout nel mio file, così' ogni volta che faccio una printf scrivo nel mio file.

I file aperti durante una fork rimangono aperti per entrambi i processi, ma condividono l'elemento della tabella dei FD, quindi anche l'offset! Se uno così' legge parte del file spostando il cursore di lettura, il cursore sarà' spostato anche per l'altro processo.

Infatti, il processo scrive nel file
"MARE
CIAO"
E non solo uno dei due, come appunto dovrebbe succedere.

La `dup2()` permette di copiare un FD dentro un altro un FD. Inoltre viene anche usato durante le operazioni che fanno uso dell'operatore ">" della shell. Se invece usassimo una `dup()` (senza il numero 2 quindi), il comando ritorna il FD equivalente a quello che abbiamo inserito (es: una `dup` con FD 42 potrebbe ritornare un altro FD di valore 45, e semplicemente questo nuovo FD indicherebbe un modo equivalente di accedere al file 42).

Lezione del (4/11/2020) – Semafori fair e problema dei filosofi

Abbiamo visto nella lezione del 20/10 come nonostante avessimo provato a risolvere il problema della critical section con i test and set, la starvation fosse ancora possibile usando i TS. Dunque abbiamo introdotto il concetto di semaforo (introdotta da Dijkstra). Tenendo a mente che nel semaforo vale l'invariante $nP \leq \text{init} + nV$, abbiamo descritto il semaforo in questo modo:

```
class semaphore
{
    semaphore(init);
    void P(void);
    void V(void);
}

semaphore mutex(1);

while(true)
{
    //codice non critico
    mutex.P();
    //codice critico
    mutex.V();
}
```

Classe semaforo

Istanza della classe

Come viene usato il semaforo effettivamente

Andiamo a vedere quali proprietà soddisfa il nostro codice: in questo codice non possono entrare nella sezione critica più di un processo, dunque soddisfa la proprietà di safety. Esaminiamo il perché questa proprietà viene rispettata. Chiamiamo la somma dei 3 termini $\text{init} + nV - nP$ il **valore del**

semaforo. Quando il valore del semaforo e' 1, vuol dire che la critical section e' libera, e appena uno dei processi esegue P(), allora il v.d.s. tornera' a 0, indicando cosi' che la critical section e' occupata. Se uno dei processi proverà ad eseguire P() quando la c.s. e' occupata, semplicemente verra' bloccato. Dunque la proprieta' di mutua esclusione verra' rispettata.

Osserviamo se anche le altre 3 proprieta' (ovvero no deadlock, no starvation e no ritardi inutili) sono effettivamente rispettate.

Se abbiamo una implementazione del semaforo che e' deadlock free, allora l'intera struttura sara' deadlock free.

Quanto alle attese inutili, se un processo vuole entrare e uscire 1000 e all'altro non frega nulla di entrare, potra farlo senza alcun slowdown.

Come sempre, il problema rimane la starvation, che non e' garantita dalla sola invariante. Infatti se abbiamo due processi, uno e' velocissimo e l'altro lentissimo, che cercano entrambi di entrare nella sezione critica, avremo che quello lento non riuscirà mai ad entrarci.

Abbiamo quindi bisogno di una regola in piu' per il semaforo, oltre al vincolo dell'invariante, che permettera' di risolvere questo problema. Chiameremo questi semafori "**semafori fair**", e saranno semafori che oltre a rispettare l'invariante garantiscono che i processi che si mettono in attesa per effetto di una P() verranno "risvegliati" in ordine **FIFO** tramite effetto di una V().

D'ora in poi, almeno che non venga specificato, parleremo solo di semafori fair e non di semafori generali.

Applicando questa regola, i semafori riescono a creare una vera critical section. Tuttavia, questa non ci permette di risolvere tutti i problemi, ma e' un ottimo strumento nonetheless.

Riguardiamo il problema del produttore-consumatore. Se noi usassimo il codice del problema senza aggiungere i semafori (quindi il primissimo codice del produttore-consumatore che abbiamo visto tempo fa) saremmo pieni di race conditions. Invece, usando l'implementazione dei semafori fair, lo abbiamo risolto completamente.

```
semaphore empty(1)
semaphore full(0)
shared buf
```

```
process producer:
    while(1):
        x = produce()
        empty.P()
        buf = x
        full.V()
```

```
process consumer:
    while(1):
        full.P()
        y = buf
        empty.V()
        consume(y)
```

Facendo cosi', se il consumatore arriva veloce, trova che il semaforo full e' 0, e quindi deve fermarsi, quindi il produttore e' sempre il primo. Il codice e' abbastanza facile da leggere....

I semafori designati in questo modo servono per segnalare e indicare una certa condizione, infatti empty ci permette di indicare che il buffer sia vuoto (P ci dice che questo sia vero, e V che questo sia falso), e full che sia pieno.

COOL SOOM my child
O_O

La prossima domanda che ci poniamo e' come possiamo implementare queste strutture nell'architettura del sistema operativo. Per rispondere a questo, consideriamo questo pseudocodice:

```
class semaphore
{
    private value;
    // coda dei processi bloccati
    private queue semq;

    void semaphore(init)
    {
        mutex_in();
        value = init;
        mutex_out();
    }

    void P(void)
    {
        mutex_in();
        if (value > 0) {
            value--;
        } else {
            // si blocca il processo corrente
            readyq.remove(current);
            semq.enqueue(current);
        }
        mutex_out();
    }

    void V(void)
    {
        mutex_in();
        // controlla se ci sono processi bloccati
        if (!semq.empty()) {
            // sblocca il primo
            proc = semq.dequeue();
            readyq.enqueue(proc);
        } else {
            value++;
        }
        mutex_out();
    }
}
```

Coda dei processi che hanno fatto richiesta.

Abbiamo bisogno di trovare un modo per fare la mutua esclusione dentro ai semafori senza usare i semafori...

Blocca il processo togliendolo dallo scheduler della CPU (sotto e' spiegato meglio questo concetto).

Coda dei processi FIFO che rende il semaforo un semaforo fair.

Tornando all'esempio del cuoco di inizio corso: in un sistema multitasking la CPU viene assegnata a turno ai processi che sono nella coda dei processi pronti, quindi per fare in modo che un processo non avanzi basta toglierlo dalla coda dei processi pronti, in modo che lo scheduler non possa piu' sceglierlo e che quindi non avanzi. Quando invece un processo dovra' essere risvegliato basta metterlo nella coda ready, e allora prima o poi verra' il suo turno ed andra' avanti.

Come possiamo invece implementare `mutex_in()` e `mutex_out()`? La risposta e':

- Se abbiamo un sistema monoprocesso possiamo usare le maschere di interrupt: `mutex_in()` maschera le interruzioni (e quindi le disabilita), mentre `mutex_out()` le demaschera.
- Se invece abbiamo un sistema multiprocessore, semplicemente ci bastera' fare uso delle test and set.

Siamo certi che questa implementazione sia fair, ovvero che il primo che si blocchi sia il primo che si sblocchi? Per verificarlo, possiamo usare una dimostrazione per assurdo. **[META-NOTA]**: questa dimostrazione e' spiegata male, ma puo' essere ignorata]

Prendiamo due processi A e B: vogliamo che vada prima A poi B, e che appena A abbia finito, continui B. Il ragionamento simmetrico dovrà essere usato su B.

Se il primo a fare la P() è A, in mutua esclusione è il primo che fa la enqueue. E se invece poi arriva B che vuole fare la P(), sarà accodato. Quando A farà la j(), allora tolgo dalla coda. Siccome i due processi accedono alla coda (che è FIFO) in mutua esclusione, allora il semaforo è fair.
[fine parte della dimostrazione]

In tutto questo c'è però un piccolo problema. Che cosa ci guadagno a usare un semaforo per fare la mutua esclusione, quando la potrei fare con la TS?

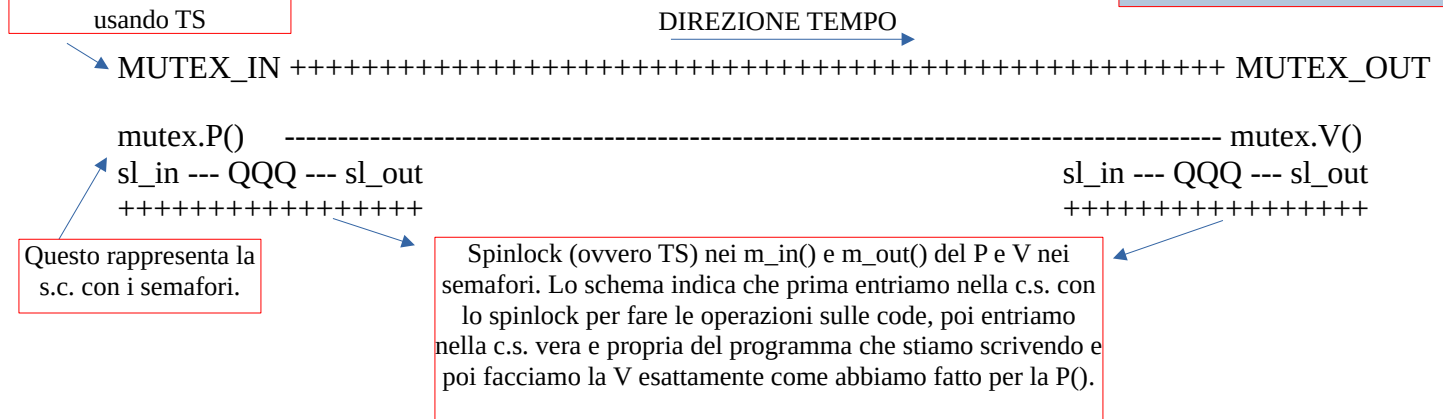
Usando il semaforo, cambiano due cose fondamentali:

1. Col semaforo, non ho busy wait; non mi metto a controllare sempre se la sezione si libera o no, siccome ci basta usare le funzioni P() e V().

Uno può pensare che comunque, siccome in un multiprocessore per implementare i mutex usiamo i TS, abbiamo comunque un grado di busy wait, che è vero in un certo senso, ma facciamo allora un esempio. Prendiamo un processo che si blocchi perché la c.s. è occupata:

Questa parte dello schema indica l'entrata in una c.s. usando TS

I "+" rappresentano il periodo in cui ho busy wait nelle due implementazioni



Quello che possiamo dedurre da questo schema è che sì, c'è un certo grado di busy waiting anche nell'implementazione con semaforo, ma solo per poche istruzioni veloci sulle code; inoltre, non abbiamo busy wait nella zona della s.c. che può essere anche molto lunga. Al contrario, usando l'implementazione con TS, abbiamo busy waiting per tutta la sezione critica.

Quindi possiamo effettivamente usare la TS anche a livello utente (ovvero nel codice scritto dall'utente) ma non conviene siccome sarebbe un grande spreco di risorse.

Con il semaforo invece, lo spinlock è presente ma solo quando "sistemiamo" le strutture dati del kernel. Dunque, i tempi di attesa applicativi non comportano busy wait.

[METANOTA]: ho riascoltato mille volte questa parte della lezione, e tutt'ora non ho capito cosa intendesse il prof dicendo questo. Mi sono limitato a trascrivere le sue parole esatte. Se qualcuno l'ha capito mi avvisi plz].

C'è inoltre da fare una precisazione molto piccola da fare rispetto alla fairness: si potrebbe dire che il semaforo fair non è FIFO in un certo caso speciale: infatti è effettivamente FIFO solo per il primo che riesce a prendere la mutex, quindi uno spinlock potrebbe arrivare per secondo e prendere quell'altro.

Dopo aver fatto queste spiegazioni più complesse del previsto, torniamo ai nostri epici semafori fair. Linux contiene una sua implementazione dei semafori, contenenti i comandi `up()` e `down()` che sono l'equivalente dei P() e V(), e anche delle implementazioni dello spinlock. Ok tutto GUCCI quindi. Ora che abbiamo uno strumento davvero potente, come disse zio Ben: "da grandi poteri derivano grandi responsabilità" e quindi anche altri problemi da risolvere.

Abbiamo visto il problema del produttore-consumatore, ma perché ora non guardiamo altri problemi campione che permettono di capire come usare i semafori: un problema interessante è quello dei producer/consumer, ovvero una versione più complessa del nostro caro e affezionato problema del producer/consumer; la difficoltà consiste nel fatto che non abbiamo più un solo consumatore ed un solo produttore, bensì dobbiamo prendere in considerazione un array di produttori e consumatori di lunghezza n. La nostra implementazione dei semafori permetterebbe comunque il funzionamento del programma, ovvero tutti i nostri elementi prodotti verranno effettivamente consumati? La risposta è sì, perché anche se abbiamo 12 produttori, i consumatori e i produttori procederanno per turni grazie al fairness del semaforo.

Un altro problema è quello del **buffer limitato**. Assomiglia al problema del p/c, tuttavia il buffer non è più composto da un solo elemento, ma da un array di c elementi (oppure come nel nostro caso da una coda di elementi). Il resto delle regole rimane invariato. Per risolvere il problema basta cambiare il valore di init di `empty()` alla grandezza del buffer... No? Assumendo che i consumatori e produttori siano uno solo, la risposta corta sarebbe sì. Se però avessimo più produttori e più consumatori, la mutua esclusione non verrebbe rispettata. Mettiamo caso per esempio che due produttori vogliano inserire entrambi un elemento nel buffer. La gestione del conteggio degli elementi funzionerebbe benissimo, ma appena entrambi si mettono a fare l'enqueue, avviene una race-condition. Questo perché le operazioni di deque ed enqueue nel buffer infatti non sono atomiche. Per risolvere questo problema mi occorre un altro semaforo (con `init = 1`) e l'accesso alla coda dev'essere fatto in mutua esclusione (quindi si mettono le operazioni di dequeue ed enqueue tra il P() e V() del nuovo semaforo).

Questo esercizio ci rivela tre importanti cose sui semafori:

- L'init dei semafori può essere utilizzato per definire quante siano effettivamente le risorse disponibili
- Ovunque ci siano variabili condivise, il loro accesso dev'essere fatto in mutua esclusione. (Nella prima versione del p/c, non c'era bisogno siccome avevo un solo elemento nel buffer).
- I semafori non risolvono automaticamente tutti i problemi: in questo problema, la mutua esclusione del buffer non è risolta nella seconda variante. O meglio, la gestione del numero degli elementi del buffer sì, ma non del buffer in sé.

Piccola digressione sugli invarianti: **[METANOTA]**: anche qui il concetto è confuso e non particolarmente utile, consiglio di skippare.]

L'invariante è una condizione che non deve variare, ovvero deve essere sempre vera e cioè deve essere mantenuto. Ma sempre vero quando? Beh quando le funzioni della libreria sono state completamente eseguite, ovvero ogni volta che una funzione di una certa classe (contenente l'invariante) vengono eseguite. In poche parole, non ci importa se l'invariante è stato modificato mentre eseguiamo la funzione, però all'inizio e alla fine dell'esecuzione della funzione sì. Spesso gli invarianti vengono usati sotto forma di asserzione nel codice, per esempio in C esiste una libreria di macro chiamata `assert.h` che consente di mettere in mezzo al codice la funzione `assert()` con una espressione booleana come parametro... e cioè serve per il binding. Ad esempio, ho bisogno che in una certa sezione del codice il puntatore non sia mai diverso da NULL, allora scrivo `assert(ptr != NULL)`. Se io esegui il codice in modalità debug e qualche asserzione fallisce, allora mi viene segnalato. Quando usiamo questo tipo di approccio, non conviene fare molti controlli siccome abbiamo avremmo un calo di performance inutile.

Un altro problema molto famoso e importante della programmazione concorrente è "**la cena dei filosofi**". Immaginiamo di avere un tavolo rotondo con 5 filosofi, e sul tavolo abbiamo 5 piatti e 5 chopstick. Un filosofo fa solo 2 cose: pensa (ovvero tempo di attesa) e mangia (tempo di consumo delle risorse). Ogni filosofo però per mangiare avrà bisogno di 2 chopstick, e ciascun chopstick è posto a sinistra e a destra del piatto. Se però ciascuno di essi non mangia e mangia sempre solo uno, vanno in



starvation. Dunque i chopstick dx e sx possono essere visti come delle risorse che i 5 processi devono condividere, ma ai quali non possono accedere contemporaneamente.

Esaminiamo un esempio del codice:

```
semaphore chopstick[5] int (1,1,1,1,1)
process philo(i), i = 0...4
    while 1:
        //think
        chopstick[i].P()
        chopstick[(i+1) % 5].P()
        //eat
        chopstick[i].V()
        chopstick[(i+1) % 5].V()
```

→ Vettore dei semafori inizializzati a 1

→ Con P(), un filosofo i prende i chopsticks i e i+1 e li usa per mangiare.

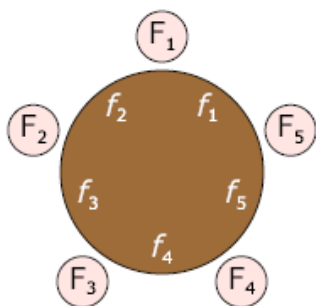
→ V() simboleggia che il filosofo ha appoggiato il chopstick sul tavolo e sono disponibili per gli altri.

Con questo codice, se tutti i filosofi diventano affamati nello stesso momento e quindi partono contemporaneamente alla stessa velocità, abbiamo deadlock. Infatti, ponendo che tutti e 5 facciano la prima P() contemporaneamente, e quindi tutti e 5 tentano di prendere le bacchette, andranno in deadlock. Tutto ciò è dovuto al fatto che abbiamo ragionato in modo del tutto simmetrico per ciascuno dei filosofi. Se per esempio un filosofo fosse mancino e prendesse per prima il chopstick a sx, non ci sarebbe più deadlock e non avremo più l'effetto del serpente che si mangia la coda.

```
semaphore chopstick[5] int (1,1,1,1,1)
process philo(i), i = 0...3
    while 1:
        //think
        chopstick[i].P()
        chopstick[(i+1) % 5].P()
        //eat
        chopstick[i].V()
        chopstick[(i+1) % 5].V()
process philo(i), i = 4
    while 1:
        //think
        chopstick[(i+1) % 5].P()
        chopstick[i].P()
        //eat
        chopstick[(i+1) % 5].V()
        chopstick[i].V()
```

→ Abbiamo aggiunto un processo in più che rappresenta il filosofo mancino (che prende prima i+1 e poi i)

Per dimostrare che effettivamente non c'è più deadlock, facciamo un esempio.



[L'immagine a fianco non è correlata al codice. Per fare in modo che sia correlata, immaginiamo che al posto di i ci sia i+1 e al posto di i+1 ci sia i+2 nelle P() e nelle V()]

Il filosofo F₁ prende la posata f₂ poi F₂ prende f₃, poi F₃ prende f₄ e poi F₄ prende f₅. L'ultimo filosofo sfigatello F₅ non potrà prendere la posata alla sua sinistra, perché sarà presa, e quindi sarà bloccato in attesa. F₁ potrà così prendere l'altra forchetta in pace (che non sarà presa da F₅ siccome è bloccato).

Questo problema ci fa vedere che alcuni esercizi possono essere abbastanza ostici, e non basta fermarci ad applicare una soluzione che sia simmetrica per tutti i componenti.

La scelta del numero di filosofi non è casuale: cinque è infatti il più basso numero che non ammette soluzione "ovvia", cioè simmetrica o per mutua esclusione. Un filosofo con una forchetta morirebbe di fame; il "ritmo di mangiata" di due filosofi sarebbe mutualmente esclusivo; con tre filosofi ognuno

sarebbe in competizione con tutti gli altri, in quanto le loro risorse sarebbero condivise fra loro; quattro filosofi si alternerebbero tra numeri pari e numeri dispari.

Un altro problema e' quello dei **lettori-scrittori**. In questo caso, abbiamo dei processi di tipo lettore e scrittore. Questi processi condividono una struttura dati, pero' la struttura dati puo' essere letta da piu' lettori senza produrre nessuna interferenza, ma NON puo' essere scritta quando ci sono lettori o scrittori che stanno lavorando. Dunque, se contiamo le componenti in atto, potremmo avere un numero arbitrario di lettori oppure un solo scrittore.

```
process reader[i] 1...tanti //sono tanti lettori
    ....
    beginread()
    read
    endread()

processs writer[i] i...tanti
    ....
    beginwrite()
    write
    endwrite()
```

detti nr = numero lettori attivi e nw = numero scrittori attivi, avremo dunque l'invariante:

$$(nw == 0 \ \&\& \ nr \geq 0) \mid \mid (nr == 0 \ \&\& \ nw == 1)$$

che indica appunto la condizione descritta prima.

Una soluzione puo' essere fatta usando un **semaforo a due stati**: questo semaforo sara' occupato o da uno scrittore solo oppure da tanti lettori. Inoltre, il processo degli scrittori, siccome uno scrittore scrive alla volta, sembra molto un problema di mutua esclusione risolvibile con una critical section.

Dunque:

```
semaphore rw(1)
```

```
beginwrite:
    rw.P()
    nw++
```

```
endwrite:
    nw--
    rw.V()
```

Il processo di scrittura avviene in questo modo in mutua esclusione. Come possiamo notare, nw-- e nw++ lo faccio tra la P() e la V(), siccome e' una variabile condivisa e quindi le modifiche vanno fatti in mutua esclusione.

Il nuovo codice usando questo tipo di implementazione sara':

```
semaphore rw(1)
semaphore mutexr(1)
```

```
beginread:
    mutexr.P() // a un semaforo che vale 1 quando vi
               // passo lo 0, quindi mutex diventa 0
    if nr==0:
        rw.P() //quando rw = 0, rw.P() rimane in a
    nr++
    mutexr.V()
```

```
endread:
    mutexr.P()
    nr--
    if nr==0:
        rw.V()
    mutexr.V()
```

```
beginwrite:
    rw.P()
```

Queste sono le implementazioni delle funzioni che vengono usate nel codice sorgente iniziale.

Proviamo ad eseguire "mentalmente" questo codice. Le variabili nw e nr inizialmente sono inizializzate entrambe a 0.

Cominciamo con un lettore. La prima operazione che fa e' mutexr.P(), in questo modo mutexr diventa 0 ed rw diventa anch'esso 0 siccome facciamo rw.P(). nr poi diventa 1. mutexr diventa poi 1, siccome eseguo mutexr.V().

In tutto questo l'invariante rimane verificato.

Uno scrittore poi prova ad entrare mentre si sta ancora facendo la read e fa rw.P(). Non riesce siccome rw==0. Allora il processo (che chiameremo lorenzo) aspetta in coda il suo turno (siccome uso un semaforo fair). Il resto e' abbastanza intuitivo.

```
nw++
```

```
endwrite:
    nw--
    rw.V()
```

In questo codice pero' abbiamo un problema non banale, ovvero la starvation degli scrittori. Infatti, se ho due lettori velocissimi e uno scrittore lento, avro' che nr sara' sempre altalenante fra 1 e 2 e non sara' mai 0.

[**METANOTA**: il prof non ha spiegato come risolvere questo problema, quindi la soluzione e' stata tratta dagli appunti dell'anno precedente.]

Per rimediare a questo problema, implementiamo una strategia diversa:

- L'arrivo di uno scrittore durante l'esecuzione di diversi processi lettore comporta il rifiuto dei processi lettore successivi, che saranno invece inseriti in una coda ed eseguiti solo quando il processo scrittore sara' terminato.
- Per impedire l'eventuale starvation dei lettori che puo' essere scaturita da questa tecnica, si fara' in modo che una strategia simile venga implementata anche dagli scrittori. In questo modo, gli scrittori favoriscono la precedenza ai processi lettori, ed in secondo luogo i lettori favoriranno l'azione dei processi scrittori.

Perfectly balanced, as everything should be

Lezione del (06/11/2020) – Device File e altre syscall

Ricordiamo che i veri utenti del sistema operativo non sono gli utenti ma i processi, che prendono le risorse attraverso l'uso di syscall.

Abbiamo visto la syscall fork(), che rappresenta l'unico modo per generare processi in Linux attraverso una syscall. Ogni volta che si esegue un programma dalla shell, la shell stessa esegue una fork(). Per far vedere meglio come funziona, ecco un esempio di shell minimale:

```
#include <stdio.h>
#include <stdlib.h>
#include <execs.h> // nota: la execs va importato come pacchetto, altrimenti si
                  // puo' usare la execve
```

```
int main(int argc, char *argv[]) {
    char* line = NULL;
    size_t linelen = 0;
    for(;;) {
        int status;
        pid_t pid;
        printf("prompt> ");
        fflush(stdout);
        if(getline(&line, &linelen, stdin) <= 0 )
            break;
        switch(pid = fork()) {
            case 0:
                execsp(line);
                exit(255);
            default:
                waitpid(pid, &status, 0);
                break;
            case -1:
                break;
        }
        printf("cmd %s\n", line);
    }
    if(line) free(line);
    return 0;
}
```

getline() legge da stdin.
Ritorna il numero di
caratteri letti; -1 se fallisce

Evito che diventi una
forkbomb assicurandomi
che esca

Questa e' una mini shell
homebrew, che permette di
eseguire tutti i programmi
della shell, tranne quelli che
per esempio cambiano
attributi del processo (es:
cd ..).

`rmdir()` e' la syscall che permette di rimuovere la directory, ma solo se questa e' vuota.
`access()` e' una syscall deprecata (ovvero non piu' in uso) che permette di vedere se il processo che la chiama puo' accedere a un file con una certo permesso (che viene specificato con un parametro `mode`, che e' diverso dalla codifica dei permessi solita `rwX` o in base 8, ma e' nel formato `R_OK`, `W_OK` etc..). Il motivo per cui la syscall e' deprecata sta nel fatto che questa syscall crea un problema di sicurezza, sempre per un problema di atomicita'. Infatti questa syscall non e' atomica, e qualcuno quindi potrebbe cambiare i permessi di un file mentre sto facendo l'accesso e potrei non accorgemene. E' dunque necessario che il controllo e l'accesso debbano essere fatti in maniera atomica.

Cominciamo ora con la lezione vera e propria: in Linux non abbiamo solo file regolari, ma abbiamo anche dei file speciali (che in realta' non sono veri file, hanno solo il descrittore del file) che hanno varie funzioni diverse. Questo perche' UNIX e' filesystem centrico ("everything is a file, otherwise it's a process"), ovvero viene dato un nome a qualsiasi cosa. Alcuni di questi file speciali possono essere mostrati col comando `ls -l /dev/tty*`. In questa cartella, la prima cosa che notiamo e' che la stringa dei permessi del file comincia con una "c". Normalmente, se guardiamo la stringa dei permessi di un file avremo un "-" (dash) come primo carattere, mentre quella di una cartella comincera' con la lettera "d". Inoltre, al posto delle dimensioni del file, abbiamo una coppia di numeri. Tutto questo deriva dal fatto che questo preciso tipo di file speciale (ovvero quelli che abbiamo trovato in questa cartella, ma ce ne sono molti altri tipi con caratteristiche diverse) non contiene dati. Di questi file esiste solo il descrittore, che indica appunto che questo e' un file speciale usato per dare un nome a un certo device, e il nome che questo file dara' al device sara' la stringa che si trova nella colonna delle dimensioni, composta da due numeri chiamati major e minor number. Il major number indica al kernel a quale device driver si fa riferimento, il minor number indica varie unita' gestite dallo stesso device driver o vari formati e peculiarita'. Questo tipo di file speciali viene detto **device file di tipo carattere**, siccome interagiscono carattere per carattere.

Altri tipi di file speciali sono i **device file a blocchi** (e infatti hanno come primo carattere della stringa dei permessi la lettera "b"), che possiamo trovare per esempio attraverso il comando `ls -l /dev/vd*`; i file in dev con il prefisso `vd` rappresentano i dischi virtuali del nostro sistema. Per vedere quale sia il file del device terminale sto scrivendo, mi basta usare il comando `tty`, che ci mostrera' il device file del nostro terminale. Se infatti proviamo a scrivere su questo file, cio' che scriveremo all'interno comparira' anche sul nostro terminale. Questa logica vale per ogni dispositivo: ogni device quindi avra' un finto file attraverso il quale Linux riuscirà a comunicare con esso. In entrambi gli esempi abbiamo notato che alcuni di questi file speciali si trovano nella cartella `/dev`: questa cartella in realta' non esiste veramente sul disco, bensì e' gestita automaticamente dal processo demone `udev` (quindi non lo fa il kernel in automatico) che quando si collega un device crea una directory.

Come faccio a creare dei file speciali o un device nostro? Mi basta usare la syscall `mknod()`. Ecco un esempio:

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

```
int main(int argc, char *argv[]){
    dev_t devno = makedev(atoi(argv[2]), atoi(argv[3]));
    mknod(argv[1], S_IFCHR | 0777, devno);
    // chmod(argv[1], 0666);
}
```

`makedev()` si usa anche per creare i file vuoti.

Cambiando i permessi del file riusciamo a crearlo.

Il flag `S_IFCHR` dice alla funzione che stiamo creando un file speciale di tipo carattere.

Il kernel per usare i file di device non fa uso del nome, bensì guarda unicamente il minor e il major number del file. [Questo codice fatto dal prof non funzionava. E' stato corretto nella lezione del 13/11].

[METANOTA]: Tutte le informazioni aggiuntive che il prof ha accennato sui comandi e syscall che abbiamo già trattato tempo fa sono già state aggiunte nelle loro rispettive lezioni.]

Torniamo a vedere la `lseek()`, la funzione vista qualche lezione fa che permette di posizionarsi all'interno di un dato file. Abbiamo come i processi siano composti da più thread in modo da avere una gestione parallela; con la funzione `lseek()`, ci sarebbero problemi di atomicità. Infatti, se un processo volesse scrivere in posizione 100 del file tramite un `lseek()` e un altro processo volesse scrivere concorrentemente al primo in posizione 200, potresti avere un interleaving strano e i risultati potrebbero essere diversi da quelli voluti (siccome il file è condiviso fra i thread).

Per risolvere questo problema esistono syscall come `pread()` e `pwrite()` che permettono a più thread di scrivere e leggere un file contemporaneamente senza interferire fra loro, siccome queste operazioni non comportano la modifica dell'offset.

Altre forme di read o di write sono `readv()`, `writew()` ovvero la read e la write vettoriali. Queste ci vengono in aiuto nel caso volessimo scrivere dei dati che non sono in un buffer unico, ma che provengono da più buffer. Come prima abbiamo alcuni problemi di concorrenza nel caso avessimo un accesso in contemporanea in questi buffer. In più, dobbiamo tenere in conto che usare le syscall richiede un certo numero di risorse, non funzioni a costo nullo; quindi vogliamo minimizzarne l'uso. Lo stesso problema si ha quando si vuole spedire un pacchetto in rete. Anche la comunicazione in rete è gestita da un file, e scrivere un pacchetto significherebbe scrivere su questo file. Come sappiamo dal corso di reti, in un pacchetto dobbiamo aggiungere vari header per ogni layer dello stack ISO/OSI, le cui informazioni possono essere in buffer differenti. `readv()` e `writew()` vengono fatte in maniera atomica.

Abbiamo inoltre visto chiamate come la `open()`, la `stat()`, `chmod()`, `truncate()` etc...

Per ciascuna di queste syscall abbiamo delle sorelle che finiscono col suffisso "at" (es. `openat()`). Anche qui, ciò è dovuto alla programmazione concorrente: l'uso di `open()`, `stat()` etc.. fallisce nel parallelismo quando si ha una risorsa condivisa, e in questo modo le funzioni diventano obsolete. Le funzioni che finiscono con "at" servono per non avere più il concetto di current working directory. Tutte queste funzioni hanno come primo parametro un file descriptor di una directory. Tale file descriptor indica cosa considerare attualmente come current working directory. Notiamo che come secondo parametro, abbiamo anche un pathname. Se questo pathname è assoluto, allora il file descriptor (ovvero il primo parametro) non viene nemmeno considerato, altrimenti se il path è relativo viene visto come il punto di partenza di scansione dell'albero del file system. Questa syscall inoltre si può usare anche per indicare un punto preciso nell'albero del filesystem attraverso il flag `O_PATH`, e dopo possiamo usare il fd ottenuto con altre funzione.

La volta abbiamo visto che i file interagiscono con le operazioni base del set di syscall, quindi se abbiamo dei file aperti e facciamo una `fork()`, tutti e due i processi continueranno ad avere il file aperto.

Normalmente dopo una `execve()` un file aperto viene ereditato dal nuovo programma, che quindi si ritroverà dei descrittori già disponibili e aperti per poter continuare le operazioni. Tuttavia se guardiamo la `open()`, tra i vari flag che possiamo mettere c'è la `O_CLOEXEC`: settando questa flag, al momento della `exec()`, il file verrà chiuso.

Abbiamo anche un'altra syscall, ovvero `fcntl()`, che consente di fare tante operazioni, per esempio consente di fare la duplicazione di un file descriptor (anche se si può usare `dup()`), o di fare forme di locking, oppure di fare la `O_CLOEXEC` se ci siamo dimenticati di farlo con `openat()` in precedenza.

Anche se i device vengono visti come file, ci sono molte operazioni sui dispositivi che non si possono fare solo con operazioni di lettura e scrittura (Per esempio: controllare volume, alti e bassi di un dispositivo audio). Per ovviare a questo problema esiste la syscall `ioctl()`.

Questa syscall prende come parametro un file descriptor che rappresenta un device e un altro parametro che indica la richiesta che facciamo al device. Per vedere la lista delle richieste che possiamo fare, possiamo usare il comando `man ioctl_list`.

Esistono anche syscall per la gestione degli utenti.

Un processo ha almeno 3 utenti e 3 gruppi. Esiste infatti **l'utente reale, l'utente effettivo e l'utente salvato**, stessa cosa per i gruppi. Normalmente queste 3 entità coincidono fra loro. Per vedere le identità di questi 3 utenti, basta usare il comando `cat /proc/`echo $$`/status`. L'utente e il gruppo servono per determinare chi può fare cosa. Tornando alla lezione dei permessi, avevamo visto che il file `/usr/bin/passwd` aveva il `setuserid` nella stringa dei permessi. Questo file, quando viene eseguito da un `exec`, fa in modo che gli effetti di questa esecuzione siano quelli che avrebbe se fosse lanciato da `root`, ovvero cambia l'utente effettivo (ovvero l'utente che fa effetto su quel file). Quando creiamo un file, l'utente effettivo è uguale all'utente reale. Ma quando uso il comando `chmod 4755 [file]` (che mi permette di "accendere" il bit "s" nella stringa dei permessi), noterò che se apro questo file in una shell diversa loggandomi con un utente diverso da quello che ha creato il file, l'utente effettivo e l'utente reale saranno diversi: l'utente reale sarà l'id della shell che sto usando, mentre l'utente effettivo sarà l'id della shell con cui ho fatto `chmod`. Quindi, il file verrà eseguito dalla seconda shell "impersonandosi" come la prima in ogni caso.

La syscall `setuid()` può essere fatta solo da `root`, `seteuid()` da qualsiasi utente ma potrà solo cambiare fra uno dei 3 utenti del file.

Ma in tutto questo, a che cosa serve l'utente salvato? L'utente salvato viene utilizzato quando un processo privilegiato è in esecuzione (come `root` ad esempio) e deve eseguire alcune attività non privilegiate.

Lezione del (11/11/2020) – Semafori binari, passaggio del testimone

Detto in modo molto brutale, un **semaforo binario** è un semaforo che può assumere come valori esclusivamente 0 e 1. Detto così, basta che prendiamo il nostro invariante che abbiamo introdotto con la definizione di semaforo, e lo modifichiamo:

$$0 \leq \text{init} + nV - nP \leq 1$$

Ovviamente, **come prima il valore con cui inizializziamo il semaforo non può essere negativo e può solamente essere 0 o 1.**

Questo semaforo può assumere quindi solo due stati possibili. Dunque se il semaforo è 0 e tendo di fare la `P()`, ovviamente come prima il processo si bloccherà, mentre se voglio fare una `V()` non sarà bloccante.

Le cose cambiano quando il semaforo è 1. In questo caso se voglio fare una `V()`, sarà bloccante!

Abbiamo quindi un semaforo che avrà ben due condizioni di blocco

Possiamo implementare questi semafori binari con i semafori normali.

```
class binarysem:
    private semaphore S0, S1
    public binarysem(v): S0(v), S1(1-v)
    public binaryP(): S0.P(), S1.V()
    public binaryV(): S1.P(), S0.V()
```

Supponiamo che il programma dia errore se `v` è un numero negativo o non valido

Essenzialmente sto usando due semafori che "adopero al contrario"

Per illustrarne il funzionamento, facciamo un esempio:

Creiamo un semaforo binario `bS` che ha valore iniziale 0. Questo significa che `S0 = 0` e `S1 = 1`. A un certo punto, un processo `A` esegue il metodo `bS.binaryV()`. Riguardando il codice scritto sopra, noteremo che prima di tutto esegue `S1.P()` (che può farlo siccome il valore del semaforo `S1` è 1) ed `S1` sarà uguale a 0, e subito dopo farà `S0.V()`.

Abbiamo poi un altro processo `C` che tenterà di fare `bS.binaryV()`. E in questo caso, `C` non potrà proseguire, e rimarrà bloccato.

HERE COMES A NEW CHALLENGER! Si unisce anche un altro processo D, che fara' bS.binaryP(), che potra farlo siccome il valore di bS (e di S0) e' 1. Finalmente il processo C potra' cosi' continuare a fare la sua swag operazione, che completera la bS.binaryV() mettendo cosi' S0 a 1 e S1 a 0. Se il processo A vorra' fare una binaryP(), potra' (siccome il valore di bS e' a 1). E D invece non potra fare un'altra binaryP().

La prossima domanda che ci poniamo e': possiamo fare semafori fair non binari con semafori binari? # Proviamo a costruire una prima implementazione:

```
class semaphore:
    binarysem mutex(1)
    binarysem blocked(0)
    int value;
    int waiting = 0;
    public init(v): mutex.bP(); value=v; mutex.bV();

    public P(): mutex.bP();
        if value == 0:
            waiting++;
            blocked.bP()
        else:
            value--
            mutex.bV()

    public V(): mutex.bP()
        if waiting > 0:
            waiting--
            blocked.bV()
        else:
            value++
            mutex.bV()
```

Questo semaforo rappresenta un modo per fare mutua esclusione, siccome come sappiamo tutte le operazioni fatte sulle variabili condivise vanno fatte in mutua esclusione

Numero dei processi bloccati

Inizializza il semaforo in mutua esclusione.

Se il valore del semaforo e' 0 allora conto che c'e' un processo bloccato in piu' e faccio blocked.P() (ovvero il fermo il processo). Altrimenti, se il valore e' maggiore di 0 calo di uno il valore del semaforo.

Se ci sono semafori bloccati, sblocca il primo. Altrimenti, aumenta il valore del semaforo.

Questa implementazione crea deadlock. Esaminiamo il perche'.

Creiamo un semaforo che ha come valore iniziale 0 (dichiariamolo con semaphore Sem(0)).

Il nostro semaforo Sem inizialmente avra' questo stato:

(1) Sem → mutex:1 blocked:0 value:0 waiting:0

Prendiamo adesso un processo A che faccia Sem.P(). Lo stato in cui sara' il semaforo sara':

(2) Sem → mutex:0 blocked:x value:0 waiting:1

Il processo A si blocca nella blocked.bP(). Notiamo che a questo punto, mutex e' rimasto 0, quindi nessun altro processo potra' fare mutex.bP() ed entrare nella critical section!! Il risultato sara' quindi la deadlock.

Da questo ne traiamo che una regola importantissima quando si lavora sui semafori e' che non bisogna mai fare una P() bloccante dentro una mutua esclusione, in quanto la mutua esclusione non verra' mai rilasciata e non avro' modo di proseguire.

Per risolvere questo problema, possiamo per esempio modificare il codice della P() in questo modo:

```
public P(): mutex.bP();
    if value == 0:
        waiting++;
        mutex.bV()
        blocked.bP()
    else:
        value--
        mutex.bV()
```

Abbiamo aggiunto la bV() della mutex dentro al primo ramo dell'if...

...e anche nel ramo else dell'if. In questo modo, non ho piu' un mutex.bV() fuori da questi due rami e non avro' piu' deadlock.

Esaminiamo la correttezza di questa soluzione eseguendola mentalmente. Il passaggio (1) rimane invariato. Nel passaggio (2) avremo che il processo A rimarra' comunque bloccato in blocked.bP(), ma che avremo che il valore della mutex sara' 1 (siccome abbiamo anche eseguito una mutex.bV()), quindi:

(2) Sem → mutex:1 blocked:x value:0 waiting:1

Immaginiamo ora che un altro processo C esegua Sem.V() del semaforo... avremo come risultato:

(3) Sem → mutex:1 blocked:1 value:0 waiting:0

La variabile waiting (ovvero il numero di processi che stanno aspettando) tornera' a 0 siccome il processo A riuscirà a sbloccarsi in questo modo, e blocked tornera' a 0 siccome A riuscirà a fare blocked.bP().

Esaminiamo un caso piu' incasinato: partendo da (3), ma senza sbloccare A, immaginiamo ora che ci sia un altro processo D piu' veloce di A che faccia la Sem.P(). Avremo che il processo anch'esso si blocchera' (senza finire la sua esecuzione):

(4) Sem → mutex:1 blocked:x value:0 waiting:2

Abbiamo cosi' due processi A e D entrambi in attesa su blocked.bP(). Arrivera' poi il processo C che fa Sem.V(). C prende il mutex, ma facendo blocked.bV() succede un errore... :

(5) Sem → mutex:0 blocked:!! value: (undefined) waiting:1

Succede la DEADLOCK! Questo in quanto il semaforo blocked e' binario, siccome il suo valore non puo' essere maggiore di 1 (e non viene ridotto siccome sia A che D sono bloccati in P()) e la mutex viene occupata senza essere liberata.

Per risolvere questo problema possiamo usare la tecnica che abbiamo usato prima, ovvero mettiamo spostiamo la mutex.bV() mettendola in ciascuno dei due rami if e else (e prima di blocked.bV()).

```
public V(): mutex.bP()
    if waiting > 0:
        waiting--
        mutex.bV()
        blocked.bV()
    else:
        value++
        mutex.bV()
```

Abbiamo aggiunto il mutex.bV()

E abbiamo spostato il mutex.bV() nell'else

Notiamo come la P() e la V() abbiano la stessa struttura simmetrica.

Nonostante cio', abbiamo comunque delle complicazioni. Per esempio, torniamo al caso precedente in cui il codice di V() non era ancora stato modificato. Essendo il valore di blocked gia' ad 1, non potevamo fare la V() una seconda volta. Cio' non crea un vero e proprio deadlock, siccome A potra' andare avanti, ma potrebbe comunque creare alcune complicazioni.

Proviamo a pensare a un modo diverso di risolvere questo problema. Pensiamo a come risolvere questo problema creando un modello:

```
task:
    .... aspettare la condizione di poter fare l'operazione
    fare l'operazione
    riattivare tutti i processi che ora possono andare avanti

P():
    aspetta che value>0
    value--

V():
    value++
    se c è una P in attesa bloccata
    possiamo pensare in maniera descrittiva di fare questa cosa, ma queste cose
    vanno fatte in mutua esclusione
```

Quello che dovra' fare il nostro programma

Queste operazioni devo farle in mutua esclusione.

Per risolvere questo problema, possiamo provare a pensare alle mutex in modo diverso. Nel primo problema infatti abbiamo ragionato pensando che lo stesso processo che apre la mutex debba anche essere quello che lo chiude. Ma chi ha detto che deve essere obbligatoriamente così? Potremmo per esempio pensare alla mutex come a un *testimone* che viene passato fra i corridori durante una staffetta. Introduciamo quindi una tecnica detta **passaggio del testimone**, dove il testimone è la mutex. Questo modo di agire ci consente di scrivere codice facendo in modo che se un processo cambia lo stato (in questo caso il valore del semaforo), si possa attivare un altro processo lasciandogli la mutex. Proviamo a vedere come possiamo implementare questo.

```
class semaphore:
    binarysem mutex(1)
    binarysem blocked(0)
    int value;
    int waiting = 0;
    public init(v): mutex.bP(); value=v; mutex.bV();
```

```
public P():    mutex.bP();
               if value == 0:
                   waiting++;
                   mutex.bV()
                   blocked.bP()
                   waiting--
               value--
               mutex.bV()
```

```
public V():    mutex.bP()
               value++;
               if waiting > 0:
                   blocked.bV()
               else:
                   mutex.bV()
```

La P() procede solo se il valore del semaforo è >0. In caso sia 0, mi devo fermare. Quindi scrivo che c'è un processo in attesa in più, rilascio la mutua esclusione e mi fermo su blocked. Uno può pensare che le operazioni che facciamo di diminuzione dei "waiting" dopo siano errate, siccome non si fanno in mutex, **ma quando un processo una blocked.bV(), lo faccio senza rilasciare la mutua esclusione!** Questo mi dà una proprietà fondamentale, che mi permette di dire che se la V() sblocca un processo, **io riattivo il processo che avevo bloccato PASSANDOGLI LA MUTUA ESCLUSIONE!**

Se non c'è nessun processo in attesa, rilascio la mutua esclusione.
DAMN BRO THATS DEEP AF! O_O

Per capire come opera questo programma, facciamo un esempio di esecuzione esattamente come abbiamo fatto per i codici precedenti. Considerando che il passaggio (1) è uguale anche in questo caso, partiamo dal passaggio subito dopo (senza ancora averlo eseguito però).

Abbiamo ancora il nostro processo A che esegue una Sem.P(). A questo punto avremo che:

(2b) Sem → mutex:1 blocked:x value:0 waiting:1

Il processo A, come prima, si fermerà nella blocked.bP(). Adesso, prendiamo un processo B esegua Sem.V(). A questo punto, attiva la mutex e quindi mutex sarà ad 0, value aumenta di 1, e siccome abbiamo un processo che sta aspettando (siccome waiting è > 0), eseguirà anche blocked.bV(), che porterà il valore di blocked a 1. Prima di eseguire quest'ultima funzione, però, il semaforo avrà questi valori:

(3b) Sem → mutex:0 blocked:0 value:1 waiting:1

Eseguendo blocked.bV(), il processo B passa il controllo della mutex al primo processo bloccato (in questo caso A) che continuerà in mutua esclusione. Dopo che A continuerà, avremo che waiting e value torneranno a 0, e si chiuderà la mutua esclusione (di cui il valore tornerà ad 1).

(4b) Sem → mutex:1 blocked:0 value:0 waiting:0

Torniamo alle condizioni di (2b), in cui il processo A era bloccato. Immaginiamo poi che un altro processo C esegua Sem.P(), bloccandosi in blocked.bP(). Questo vuol dire che prima mutex sarà settato ad 1, poi waiting aumenterà a 2 (siccome infatti ho due processi bloccati, A e C) e mutex tornerà nuovamente a 0. Le condizioni del nostro semaforo ora saranno:

(3c) Sem → mutex:1 blocked:x value:0 waiting:2

Adesso poniamo che un processo D faccia Sem.V(). In questo caso D prende la mutua esclusione, aumenta value a 1, e siccome waiting > 0, allora eseguirà blocked.bV().

(4c) Sem → mutex:0 blocked:1 value:1 waiting:2

In questo modo il processo A, il primo nella coda dei processi bloccati, procederà e il valore di blocked sarà settato a 0, waiting sarà ridotto di 1 e value pure.

(5c) Sem → mutex:1 blocked:0 value:0 waiting:1

Il passaggio del testimone è una tecnica estremamente potente. Come abbiamo visto prima infatti, ci permette di far ripartire la P() “proteggendoci” da eventuali cambi di stato o variabili globali fatte da altri processi, siccome la P() viene fatta ripartire mantenendo la mutua esclusione. Inoltre, il fatto che siamo riusciti a implementare i semafori normali con quelli binari ci dice anche che noi possiamo risolvere qualsiasi problema risolvibile coi semafori “normali” utilizzando i semafori binari. Vale anche il viceversa siccome abbiamo implementato i semafori binari con i semafori normali. Dunque, i due paradigmi hanno lo stesso potere espressivo.

Molti problemi si possono risolvere con il layout del modello del passaggio del testimone, traducibile in pseudocodice come:

```
< ASPETTA cond_i -> azione_i >  
< azione_j >
```

```
P-> < ASPETTA value > 0 -> value-- >  
V-> < value++ >
```

Se il problema ha la struttura “aspetta una condizione e fai una azione, il tutto in mutex”, allora si può riscrivere in questo modo a cervello spento. Con value++ e value-- si intendono i cambiamenti di stato del semaforo.

```
< ASPETTA cond_i -> azione_i >: azione P()  
    mutex.P()  
    if (!cond_i)  
        contatore_cond_i++  
    mutex.V()  
    sem_cond_i.P()  
    contatore_cond_i--  
    azione_i  
    NUOVO_STATO()
```

Contatore condizione di continuità

Qui c'è il passaggio del testimone

Qui cambia lo stato del nostro semaforo. La condizione di continuità ora era già verificata oppure lo è diventata. Facendo NUOVO_STATO(), il sistema si chiede “visto che lo stato è cambiato, quali sono i processi che possono andare avanti a causa di questo cambiamento?”

```
< azione_j >  
    mutex.P()  
    azione_j  
    NUOVO_STATO()
```

Azione non condizionata, questa va direttamente a cambiare lo stato. (azione V())

```
NUOVO_STATO():  
    if (cond_0 && contatore_cond(0) > 0) sem_cond_0.V();  
    else (cond_1 && contatore_cond(1) > 0) sem_cond_1.V();  
    else (cond_2 && contatore_cond(2) > 0) sem_cond_2.V(); //same e così via  
    ....  
    else  
        mutex.V()
```

Guarda tutte le condizioni di attesa, e se la condizione di attesa è verificata e c'è qualcuno che la sta aspettando, allora se così non rilascio la mutua esclusione e va avanti lui. Solo quando nessuno può andare avanti, rilascio la mutua esclusione (else finale).

(AGGIUNTO DALLA LEZ. 18/11) Questa parte è non deterministica, non c'è regola automatica per permutare come definire cond_0, cond_1 etc. Potrei permutare le condizioni e il mio codice avrebbe ancora senso.

Questo modo di pensare risolve una buona gamma di problemi. C'è da considerare però che questo metodo non è usabile completamente a cervello spento: la parte NUOVO_STATO() infatti nasconde un problema che il programmatore dovrà risolvere. Infatti, come possiamo decidere quali dei processi delle varie condizioni mandare avanti se più di essi sono verificati? Lo vedremo la volta prossima.

Lezione del (13/11/2020) - Event Programming

I socket, che abbiamo anche visto al corso di Reti, sono implementati a livello del sistema operativo attraverso syscall.

Piccola parentesi: l'`execs()`, o `execsp()` che si trovano nella libreria `execs.h` e che abbiamo usato per la scrittura di una shell fatta in casa NON sono system calls! Solamente `execve()` e' una vera syscall. La differenza sta nel fatto che mentre le varie `execve()` (o anche `execvp()` per esempio) fanno parte della `libc` e sono definite anche nello standard POSIX, l'`execsp()` e' una libreria da caricare a parte (creata proprio da Davoli tra l'altro, BIG FLEX) e scaricabile da Debian o Ubuntu come pacchetto. Se non vogliamo usare questa funzione, dobbiamo convertire la stringa di caratteri che passiamo come comandi della shell in un vettore di stringhe. Per fare cio', possiamo usare la funzione `strtok()` della libreria `string.h`, che divide una stringa in token dato un delimitatore.

Il comando `losetup` fa in modo che `/dev/loop0` sia un device che rappresenta l'immagine del disco.

[Correzione errore del 06/11 di `mknod()`]

L'errore del codice era il tipo del file speciale che usavamo. Il prof creava un file speciale di tipo carattere, mentre per fare cio' e' necessario un file di tipo blocco. Per creare tale tipo di file, usiamo il flag `S_IFBLK`.

[Fine correzione]

Chiariamo la definizione di special file. Uno special file serve solo per dare il nome a un device (ad esempio il device "7, 0"), che permette un collegamento fra un pathname e l'indicazione al kernel di un device, e in questo modo si possono usare le syscall standard usando il nome che abbiamo definito attraverso il file.

Iniziamo ora la lezione vera e propria.

La syscall `pipe()` e' una syscall abbastanza particolare, che prende come parametro un vettore di due file descriptor e ritorna un intero. Ecco un esempio del suo utilizzo.

```
#include<stdio.h>
#include<unistd.h>
```

```
int main(argc, char*argv[])
{
    int pipefd[2];
    pipe(pipefd);
    write(pipefd[1], "ciao", 5);
    char buff[10];
    int rv = read(pipefd[0], buf, 10);
    if (rv < 0)
        return 1;

    buf[rv] = 0;
    printf("%s\n ", buf);
}
```

Il file descriptor di indice 0 serve per leggere, quello di indice 1 per scrivere.

Istanza due file descriptor dai file descriptor passati.

La syscall `pipe()` lavora essenzialmente come un tubo: tutto cio' che scriviamo nel descrittore 1, lo possiamo poi leggere nel descrittore 0, ovvero crea un canale di dati unidirezionale fra il file descriptor `pipefd[0]` e il FD `pipefd[1]`. Ogni byte scritto in `pipefd[1]` verra' memorizzato in un buffer collegato a `pipefd[0]`. La domanda che sorge spontanea e': a che serve sta roba? Sembra quasi

inutile... Ma invece ha un senso: ricordiamo infatti che i file descriptor infatti vengono ereditati durante una `fork()`! La `pipe()` inoltre, come abbiamo accennato prima, fa uso di un bounded buffer, ovvero il suo contenuto deve essere drenato col tempo altrimenti si satura. Facciamo un altro esempio dell'uso di `pipe()`.

```
#include<stdio.h>
#include<unistd.h>
#define BUFSIZE 1024
```

```
int main(int argc, char*argv[])
{
    int pipefd[2];
    pipe(pipefd);
    char buf[BUFSIZE];
    switch(fork())
    {
        case 0:      //figlio
        {
            close(pipefd[1])
            while (( len = read(pipefd[0], buf, BUFSIZE)) > 0){
                write (STDOUT_FILENO, buf, len)
            }
            break;
        }
        default:{    //padre
            close(pipefd[0])
            while (( len = read(STDIN_FILENO, buf, BUFSIZE)) > 0 )
            {
                write (pipefd[1], buf, len)
            }
            break;
        }
        case -1: return 1    //errore
    }
}
```

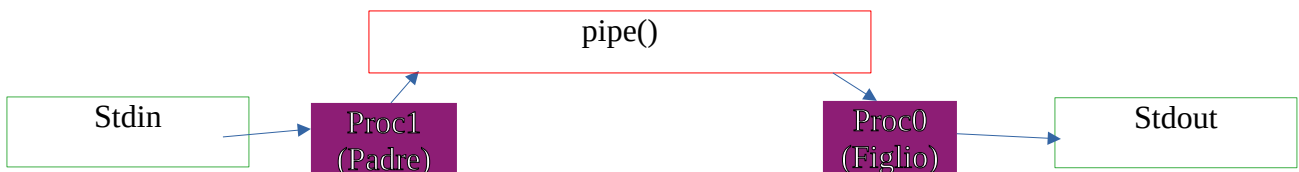
L'effetto complessivo di questo programma e' quello di un **cat**, ma fatto passando dati fra due processi (li legge il padre e li riscrive il figlio).

Il processo figlio legge dalla pipe() e scrive in output

Il processo padre legge dalla input e scrive sulla pipe()

Se non chiudo i file, potrei avere dei problemi. Per esempio, l'endoffile potrebbe non essere rilevato correttamente, perche' l'altro processo potrebbe stare ancora scrivendo. In questo modo, la lettura del figlio loopa all'infinito.

Ecco una rappresentazione grafica di cio' che succede:



Proviamo ora a creare un codice che faccia la stessa cosa del comando `ls | sort -r` usa la `pipe()`:

```
int main(int argc, char*argv[]){
    int pipefd[2];
    pipe(pipefd);

    switch(fork()){
        case 0:{ //figlio
            dup2(pipefd[0],0)
            close(pipefd[0])
            close(pipefd[1])
            execlp("sort", "sort", "-r", NULL);
        } break;

        default:{ //padre
            dup2(pipefd[1],1)
            close(pipefd[0])
            close(pipefd[1])
            execlp("ls", "ls", NULL);
        }break;

        case -1: return 1 //errore
    }
}
```

Copia il FD dello
stdin dentro al
lato lettura della
pipe.

Il figlio fa in modo che il `pipefd[0]` diventi lo `stdin`. Poi chiudo le pipe siccome ho già duplicato i file descriptor (attraverso `dup2()`, inoltre il secondo parametro è 0 siccome è il primo file della tabella degli fd aperti, come abbiamo detto qualche lezione fa). Quindi il figlio prenderà lo `stdin` come file di input della pipe.

I parametri di `execlp` sono:
programma da lanciare,
`argv[0]`, `argv[1]` e fine
comando.

Il padre, che invece dovrà fare il comando `ls`, duplica il lo `stdout` nell'output del pipe e poi chiude entrambi i fd siccome ha già duplicato l'`stdout` nel pipe.

Copia il FD dello
stdout dentro al
lato scrittura della
pipe.

Devo chiudere entrambi i file in entrambi i processi, siccome ogni processo ha la sua tabella dei file descriptor.

Abbiamo così capito cosa succede a livello di syscall per usare la funzionalità dell'operatore `|` nella shell. La `pipe()` dunque ci permette di creare un canale fra due FD, e con la `fork()` si creeranno processi che ereditano i FD. Ma come si fa a far parlare fra loro due processi che non sono parenti? In questo caso non possiamo usare la `pipe()`, ma comunque il processo sarà abbastanza simile. La peculiarità che dovremo aggiungere sarà quella di "dare un nome" a questa nuova `pipe()`, e per fare tutto ciò, siccome siamo in UNIX, il modo classico è farlo attraverso il file system. Queste nuove pipe saranno chiamate **fifo** [wow non ci confonderà per niente questo nome nono] (o **named pipe**). Il comando che ci permette di creare le fifo sarà `mkfifo [file]`, che creerà un file speciale di tipo fifo. Se noi scriviamo su questo file da un terminale e in tanto lo leggiamo con un altro, i messaggi scritti verranno passati.

Per tradurre questo comando in codice, possiamo usare la syscall `mknod()`, che abbiamo usato in precedenza per creare nuovi file, e ci permette di creare anche dei file di tipo fifo usando il flag `S_IFIFO`! Una volta costruita la named pipe, si può usare esattamente come se fosse un file. Un file fifo vuole un writer e un reader, e quando uno dei due viene chiuso si riceve l'endoffile.

Proviamo a fare il codice di prima usando i named pipe.

```
int main(int argc, char*argv[]){
    int pipefd[2];
    char buf[BUFFLENGTH];
    size_t len;
    char s[] = "tmp/tmpfifoXXXXXX";
    mkstemp(s);

    mknod(s, S_IFIFO | 0666);

    switch(fork()){
        case 0:{           //figlio
            int fd = open(s, O_RDONLY);
            dup2(fd,0);
            close(fd);
            execlp("sort", "sort", "-r", NULL);
        } break;

        default:{          //padre
            int fd = open(s, O_WRONLY);
            dup2(fd,1);
            close(fd);
            execlp("ls", "ls", NULL);
        }break;

        case -1: return 1    //errore
    }
}
```

Mi permette di creare un file temporaneo con un nome unico, rimpiazzando le X di s con un numero. Se non usiamo questa funzione potremmo avere una race condition. Ogni volta che viene lanciato, si ha un nome diverso.

Crea il file della named pipe.

Il FD 0 e' corrispondente allo stdin, il FD 1 allo stdout e FD 2 allo stderr

[Il programma purtroppo non funziona, ma comunque era un esempio di non troppa importanza].

Proviamo ora a creare una "chatroom" tra due terminali diversi usando le named pipe (due terminali che sono attivi nello stesso pc ovviamente). Quindi, avro' bisogno di due pipe. Quello che si scrivera' su un terminale dovra' entrare in una pipe ed essere letto dal secondo terminale, e la seconda pipe dovra' fare la stessa cosa simmetrica.

Per fare qualcosa del genere possiamo procedere in due modi: potremmo fare un programma multithread, dove un thread legge dal terminale e manda sulla pipe e l'altro invece legge dall'altra pipe e scrive sull'altro terminale (avremo un thread per terminale quindi, e una pipe sx-dx e un'altra dx-sx); l'altra opzione sarebbe farlo a **eventi**. Esistono delle syscall che sono capaci di aspettare degli eventi da insiemi di descrittori (la piu' famosa e' la syscall `select()`). Essenzialmente, una parte del nostro programma sta fermo in attesa che succeda qualcosa, e al nostro programma possono arrivare o dati dallo stdin o arrivare dati dal canale. Per gestire questi due casi diversi, si usa appunto la `select()`. Tra i parametri di questa funzione abbiamo un insieme di descrittori (detti `fd_set`) ai quali possiamo aggiungere e togliere elementi, e controllare se un elemento e' acceso o spento. Il primo parametro invece e' `l'nfds`, ovvero il numero massimo del descrittore definito in uno degli insiemi +1 (scomodissimo). Negli argomenti di questa funzione abbiamo 3 insiemi di descrittori: uno che indica l'insieme dei descrittori dai quali vogliamo leggere, l'altro l'insieme dei descrittori su cui possiamo scrivere e l'ultimo l'insieme degli eventi eccezionali (es. i messaggi `out_of_band`, ovvero se dobbiamo interrompere una operazione con CTRL+C durante un trasferimento dati. Se non usassimo un evento eccezionale, alcuni file continueranno a inviarsi prima che il canale venga effettivamente chiuso). In ogni caso l'`exceptfds` (gli fd degli eventi eccez.) si usa molto raramente, quelli piu' usati sono i `readfds` e `writelfds`. Se devo aspettare di leggere dallo stdin o dal canale, usiamo i `readfds`. L'ultimo parametro consente di avere un timeout, che permette di stabilire il tempo di attesa di un evento. Il valore di ritorno e' il numero di fd degli eventi che sono stati rilevati (e quindi hanno il bit acceso).

Proviamo ora a scrivere il codice della nostra chatroom tra terminali.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#define BUFSIZE 1024
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
```

```
int main(int argc, char *argv[]){
    int fdin = open(argv[2], O_RDONLY | O_NONBLOCK);
    int fdout = open(argv[1], O_WRONLY);
    char buf[BUFSIZE];

    for(;;){
        fd_set readset;
        FD_ZERO(&readset);
        FD_SET(0, &readset); // 0 vuol dire standard input
        FD_SET(fdin, &readset);
        select(fdin+1, &readset, NULL, NULL, NULL);

        if(FD_ISSET(0, &readset)){
            size_t len = read(0, buf, BUFSIZE);
            if(len == 0)
                break;
            write(fdout, buf, len);
        }

        if(FD_ISSET(fdin, &readset)){
            size_t len = read(fdin, buf, BUFSIZE);
            if(len == 0)
                break;
            write(1, buf, len);
        }
    }

    close(fdin);
    close(fdout);
}
```

Apri pipe per leggere

Apri pipe per scrivere

Se e' arrivato un dato da leggere su stdin

Leggo da stdin

Scrivo su fdout (pipe lato scrittura)

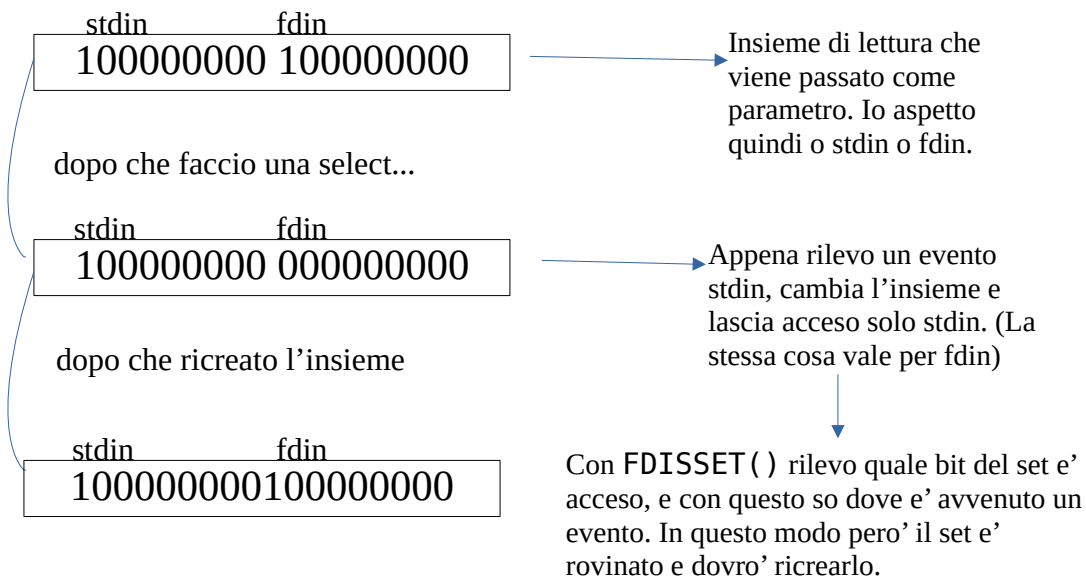
Se mi arriva qualcosa da fdin

Leggo da fdin

Scrivo su stdout (pipe lato lettura)

Questo metodo funziona, ma e' scomodo, siccome tutte le volte gli insiemi di file descriptor vengono ricalcolati dal programma, dunque non e' esattamente una soluzione scalabile (nel codice d'esempio il nostro insieme di file era molto semplice, se ne avessimo uno piu' complesso ci potrebbero essere degli slowdown).

Ecco uno schema di come funziona il programma la gestione degli eventi con `select()`:



In sostituzione alla `select()`, qualche anno dopo si e' inventata la `poll()`. Questa syscall non ha piu' gli insiemi, bensì ha un vettore di strutture `pollfd`, un numero che indica quante sono le strutture prima indica e un timeout in millisecondi. Un `pollfd` ha 3 campi: un file descriptor, uno short che indica gli eventi che vogliamo attendere, e un altro che gli eventi che sono avvenuti. Al contrario dei set di prima, gli elementi del vettore di `pollfd` (nelle loro parti `fd` e `events` si intende) non viene modificato, quindi posso ottenere il vettore già settato e usarlo per ogni iterazione. Proviamo ora a riscrivere il codice di prima usando la `poll()`:

```
int main(int argc, char *argv[])
{
    int fdin = open(argv[2], O_RDONLY | O_NONBLOCK);
    int fdout = open(argv[1], O_WRONLY);
    struct pollfd pfd[] = {{0, POLLIN | POLLHUP, 0}, {fdin, POLLIN | POLLHUP, 0}}
    char buf[BUFILENGTH];

    for(;;)
    {
        poll(pfd, sizeof(pfd)/sizeof(*pfd), -1 );
        if(pfd[0].revents & POLLHUP) return 0;
        if(pfd[1].revents & POLLHUP) return 0;

        if(pfd[0].revents & POLLIN){
            size_t len = read(0, buf, BUFILENGTH);
            if(len == 0)
                break;
            write(fdout, buf, len);
        }

        if(pfd[1].revents & POLLIN){
            size_t len = read(fdin, buf, BUFILENGTH);
            if(len == 0)
                break;
            write(1, buf, len);
        }
    }
    close(fdin);
    close(fdout);
}
```

Tipi di eventi che dobbiamo aspettarci da `fdin` e 0.
L'evento che dobbiamo aspettarci dai due file sono
POLLIN (ovvero vedere se c'è una scrittura) e
POLLHUP (ovvero se una delle due parti ha chiuso lo
stream)

Con questo truccetto di /davoli, la
lunghezza del vettore viene cambiata
automaticamente, senza dover
aggiornare manualmente l'argomento.
Con -1 si intende "aspetta per sempre".

Se `pfd[0]` negli eventi avvenuti ha il bit
POLLIN acceso, allora devo leggere lo
stdin.

Se `pfd[1]` negli eventi avvenuti ha il bit
POLLIN acceso, allora devo leggere lo `fdin`
(perché vuol dire che sono arrivati dei dati
da leggere).

Passiamo al prossimo capitolo, che è quello di un segnale. Con **segnale** si intende una indicazione o *evento* che avviene in maniera sincrona rispetto all'esecuzione del programma. I segnali sono molto usati per gli errori, in particolare quelli che avvengono durante la vita di un processo e dovuti a un'operazione sbagliata dell'utente (e quindi non per un errore causato da syscall, per quelli usiamo gli `errno`). Il kernel così, quando avviene un errore di questo tipo, segnala l'accaduto tramite l'uso, appunto, di segnali.

Tra questi segnali abbiamo:

- SIGHUP, che avvisa di un hang-up di un terminale
- SIGINT, che segnala un interrupt da tastiera (avviene per esempio quando si preme CTRL+C nel terminale, per uscire dal processo)
- SIGQUIT, altro modo per terminare da tastiera
- SIGILL, segnala una istruzione illegale dentro all'eseguibile.
- SIGFPE, segnala una eccezione causata da errore di floating-point (es. divisione per 0).
- SIGKILL, segnala la terminazione di un processo per mano dell'utente attraverso `kill`.
- SIGSEGV, segnala un seg-fault.
- SIGTERM, molto usato, semplicemente è il segnale di terminazione di un processo.
- SIGCHLD, segnala che un figlio ha cambiato stato (usato dalla `wait` per avvisare che c'è qualcosa da leggere)
- Etc...

Ad ogni segnale e' associato anche il tipo di azione di default che avviene quando si manifesta uno di questi segnali: con Term si indica che termina il processo corrente, con Core si intende che il processo termina e in piu' mi salvo un Core Dump (ovvero una immagine della memoria al momento del segnale, in modo da poter vedere con un debugger esattamente che cosa e' successo), con Ign si intende che essenzialmente si fa finta che non sia successo nulla (cio' succede con SIGCHLD per esempio), con Stop blocca l'esecuzione del processo, con Cont che lo faccio ripartire.

Le azioni di default possono essere modificate (tranne per il segnale 9), quindi essenzialmente l'utente puo' dire di per esempio di eseguire una certa funzione quando appare un certo segnale (volendo posso far proseguire anche se ci sono sefault o divisioni per 0 ommioddio o.o).

Possiamo dire che il segnale, a livello di processo, ha una funzionalita' simile a cio' che avviene a livello di processore quando avviene un interrupt (anche se sono due concetti totalmente differenti), e come infatti se avessimo creato sopra all'O.S. un meccanismo che permette di interrompere il processo in esecuzione.

(IMPORTANTE: interrupt ed eccezioni sono sinonimi!!, ma ciò non vuol dire che le eccezioni in programmazione sono considerate trap o interrupt! [aggiunto dalla lezione del 26/02] Try-catch non c'entra assolutamente nulla con il concetto di interrupt! Bensì è solo una struttura del linguaggio di programmazione per gestire degli errori nell'esecuzione di un pezzo di codice)

Per implementare queste funzionalita' a livello codice, possiamo fare uso delle syscall `kill()` e `signal()`. Il nome di queste syscall pero' e' fuorviante: la syscall `kill()` infatti NON uccide un processo e `signal()` non serve per mandare un segnale. Al contrario, la `kill()` serve per mandare un segnale qualsiasi (si ok, anche per uccidere un processo, ma in generale si usa per mandare un segnale qualsiasi), e la `signal()` non serve per mandare un segnale, ma per definire il gestore del segnale; infatti come parametri ha il segnale stesso e il gestore del segnale (signal handler).

```
#include <stdio.h>
#include <signal.h>
```

```
typedef void strlc_handler(int signo){
    printf("PRRR \n")
}
```

Ci fa una pernacchia se premi CTRL+C. (E se non riesci a passare l'esame)

```
int main(int argc, char *argv[]){
    int c;
    signal(SIGINT, strlc_handler)
    while ((c=getchar()) != EOF){
        putchar(c);
    }
}
// si puo sempre chiudere con ctrl + d
```

Signal essenzialmente dice "quando avviene un certo segnale, chiama quell'handler".
Il segnale che ci interessa e' il SIGINT, che corrisponde all'interrupt da tastiera (come scritto prima).

CTRL+D (^D) corrisponde al carattere EOF. Inoltre mostra i caratteri diversi da EOF sul terminale.

Lezione del 18/11/2020 - Lettori/Scrittori, Monitor

Riprendiamo il problema dei lettori/scrittori.

processo Reader:

```
while(1):
    startread()
    read()
    endread()
```

process Writer:

```
while (1):
    startwrite()
    write()
    endwrite()
```

Si ricorda che in questo problema, posso avere uno scrittore alla volta o piu' lettori alla volta. In piu', lettori e scrittori non si possono mettere assieme. Le funzioni che dovremo implementare saranno:

nr=nw=0

Lo stato sara' definito dal numero dei lettori e dal numero degli scrittori (ci serve per l'invariante).

startread() -> <ASPETTA nw==0 -> nr++>

endread() -> <nr-->

startwrite() -> <ASPETTA nr==0 && nw==0 -> nw++>

endwrite() -> <nw-->

/** IMPLEMENTAZIONE **/

nr=nw=0

wr=ww=0

sem sr(0), sw(0), mutex(1)

ww e wr sono il numero di scrittori e di lettori rispettivamente che sono in attesa. (Sono quelli che nello pseudocodice chiamavamo contatore_cond)

startread:

mutex.P()

if (nw!=0):

wr++; mutex.V(); sr.P(); wr--;

nr++

CAMBIO_STATO

endread:

mutex.P()

nr--

CAMBIO_STATO

startwrite:

mutex.P()

if(!(nw==0 && nw==0)):

ww++; mutex.V(); sw.P(); ww--;

nw++

CAMBIO_STATO

endwrite:

mutex.P()

nw--

CAMBIO_STATO

In questa funzione o attivo uno dei processi in attesa, oppure lascio la mutex.

CAMBIO_STATO:

if (nw==0 && wr>0) sr.V()

else if ((nr==0 && nw==0) && ww>0) sw.V()

else mutex.V()

Corrispondo alle cond_i e contatore_condizione del codice campione dei lettori/scrittori.

Questa e' una soluzione con starvation, in particolare nel caso in cui continuino ad arrivare costantemente dei lettori!

Proviamo a capire perche' eseguendo il codice in sequenza: entra il primo lettore, ed nw e' 0. nr++ incrementa ed entra in CAMBIO_STATO. Siccome non ho processi in attesa, rilascio la mutex. Arriva un secondo lettore che fara' la stessa identica cosa del primo. Intanto, il primo se ne va facendo un endread, e si prende e poi rilascia la mutex. In tutto questo, il numero dei lettori non e' mai andato a 0. Bastera' quindi fare una prima startread e poi un continuo di startread/endread in modo da non portare mai nr a 0. Gli scrittori cosi' entreranno in starvation siccome non potranno mai entrare a lavorare.

Proviamo dunque a risolvere il problema della starvation, usando un approccio basato sul “spegnere il cervello” e procedere in modo meccanico.

Sappiamo che se eseguiamo la **startread** fino a esattamente prima della chiamata di **CAMBIO_STATO**, siamo certi che **nr** sia maggiore di 0 e che **nw** sia uguale a 0 (questo è vero anche se ci sono 10000 scrittori e 1000 lettori che hanno eseguito questo codice 10000 volte per esempio).

L'unica condizione che potremo avere sarà quella di avere più lettori in attesa.

```
startread:
    mutex.P()
    if (nw != 0):
        wr++; mutex.V(); sr.P(); wr--;
    nr++
    if (wr > 0) sr.V() else mutex.V()
```

Condizione aggiunta al posto di **CAMBIO_STATO**.
Se ho un lettore in attesa, passo la mutex al primo.

In **endread**, siamo sicuri che **nw** sia 0 perché è appena uscito un lettore, e se l'invariante è verificato vuol dire che non c'era nessun scrittore, quindi si potrebbe pensare di fare qualcosa del genere:

```
endread:
    mutex.P()
    nr--
    if (wr > 0) sr.V()
    else if (nr == 0 && ww > 0) sw.V()
    else mutex.V()
```

Questa condizione volendo si potrebbe anche togliere, perché se il numero di scrittori era 0 i lettori sarebbero comunque entrati a loro tempo, se non ci sono scrittori quando un lettore tenta di entrare, lui entra. (nella **startread**, infatti, abbiamo già una condizione che testa se $nw > 0$, e se è $ww > 0$ allora anche $nw > 0$)

Questi sono i 3 casi in cui potremmo trovarci se entriamo in **endread**.

Iniziamo ora a scrivere il codice degli scrittori. Nella funzione **CAMBIO_STATO**, nessuna delle casistiche corrisponde a una situazione in cui potremmo trovarci in **startwrite**. Dunque l'unica soluzione dopo aver svolto le modifiche alle variabili globali sarà uscire dal mutex (ricordiamo che il numero degli scrittori può essere solo di uno alla volta, non posso passare quindi il testimone a un altro scrittore).

```
startwrite:
    mutex.P()
    if (!(nw == 0 && nw == 0)):
        ww++; mutex.V(); sw.P(); ww--;
    nw++
    mutex.V();
```

Nell'**endwrite**, sappiamo sicuramente che **nw** è uguale 0 (siccome posso avere un solo scrittore alla volta), dunque:

```
endwrite:
    mutex.P()
    nw--
    if (wr > 0) sr.V()
    else if (ww > 0) sw.V()
    else mutex.V()
```

Sappiamo già che il numero dei lettori è uguale a 0 grazie all'invariante del problema.

Per quanto possa sembrare che questi cambiamenti possano risolvere il problema, non lo risolvono assolutamente, anzi peggiorano la situazione. Vediamo il perché di questo. Poniamo che ci sia il problema di prima, con i primi due lettori che arrivano e un terzo lettore che arriva subito dopo che il primo ha finito la lettura (quindi non ci sono mai, appunto, zero lettori): anche questa volta gli scrittori non entrano mai. Inoltre, con questo codice, nel caso in cui entri uno scrittore e nel mentre si mette in coda un secondo scrittore, il primo scrittore quando esce manderà avanti il secondo scrittore, e se arriva un terzo scrittore che si mette in coda, appena il secondo finisce la scrittura manderà avanti il terzo e così via, creando una starvation anche dei lettori! Mio dio che casino che abbiamo creato :(

Quindi, come possiamo fare per risolvere?

Semplicemente, come direbbe il nostro amico Coen, dobbiamo accendere il cervello. Non abbiamo più modo infatti di procedere meccanicamente, ma dovremo usare la fantasia.

Questo dimostra quindi che, come abbiamo visto nella lezione della settimana scorsa, non possiamo usare il “template” del problema dei lettori/scrittori in modo mindless, in quanto appunto ci sarà una parte (quella del cambio di stato) che richiederà di usare il cervello.

Dunque, tornando alla risoluzione del nostro problema: in questo tipo di esercizi, la cosa migliore da fare è trovare delle soluzioni che si basano sul procedere a turni, in cui quindi andranno prima tutti i lettori, poi gli scrittori in attesa etc.

Analizziamo la starvation causata dai lettori: il problema sta nel fatto che se ho uno scrittore in attesa, i lettori “se ne fregano” e continuano ad entrare ed uscire ignorandolo. Dovrei trovare un modo per impedire ai lettori di entrare in attesa appena c’è uno scrittore in attesa, dunque i lettori che sono in attesa finiranno di leggere, ma poi manderanno avanti il nostro scrittore che sarà ormai impaziente.

```
startread:
    mutex.P()
    if (nw > 0 || ww > 0):
        wr++; mutex.V(); sr.P(); wr--;
    nr++
    if (wr > 0) sr.V() else mutex.V()
```

Se abbiamo scrittori in attesa, allora faccio finire i lettori in attesa rimasti e blocco il resto (tutto grazie alla sr.P())

Abbiamo così risolto uno dei due problemi della nostra soluzione iniziale. Dobbiamo risolvere l’altro. Nella versione precedente del nostro codice, lo scrittore in uscita controlla se ci sono dei lettori e nel caso in cui ci siano risveglia il primo lettore. Altrimenti, se non ci sono lettori, può andare avanti un altro scrittore.

Se mentre lo scrittore scrive arrivano 10 lettori, si fermeranno tutti nella sr.P() della startread (com’è giusto che sia). Quando l’unico scrittore esce, entrerà nel primo if (la cui condizione è $wr > 0$, che è vera siccome ne abbiamo 10) e risveglierà il primo dei lettori. Il primo scrittore poi risveglierà tutti gli altri (grazie al secondo if della startread). Se continuano ad arrivare lettori e scrittori faranno a turni, in modo che abbiamo uno scrittore che scrive e subito dopo tutti gli altri lettori leggeranno.

Come facciamo a risolvere questo problema? Beh, piccolo spoiler: se hai letto bene il testo di prima ti accorgerai che corrisponde esattamente a ciò che il nostro programma dovrebbe effettivamente fare, e quindi non c’è più alcun problema. Aggiungendo quell’OR nella condizione dello startread abbiamo davvero risolto tutto, sistemando il codice della nostra funzione che ora è 100% funzionante senza starvation.

I semafori però sono davvero scomodi da usare, e volendo si potrebbe anche fare di meglio. Infatti, ha forti problemi di leggibilità. A livello utente quindi, riusciamo a fare di meglio? La risposta è sì.

Uno potrebbe pensare che quindi tutto quello che abbiamo fatto finora sia inutile, ma in realtà bisogna pensare che tutto ciò che vogliamo effettivamente usare va costruito (e’ anche il motivo per cui abbiamo studiato la TS, perché appunto si usa per costruire i nostri semafori).

Possiamo pensare di avere una protezione simile a quella che si può trovare nel concetto di programmazione oggetti, in cui ho variabili private che sono visibili solo all’interno della classe stessa, ed accessibili (e modificabili) solo grazie a dei metodi appositi che offre la classe. Il paradigma che ci permette di fare esattamente ciò in programmazione concorrente è il **monitor**.

Il monitor ci permette di implementare un critical section attraverso metodi che sono accessibili solo da un processo alla volta (senza eccezioni) e risolvendoci il problema della race condition.

[**META-NOTA**: poco importante...]

Tuttavia, permettere esclusivamente la mutua esclusione non basta per risolvere i problemi del buffer limitato, p/c, cena dei filosofi etc... Abbiamo bisogno di qualcosa che permetta a un processo di indicare ad un altro processo che nel nuovo stato si può fare una certa operazione.

Nel metodo “generico” visto poco prima sulla risoluzione dei problemi di programmazione concorrente, possiamo notare che ci sono due caratteristiche principali:

- Lo stato interno, che deve essere sempre acceduto in mutua esclusione
- Ci sono meccanismi per indicare agli altri processi che il nuovo stato consente di attivare qualcuno che era in attesa.

Il monitor assicura solo la prima, dobbiamo invece trovare un modo per implementare anche la seconda caratteristica.

[fine sez. poco importante]

Il monitor prevede:

- Una funzione di inizializzazione
- Variabili locali e private
- Eventuali funzioni locali e private
- Funzioni accessibili dall'esterno, cioè pubbliche, identificate (nel nostro pseudocodice) dalla keyword **entry**. Ognuna di queste funzioni, quando invocata, rispetterà automaticamente e senza la necessità di ulteriori costrutti la **mutua esclusione** tra processi.

Nella definizione di monitor intervengono inoltre delle cosiddette **variabili di condizione** (identificate, nella sintassi del nostro pseudocodice, dalla keyword **condition**) cioè oggetti muniti delle seguenti due operazioni:

- **wait()**: chiamata sempre bloccante che rilascia la mutua esclusione sul monitor ed inserisce il processo invocante in una coda, in attesa di una chiamata di tipo **signal()** sulla stessa variabile.
- **signal()**: che comporta la riattivazione immediata di un processo (secondo una politica FIFO) in attesa sulla variabile, ponendo il processo invocante all'interno di una struttura dati detta **urgent stack** in attesa che quello risvegliato completi la sua operazione. **signal()** non ha alcun effetto se nessun processo è in attesa sulla variabile di condizione.

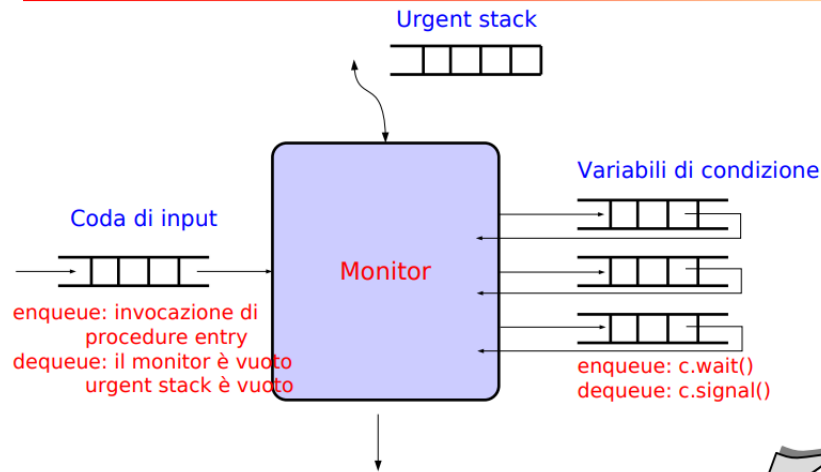
Facciamo un esempio. Se ho una variabile di condizione **c**, allora con **c.wait()** si intenderà che si vuole aspettare una certa condizione **c**. Se un processo eseguirà questa istruzione, sarà bloccato in essa. Con **c.signal()**, si intende che quando la condizione è vera, allora liberiamo il primo processo bloccato in **c.wait()**. Essenzialmente quindi segnala che una certa condizione è vera.

Semaphore vs condition:

- wait sempre bloccante nei monitor --- mentre la P lo è solo se val == 0.
- signal viene persa se nessuno aspetta --- mentre la V “ricorda”.
- Signal (urgent) il processo riattivato parte prima che altre variazioni possano avere luogo nelle var del monitor.

Ora che conosciamo come è strutturato un monitor, abbiamo un problema da risolvere: consideriamo di avere due processi, uno che aspetta in **c.wait()** e uno che esegue la **c.signal()**; se facessimo partire quello che aspetta la condizione e concorrentemente lasciassimo andare avanti quello che ha fatto la signal, sarebbe davvero un bel casino siccome avremmo due processi che stanno avanzando all'interno del monitor, e avremmo quindi violato la definizione di monitor. Per questo motivo esiste la **signal urgent**, che consiste nel sospendere il processo che ha fatto la signal (solo in caso ci siano effettivamente processi bloccati in **c.wait()**) e far andare avanti il primo processo nella **c.wait()**; quando quest'ultimo sarà uscito dalla wait avrà due possibilità: uscire dal monitor oppure bloccarsi in un'altra condizione, e in quest'ultimo caso il processo che inizialmente aveva fatto la **signal()** viene poi riattivato, tramite una **pop()** nello **urgent stack**.

Ecco uno schema di funzionamento del monitor:



Per spiegare questo schema potremmo usare la metafora di degli studenti che salgono su un piedistallo per scrivere su una lavagna. Se abbiamo un solo studente, allora questo scrive sulla lavagna ed esce. Se invece uno studente vuole anche lui scrivere mentre scrivo io, aspetta nella **coda di input**, aspettando che finisca, senza variabili di condizione. Chi deve aspettare una certa condizione prima di scrivere qualcosa entra nella coda di una delle variabili di condizione, e quando si mette in coda rende il monitor libero; dunque un altro processo potrà così entrare nel monitor. Appena uno degli studenti segnala che la condizione si è *verificata* (ovvero il processo ha fatto la signal), se la coda della sua variabile di condizione di cui ha fatto la signal è vuota allora non succede nulla [METANOTA: ho abbandonato l'uso della metafora fatta dal prof siccome la ritenevo poco pratica (e poco dopo l'ha abbandonata pure lui)], altrimenti il processo che ha fatto la signal va nello **urgent stack** (di cui abbiamo parlato prima nella definizione di monitor). In questo modo il processo in attesa nella queue della condizione entrerà nel monitor. A questo punto avremo due casi:

- *Caso semplice*: il processo che è appena stato risvegliato esce dal monitor, e il processo che ha fatto la signal (e che si trovava nella urgent stack) viene rimesso nel monitor e continua il suo corso.
- *Caso complesso*: il processo che è appena stato risvegliato potrebbe anche lui fare una signal (sia della sua stessa condizione che di altre) e, se avrò un processo in attesa nella condizione di cui ha fatto la signal, il processo ora in esecuzione andrà nella urgent stack.

In cima allo stack avrò quindi questo processo di cui stavamo parlando, e subito sotto di esso il processo che lo ha risvegliato (ovvero il primissimo processo, quello che ha iniziato tutto) permettendogli di creare sto casino enorme.

Quando non si possono più fare signal perché qualcuno si blocca sulla wait o lascia il monitor, viene preso il processo in cima dello stack e messo nel monitor.

Il monitor ammette nuovi elementi in input solo quando lo urgent stack sarà vuoto, altrimenti questo avrà la precedenza sulla coda di input.

Proviamo ora ad implementare i semafori attraverso i monitor.

```
monitor semaphore:
    int value
    condition ok2p // condizione: value > 0
```

```
def semaphore(x):
    value = x
```

```
entry P():
    if (value <= 0):
        ok2p.wait()
    value--
```

```
entry V():
    value ++
    ok2p.signal();
```

Il fatto che sia un monitor ci garantisce già la mutua esclusione nei metodi P() e V() e nel costruttore. Non abbiamo dunque bisogno di definirla ogni volta.

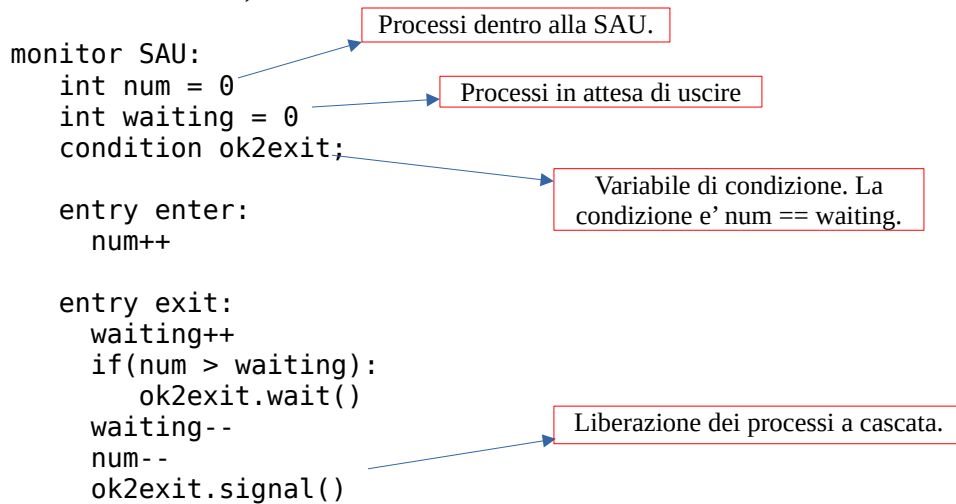
Posso fare la P() del semaforo solo se il suo valore è > 0.
Questa condition ci dice esattamente questo.

Se riesco a fare value--, son sicuro che la condizione di ok2p fosse vera (subito prima di fare value--).

Testiamo se questa implementazione funziona facendo finta di eseguire mentalmente il codice con 3 processi A,B,C e un semaforo mutex(1) che permette la mutua esclusione. Supponiamo che ciascuno di questi 3 processi voglia accedere alla mutua esclusione. Quando A chiama `mutex.P()`, la prima cosa che fa e' testare se `value` non e' maggiore di 0. Tuttavia, il valore del semaforo e' maggiore di 0 e quindi l'unica cosa che succede e' che il valore del semaforo scende a 0. B poi fara' anch'esso `mutex.P()`, e si blocchera' siccome `value` e' 0, dunque sara' messo nella coda della variabile di condizione `ok2p`. Stessa cosa accadrà con C, dunque nella coda di `ok2p` ci saranno sia B che C. Quando A fara' `mutex.V()`, `value` andra' ad 1. Chiamando `ok2p.signal()`, A va nello `urgent stack` e B si sblocchera', uscendo dalla coda e calando il valore di `value` a 0. B uscirà poi dal monitor, ed A potrà ripartire uscendo anche lui dal monitor dopo di B. C rimarra' bloccato nella `wait()` finche' un processo non eseguirà `semaphore.V()`.

Il fatto che possiamo implementare i semafori attraverso l'uso dei monitor ci dice che noi possiamo risolvere coi monitor tutti i problemi risolvibili coi semafori.

Proviamo ora a fare un altro esercizio coi monitor, ovvero quello della Sezione Acritica Unificante (SAU). In questa sezione, ogni processo entra quando vuole, ma poi tutti i processi devono uscire assieme (es: uno chiede di entrare, esce. due entrano, allora potranno uscire solo dopo che entrambi chiedono di uscire).



Proviamo ad eseguire a mente questo codice. Un processo A entra e `num` va a 1, poco dopo chiede di uscire, quindi `waiting` va 1 momentaneamente, ma poco dopo sia la `num` che `waiting` vanno a 0. A esegue il `signal`, ma non c'e' nessun processo in attesa quindi esce subito dal monitor.

A poi torna dentro alla sezione e `num` torna ad 1. Subito dopo un processo B entra pure lui, e poi anche un processo C, dunque `num` sara' a 3.

Poniamo poi che B voglia uscire, quindi `waiting` va ad 1 e il processo B si blocca entrando nella coda della variabile di condizione, A pure fa la stessa cosa quindi `waiting` andra' a 2. Un processo D poi entra nella SAU e `num` va a 4, e chiama poi la `exit` e `waiting` passa a 3. La coda di `ok2exit` sara' quindi B-A-D. C poi sara' la ultima a fare la `exit`, dunque `waiting` andra' a 4, non si blocca e diminuisce la `waiting` e il `num`, cosi' entrambi andranno a 3. Fara' poi la `signal` e passera' nella `urgent stack`.

Il processo B riparte, `num` e `waiting` vanno a 2 ed andra' anche lui nella `urgent stack` siccome fara' la `signal`. Anche A potra' ripartire, e anch'esso andra' nello `urgent stack`. L'ultimo a ripartire sara' D, e siccome nella coda di `ok2exit` non c'e' nessuno, potra' uscire senza problemi. Poi esce la A, poi B e poi C.

Questi che abbiamo visto sono appunto dei `signal` in cascata di cui parlavamo prima.

Proviamo ora a risolvere il problema dei lettori/scrittori coi monitor:

```
monitor rw:
    nr=nw=0
    ww=0
    condition ok2read
    condition ok2write

startread:
    if ( nw>0 || ww>0 ) ok2read.wait()
    nr++
    ok2read.signal()

endread:
    nr--
    ok2read.signal()
    if ( nr==0 ) ok2write.signal()

startwrite:
    if ( !(nw==0 && nr ==0) ) ww++; ok2write.wait(); ww--
    nw++

endwrite:
    nw--
    ok2read.signal()
    if ( nr==0 ) ok2write.signal()
```

La condizione e' nw==0

La condizione e' nw==0 && nr==0.

Lezione del (20/11/20) – Syscall

Riprendiamo il concetto di segnale che abbiamo visto la settimana prima.

Cosa succede se mentre sto facendo funzionare di un gestore di un segnale arriva un altro segnale? E se per caso arriva un segnale mentre sto eseguendo una syscall, ad esempio quando faccio una `fork()` o una `exec()`? Queste domande hanno portato ad una grossa evoluzione nella gestione dei segnali. Oggi esistono delle interfacce molto piu' dettagliate del `signal()`, che e' considerata inaffidabile perche' non garantisce che tutti i segnali vengano effettivamente catturati. Nonostante cio', e' stata migliorata per fare in modo che non ci fossero problemi di nidificazioni di chiamata durante la gestione di un handler di un segnali.

Il modo piu' corretto per definire la gestione dei segnali oggi e' attraverso l'uso di **`sigaction()`**, che prende 3 parametri: 2 di questi sono di tipo `sigaction`, detti "action" e "oldaction" che permettono di settare le azioni da fare, leggere quali azioni sono provviste oppure fare le due cose assieme in maniera atomica, ovvero mettere una nuova azione salvando al tempo stesso anche la precedente. I primi due campi dello struct `sigaction` infatti sono mutualmente esclusivi, ovvero possiamo solo mettere il primo (l'handler) oppure il secondo (l'action). Questa e' la sintassi dello struct `sigaction`:

```
struct sigaction {
    void      (*sa_handler)(int);
    void      (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
};
```

Contiene moltissime informazioni, tra cui id del processo che ha mandato il segnale, se e' successo un errore e quale, il numero del segnale etc

Permette di mascherare i segnali, facendo vedere quali il gestore puo' vedere e quali no. Questo fa apparire la gestione dei segnali simile alla gestione degli interrupt.

Nel tempo essenzialmente e' cambiata la sintassi del **callback** (ovvero funzioni che vengono passati come parametri, queste indicate dalla freccia).

Proviamo ora a rifare il codice di `noctrlc.c` (ovvero il codice che ci permetteva di evitare di usare il `ctrl+c`) con il `sigaction`.

```

#include <stdio.h>
#include <signal.h>

typedef void strlc_handler(int signo){
    printf("PRRR \n")
}

int main(int argc, char *argv[]){
{
    int c;
    static struct sigaction act;
    act.sa_handler = strlc_handler;
    act.sa_flags = SA_RESTART;
    sigemptyset(&act.sa_mask);
    sigaction(SIGINT, &act, NULL);
    while ((c = getchar()) != EOF)
        putchar(c);
}
}

```

Usiamo questo flag per dire che se ci sono delle syscall che potrebbero essere interrotte dal nostro segnale (in questo caso la read() contenuta nel getchar()) queste devono ricominciare e non devono ritornare un errore EINTR.

(EINTR e' il codice d'errore causato dall'interruzione di una syscall) .

Se non si mette questo flag, il codice va alcune volte ed altre no, a causa dello STACK POINTER che non puo' puntare sia sul codice dell'handler che sul programma contemporaneamente (non abbiamo piu' processi ma uno singolo). A volte ritorna codice 4 (ovvero EINTR).

NULL serve a settare la new action senza tenere traccia di com'era prima.

Questa funzione serve per gestire gli insiemi di segnali, e ci serve per specificare che quando noi usiamo questo handler nessun altro possa interferire all'interno di un processo.

Per evitare che i segnali arrivino in momenti non opportuni, abbiamo tante funzioni per mettono di fare un mascheramento dei segnali, tra cui `sigprocmask()` che ritarda i segnali dopo che un sigaction viene rilasciato. Inoltre e' possibile usare una `ppoll()` che fa uso di una maschera dei segnali, e possiamo usarla per aspettare in modo sicuro che un FD sia pronto o un segnale venga catturato.

Se nel codice sopra avessimo messo due sigaction `sigaction(SIGUSR1, NULL, &act)` e `sigaction(SIGUSR1, &act, NULL)` allora potevamo fare la stessa cosa ma attraverso il comando `kill -USR1 [numero proc]`.

Facendo una piccola deviazione, come possiamo rappresentare un insieme, quindi un set, in programmazione? Potremmo usare una struttura dati non ordinata tipo un vettore o una lista, oppure un altro modo e' usando la funzione caratteristica dell'insieme. Ad esempio, se ho un insieme {a, b, c, d} e voglio rappresentare un insieme che contiene a e b, accendero' solamente i bit di a e b nella struttura dati. Un ragionamento simile si usa per le funzioni `sigemptyset()` e `sigfullset()`. Per capire meglio questo concetto, proviamo a scrivere un codice che stampa tutti gli elementi dell'insieme dei segnali:

```

#include <signal.h>
#include <stdio.h>

void printsigset(sigset_t *set) {
    int i;
    for(i=1; i<32; i++){
        printf("%d", sigismember(i,set));
        printf("\n")
    }
}

int main (int argc, char* argv[]){
    sigset_t se, sf;
    sigemptyset(&se);
    printsigset(&se)
    sigfillset(&sf)
    printsigset(&sf)
}

```

Questo codice stampa una serie di 0 e poi una serie di 1, che indicano gli elementi dei due insiemi. Se aggiungiamo un elemento all'insieme vuoto, avremo un 1 nella serie di 0.

La funzione crea un nuovo insieme vuoto di segnali. Se vorro' aggiungere un segnale in questo insieme mi bastera' usare la funzione `sigaddset(&se, [SEGNALE])` e si aggiungera' tale segnale all'insieme.

Crea un insieme di segnali pieno, che contiene essenzialmente tutti i segnali che si possono usare.

Modifichiamo ora il codice della scorreggia coi segnali modificandola in modo che possiamo vedere l'insieme dei segnali quando ne facciamo uso.

```
/*il main e le librerie sono le stessa del codice del noctrlc.c*/
```

```
void printsigset(sigset_t *set) {
    int i;
    for(i=1; i<32; i++)
        printf("%d", sigismember(i,set));
    printf("\n");
}

typedef void strlc_handler(int signo){
    sigset_t current;
    sigprocmask(SIG_BLOCK, NULL, &current);
    printsigset(&current);
    printf("PRRR %d\n", signo);
}
```

Se avessimo piu' segnali (di tipo diverso o meno che siano) che vengono catturati dal nostro handler, ci possono essere problemi nel caso in cui venissero catturati mentre stiamo modificando una struttura dati. Infatti, in questo modo per esempio si modificherebbe solo una parte di essa e questo prima che inviamo il segnale... per questo, come dicevamo prima, esiste l'errore EINTR.

Inoltre, se io mando un segnale di tipo 10 mentre sto gestendo quello di tipo 2, allora quello di tipo 10 viene accodato e aspettera' che il segnale di tipo 2 finisca di essere gestito.

La syscall **pause()** non prende nessun parametro, e il suo scopo vitale e' quello fermare l'esecuzione di un programma e aspettare che un segnale qualsiasi venga emesso. Ovviamente questo e' un modo molto grezzo di lavorare, perche' magari sarebbe piu' efficiente aspettare un segnale di un certo tipo. Per questo motivo esiste un equivalente piu' raffinato, ovvero **sigsuspend()**, che prende una maschera di segnali come input e la sostituisce temporaneamente con quella del process che chiama la funzione.

Come facciamo a capire l'id di un processo che ci invia un segnale? Per capirlo, scriviamo il codice di un programma "permaloso", che uccide un qualsiasi

```
static void handler(int signo, siginfo_t *info, void *arg)
{
    printf("received %d from %d\n", signo, info->si_pid);
    kill(info->si_pid, SIGKILL);
}
```

Strage di processi
attuata

Fun fact: se faccio
ctrl+c il processo si
uccide da solo, rip.

```
int main(int argc, char *argv[])
{
    int c;
    static struct sigaction act ;
    act.sa_sigaction = handler ;
    act.sa_flags = SA_RESTART | SA_SIGINFO;
    sigfillset(&act.sa_mask);
    int signo;

    for(signo=0; signo<32; signo++);
        sigaction(signo, &act, NULL);

    for(;;)
        pause();
}
```

Sta volta usiamo
sa_sigaction al posto di
sa_sighandler siccome
e' piu' completo

Mi permette di ricevere
l'indicazione di chi ha
mandato il segnale.

Controlliamo tutti i 32
segnali

La ragione per cui possiamo usare sia `sa_sigaction` che `sa_sighandler` e' per il fatto di backward compatibility, ovvero per fare in modo che programmi vecchi possano comunque essere compilati. Se facciamo uso di una `sigaction`, dobbiamo dargli come secondo parametro uno struct `siginfo_t` che contiene numerosissimi campi utili, tra cui il numero di errore, il numero del segnale, il tempo di esecuzione etc... (si possono trovare tutte nel man ste cose). Inoltre la struttura e' anche cambiata nel tempo, ma vabbe'...

Passiamo ora a parlare della gestione degli utenti e dei gruppi in dettaglio.

Sappiamo che esistono 3 tipi di utenti (ovvero appunto utente reale, utente effettivo, utente salvato) e 3 tipi di gruppi (ovvero gruppo reale, gruppo effettivo e gruppo salvato). Normalmente non ci si accorge di questo, perche' nei casi normali in cui facciamo un programma e lo lanciamo, l'utente effettivo, reale e salvato coincidono con l'utente che ha lanciato il programma.

Le cose cambiano se abbiamo gli eseguibili di tipo **setuserid**, che come abbiamo visto permettono di cambiare momentaneamente l'utente a root attraverso la loro esecuzione. Questi sono creabili solo da root o da un utente con le capacilta' necessarie a farlo.

Proviamo a scrivere un programma che ci permette l'id di un utente di vedere il suo user id.

uid.c

```
#include <stdio.h>
int main (int argc, char*argv[])
{
    printf("%d %d\n", getuid(), geteuid())
    printf("%d %d\n", getuid(), getegid())
}
```

Stampa l'utente reale e quello effettivo.

Usando il comando `chown 6755 uid` (dove uid e' programma che abbiamo appena scritto) possiamo mettere il permesso `setuserid` (quindi il bit "s" nella stringa dei permessi) al programma uid appena scritto.

Eseguendo il programma noteremo che ora che mostrera come user effettivo 0, e gruppo effettivo 29.

Notiamo subito che un programma del genere potrebbe dare problemi di sicurezza, infatti se magari un programma del genere (ovvero con `setuserid`), avesse tra gli user parameters un pathname di un file da leggere, potrei leggere file che un utente normale non potrebbe normalmente leggere (privilege escalation, o semplicemente disclosure siccome stiamo leggendo dei file che non dovremmo). Per risolvere questo, si potrebbe pensare che basterebbe fare una semplice verifica per testare se l'utente reale puo' accedere al file, ma cio' comunque non risolve il problema in quanto se separiamo i momenti in cui controlliamo dal momento in cui facciamo effettivamente l'azione, qualcuno potrebbe cambiarci le carte in tavola nel momento in cui stiamo facendo il controllo dei permessi.

Dunque, l'unico modo per risolvere questo e' facendo un trick epico e facendo in modo che l'azione venga fatta obbligatoriamente dall'utente reale, e dopo questo vogliamo tornare ad essere root.

Per fare tutto cio', ci basta usare la syscall `seteuid()`. Ecco qui un esempio:

```
#include <stdio.h>
#include <stdio.h>
int main (int argc, char*argv[])
{
    uid_e effective = geteuid();
    seteuid(getuid())
    printf("%d %d\n", getuid(), geteuid())
    // accedo al file
    seteuid(effective)
    printf("%d %d\n", getuid(), geteuid())
}
```

Salvo l'utente effettivo (che e' anche quello piu' potente in questo caso)

Setta l'utente effettivo a quello reale

Setta l'utente effettivo a quello effettivo iniziale

N.B: ogni volta che si sovrascrive un programma dopo la sua ricompilazione, non si mantengono i bit permessi. Dunque, bisognerà resettarli e riautorizzare il programma.

Per visualizzare le statistiche e le informazioni su un processo in esecuzione sulla shell, basta usare il comando `cat /proc/$$/status` (il prof ha fatto questo comando in root, ma non so se sia effettivamente necessario)

Torniamo però a sta storia della gestione degli utenti: esistono delle chiamate più moderne (ma non standard, secondo il prof infatti non vanno su BSD) che fanno cose simili a quello che abbiamo appena visto, che permettono di leggere e settare l'utente reale, effettivo o salvato. Questi sono `setresuid()` (che sta per set utente reale, effettivo e salvato) e il suo equivalente per i gruppi `setresgid()`. Per ragioni di sicurezza, quando root fa una di queste operazioni, cambia tutti e 3 gli utenti in modo che il processo non possa più tornare a root così'.

Un utente può appartenere a tanti gruppi. Per sapere a quali gruppi un utente appartiene c'è la syscall `getgroups()`. Questa vuole come parametri l'ampiezza e un vettore di `gid_t` nel quale scriverà le informazioni sul gruppo al quale l'utente che esegue tale syscall appartiene. L'utente root (o anche chi ha permessi necessari) inoltre può settare l'insieme dei gruppi attraverso `setgroups()`.

groups.c

```
#include <stdio.h>
#include <gr.h>
int main (int argc, char*argv[])
{
    int size = getgroups(0, NULL);
    gid_t gid_list[size];
    getgroups(size, gid_list);
    int i;
    for(i = 0; i < size; i++)
    {
        struct group *g = getgrgid(gid_list[i]);
        printf("%d -> %s\n", gid_list[i], g->gr_name);
    }
}
```

Settando il primo parametro a 0, la funzione ritorna il numero dei gruppi.

Inserisce nell'array di `gid_t` le info su ciascun gruppo al quale appartiene l'utente che ha eseguito il programma.

Funzione di libreria scandisce le informazioni nell'array di `gid_t` altre info aggiuntive.

Siccome `getgrgid()` restituisce un puntatore, e quindi una quantità di memoria allocata dentro al main, queste funzioni non sono **reentrant**, ovvero non possono essere usate nel multithreading. Se usiamo l'equivalente di tale funzione con il suffisso "_r" (`getgrgid_r()`), allora questa prenderà un altro parametro corrispondente a un buffer che vogliamo utilizzare e potremo utilizzarla nel multithreading.

ERRORI GRAVI SUI SEMAFORI:

- Fare una P() dentro una mutex è un erroraccio, che causa deadlock!!
- Una P() NON DEVE MAI avere parametri.

Lezione del (25/11/2020) – Risoluzione di problemi di concorrenza con i monitor

[Prima di questi punti, ha fatto un leggero ripasso generale dei semafori e dei monitor.]

Proviamo a risolvere il problema dei **produttori consumatori** attraverso l'uso dei monitor.

```
monitor pc:
    T buffer
    int full = 0
    condititon ok2read // full == 1
    condititon ok2write // full == 0

    entry add(T x)
        if full != 0 ok2write.wait()
        buffer = x: full = 1
        ok2read.signal()

    entry get():
        if full != 1 ok2read.wait()
        z = buffer; full = 0
        ok2write.signal()
        return z
```

Nella zona subito dopo l'if siamo sicuri che il valore di full sia = 0, o perché lo era già quando abbiamo chiamato l'add, oppure perché abbiamo ricevuto una segnalazione di ok2write, e quindi ci è stato segnalato che full fosse uguale a 0.

Qui è necessario invece che il buffer "full" sia pieno, a questo punto possiamo leggere e dichiarare l'elemento come vuoto, consumandolo.

Se avessimo scritto una `get()` del genere, il nostro approccio sarebbe stato errato:

ATT.ne: QUESTO CHE SEGUE E' ERRATO! -> possibile sovrascrittura (**)

```
entry get():
    if full != 1 ok2read.wait()
    full = 0
    ok2write.signal()
    return buffer
```

Appena riattiviamo un processo con signal, questo potrebbe modificarlo. Dunque, è necessario prima salvare il buffer in una variabile, e poi ritornarlo.

Proviamo ora a risolvere il problema del **bounded buffer** (ovvero del buffer limitato) coi monitor.

```
#define BUFSIZE
```

```
monitor bb:
    queue of T q
    condititon ok2read // q.lenght() > 0
    condititon ok2write // q.lenght() < BUFSIZE

    entry add(T x)
        if q.lenght() >= BUFSIZE ok2write.wait()
        q.enqueue(x)
        ok2read.signal()

    entry get():
        if q.lenght() == 0 ok2read.wait()
        z = q.dequeue()
        ok2write.signal()
        return z
```

Come si può notare, abbiamo cambiato poche cose dal codice precedente.

Proviamo a risolvere il **problema dei filosofi** attraverso i monitor.


```
process philo(i) i = 0,..,4:
    think
    dp.starteat(i)
    eat
    dp.endeat(i)
```

```
monitor dp:
    bool chopstick[5] // true = in_use
    condition ok2get[5] // chopstick[i] == 0

    entry starteat(i) {
        if (chopstick[i] == true) ok2get[i].wait()
        chopstick[i] = true //(**)
        if (chopstick[(i+1) % 5] == true) ok2get[(i+1) % 5].wait()
        chopstick[(i+1) % 5] = true
    }

    entry endeat(i) {
        chopstick[i] = false;
        ok2get[i].signal();
        chopstick[(i+1) % 5] = false;
        ok2get[(i+1) % 5].signal();
    }
```

Prendo prima la
bacchetta a destra...

...e poi se riesco
quella a sinistra

Se usavamo i semafori, in questo punto c'era context switch, ovvero il processo in esecuzione poteva cambiare... tuttavia, usando i monitor, la mutex NON viene MAI RILASCIATA finché non si esce dalla funzione entry.

Questa soluzione ha deadlock? Per verificare ciò, esaminiamo un caso particolare.

Il filosofo 0 prende il chopstick 0, però il chopstick 1 è occupato. Se il chopstick 1 è occupato, ciò vuol dire che sicuramente il filosofo 1 se l'è preso. In più, se non lo rilascia, vuol dire che prima ancora che il filosofo 2 si è preso il chopstick 1, e che a sua volta se anch'esso è occupato, vuol dire che il filosofo 3 si è preso il chopstick 2, che se si è bloccato, vuol dire che non ha preso il chopstick 3, che invece è stato preso dal filosofo 4.

[METANOTA: devo controllare il problema vecchio risolto coi semafori per verificare se questa ultima frase è giusta (che non mi sembra ma vabb)] Normalmente il processo 4, quando andava a prendere 0, se lo trovava occupato. Ma siccome siamo coi monitor, NON potrà mai trovare lo 0 occupato, siccome il monitor garantisce mutua esclusione. Infatti, fra prendere 4 e prendere 0, il filosofo 4 non rilascia mai la mutua esclusione.

Proviamo ora a risolvere lo stesso problema cambiando soluzione.

```
monitor dp:
    bool eating[5]
    condition ok2eat[5] // eating[(i - 1) % 5] == false
                        // && eating[(i + 1) % 5] == false

    entry starteat(i)
    {
        if eating[(i - 1) % 5] == true || eating[(i + 1) % 5] == true:
            ok2eat[i].wait()

        eating[i] = true
    }

    entry endeat(i) {
        eating[i] = false
        if (eating[(i-2) % 5] == false) ok2eat[(i - 1) % 5].signal()
        if (eating[(i+2) % 5] == false) ok2eat[(i + 1) % 5].signal()
    }
```

"eating" indica che un
filosofo sta mangiando.

Questi if sono quelli che
impediscono alla nostra
soluzione di essere
ottimale.

N.B.: quando usiamo i mod (ovvero l'operatore del resto) sarebbe meglio fra in modo che si usino in questo modo, siccome alcuni compilatori hanno problemi con i mod con i numeri negativi:

$(i-1) \% 5 \rightarrow (i+4) \% 5$ ----- $(i-2) \% 5 \rightarrow (i+3) \% 5$ (ovvero, al posto di usare $i-1$ oppure $i-2$, usiamo il corrispondente col +)

Torniamo al nostro problema dei filosofi. In questo codice c'è un problema fondamentale. Per capirlo, facciamo un esempio:

Filosofo 0	Filosofo 1	Filosofo 2	Filosofo 3	Filosofo 4	Azione svolta
x					Inizia a mang. 0
x		x			Inizia 2
x					Fine 2
x			x		Inizia 3
			x		Finisce 0
	x		x		Inizia 1
			x		Finisce 1
x			x		Finisce 0

Come si può notare, il 4 non mangia mai. Questa soluzione quindi genera starvation.

Proviamo ora a risolvere il problema dei **lettori-scrittori** coi monitor.

```
reader: rw.startread(); READ; rw.endread()
writer: rw.startwrite(); WRITE; rw.endwrite()
```

```
monitor rw:
  int nr, nw, ww
  condition ok2read // nw == 0
  condition ok2write // nr == 0 && nw == 0
```

```
entry startread():
  if (nw > 0 || ww > 0) ok2read.wait()
  nr++
  ok2read.signal()
```

```
entry endread():
  nr--
  if nr == 0 ok2write.signal()
```

```
entry startwrite():
  if (nr > 0 || nw > 0) ww++; ok2write.wait(); ww--
  nw++
```

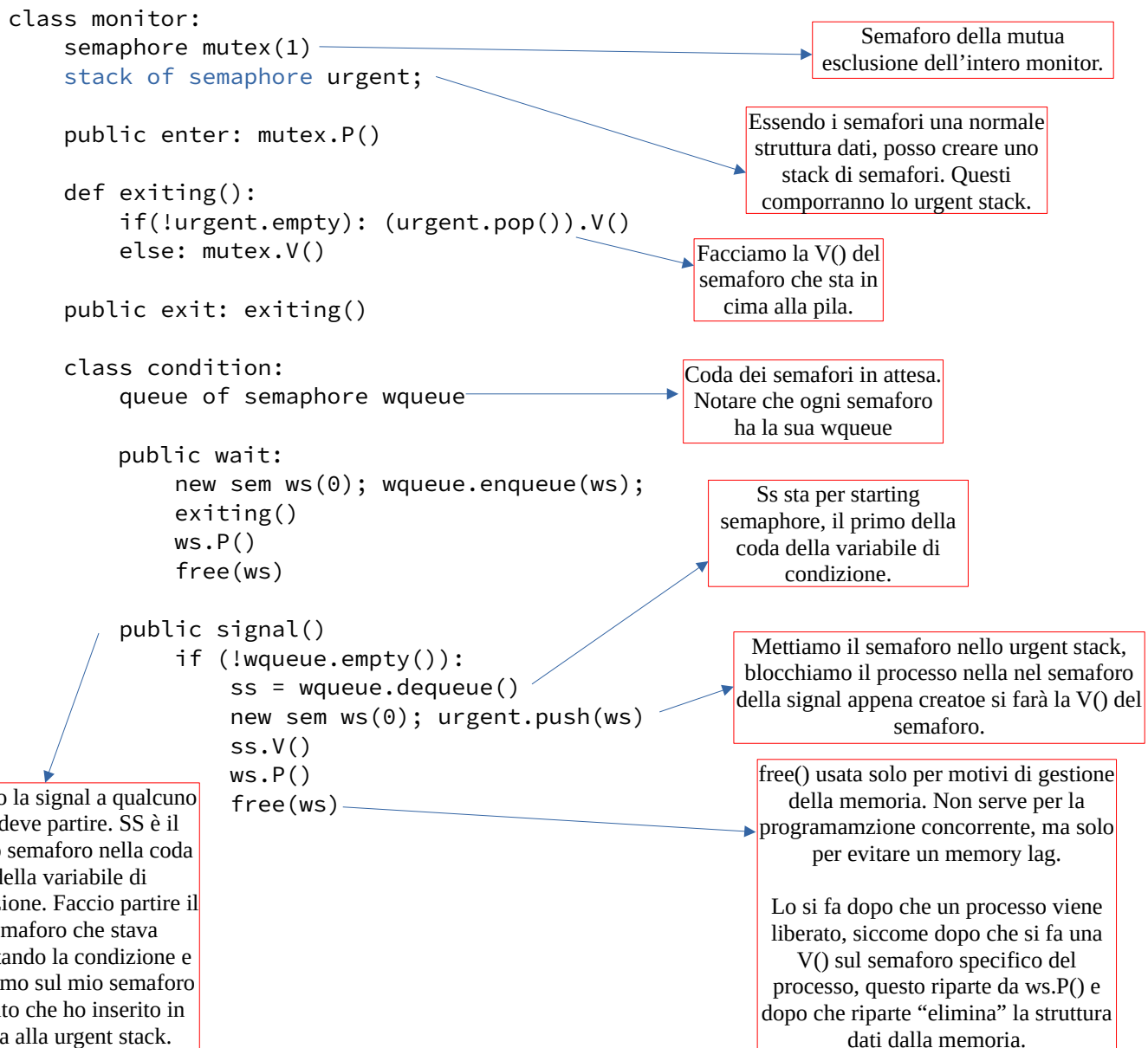
```
entry endwrite():
  nw--
  ok2read.signal()
  if (nr == 0) ok2write.signal()
```

Se arrivavano sempre lettori, e il numero di lettori non riesce mai a scendere a 0, allora gli scrittori non avrebbero scritto mai. Fortunatamente, la variabile ww risolve questo problema.

Senza questo signal, il programma sarebbe inefficiente.

Infatti, in questo modo, si riattivano a cascata tutti i lettori una volta che si ha via libera (o meglio, che un lettore ha via libera), infatti i lettori possono tutti leggere contemporaneamente il buffer, al contrario degli scrittori che DEVONO procedere una alla volta in sequenza.

Proviamo a creare una implementazione dei monitor attraverso i semafori (di nuovo), ma usando un approccio diverso dall'ultima volta.



Come si può notare, c'è un sacco di passaggio di testimone: exiting() è l'unico che può lasciare la mutua esclusione, e quando viene risvegliato qualcuno, a questo qualcuno viene passata la mutua esclusione. Infatti, si sveglia quello che stava aspettando la condizione (in ss.V()) e ci si ferma (in ws.P()). Quindi sia che uno venga risvegliato dal signal di una condition, oppure sia che uno venga svegliato dallo urgent stack (attraverso la funzione exiting()), **la mutua esclusione viene passata**. Altra cosa: come abbiamo, quindi, nelle istruzioni subito la ss.V() e subito la exiting(), ci può essere concorrenza, mentre le operazioni sullo stack e sulla waiting vengono fatte in momenti in cui non ci poteva essere concorrenza (o meglio, interferenza). Il semaforo che si crea con ws (sulla waiting queue o sullo urgent stack che sia) è un semaforo specifico per quel processo che l'ha creato, dove nessun altro si potrà fermare.

Lezione del (02/12/2020) – Message Passing e indicazioni sugli esercizi

Chiarimenti sul significato di thread e processo, e introduzione a concetti di memoria privata e condivisa

Nella visione teorica della programmazione concorrente, non esiste la parola "thread", ma solo quella di processo. A livello dei processi utente, nella visione implementativa (dunque non nella teoria), si

parla di entità depositarie delle risorse per l'esecuzione, e i thread sono i fili esecutivi (ovvero ogni percorso esecutivo che c'è all'interno del processo).

Nel caso della visione teorica, i processi compongono il modello a memoria privata o memoria condivisa:

- Se a livello implementativo ho un processo con più thread, allora il modello teorico è a **memoria condivisa**.
- Mentre se a livello implementativo ho più processi tutti con un solo thread ciascuno, allora il modello teorico sarà a **memoria privata**.

Nel caso del kernel, abbiamo una implementazione con **memoria condivisa**. Infatti, il kernel può vedere la memoria di tutti i fili conduttori.

Altri modelli a memoria condivisa sono i semafori, i monitor e i semafori binari (che come abbiamo visto, hanno tutti lo stesso potere espressivo). Test and set, anche se non ha lo stesso potere espressivo dei semafori/monitor, siccome richiede busy wait, è anch'esso un modello a memoria condivisa.

Memoria condivisa: il message passing

Quando invece abbiamo un modello a memoria condivisa, dobbiamo usare il **message passing**. Il concetto chiave del modello a memoria condivisa sta nell'uso della sincronizzazione. In questo modo, possiamo dire che la comunicazione fra i processi è data dal congiungimento della condivisione con la sincronizzazione.

Comunicazione = condivisione + sincronizzazione

Per esempio, ogni volta che noi esaminavamo problemi con comunicazione (come il bounded buffer o il producer/consumer), in realtà quello che facevamo era creare uno spazio di scambio nella memoria per il passaggio dei dati, e per fare in modo che tutto avvenga correttamente, necessitavamo di sincronizzazione.

Il concetto chiave della memoria privata è dunque la comunicazione, e la sincronizzazione stessa verrà implementata attraverso la comunicazione.

Le chiamate del message passing sono usualmente le send e receive.

Abbiamo vari meccanismi di message passing:

- **Message passing sincrono**: in questo schema, sia la send che la receive sono chiamate bloccanti (solitamente le chiamate vengono indicate con `send(msg, dest)` e `srecv(sender) → msg`).
- **Message passing asincrono**: in questo schema, la send NON è bloccante, mentre la receive è bloccante (solitamente le chiamate vengono indicate con `asend(msg, dest)` e `arecv(sender) → msg`).
- **Message passing completamente asincrono (non-blocking)**: in questo schema, sia la send che la receive sono chiamate NON-bloccanti (solitamente le chiamate vengono indicate con `nbsend(msg, dest)` e `nbrecv(sender) → msg`).

“??” come prefisso perché non sappiamo ancora il tipo di MP

Facciamo un esempio, se abbiamo Alice e Bob (nice reti reference), e Alice fa `??send(“ciao bob!”, Bob)` e Bob fa `??recv(Alice)`, data questa situazione abbiamo due casi, o arriva prima Alice, oppure arriva prima Bob. Nel caso del message passing sincrono, se arriva per prima Alice, lei si ferma e Bob fa la receive (bloccandosi anche lui), finché non riceve il messaggio, permettendo ad entrambi di sbloccarsi. Se fosse arrivato prima Bob, e quindi se avesse fatto lui per prima la send, allora Bob si sarebbe bloccato, e si sbloccherà solo quando Alice avrà mandato il messaggio.

Nel caso del message passing asincrono, Alice manda il messaggio, senza bloccarsi, e continuando quindi a fare ciò che doveva fare, e successivamente quando Bob farà la receive si bloccherà, e troverà un messaggio da parte di Alice per Bob, permettendo di sbloccarsi.

Il modello di message passing asincrono ha bisogno di una coda, siccome Alice può mandare i messaggi a Bob, a Charlie etc, e se i messaggi sono stati inviati, ma non sono ancora stati ricevuti, allora dovremo memorizzarli [Il message passing sincrono, invece, non avrà bisogno di alcun tipo di

buffer]. Quando noi memorizziamo i dati nel buffer, in caso MP async, non facciamo caso al fatto che ci possa essere una perdita di messaggi a causa del buffer pieno (simile al packet loss che avveniva nei router, ma noi non lo consideriamo, chadOS).

Nel caso del Non-Blockin, nessuno aspetta. Quindi, se Alice manda un messaggio e poi Bob fa la ricezione, allora Bob riceve il messaggio di Alice, se invece Bob fa per primo la ricezione, riceverà un NaN o un null, ovvero una segnalazione asincrona che indicherà che non c'è nessun messaggio da parte di Alice. Anche in questo caso, però, occorre un buffer.

Nel modello che stiamo studiando sono necessarie alcune assunzioni, che però non sono sempre valide nell'implementazione:

1. I messaggi **sono affidabili**, ovvero non viene mai perso nessun messaggio.
2. Quando spediamo un messaggio, lo mandiamo **a uno specifico interlocutore**, e non in broadcast o multiscast, in quanto creerebbe un problema di sincronia nel nostro modello teorico (ovviamente nell'implementazione reale, queste cose invece possono succedere).

Altra cosa da notare di questo modello è che quando si chiama una receive, il valore del parametro di mittente può essere ANY, ovvero che può ricevere un messaggio da un qualsiasi mittente. Nel receive inoltre abbiamo anche una coda FIFO, in cui i messaggi vengono bufferizzati, appunto, per non essere perduti. Tuttavia, per leggere più messaggi salvati nel buffer, dovranno chiamare più rcv.

Implementazione MP sincrono dal MP asincrono

Proviamo ad implementare MP sincrono partendo dall'asincrono. Per implementarlo, sarà necessario mandare una conferma di lettura e il mittente dovrà usare una rcv per ricevere una conferma.

Ecco una prima implementazione:

```
def ssend(msg, dest):
    asend(msg, dest)
    arecv(dest)

def srecv(sender):
    msg = arecv(sender)
    asend(ACKNOWLEDGE, sender)
    return msg
```

Il problema di questo codice è che funziona solo se mittente e destinatario sono processi ben determinati, dunque non potremmo mettere il valore "ANY" nel campo della receive, siccome sennò andrebbe nell'asend, che non accetta il valore ANY.
Non possiamo quindi accettarlo come soluzione finale.

Dunque, per risolvere il problema, mandiamo un messaggio esplicitando il mittente (nell'ssend) e in questo modo, sappiamo a chi mandare l'ack anche in caso di ANY.

```
def ssend(msg, dest):
    asend((getpid(), msg), dest)
    arecv(dest)

def srecv(sender):
    (realsndr, msg) = arecv(sender)
    asend(ACKNOWLEDGE, realsndr)
    return msg
```

Prendiamo l'id del processo, inserendolo nel msg....

...e quando lo riceviamo, lo "salviamo", usandolo come parametro per rimandare il tutto.

Notare come questa implementazione è infatti bloccante, siccome la send, nel MP asincrono, è bloccante e viene chiamata sia nel ssend che nell'srcv.

Implementazione MP asincrono dal MP sincrono (molto interessante)

Implementare qualcosa del genere sembra quasi impossibile, ma diventa più chiaro se pensiamo che possiamo creare un processo secondario, che "*smista*" i messaggi per conto del processo principale, che chiameremo **processo server**.

```
def asend(msg, dest):
    ssend((msg, myself(), dest), server)

def arecv(sender):
    ssend((NULL, sender, myself()), server)
    return srcv(server)
```

Fa la stessa cosa del processo getpid(), ovvero ritorna il proprio id.

Notare inoltre come il sender invia il messaggio al processo server.

Mandiamo al server un messaggio dicendo che siamo in waitMode (ovvero che stiamo aspettando un messaggio), che ci blocca.

process server:

```
boolean waiting[]
msg db ---- metodi: db.add(msg, mittente, destinazione)
                  db.get(mittente, destinazione)
```

Aggiungiamo una struttura dati msg database in cui aggiungiamo memorizziamo i messaggi arrivati.

while True:

```
(msg, s, d) = srecv(ANY)
```

```
if (msg == NULL):
```

```
    if ((msg = db.get(s,d)) == NULL):
```

```
        waiting[d] = s
```

```
    else:
```

```
        ssend(msg, d)
```

```
else:
```

```
    if (waiting[d] == s || waiting[d] == ANY):
```

```
        ssend(msg, d)
```

```
        waiting[d] = NULL
```

```
    else:
```

```
        db.add(msg, s, d)
```

db.get ritorna null se non c'è un messaggio in "attesa" per il processo. Inoltre, il valore del parametro destinazione può anche essere "ANY".

In caso il messaggio che arriva sia uguale a NULL, allora probabilmente sarà arrivato da un destinatario (e che per questo sta inviando un messaggio con NULL al server).

Caso in cui una rcv viene eseguita prima di una send: mettiamo che il destinatario d sta aspettando s.

Se c'è effettivamente un messaggio che stava arrivando, allora mandiamo subito al destinatario, che stava aspettando un messaggio.

Abbiamo bisogno di un vettore, con un elemento per ogni processo, che ci indica se quel processo a nostra conoscenza non sta aspettando, sta aspettando da uno specifico sender oppure abbia valore ANY se non aspetta un sender specifico. Il valore del waiting del processo è uguale a NULL se non è in attesa. [IMO il prfo si è sbagliato a definirlo come boolean].

Avendo esaminato per bene il codice, facciamo un esempio:

Parte il processo server, e appena parte, si ferma sulla srecv. Dopidiché, Alice fa una send a Bob con la chimata asend("ciao bob!!", bob), ma di fatto farà una ssend(("ciao bob!!", Alice, Bob), server). A questo punto il server riceve il messaggio, e siccome riceverà il messaggio sbloccherà anche Alice (e sé stesso), ed è qui che abbiamo l'asincronia, **Alice infatti non ha aspettato che Bob ricevesse effettivamente il messaggio**. Dunque, il server va avanti, siccome il messaggio non è NULL va nel ramo else, e siccome Bob non è nemmeno stato avviato, quindi non era in waiting, viene fatta una db.add("ciao bob!!", Alice, Bob). Arriva Bob, che vuole fare una rcv, e allora manda un messaggio ssend((NULL, Alice, Bob), server) al server, che nota che c'è un messaggio in giacenza nel database per lui, e glielo manda.

Esaminiamo invece un caso in cui ci sia un destinatario che stia attendendo un messaggio da qualcuno, in questo caso da uno qualsiasi.

Charlie quindi fa una ssend((NULL, ANY, charlie), server) (che ovviamente sarà vista da fuori come una normale arcv()). Quando arriva al server, siccome Charli non ha un messaggio da nessuno in giacenza (ovvero nel database), allora il server assegnerà waiting[charlie] = ANY.

Adesso, proviamo ad implementare il MP asincrono con MP completamente asincrono (non blocking).

```
def asend(msg, dest):  
    nbsend(msg, dest))
```

```
def arecv(sender):  
    while ((msg = nbrecv(sender)) == NULL)  
        ;  
    return msg
```

Essenzialmente, per rendere bloccante una chiamata non bloccante, facciamo *busy waiting*, ciclando una istruzione vuota.

Proviamo ad implementare il MP completamente asincrono (non blocking) attraverso MP asincrono. Per farlo, possiamo fare uso di un server come abbiamo fatto prima, ma possiamo anche non usare un processo server ed adottare un sistema più semplice e leggero.

```
def nbsend(msg, dest):  
    asend((myself(), msg), dest)
```

```
def nbrecv(sender):  
    static local db  
    asend((myself(), TAG), myself())  
    while True:  
        (msgsnd, msg) = arcv(ANY)  
        if (msgsnd == myself() && msg == TAG): break;  
        db.add(msgsnd, msg)  
    return db.get(sender)
```

Ci “auto-mandiamo” un messaggio a noi stessi. In questo modo siamo certi che la recv non possa essere bloccante.

Nel caso il messaggio arrivato sia l’automessaggio, allora esce dal ciclo di attesa.

Se arriva subito l’automessaggio, allora vuol dire che non c’erano più altri messaggi in attesa, altrimenti se non era l’automessaggio, lo mette nel “database”(che può essere anche una coda volendo).

Come abbiamo visto, per implementare il completamente asincrono con l’asincrono, occorre busy wait, di conseguenza il MP completamente asincrono ha una potenza espressiva minore del MP asincrono.

Inoltre, se vogliamo implementare MP asincrono attraverso il MP sincrono, c’è bisogno dell’aggiunta di un processo, mentre se vogliamo implementare il MP sincrono con il MP asincrono, basta solo aggiungere un semplice ACK, dunque l’asincrono ha anche maggiore potenza espressiva del MP sincrono, dunque l’MP asincrono è il paradigma con il maggiore potere espressivo dei 3.

Alcune indicazioni date dal prof per lo svolgimento degli esercizi

Nel caso del MP, in caso sia asincrono, non c’è bisogno di chiedere se è completamente asincrono se non è specificato.

Inoltre, i servizi di MP sono FIFO almeno che non sia indicato diversamente, ciò è riferito ovviamente alla ricezione/lettura dei messaggi.

Nel caso dei **monitor**, bisogna specificare più cose:

- Se c’è busy wait all’interno di un monitor, allora ci sarà deadlock.
- Le code delle variabili di condizione, lo urgent stack sono strutture private gestite dall’implementazione del paradigma e NON sono accessibili.
- L’inizializzazione di un monitor non può contenere wait e signal.
- È sicuramente errato un programma che su una variabile condizione effettua solo wait e mai signal. (viceversa la condizione è inutile).
- È sicuramente errato un programma che ha una wait come prima istruzione di ogni procedure entry, siccome saremmo in deadlock sicuramente.

Altre considerazioni generali:

- Soluzioni con semafori, monitor o message passing non devono contenere busy-wait (sono stati creati per evitarlo).

- È possibile definire strutture dati senza implementarle a patto che (1) non contengano primitive concorrenti (sincronizzazione) (2) non siano miracolose, devono poter essere implementabili quindi i parametri devono contenere tutte le informazioni necessarie per fornire i servizi richiesti. [Questo è particolarmente utile, siccome implica che possiamo usare code e hashmap, che sono quasi fondamentali.]
- È sicuramente errato un programma che usa variabili prima di assegnare loro un valore iniziale [AKA ricordatevi dell'inizializzazione].

Lezione del (09/12/2020) – Python programming 1

Lezione del (11/12/2020) – Python programming 2

Lezione del (23/02/2021) – Storia dei sistemi operativi, accenni all'architettura.

Un sistema operativo è un programma, che crea l'astrazione per gestire le risorse (es. File System) e fornisce l'astrazione per gestire i processi (che consente di eseguire i programmi).

Se una macchina deve fare una sola cosa (ovvero eseguire un solo programma) non ha bisogno di un sistema operativo.

Come si valuta un sistema operativo? In base all'efficienza, affidabilità, sicurezza etc..., questi inoltre vengono visti come i requisiti di un sistema operativo.

Alcune astrazioni possono essere gestite da processi utente (es. il file system) ma altre devono essere gestite obbligatoriamente dal kernel (es. gestione della CPU).

Storia dei sistemi operativi

Tra il 1800-1945 abbiamo personaggi come Ada Lovelace, Menambrea. Le macchine in questo periodo erano calcolatori, e non elaboratori. Ovviamente un tempo non c'era alcun sistema operativo. Il passaggio ulteriore si ha con la generazione 1. Babbage creò il primo prototipo di computer meccanico, chiamata macchina analitica.

Torniamo all'architettura degli elaboratori. Cosa è essenziale per fare un pc? È necessario un meccanismo che cerchi di perturbare un impulso ad un altro circuito. Posso quindi spegnere ed accendere un interruttore, ma non basterebbe per fare un elaboratore, avrei invece bisogno che questo ne faccia scattare altri. Per fare ciò, ho bisogno di componenti che dato un impulso elettrico permettano di cambiare lo stato della porta NAND (o NOR). Per costruire tali componenti uso dunque un **relais**. Ma qual'è il problema di usare un relay, che è uno strumento meccanico? Usura, tempo e consumo di corrente.

Un passaggio ulteriore si ha quando Edison accende una lampadina e mette una piastra metallica dall'altro lato rispetto al filamento, in più fa passare una corrente continua fra il filamento e la piastra; si accorge che ci sono nuvole di elettroni che passano dal filamento riscaldato alla piastra, da qui si ottiene l'effetto diodo, e a seconda della tensione della placca si può far passare più o meno la nuvola di elettroni. E così, abbiamo un interruttore, sottoforma però di **valvola**. Che guadagno c'è nell'usare le valvole rispetto ai relay? LA VELOCITÀ! Ma cosa viene perso? Le dimensioni, e altri due fattori: se abbiamo tantissime valvole, la probabilità che si fulmini è molto alta, in più hanno bisogno di alte tensioni per funzionare → alto consumo di energia! Colossus, la macchina fatta da Turing per decifrare l'enigma, era fatta a valvole.

Terzo passaggio: il **transistor**, ha consentito di cambiare le regole del gioco. Infatti, fa la stessa cosa di una valvola o di un relay, ma le correnti in gioco sono minime, ed è molto meno delicato rispetto alle valvole, quindi posso far funzionare sistemi d'elaborazione molto più ampi.

Perché nascono i sistemi operativi? Per capirlo bisogna studiare l'ambiente in cui questi elaboratori venivano usati. Le macchine della gen 0 e della gen 1 erano pochissime e riservate ai ricercatori, e le loro macchine facevano una cosa sola, nessuna pensava che le macchine potessero effettivamente fare più cose contemporaneamente. Col tempo però le macchine diventano sempre più piccole e vengono usate sempre di più in ambiti diversi, ma sono ancora molto costose. Perciò c'è l'esigenza di fare in modo che queste possano essere usate per più cose e che venga ridotto il tempo in cui non sono produttive.

Quarto passaggio: **schede per programmare**. Sono un BATCH, ovvero non interattive (non c'è un intervento manuale). Ogni scheda rappresenta una linea di programma. Il primo comando era FTS, c'era bisogno di un separatore per ogni tipo di dato all'altro. Col tempo, è arrivata l'esigenza di usare

ottimizzare i tempi, così', mentre si elaborava la scheda vecchia, ne veniva caricata e letta una nuova (meccanismo di SPOOL, usato anche nelle stampanti).

Il passaggio più importante però è quello successivo, con l'invenzione del **monitor**.

Il monitor viene messo in memoria e controlla il caricamento delle schede, l'elaborazione e che permette al ciclo di ricominciare una volta che tutto è finito. Però, questa memoria doveva essere sufficiente per far eseguire il più grande programma presente che uno voleva elaborare, e il monitor residente doveva aspettare i tempi di elaborazione per caricare il prossimo pacco di schede. Allora, per rendere più efficiente la cosa, avvengono due cose contemporaneamente: 1) si passa dai transistor separati a poter creare circuiti comprendenti molti transistor utilizzando i procedimenti tot. climici. Per fare un transistor occorre un blocco di silicio diviso in 3 strati che abbiamo delle impurità diverse. Il silicio è un cattivo conduttore (ma non un isolante). Il semi-conduttore si ha quando si toccano due parti, una di una impurità di un tipo, e una dell'altro, e la corrente così va in una sola direzione. Quando si toccano 3 parti, si ha un transistor. **(METANOTA)** ha poi spiegato roba sui circuiti integrati che sinceramente non mi interessava).

Come faccio con un solo processore ad eseguire più processi contemporaneamente? Andiamo ad esaminare il codice di un programma: ci sono operazioni di i/o, e programma in sé. Le prime, che sono operazione meccaniche, sono ordine di grandezze più lento di quelle di calcolo, quindi posso far finta di eseguire contemporaneamente più cose, perché quando un processo è in fase di i/o, posso fare intanto eseguire un'altra fase di un altro programma.

Questo però non è a costo 0: un processore deve avere determinate caratteristiche, deve consentire di fotografare lo stato del processore in un punto del programma e ricaricarlo in un altro. In più, i programmi che funzionavano nei microcontrollori (per esempio delle lavatrici) non si danno tante regole, ma accendono direttamente i device, quindi occorre che tutte le operazioni di i/o non vengano scritte dai programmi come operazioni dirette, ma come richieste al sistema operativo. L'altro problema è che questo concetto funziona se abbiamo dei programmi che fanno abbastanza spesso i/o, se io scrivo un programma che fa una elaborazione enorme, blocca il sistema. Ancora oggi esistono diversi tipi di programmi detti **i/o bound** (o **cpu bound**), ovvero che fanno tantissimo i/o e poco calcolo. Alcuni esempi sono cat, oppure un database management system.

Allora nasce l'idea successiva chiamata **time-sharing**, che si realizza facendo in modo che succeda qualcosa nel sistema ad intervalli regolari.

Nella quarta generazione abbiamo l'avvento del **microprocessore**. Prima del microprocessore, il processore era fatto da tanti componenti, ma adesso finalmente è un componente unico. Questo fa risparmiare ed aumenta le possibilità. I primi sistemi operativi sono CTSS e MULTIX (che poi evolve in UNIX). Dopo si passa a UNIX, che viene creato nei laboratori della società telefonica AT&T. Da allora UNIX si è evoluto in numerose interpretazioni siccome il codice sorgente era libero.

Alcune definizioni:

Sistema real-time: è un sistema di cui la correttezza non dipende solo dal valore del risultato ma anche dal tempo in cui il risultato viene prodotto. Esistono sistemi real-time lentissimi, es. sistemi di controllo delle centrali nucleari. Ci sono due tipi di sistemi real-time: hard-realtime, che produce risultati catastrofici se i tempi non vengono rispettati, soft-realtime che sono per esempio i sistemi multimediali etc. Questi tipi di sistemi non possono essere fatti come un sistema normale.

Sistemi distribuiti: esistono sistemi **loosely coupled** e **tightly coupled**. Nel TC ho tante CPU che condividono la memoria, e avremo tantissimi problemi (es. accesso alla memoria, parallelismo limitato) che l'SO dovrà risolvere. Nel LC ogni processore ha la sua memoria e collaborano cambiando di stati (es: comunicano attraverso message passing). (per fare in modo che ogni processore possa comunicare con un altro in logn passi si usano quadrati per 4, cubi per 8 e ipercubi per 16 processori).

Architetture Computer:

La macchina di **Von Newmann** e' composta da una memoria primaria di lavoro contenente sia dati che programmi, che può essere acceduta tramite un bus condiviso, connesso alla cpu e ai dispositivi di I/O (al giorno d'oggi c'è anche una gerarchia dei bus). Il bus è suddiviso in tre parti: il bus indirizzi, il bus dati, e a ritroso il bus interrupt.

Per fare operazioni di I/O si utilizza il bus. Un processore può effettuare due operazioni sul bus:

- Lettura (si mette indirizzo sul BUS indirizzi e poi si dà comando di lettura. Dopo un po si guarda cosa c'è sul BUS dati)
- Scrittura (si legge il BUS indirizzi e si modifica il valore in tale indirizzo col BUS dati)

Inoltre, ci saranno degli indirizzi che non sono della memoria, ma degli indirizzi al quale il BUS risponde col controller, che vengono usati per esempio per stampare i dati o fare altre operazioni. Per un processore infatti scrivere in una memoria e stampare dei dati non e' molto diverso.

Quindi per la macchina di Von Newmann, fare delle operazioni I/O e fare degli accessi in memoria e' equivalente.

Per informare il processore che una certa operazione e' terminata si fa uso degli **interrupt**, ovvero dei segnali che sono proprio dei fili (bus interrupt). Come funziona gli interrupt? Quando una operazione esterna dice che ha finito, accende il suo livello di bit; la CPU, ad ogni ciclo FDE, ha un punto morto, ovvero il punto in cui ha finito una istruzione ed inizia quella successiva e solo a quel punto li' si chiede se c'e' qualche interrupt. Se gli interrupt sono macherati semplicemente li ignora.

Lezione del (26/02/2021) – Architettura degli elaboratori

Il sistema parallelo LC e' un passo verso i sistemi distribuiti (che solitamente sono costituiti da tanti sistemi), ma che sono fatti in modo da apparire come un'unica macchina. Quando parliamo di macchine parallele (ad esempio sistemi multicore (SMP)), parliamo di macchine LC, quindi con memoria condivisa. Il problema sta nel fatto che, come abbiamo visto nelle lezioni del primo semestre, le CPU "combattono" fra loro per avere accesso al BUS condiviso. Per questo motivo, spesso avere piu' di 16 processori e' controproducente. Infatti, quando uno pensa ai sistemi paralleli, pensa che ogni processore del sistema lavori alla stessa velocita', ma non e' cosi' siccome abbiamo dei rallentamenti a causa dei vari accessi in memoria; Se volessimo fare un grafico con in ascissa il numero dei processori e in ordinata la performance, otterremmo un grafico che tende a raggiungere un asintoto spostandoci sull'ascissa.

La cosa positiva di tutto cio' e' che se abbiamo tanti elementi d'elaborazioni indipendenti, che eseguono tanti processi per tanti utenti, anche se i singoli processi non sono pensati per sistemi paralleli, si ha un aumento di performance.

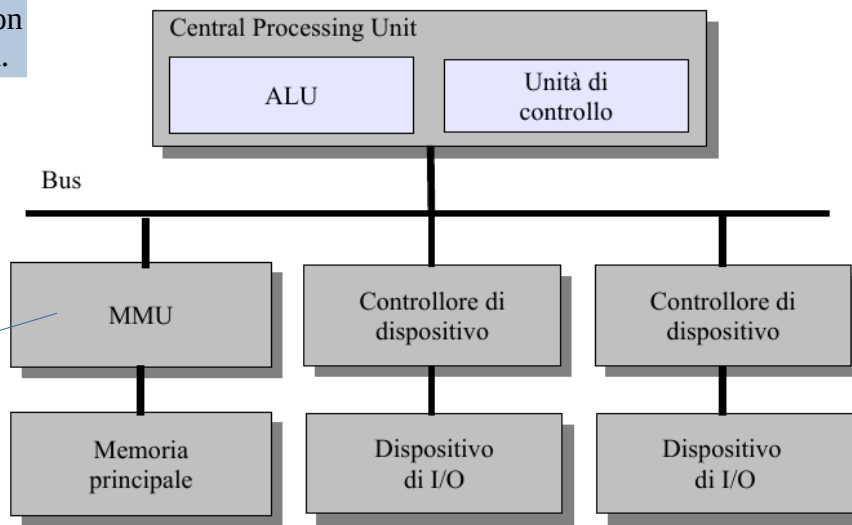
Tempo fa si era anche provato ad usare una architettura basata su moltissimi processori (circa 65000) poco potenti, al posto di usare 4 processori molto potenti, detta Connection Machine. Il problema di un architettura del genere e' il fatto che se abbiamo tanti processi indipendenti, dobbiamo caricare il processo sulla memoria di ciascun processore. Percio' questi sono molto efficaci su macchine specifiche, e usando programmi ad hoc fatti apposta per quel tipo di struttura.

La Connection Machine pero' falli'... uno dei problemi delle macchine infatti e' che complessita' = fragilita' (come abbiamo visto con le valvole, che funzionavano un giorno si e gli altri no lol). Il singolo processore normalmente e' molto robusto e resistente, ma quando se ne mettono 65000 si iniziano a creare problemi.

I programmi stanno in una memoria, unica se pensiamo a Von Neumann, separata dagli altri dati se stiamo parlando di Harvard.

Architettura di Von Newmann tipica.

La MMU e' dentro la "scatola" del processore, ma dal punto di vista logico non e' parte del processore, e' separato.



Gli **Interrupt** sono monodirezionali, e sono controller per “attirare l’attenzione” della CPU. L’evento più comune assegnato ad un interrupt e’ la terminazione di una operazione I/O. In un sistema base, monoprocesso, il processore e’ il sovrano assoluto del bus. Quindi il processore vuole parlare con la memoria, mettere l’indirizzo, dice che operazione vuole fare e le fa. Le operazioni che si fanno sul BUS solitamente sono 2: metti in un valore negli bus indirizzi, metti un valore nel bus dati e accendi il bit W (quindi faccio una operazione di scrittura) oppure metti un valore nel bus indirizzi, accendi il bit R e sul bus dati dopo un certo un periodo ricevo il dato da leggere.

Ci sono degli indirizzi (detti device register in umps3) in cui dice se 0 e’ la RAM se e’ 1 sono di I/O, quindi il processore per dire “fai tale cosa” scriverà qualcosa negli indirizzi a cui risponde il controller. Avendo tutti i fili in parallelo, quello che passa sul BUS e’ visto da tutti quelli che sono collegati al BUS, in modo simile ad un grande display che tutti possono guardare. Allora quando passa un indirizzo col primo bit spento lavora la RAM, altrimenti quando passa un indirizzo nel range degli indirizzi dell’I/O, e se c’è un unità che riconosce l’indirizzo come suo, lo prende e lo fa suo. Può essere anche un ordine di lettura/scrittura, siccome dentro ai controller dei device ci possono essere dei buffer. Se invece dobbiamo lavorare con un disco, non dobbiamo leggere un byte alla volta, bensì viene letto per blocchi (se ci ricordiamo dalla vecchia lezione sui file di device di Linux, noteremo che appunto ci sono device a blocchi e altri a carattere c, quanti a byte. La distinzione che si fa fra questi e’ esattamente di questo tipo, ovvero da come essi vengono letti, quindi a blocchi o byte per byte).

Pero’, per scambiare i blocchi occorre che nel controller ci sia un buffer. Facciamo un esempio: se volessimo fare una scrittura, assumendo di avere una macchina single-core in cui il solo processore e’ il signore del bus, con una operazione di lettura o scrittura possiamo leggere o scrivere solo una quantità di dati corrispondenti alle dimensioni del BUS, quindi in una macchina moderna sono 32/64 bit.

Infatti quando parliamo di processori a 32 o 64 bit, intendiamo che sia l’architettura interna del processore che quella del bus sono a 64 bit, questo vuol dire che con un ciclo solo trasmettiamo 8 byte. **(Fun fact:** un tempo c’erano macchine con architettura di cpu e bus diverse, e queste due architetture erano separate da una barra, ad esempio 128/64.). Se nel nostro device abbiamo un blocco da 4096 byte, non possiamo scrivere un blocco alla volta, dunque si fa uso di un buffer nel controller.

L’interrupt, quindi, avviene quando finisce una operazione di I/O (se c’è), o anche quando c’è da segnalare qualcosa, ad esempio, errori di lettura del disco etc. L’**interrupt “vero”** (quindi **hardware**) e’ completamente asincrono rispetto all’esecuzione di un processo. Se ho un programma in esecuzione (quindi un processo), non e’ correlato al processo corrente, ma arriva da un operazione che era stata lanciata da qualcun altro.

L’interrupt fanno in modo che il processore, quando ha finito di eseguire le istruzioni, se c’è un interrupt pendente (e che non e’ stato mascherato), salta a fare altre cose. Questo meccanismo, che interrompe il funzionamento normale del processo in esecuzione e consente al kernel di intervenire, e’

stato esteso agli **interrupt software (trap)**: esistono infatti delle istruzioni che chiunque può chiamare (anche in modalità utente, più avanti poi viene spiegata meglio la distinzione fra kernel mode e user mode) che permettono all'interrupt di intervenire. Inoltre, sono sincrone, ovvero il processo stesso chiama la trap. In questo modo, si estende il “chiede aiuto al OS perché è successo qualcosa” al processo stesso, e le syscall sono create come interrupt software.

Gli interrupt inoltre possono essere **mascherati** (l'abbiamo anche visto quando dovevamo fare la critical section) perché talvolta, per poter scrivere codice del sistema operativo, abbiamo bisogno di fare delle cose senza soluzioni di continuità, e quindi senza dover essere “sviati” dal cammino principale e dover eseguire codice che si trova in un'altra zona. Se c'è un interrupt mentre gli interrupt sono mascherati, l'interrupt NON viene perso, ma rimane in attesa e quando viene tolto il mascheramento (che è un registro della CPU, dove se il bit è a 1 allora il mascheramento è attivato), la CPU si accorge che c'è un interrupt e lo salta. Un interrupt si spegne quando si manda al controller (sempre via BUS) una istruzione di acknowledgement, (es: il controller dice “ho finito una operazione di I/O”, alza la bandierina, arriva il kernel a gestire, e poi gli manda una operazione ack).

Fra i dispositivi di I/O, per fare scheduling (che permette il multitasking attraverso time sharing, che permette a un processo di non usare la CPU per tutto il tempo grazie ad un timeout, lo vedremo nel prossimo capitolo), c'è un device che letteralmente non fa niente, si gira i pollici letteralmente, per una certa durata. Per indicare al kernel quando questo periodo sia effettivamente finito, si manda un interrupt. Questo device è detto **interval timer**.

Gli interrupt software invece vengono usati per segnalare errori in cui il sistema “non deve saltare del tutto”, e in cui quindi deve essere comunque affidabile. Un esempio sono errori quali segfault o divisioni per 0. Si usano anche per segnalare eventi eccezionali come syscall.

Cosa succede quando avviene un interrupt?

Generalmente avviene un salto, ovvero viene caricato il Program Counter (PC) a un valore predeterminato corrispondente al codice del gestore degli interrupt, all'insaputa del codice corrente.

Il processore deve gestire correttamente questo salto, ad esempio dovrà salvare i dati nei registri (come in PC e il registro di stato, State). Occorre anche salvarsi cosa viene cambiato, per esempio sicuramente si passerà da modo user a modo kernel, e questo dovremo ricordarcelo (questo dato corrisponde ad un bit), altra cosa sarà la maschera di interrupt.

Come fanno i singoli processori? Dipende, ci sono processori che salvano l'intero stato della CPU, rendendo la vita più facile all'OS, ci sono invece altri che fanno solo operazioni minimali e quindi il sistema operativo dovrà ricordarsi di salvare altre cose. I primi sono meno efficienti, siccome salvano anche cose che non servono, e i secondi sono più efficienti ma fanno fare più lavoro all'OS. In general e però, il processore dovrà sospendere le operazioni del processo corrente e saltare a un particolare indirizzo di memoria (in alcuni processori sono più indirizzi e si parla di interrupt vector, e a seconda del tipo di interrupt si salta ad indirizzi diversi) contenente la routine della gestione degli interrupt (**interrupt handler**). Tutto ciò avviene all'insaputa del processo che l'ha causato.

Facciamo un esempio: io ho un processo che sta facendo raytracing. Ad un certo punto, l'interval timer chiama un interrupt siccome il tempo è scaduto (e il programma che fa raytracing non ha nulla nella sua parte del codice che prevede questo, quindi è proprio a insaputa di esso), e se c'è un processo in attesa sospenderà il raytracing e chiamerà quello in attesa, finché poi non scatterà di nuovo il suo timer e chiamerà nuovamente un interrupt.

Quando ci sono guasti hardware, si chiamano degli interrupt detti interrupt NMI (non-maskable interrupts).

Come abbiamo visto, try-catch non c'entra assolutamente nulla con il concetto di interrupt! Bensì è solo una struttura del linguaggio di programmazione per gestire degli errori nell'esecuzione di un pezzo di codice. Tuttavia, se noi per esempio facciamo una divisione per 0, viene lanciata una trap per il kernel, che la “acknowledgia” e invia un segnale al processo (la SIGFPE che abbiamo visto nei segnali) che se non viene catturato causa la terminazione del processo, e se invece viene catturato fa partire il gestore del segnale. In Python per esempio abbiamo questo tipo di gestori. Essenzialmente, se facciamo una divisione per 0, avviene una trap, poi questo errore può risalire i vari livelli per arrivare ad essere

gestito dal processo (se vogliamo) e poterlo controllare dal linguaggio di programmazione, ma siamo molti stati sopra rispetto al kernel.

Torniamo a come viene gestito un interrupt: arriva un segnale di interrupt alla CPU, la CPU finisce l'esecuzione della istruzione corrente e verifica se c'è stato un interrupt. A questo punto, attraverso un salto, il controllo viene trasferito all'interrupt handler, ma prima si salvano i registri critici (oppure fa una fotografia dello stato del processore, a seconda del metodo scelto). Dopodiché, si carica il registro PC con l'indirizzo iniziale dell'interrupt handler assegnato.

Queste operazioni appena spiegate sono fatte direttamente dal processore, non si trova nessun codice nel programma, e quindi non vengono decise dal programmatore, ma bensì da chi costruisce l'architettura della CPU. Possiamo trovare queste istruzioni spiegate nel data sheet del processore stesso per sapere come avvengono.

Scrivendo nel sistema operativo, in fase di partenza noi dovremmo andare a mettere a mano l'indirizzo dell'handler nella locazione di memoria dedicata ad esso.

Una volta fatta questa parte, però, il processore non sa più nulla di cosa avviene durante la gestione dell'interrupt (es: se è andato in modo kernel, se ha mascherato gli interrupt, se ha cambiato il PC...), e qui entra in gioco il kernel. La CPU infatti andrà a richiamare il gestore degli interrupt che gli abbiamo specificato, e spetterà a lui gestire le varie situazioni che avvengono.

In sostanza, il kernel lavora in questo modo: fa una inizializzazione, dà il controllo al primo processo (in libertà vigila però, siccome gli abbiamo messo il tempo massimo etc...). Quando avviene un interrupt, facciamo il codice del kernel e miracolosamente riparte dall'interrupt handler. Poi alla fine ridà il controllo al processo chiamato.

A questo punto il processore non sa più nulla dell'interrupt, e il codice deve completare il salvataggio dello stato (il kernel è responsabile del riavvio del processo in modo da non perdere nulla). Poi va a vedere che interrupt è (per esempio: è un interrupt I/O?) e si comporta di conseguenza.

(Digressione) L'interrupt handler ha gli interrupt mascherati, ma nel momento in cui carica lo stato normalmente viene smascherato. Comunque, mentre il processo utente non può smascherare/mascherare gli interrupt, volendo se sappiamo che il kernel potrebbe gestire un altro interrupt mentre ne sta già gestendo uno, allora noi possiamo smascherarli quando vogliamo. A livello utente normalmente sono tutti smascherati (o come preferisce il prof, sono non mascherati).

I sistemi operativi moderni sono detti interrupt driven, siccome l'interrupt ha un ruolo fondamentale in questi, è come un direttore d'orchestra. Ma come si faceva prima che ci fossero gli interrupt? C'erano anche dei sistemi operativi che promettevano una specie di richiesta al kernel nulla. In UNIX abbiamo una cosa simile attraverso la syscall `yield()` (tradotto "dare la precedenza"), che blocca l'esecuzione del processo che la chiama in favore dell'esecuzione di un processo che ha una priorità maggiore o uguale. Il difetto di questo sta nel fatto che si cede il controllo al processo utente, e la gestione di questa organizzazione è così totalmente in mano al programmatore, quindi anche l'efficienza. Proprio per questo sono nati gli interrupt e il concetto di time sharing etc... (spesso succedeva che si faceva il controllo della fine di output solamente quando "pareva" al processo. I sistemi così erano davvero poco efficienti.)

Un modo per trattare gli interrupt è per esempio quello che abbiamo visto poco fa (ovvero quando arriva un interrupt si mascherano tutti gli interrupt, lo gestiamo e torniamo al processo utente), ma volendo si può anche fare di meglio: mentre gestiamo un interrupt si possono lasciare non mascherati gli interrupt di priorità superiore, in questo modo gestiamo gli interrupt in maniera nidificata.

Fra i registri che dobbiamo salvare, c'è lo stack pointer. Il kernel prima di gestire l'eccezione deve salvare lo SP del programma, ed eseguire il codice dell'interrupt handler in un nuovo stack. Nel caso

degli interrupt nidificati, dobbiamo avere DUE stack differenti, siccome il codice dello stack dell'interrupt del primo livello non deve interferire con il codice dello stack del secondo livello.

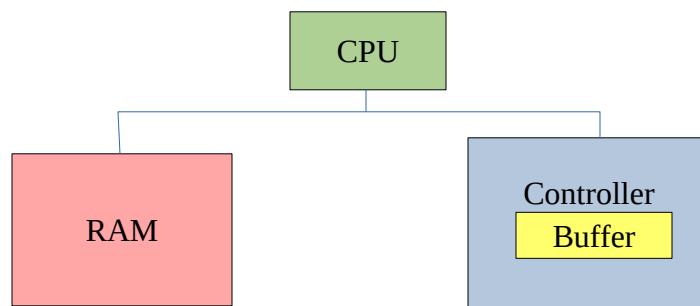
[IMPORTANTE]: ecco qua un altro concetto spiegato male. Come sempre, mi sono limitato a scrivere esattamente ciò che ha detto]

Il controllore hardware deve gestire la comunicazione fra il BUS e il device, e deve gestire una richiesta alla volta. Per ricevere le risposte si potrebbe fare busy waiting, ma questo si potrebbe fare solo se il sistema non è multitasking. ([Fun fact: il codice della fase1 del nostro progetto fa busy waiting per scrivere sul terminale]). Normalmente quello che succede è che i dati da leggere e scrivere sono nel buffer del dispositivo e bisogna copiare i dati dal buffer del dispositivo alla memoria.

Nell'interrupt driven I/O, la CPU carica (tramite il BUS) i parametri della richiesta di i/o direttamente nel buffer dell'hardware controller. Dopo che riceve il segnale del completamento dell'operazioni, la CPU copia i dati dal buffer del controller alla memoria. Ma questo non è molto efficiente, siccome occupa molto la CPU, inoltre non tutte le operazioni fanno uso del bus (es: somma fra due registri, salti...) ma vengono svolte direttamente all'interno del processore. Le macchine RISC inoltre non hanno operazioni aritmetico logiche con indirizzi di memoria, bensì dobbiamo caricare i registri della CPU e si fanno solo le operazioni aritmetico logiche sui registri della CPU. La CPU lascia così tempo libero ad un'altra risorsa che è il BUS, che invece potremmo sfruttare. Sarebbe più logico quindi lasciare la CPU libera ed usare di più il BUS durante i suoi tempi morti, e per fare ciò si fa uso del **Direct Memory Access (DMA)**.

Tempo fa avevamo spiegato che la CPU era il “signore del BUS”, ma ora la cosa si complica: si consente al controller di usare il BUS quando non viene usato dalla CPU, e si aggiunge un circuito detto “arbitro del bus” che decide chi usa il BUS (ovviamente la CPU ha la precedenza). Facendo così, possiamo direttamente dire al controller, per esempio, di scrivere su disco il blocco che si trova in memoria centrale a un dato indirizzo. Il controllore così, usando il cycle stealing, “ruba” il ciclo di BUS al processo quando non lo usa e va a prendere i dati necessari in memoria per riempire il buffer del controller, e nel mentre il processore esegue altri processi.

Facciamo un esempio schematico dell'operazione di lettura:



Mettiamo che al controller viene dato l'ordine “leggi il valore dal disco”. A sto punto il controller prende il valore dal disco e lo mette nel suo buffer (e fin qui l'operazione è uguale sia che ci sia la DMA oppure no). Se non usassimo il DMA, l'operazione finirebbe qui e mando l'interrupt per segnalare il completamento. In caso contrario, il controller prende uno dei cicli del bus della CPU e mette il risultato nella locazione RAM in cui la CPU vorrà leggere spostando 8 byte alla volta (supponendo che la macchina sia a 64 bit); Nel frattempo la CPU sta eseguendo un altro processo. Se non ci fosse stato il DMA, la CPU avrebbe fatto la stessa cosa, ma avrebbe usato il suo tempo prezioso.

Cambiamo argomento.

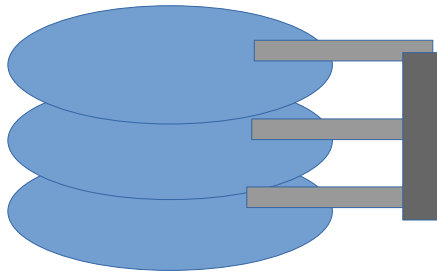
Nelle macchine moderne, l'accesso alla memoria viene fatto attraverso **MMU** (Memory Managment Unit) che traduce gli indirizzi logici in indirizzi fisici. Quando la CPU accede alla memoria, nel BUS

mette degli indirizzi che non sono i veri indirizzi di memoria, siccome ha una visione logica di essa. Questa richiesta quindi viene mediata dalla MMU, che appunto traduce l'indirizzo dato.

La MMU svolge due operazioni fondamentali. Prima di tutto ha una funzione di protezione: dobbiamo pensare alla memoria come qualcosa di molto dinamico. Se infatti un processo accedesse direttamente alla memoria senza MMU, sarebbe un bel guaio siccome se abbiamo riservato un certo spazio di memoria per un processo e magari ne ha bisogno di altra, potrebbe andare ad occupare lo spazio vicino che potrebbe essere riservato ad un altro processo. Invece in questo modo possiamo farci che ciò che il processo vede segmenti di memoria vicini e contigui siano invece separati in due spazi molto diversi, e così si riesce a gestire queste richieste (questo metodo si chiamerà paginazione e lo vedremo più avanti). [DISCLAIMER: teniamo a mente che il DMA serve per i dispositivi e l'MMU per la memoria].

Esistono dispositivi che per loro natura sono **memory mapped**, ovvero l'accesso al dispositivo è equivalente all'accesso in memoria. Un esempio di questi può essere uno schermo grafico: uno schermo grafico contiene un framebuffer, equivalente a un vettore che sta nel controllo e che contiene 24 bit per ogni pixel. Per scrivere sullo schermo, non andiamo a mandare richieste al sistema operativo, bensì andiamo a scrivere in delle zone del buffer del controller in modo da comunicargli esattamente quale pixel andremo ad accendere. Siccome non comunichiamo con l'os per accedere al dispositivo, non abbiamo interrupt. Lo svantaggio di questo approccio è il fatto che necessita di tecniche di polling (ovvero ripetizione, busy wait, di una operazione di check del dispositivo) siccome appunto non abbiamo interrupt che ci dicono se il processo ha finito o meno.

Memoria a Dischi:



Se mettiamo la testina in una certa posizione, noi possiamo leggere o scrivere tutti i dati che stanno a quella distanza dal centro.

Possiamo solo usare una testina alla volta.

Per andare ad un'altra distanza dal centro, dobbiamo spostarci fisicamente.

Se noi facciamo uso del DMA col disco, noi dobbiamo comunicare esattamente quale cilindro, settore e testina (in questo ordine esatto) usare, che operazione vogliamo fare e dov'è il buffer. In realtà le operazioni che dovremo fare saranno 3, e siccome lo spostamento delle testine è una operazione lenta, esiste una operazione di I/O per fare esattamente questo. Questa operazione di movimento delle testine viene chiamata SEEK.

Le operazioni saranno quindi SEEK, READ e WRITE.

SSD:

Le memorie a stato solido sono dispositivi per la memoria non volatile dei dati. Queste memorie si scrivono a blocchi e si leggono a banchi.

Protezione del sistema:

Un processo, come dicevamo prima, non deve mai compromettere un altro processo o il sistema. Per fare questo abbiamo bisogno di hardware dedicato e non si può controllare via software (svolto appunto dalla MMU come dicevamo prima).

Inoltre è presente un bit di "modo", che stabilisce se siamo in **modo kernel** o in **modo user**. Quando un processore è in modalità kernel, allora potrà fare qualsiasi operazione esso voglia. Altrimenti, se è in user mode, potrà solo eseguire le istruzioni che accedono al processore stesso o alla memoria che è stata assegnata a quel processo che stiamo eseguendo.

Il modo kernel e il modo utente non centrano assolutamente nulla con la multi-utenza oppure con l'utente root, infatti anche i processi root vengono svolti in user mode, siccome anche essi devono essere protetti! Le richieste che fa al sistema sono fatte attraverso le syscall, che sono a un livello più alto di quello che stiamo trattando e in con un certo grado di protezione.

All'accensione del macchina (e quindi in partenza) il processore è in modalità kernel. Una volta caricato il sistema operativo (attraverso il bootstrap) e ad averlo eseguito, si inizializzano le sue tabelle e variabili di funzionamento, crea a mano il primo processo (che è un processo utente) e si passa così al modo utente. Ogni volta che si entra nel gestore delle eccezioni, e quindi ogni volta che avviene un interrupt, si ritornerà in modalità utente.

Se c'è un bug a livello del kernel, avviene un kernel panic e crolla tutto :(.

Le istruzioni di I/O sono privilegiate, ovvero in modo user non si accede a nulla che fa I/O. Sarà l'OS che dovrà fornire agli utenti delle primitive per accedere ad esso, e passeranno quindi attraverso codice del OS, sottoforma per esempio di syscall. Infatti, le uniche richieste che possiamo fare al sistema da processi utente è attraverso le syscall, che fanno uso di interrupt (quindi avviene esattamente quello che abbiamo descritto negli interrupt, ovvero si salvano i registri principali, si salta al gestore etc...).

Lezione del (2/03/2021) – Struttura generale del SO, classificazione di un Kernel, VM

Alcuni chiarimenti prima di cominciare la lezione vera e propria:

[DISCLAIMER: la roba qui sotto è interessante, ma non troppo importante]

Inizialmente la RAM era costruita attraverso nuclei di ferrite, e quando il pc veniva spento, i suoi contenuti rimanevano salvati anche dopo lo spegnimento della macchina. Dopo questo, c'è stato un passaggio alla memoria costruita prima con i transistor e poi a condensatori, e più o meno velocemente perdevano il loro contenuto, e dunque i sistemi operativi iniziarono ad adeguarsi e tenere in questo tipo di memoria solo i dati in corso di elaborazione. Ora stanno comparando sul mercato memorie non volatili con velocità comparabili a quella della RAM.

Con polling si intende “tentare ripetutamente di fare qualcosa finché non va a buon fine”. In una lezione precedente (del secondo semestre) è stato citato siccome se non abbiamo gli interrupt, non possiamo essere avvisati di un particolare stato della CPU e quindi ci tocca fare un controllo continuo. Se abbiamo un microcontrollore (monotask, i.e. arduino), per vedere se un sensore ha finito di fare una certa operazione controlliamo continuamente nel loop. Se invece abbiamo un sistema multitasking è un bel guaio, siccome il controllo verrà fatto solo alcune volte quando la CPU riesce a farlo, e quindi sono molto meno responsive.

Tornando al memory mapped I/O, prendiamo l'esempio dello schermo, in cui per accendere un pixel di un certo colore dobbiamo scrivere nel framebuffer dello schermo. Questa memoria, e in generale in tutti i MMI/O, si trova nei controller del dispositivo ed è una memoria a doppia entrata, ovvero c'è un lato in cui la CPU scrive (ed eventualmente legge) e dall'altro c'è il controller che legge e in base a quello mostra l'immagine sullo schermo. Il controller del display, in questo caso, fa polling, siccome deve sempre leggere ciò che è scritto in questa memoria e mostrare le cose su schermo.

Struttura del SO:

Il sistema operativo, assieme ai compilatori, è uno dei sistemi più complessi che un informatico può costruire. All'interno del sistema operativo abbiamo molte interfacce interne che comunicano e sono collegate fra di loro. I veri utenti del sistema operativo non sono gli umani, ma i processi.

Il processo è essenzialmente un programma in esecuzione. Con programma si intende invece il “testo” vero e proprio delle istruzioni da eseguire. Nell'ambito della gestione dei processi, il SO è responsabile di diverse attività, tra cui: creazione e terminazione dei processi, comunicazione fra processi, gestione del deadlock etc...

Il sistema operativo si occupa anche della gestione della memoria principale.

La memoria principale viene vista dal calcolatore come un grande array di byte, indirizzabili singolarmente. Tra le varie cose, il SO gestisce quanta memoria allocare a un processo e tiene traccia di quali parti della memoria sono state usate e da chi, e decidere come programmare la corrispondenza fra indirizzi logici e fisici fatta dalla MMU.

La memoria secondaria è anch'essa gestita dall'OS. Poiché la memoria principale è volatile e troppo piccola per contenere tutti i dati, il computer è dotato di una memoria secondaria. Tra le varie cose,

l'OS gestisce il partizionamento e l'ordinamento efficiente delle richieste. Oltre a questo, si occupa della virtualizzazione della memoria.

Il 90% del codice del kernel sta nella gestione dell'I/O, siccome esistono moltissimi tipi di device diversi. Il SO fornisce un'interfaccia comune per la gestione dei device driver dei miei dispositivi I/O, infatti esso ha un insieme di driver possibili così da riuscire a capire come vuole comunicare ogni dispositivo, quindi avrà anche un sistema di gestione di buffer per il catching delle informazioni di questi dispositivi.

Il sistema operativo si occupa anche della gestione del **file system**, che è una astrazione, non è totalmente indipendente dal media in cui viene memorizzato, che ha caratteristiche proprie e una propria organizzazione fisica.

Il sistema operativo si occupa anche del supporto multiuser: deve fornire la protezione (controllare gli accessi dei processi alle risorse del sistema e degli utenti). Tra le varie cose quindi deve: gestire l'identità del proprietario del processo (uid gid), gestire chi può fare cosa (specificarlo per ogni risorsa/utente), fornire un meccanismo che attui / implementi la protezione (nel momento in cui vengono usate le syscall da un processo utente devono essere eseguite SOLO se ne ha il permesso). Infine, l'OS gestisce il networking: consente di far comunicare più processi su diverse macchine, di condividere risorse. Permette di far funzionare protocolli di comunicazione a basso livello come TCP/IP o UDP e servizi ad alto livello come file system distribuiti o print spooler (=coda di stampa, memorizzare le stampe degli utenti con una politica FIFO e mandarle alla stampante che può anche essere collegata in rete).

Difetti degli SO:

Se abbiamo bisogno di gestire eventi con tempistiche certe, un arduino è molto meglio di un sistema con linux. Questo perché, essendo linux un sistema multitasking, con memoria virtuale e processi, non può dare una garanzia assoluta di timing. Quindi se io voglio creare un sistema modulare, devo creare un modo per gestire la modularità, che è una cosa molto difficile da gestire coi sistemi operativi.

Classificazione degli SO:

Posso dividere i sistemi operativi in due grandi famiglie: i sistemi a struttura semplice e quelli a struttura a strati. L'uso di questi due tipi dipende da scopi diversi.

- I **sistemi a struttura semplice** solitamente sono usati nei sistemi embedded. Abbiamo per esempio sistemi embedded che non hanno bisogno di gestire complessità extra. Non ci sono livelli di funzionalità separati, le applicazioni possono accedere direttamente alle routine di base per fare I/O (un programma può far crashare tutto).

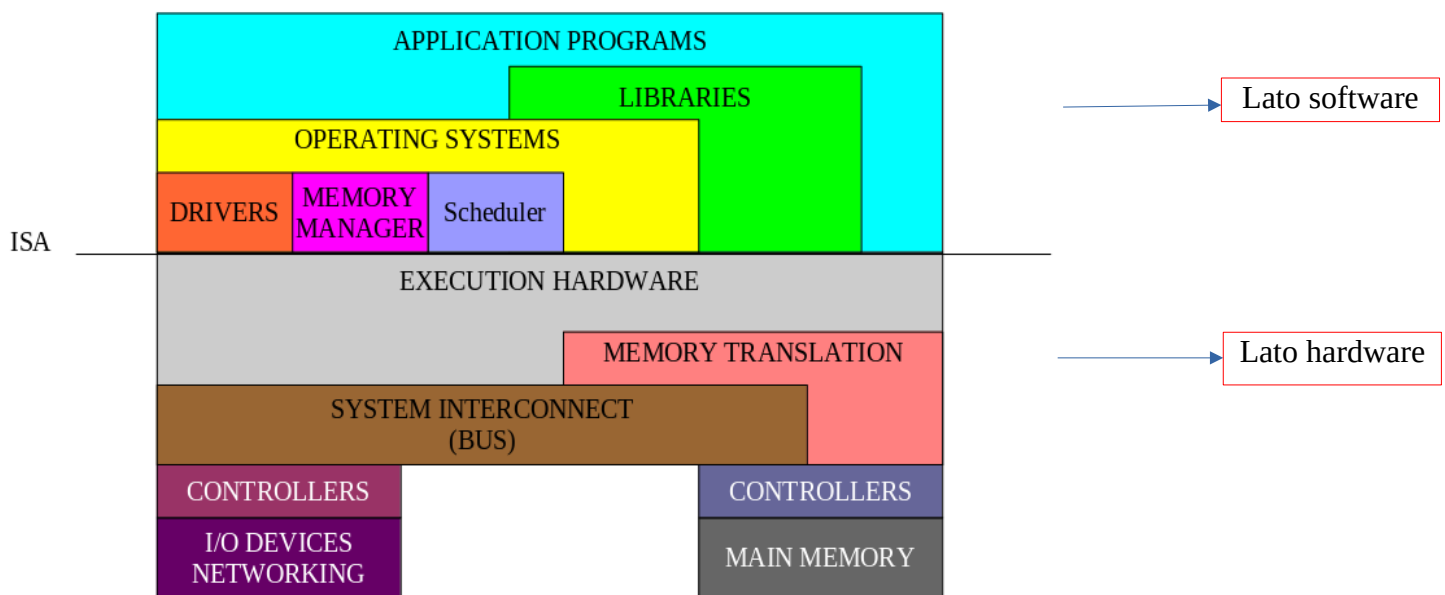
Un esempio è Free-DOS (controparte opensource di MS-DOS), che ha un kernel costituito da device driver. Free-DOS inoltre ha programmi detti **“terminate and stay resident”** (tsr), che possono essere caricati nel device e poi rimangono residenti. È monotask, e anche la shell di questo SO è caricata come un terminate and stay resident. In poche parole, si carica un programma alla volta, e quando si termina si torna alla linea di comando dove si può caricarne un altro. In Free-DOS, gli .EXE erano i programmi “normali”, mentre i .COM erano i tsr. Il motivo di queste scelte (in particolare della scarsa sicurezza in I/O) è dovuto al fatto che spesso questi SO erano dedicati a processori a basso costo, che non avevano modalità protetta (kernel mode).

Anche UNIX è poco strutturato! Nonostante infatti la struttura interna sia molto complessa, il sistema operativo è compilato come un singolo eseguibile: il programma “kernel” è quello che sta in mezzo all'hardware e al PC. UNIX, però, non è un microkernel, ma un kernel monolitico (ovvero un blocco unico e inscindibile), e questo blocco è rappresentato dall'eseguibile di questo SO che viene lanciato all'avvio.

- I **sistemi operativi a strati** sono basati sulla stratificazione degli elementi del sistema operativo, e ogni strato comunica con quello superiore e quello inferiore ad esso, in particolare offre servizi agli strati superiori e si basa su quelli inferiori, ogni strato parla una lingua “più evoluta”. In questo modo si ottiene un maggiore grado di modularità.

L'idea del sistema operativo a strati si può anche trovare nel "The O.S." di Dijkstra, che conteneva 6 strati (da 0 a 5). Oggi la maggior parte dei sistemi operativi contiene pochi strati (UNIX ne ha 3), questo perché ogni strato aggiunge un "overhead", e non sono studiati accuratamente potremmo avere degli strati in cui è presente una duplicazione di codice che ha la stessa funzionalità di uno strato inferiore ma non accessibile a uno strato molto superiore siccome quella parte è ormai troppo incapsulata (es: in Windows, la swap area è implementata attraverso un file per causa di questo, ed molto inefficiente. In Linux invece, ho una partizione dedicata allo swap).

Ecco uno schema dell'architettura HW/SW di un computer:



Politiche e meccanismi:

La politica è un concetto di ingegneria del software: una cosa è decidere che cosa bisogna fare (**politica**), e un'altra è attuare effettivamente questa cosa (**meccanismo**). È simile alla divisione fra parlamento (che decide cosa fare, fa le leggi) e governo (che attua effettivamente le leggi decise dal parlamento). Separando le cose, è più facile cambiare i meccanismi senza cambiare la politica e viceversa. Facciamo un esempio: se vogliamo allocare la memoria per un processo, una cosa è mantenere la tabella della memoria e dire "ah io metterei quel processo lì", altra cosa è andare a mettere i dati nell'MMU per mettere effettivamente i dati lì. In questo modo possiamo cambiare la MMU senza dover cambiare la parte che gestisce le politiche.

Nei microkernel, sono implementati solo i meccanismi e si delega la funzione di politica ai processi fuori dal kernel.

[**DISCLAIMER**: è spiegato meglio più sotto.] Per capire bene cos'è un microkernel, dobbiamo tenere a mente quali servizi devono essere obbligatoriamente inseriti nel kernel per definirlo come tale, e quali invece possiamo scartare in quanto gestibili come processi. Un esempio di microkernel è il MINIX, dove il gestore della memoria è un processo esterno al kernel. Bisogna però tenere a mente che il gestore della memoria non è un meccanismo, non effettua la allocazione effettiva dei processi in un'area di memoria, bensì decide solo dove un processo deve essere allocato. Dunque, è una politica e non un meccanismo. La componente del kernel che esegue (o meglio, effettua) questa decisione è detta **system task**.

[**FUNFACT**] Nel MacOS fino al 9 si tendeva a mettere più roba possibile all'interno del kernel (tra cui anche meccanismi e politiche). Addirittura i meccanismi di gestione dell'interfaccia grafica erano inseriti nel kernel, in modo da forzare un unico look and feel dell'OS. Questo però poteva risultare molto problematico, in quanto un bug grafico poteva mandare in crash l'intero sistema (e purtroppo Windows 9x non è differente, era una bomba di BsoD).

Organizzazione del kernel

Esistono 4 categorie di kernel:

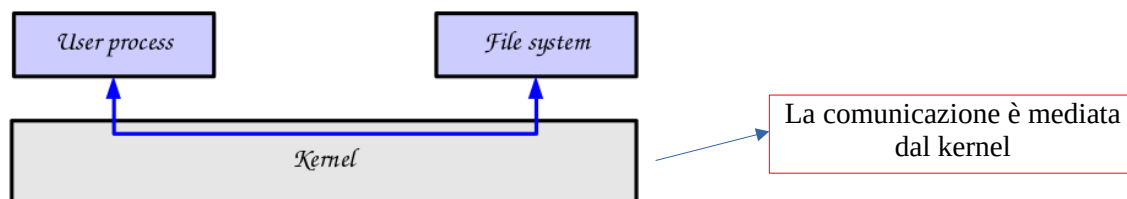
- **Kernel Monolitici:** un aggregato unico e ricco di procedure di gestione mutuamente coordinate e astrazioni dell'HW
- **Micro Kernel:** Semplici astrazioni dell'HW gestite e coordinate da un kernel minimale, basate su un paradigma client/server, e primitive di message passing.
- **Kernel ibridi:** simili a microkernel, ma hanno componenti eseguite in kernel space per questioni di maggiore efficienza.
- **ExoKernel:** non forniscono livelli di astrazione dell'HW, ma forniscono librerie che mettono a contatto diretto le applicazioni con l'HW.

Un **kernel monolitico** è un programma composto da un insieme completo e unico di procedure mutualmente correlate (come dice il prof, “può fare molte cose”). Possiede una struttura interna, e tutti i moduli fanno parte di un unico spazio di memoria (motivo per cui un device driver che va ad accedere ad un indirizzo di memoria sbagliato fa crollare l'intero sistema). Sono presenti syscalls, che come abbiamo visto sono servizi forniti dal kernel, tipicamente eseguiti in kernel mode. Esiste un certo grado di modularità, e come avevamo accennato due righe fa tutti i moduli sono eseguiti nello stesso spazio (kernel-space), ed è tale da rendere tutto l'insieme un corpo unico in esecuzione. Il problema di questo è che se fallisce un modulo, fallisce l'intero sistema generando un kernel panic (per esempio, se un segfault non viene gestito correttamente, l'intero sistema va in kernel panic). Questo perché tutti i moduli sono parte dello stesso programma. [Precisazione dalla lezione del 19/03/2021: se per esempio scriviamo un modulo kernel che fa una divisione per zero per esempio, e lo carichiamo, il sistema crolla completamente.]

I kernel monolitici sono molto-efficienti, pensiamo alla differenza di costo fra una chiamata funzione (in cui si scrive sullo stack, si fa un salto e fine) oppure fare inter-process communication (con message passing per esempio, in cui dobbiamo trovare il modo di passare dati fra un processo fuori dal kernel e il kernel stesso: dobbiamo fare tunneling, accertarsi che quello che spediamo venga letto etc... non è molto efficiente ngl). [FUNFACT: pandOS è un kernel monolitico a strati.]

Un **microkernel** è essenzialmente un programma che svolge le funzioni essenziali del kernel, tuttavia non contiene le parti non essenziali, che vengono invece implementate come processi a livello utente. Il microkernel “ideale”, ovvero nel quale non si possono più rimuovere funzioni (altrimenti non sarebbe un kernel) fornisce solo due servizi, due “vere” syscall: send e receive. Per accedere a un file quindi, non si fa uso di una syscall “open()”, ma attraverso la syscall send si invia al gestore dei file (che è un processo in questo caso) il comando “open” per accedere al file.

Generalmente, il microkernel deve offrire le funzionalità minime di gestione dei processi e della memoria, ma soprattutto dovrà quindi offrire dei meccanismi di comunicazione per permettere ai processi *client* di chiedere servizi ai processi *server*. Questa comunicazione interprocessi sarà basata su message passing, e il micro kernel si occuperà di smistare i messaggi fra i vari processi.



Esempio di open fatta con microkernel avente come syscall solo send e receive:

```
int open(char* file, ...)
{
    msg = < OPEN, file, ... >;
    send(msg, file-server);
    fd = receive(file-server);
    return fd;
}
```

Manda il messaggio al processo che si chiama file-server.

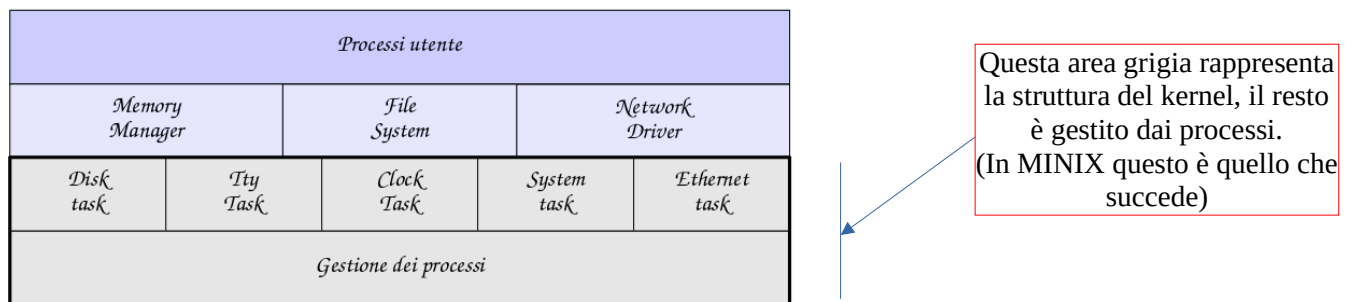
Metti ciò che ricevi da file-server nel file descriptor.

I vantaggi di un microkernel stanno nel fatto che è più facile da realizzare, è più modificabile ed espandibile (siccome basta aggiungere un processo a livello utente senza ricompilare il kernel, e la modifica è altrettanto facile). Es: se domani qualcuno inventasse un file system ext5, e dovessi implementare la possibilità di montare dispositivi aventi questo file system, essendo UNIX monolitico, sarei obbligato a creare una nuova versione del kernel. Con un microkernel, invece, devo solo aggiornare il processo che gestisce i file system, e a run-time (senza nemmeno dover spegnere il sistema) posso accedere a tale file system.

Oltre a tutto ciò, la portabilità è ottima: posso fare il porting del kernel ad un'altra architettura e poi mi basterà semplicemente ricompilare i servizi. Infine, posso assegnare al microkernel e ai vari processi di sistema livelli di sicurezza diversi, ed anche molto facile da adattare ai sistemi distribuiti.

Tra gli svantaggi, l'elefante nella stanza è rappresentato dalla scarsa efficienza, dovuta all'overhead determinato dal fatto che la comunicazione fra processi è mediata dal kernel.

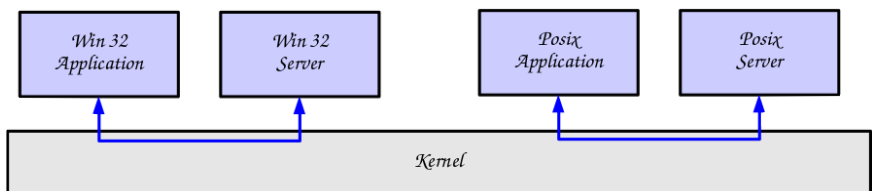
MINIX è uno dei microkernel più famosi, e la parte del nucleo è data dal gestore dei processi e dei task. Questi task sono dei thread del kernel.



Nonostante i kernel monolitici siano stati considerati obsoleti nel 1990, sono ancora quelli più in uso. Sono i più semplici da realizzare, ed è meno complesso gestire il codice di controllo in un'unica area di indirizzamento. I microkernel oggi sono usati soprattutto nei contesti in cui i failure non sono ammessi e potrebbero essere critici.

Esistono inoltre **kernel ibridi**, che sono essenzialmente microkernel modificati in modo da mantenere una parte di codice in "kernel-space" cosicché l'esecuzione sia più efficiente e allo stesso tempo adottano message passing fra i moduli in user space. Non sono da confondere con kernel monolitici in grado di caricare alcuni moduli dopo la fase di boot, la differenza sta nel fatto (almeno in Linux) che questo modulo viene caricato in kernel-space al momento di boot, e che quindi potrà avere accesso completo e sarà parte del kernel.

Windows NT può essere considerato un kernel ibrido, che ha supporto per diverse API (es. Win32, OS/2, Posix), e le funzionalità di queste API sono implementate tramite processi server



e il kernel media il message passing. Queste API aggiungono un layer di compatibilità fra applicazioni Posix per esempio. Nelle nuove versioni di Windows, c'è anche una certa compatibilità con Linux, implementato sempre grazie a processi server (WSL, Windows Subsystem for Linux).

Abbiamo poi gli **ExoKernel**, [**DISCLAIMER**: il prof considera gli Exokernel molto poco importanti e per questo decide di citarli e basta] l'idea di base sta nel coordinare l'accesso all'hardware da parte di più processi, e per fare ciò si programmano i processi in modo avere una visione diretta dell'hardware. Es: se un utente vuole allocare area X a un processo Y, l'exokernel si preoccupa solo di dire a quali settori può accedere il processo Y.

Macchine Virtuali:

Le **macchine virtuali** si basano sul fatto che, oltre a creare l'astrazione di processo, si prova a creare anche l'astrazione di una macchina virtuale. È un concetto stra-applicato nelle infrastrutture a cloud (cloud computing) e nell'IAS (Infrastructure As a Service). Facciamo un esempio e prendiamo lo schema colorato HW/SW di prima: se io togliessi la parte sotto a ISA e la sostituissi con una cosa, un qualsiasi cosa, che potesse rispondere correttamente alle richieste che la parte sopra fa, allora la parte sopra procederebbe senza problemi, senza accorgersi del trucco; penserà di usare una macchina anche se la macchina non c'è. Una VM quindi emulano (ovvero, si comportano allo stesso modo, da non confondere con simulare, siccome non simulare vuol dire fingere di fare quella cosa) una macchina fisica. Ci sono vari modi di creare macchine virtuali: all'epoca, IBM creò una macchina virtuale che rappresentava un layer subito dopo l'hardware, in modo che più kernel e più processi potessero essere eseguiti contemporaneamente. Nelle VM più popolari odierne, lo strato della VM si trova dopo quello del sistema operativo, comportandosi come un processo.

[**FUNFACT**] Le VM nascono in IBM (International Business Machine) nascono per risolvere un problema commerciale: cambiando il SO, IBM costringe il pubblico a cambiare anch'esso tutto il software per la nuova architettura. Dunque, per risolvere questo problema nasce la MVSVM.

Meccanismo per creare multi-tasking tra i diversi so. (so creato da IBM: MVSVM)

I vantaggi di una VM sono la possibilità di usare più sistemi operativi, e di sperimentarli. In più, possiamo emulare architetture diverse di un processore (es: PowerPC su x86, ARM o MIPS su x86).

Gli svantaggi sono che una soluzione del genere purtroppo è molto inefficiente.

Le macchine virtuali di java o python hanno funzioni e scopi diversi dalle VM di cui stiamo trattando ora. In poche parole, il codice viene compilato in un bytecode per una macchina che effettivamente non esiste, ma che poi viene tradotta per la end-machine.

[**DISCLAIMER**, il prof ha solo accennato queste cose e la definizione era abbastanza confusa, le seguenti informazioni sono prese da internet.] Abbiamo due tipi di VM: la Process VM e la system VM. Mentre la prima permette a un singolo processo e basta di girare su una architettura emulata, questa macchina viene dunque creata quando il processo si avvia e si distrugge quando il processo esce. La Machine VM o System VM emula un sistema intero che permette l'esecuzione di più processi e quindi anche di un sistema operativo intero. Il layer che si occupa della virtualizzazione dell'hardware viene chiamato HyperVisor (o virtual machine monitor) e può girare subito sopra l'hardware (tipo 1 o native VM) oppure sopra un sistema operativo (tipo 2 o hosted VM, simile a VirtualBOX).

[**piccola digressione, ulteriori info su busybox nella lezione dopo**] BusyBox è un sistema per il supporto di comandi Linux in un unico eseguibile in modo da non occupare spazio per "tutti" gli eseguibili dei comandi che ci sono in Linux.

XEN è un esempio di system VM di tipo 1, si trova come strato "sotto a Linux", e le macchine virtuali prendono il nome di domain. Xen in questo modo permette a più macchine virtuali di convivere sulla stessa macchina. In Xen, domain0 è la prima macchina virtuale, e tutto ciò che una macchina non riesce a risolvere (in particolare i device driver) viene dirottato al domain0 (es: device driver del disco è quello del domain0 in ogni macchina). Xen introduce anche il concetto di para-virtualizzazione in questo modo, che consiste nel creare device drivers virtuali all'interno delle macchine stesse (tranne in domain0 obv, è il concetto di cui abbiamo parlato fino adesso).

Un argomento interessante sono le macchine virtuali parziali, che emulano alcuni aspetti dell'hardware senza dover ricreare l'intero set delle syscall, e facendo così convivere parti reali e virtuali nella visione del processo utente che lancia la VM.

Progettazione di un sistema operativo:

[**DISCLAIMER**]: questa parte è stata trattata velocissimo e onestamente è molto importante, imo conviene saltarla]

Abbiamo diversi sistemi operativi per diversi scopi, ma la loro concezione sarà influenzata (o modificata) dall'hardware che dobbiamo usare e dalle applicazioni che dobbiamo eseguire (quindi questi saranno i constraint).

Lo stesso sistema operativo spesso viene usato anche su molte architetture diverse, ma possiamo avere casi in cui vogliamo un sistema usabile per uno scopo o per un altro, e spesso bisognerà ottimizzare tale sistema operativo per uno scopo o per un altro. Per UNIX e Linux esistono moduli aggiuntivi fatti apposta per questo, che aggiungono/aggiornano parti diverse del kernel (queste si trovano nella cartella `/boot/` in Linux). Linux inoltre è molto figo per i sistemi-embedded, siccome posso ricompilare il kernel con i parametri che voglio io e creare un sistema operativo per il mio sistema adatto alle mie esigenze.

Altro modo per gestire moduli aggiuntivi possono essere le estensioni del kernel (kext in MacOS 9 in giù) e in kernel-mode driver in Windows.

Lezione del (5/03/2021) – Scheduling, funzionamento del kernel, thread vs. processo, algoritmi di scheduling

Precisazioni sulle lezioni precedenti:

Per usare **busybox**, devo inizializzare una shell con la versione di busybox della rispettiva architettura (nel caso di un pc con amd64, devo usare la versione di busybox x86_64) attraverso il comando `busybox-x86_64 sh` (a patto che abbiamo messo nella cartella in cui siamo l'eseguibile di busybox) e per ogni comando che vogliamo aggiungere al nostro dispositivo, dobbiamo creare un link simbolico a ciascun comando. (es: `ln -s busybox cat`, `ln -s busybox ls`). La cosa bella di busybox, è che senza di esso dovrebbero creare un nuovo eseguibile in `/bin/` per ogni comando, mentre con busybox abbiamo un unico eseguibile che “fa le veci” di tutti i comandi basic di Linux. In questo modo posso per esempio creare un decoder tv risparmiando memoria su disco e memoria RAM (infatti, si riesce a caricare una volta sola busybox in RAM senza dover caricare più comandi ogni volta). Sono comunque versioni dei comandi semplificati, non contengono tutte le opzioni.

Parliamo invece delle **VM parziali**. Solitamente, in una macchina normale, i processi si basano sul kernel e le syscall vengono gestite dal kernel. Nelle VM parziali (un esempio, è VUOS), la VM si mette nel mezzo solo nelle parti che sono da virtualizzare (attraverso dei moduli), e per le parti non virtualizzate parla direttamente col kernel. Maggiori informazioni si trovano a [questo link](#).

Tabella dei processi e PCB:

Un sistema operativo si comporta come un contabile, nel senso che deve gestire di risorse. Fra le varie risorse, quella più importante da gestire è sicuramente il processore, e nella gestione del processore il kernel deve creare l'astrazione di processo. Partiamo da quali dati il kernel (il nostro contabile) deve tenere conto per ogni processo: il kernel per mantenere questi dati fa uso di “tabelle”. Una delle più importanti è la **tabella dei processi**, che contiene tutte le informazioni utili per poter eseguire processi in maniera fedele e senza che ci siano problemi.

Altre tabelle di cui il kernel tiene conto sono:

- *Tabelle di gestione della memoria*, che mantengono le informazioni per l'allocazione della memoria per il SO, e per l'allocazione in memoria principale e secondaria per i processi. Comprende anche la tabella con informazioni per i meccanismi di protezione (es. tabelle dell'MMU).
- *Tabelle per la gestione dell'I/O*, che mantiene le informazioni sullo stato di assegnazione dei dispositivi utilizzati dalla macchina e le informazioni della gestione delle code di richieste (o code di accesso).

- *Tabelle per la gestione del file system*, che mantengono l'elenco dei dispositivi utilizzati per mantenere il file-system e l'elenco dei file aperti e il loro stato.

In un certo senso, tutte le tabelle sono comunque dipendenti dalla tabella dei processi, i.e. le tabelle di memoria tengono conto della memoria che assegnamo a ogni *processo*, le tabelle della gestione dell'I/O mantengono lo stato della richieste del *processo* e le tabelle del file-system quali sono i file aperti dai *processi*.

Per poter eseguire un processo, dobbiamo tenere traccia di tutte le informazioni che ci servono a eseguirlo, e tra queste abbiamo:

- I segmenti dati, che rappresentano i dati su cui il processo deve operare.
- Uno stack di lavoro, per la gestione di chiamate di funzioni, passaggio di parametri etc.
- Il segmento codice, che rappresenta le istruzioni da eseguire.

Oltre a questi dati, abbiamo anche un insieme di attributi contenente tutte le informazioni per la gestione del processo stesso (i.e. dov'è lo stack, dov'è in memoria il codice da eseguire, come identifico questo processo, quali sono i file aperti etc...). Questo insieme di attributi, prende il nome di **descrittore del processo** (process control block, PCB). È possibile suddividere le informazioni contenute nel descrittore in tre aree:

- **informazioni di identificazione di processo**: in generale si parla l'identificatore del processo (pid), che può essere semplicemente un indice all'interno di una tabella di processi (tuttavia questo può causare problemi di omonimia, siccome un processo che prova a contattare il padre magari alla fine contatta un processo nuovo siccome il vecchio è terminato). Oppure può essere un numero progressivo; in caso, è necessario un mapping tra pid e posizione del relativo descrittore.
[FUNFACT] Alcuni sistemi sperimentali fanno uso dell'indice seguito da un "contatore di reincarnazione", ovvero tutte le volte che l'indice del processo viene riutilizzato, il contatore di reincarnazione aumenta.
Altre informazioni di identificazione contenute potrebbero essere per esempio l'id del processo padre.
- **informazioni di stato del processo**: servono soprattutto per poter fermare l'esecuzione di un processo e farla partire dal punto in cui esso si è fermato (es. quando avvengono gli interrupt). Normalmente, contiene una "fotografia" di tutti i registri del processore al momento dell'interruzione del processo, salvandola nel pcb. Quando avviene un interrupt, si fa un salto all'interrupt handler del kernel. Una volta che ha fatto queste operazioni, lo scheduler va a vedere il processo che deve ripartire, lo prende, si prendono le informazioni dello stato della CPU e vengono caricate, ritornando ad eseguire il processo al momento in cui era stato sospeso.
- **informazioni di controllo del processo**: contiene informazioni dello stato del processo, specificando se è ready, blocked o running.
Abbiamo poi anche informazioni sulla priorità del processo, di gestione della memoria (es: informazioni della configurazione della MMU, come puntatori alle tabelle delle pagine). Inoltre, vi sono le cosiddette informazioni di accounting, ovvero tempo di esecuzione del processo (tipo tempo di uso del processore) e il tempo trascorso dall'attivazione del processo. Tenere conto del tempo trascorso è abbastanza importante per un processo, e può essere utile per syscall come una wait.
Abbiamo poi informazioni relative all'inter-process communication, come lo stato dei segnali e dei semafori. Infine, ci sono le informazioni relative alle risorse, ad esempio i file aperti, device allocati al processo etc.

Scheduler:

Lo scheduler rappresenta la componente più importante del kernel, che gestisce l'avvicendamento dei processi decidendo quale processo deve andare in esecuzione ogni volta che esso viene richiamato. Lo scheduler interviene quando viene richiesta un'operazione di I/O e quando un'operazione di I/O termina. Infatti, per fare una operazione di I/O un processo chiama una syscall (che viene gestita esattamente come una trap) e il processo viene bloccato, e dovrà aspettare che la operazioni di I/O

termini. Essendo bloccato, il gestore della syscall chiamerà lo scheduler (dopo aver bloccato il processo, questo togliendolo dalla readyQueue) al quale toccherà scegliere un nuovo processo. Quando l'operazione di I/O termina, l'interrupt arriva e sblocca il processo che era bloccato nella operazione di I/O, rimettendolo nella readyQueue.

Come avevamo detto nella lezione del 26/02, l'interval timer si comporta come un dispositivo I/O, nel senso che quando il tempo finisce, arriva un interrupt che sblocca tutti i processi bloccati (che per esempio avevano fatto una wait). [Teniamo a mente la differenza fra interrupt e trap: le trap avvengono quando un processo commette un errore oppure chiama una syscall, e sono sincrone. Gli interrupt invece terminano le operazioni di I/O, e sono asincrone].

N.B.: Il termine "scheduler" viene utilizzato anche in altri ambiti con il significato di "gestore dell'avvicendamento del controllo", possiamo quindi fare riferimento allo "scheduler del disco", e in generale allo "scheduler del dispositivo X".

[**FUN FACT**: ogni volta che colpiamo la tastiera con un dito per scrivere un carattere, avviene un interrupt che normalmente non viene consegnato al processo per efficienza. La tastiera viene mantenuta in modalità raw.]

Con *schedule* si intende la sequenza temporale di assegnazioni delle risorse da gestire ai richiedenti, con *scheduling* l'azione di calcolare uno schedule, e con *scheduler* la componente software che calcola lo schedule.

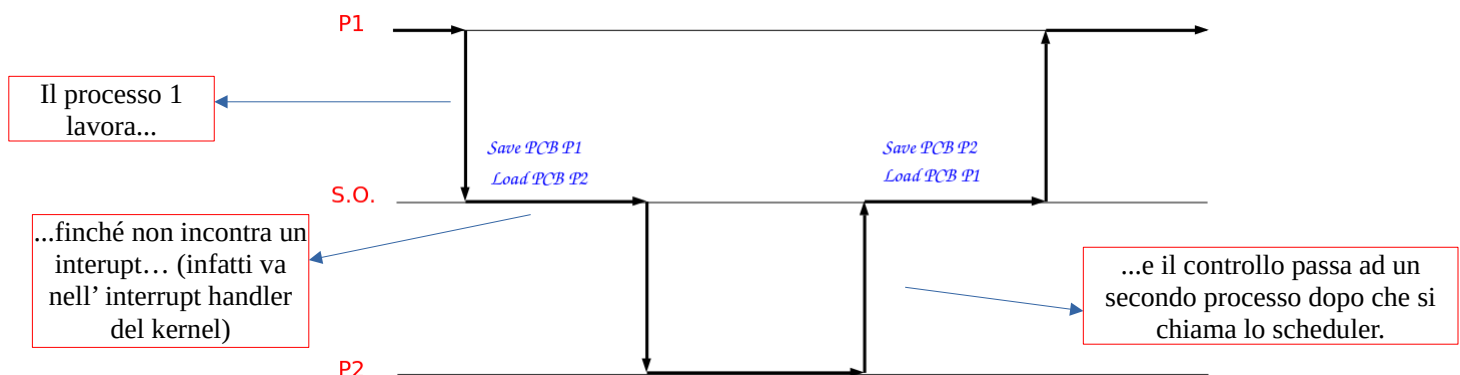
Mode switching e context switching:

Tutte le volte che un tutte le volte che avviene un interrupt (software o hardware) il processore è soggetto non solo al salto, ma anche ad un **mode switching**. È infatti fondamentale per fare in modo che i sistemi siano affidabili che si garantisca la presenza di un *modo kernel* e di un *modo user*, che rappresentano uno stato in cui lavora un processore. Quando la CPU è in modo user, può solo fare operazioni aritmetico-logiche con la memoria che gli è stata assegnata, e operazioni come accesso diretto ai bus, operazioni di I/O, cambiamento delle tabelle della MMU non le può fare. Quando però avviene un interrupt o una trap, avviene un mode switching, ovvero si cambia da modalità user a modalità kernel (siccome quando si trova all'interno dell'interrupt handler sta eseguendo codice kernel). A questo punto, dopo che l'interrupt è stato gestito, si chiama lo scheduler, che sceglie un processo fra quelli pronti (nella readyQueue) e lo prepara. [Qui il prof ha fatto un riferimento a pokemon, dicendo che lo scheduler fa come ash quando lancia pikachu dicendo "scelgo te". It was pretty funny ngl].

Quando avviene un interrupt, nello stato del processo viene salvato anche il modo in cui esso stava operando; questo perché potremmo avere interrupt nidificati, e quindi quando il controllo viene ripristinato, deve essere ceduto a un processo che si trovava in kernel mode.

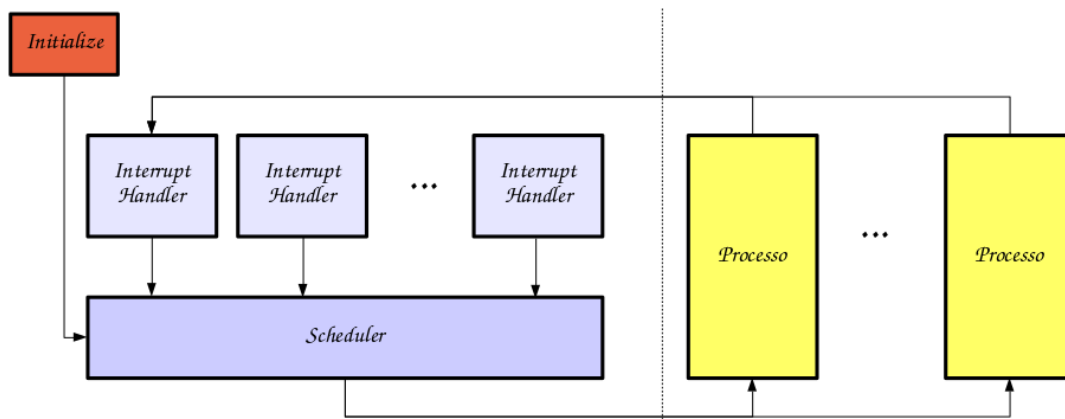
Se dopo la gestione dell'interrupt viene chiamato lo scheduler che chiama un altro processo, diverso da quello interrotto, si dice che il processo è soggetto a **context switching**. Durante questo evento, lo stato del processo attuale viene salvato nel PCB corrispondente, mentre lo stato del processo selezionato per l'esecuzione viene caricato dal PCB nel processore (ovvero l'LDST che facevamo nella Fase 2 di PandOS). Se non avviene context switching, possiamo ricaricare lo stato direttamente da un buffer temporaneo in cui è stato salvato lo stato della CPU al momento dell'interrupt.

Ecco un esempio di context switching:



Funzionamento di un kernel:

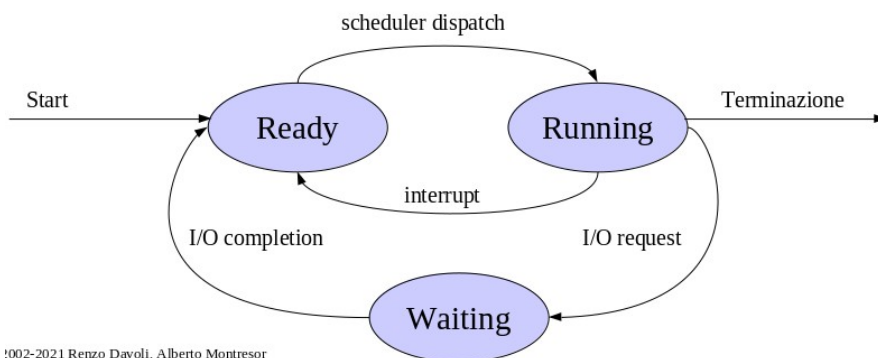
Dunque, ora sappiamo più o meno come funziona la gestione dei processi in un kernel. Qua sotto è riportato un ottimo schema del funzionamento del kernel che riassume ciò che succede.



La parte sopra in arancione corrisponde alla fase di inizializzazione delle strutture dati necessarie al funzionamento del sistema (es. tabelle dei processi, tabelle di memoria, tabelle dei device...) e viene creato poi "a mano" lo stato del primo processo, che verrà inserito nello scheduler, ovvero il processo init. L'init è il processo 1, e in molti sistemi l'init che viene chiamato si chiama "systemd". Una volta finita l'inizializzazione, viene chiamato lo scheduler, che avendo un solo processo da poter scegliere nei processi ready, sarà obbligato a scegliere quello. A quel punto, il controllo sarà passato al processo 1 (che è il processo scelto dallo scheduler, ovvero init) e il controllo ritornerà al kernel tramite un salto all'interrupt o trap handler a seconda di quello che succederà.

I processi possono trovarsi in 3 stati diversi:

- **Running:** il processo è in esecuzione
- **Waiting (o Blocked):** il processo è in attesa di qualche evento esterno (e.g., completamento operazione di I/O); non può essere eseguito, e quindi chiamato dallo scheduler.
- **Ready:** il processo può essere eseguito, ma attualmente il processore è impegnato in altre attività.



Lol non voglio
rimuovere il copyright
get fucked

!002-2021 Renzo Davoli, Alberto Montresor

I vari stati dipendono dalla presenza o no del nostro processo nella struttura dati (solitamente una coda, readyQueue) che contiene i processi "ready", che possono essere chiamati dallo scheduler. Quando un processo è bloccato per qualche motivo, lo si mette nella coda del semaforo in cui esso è bloccato (come abbiamo visto nelle lezioni del primo semestre) in modo che non sia nella coda ready, cosicché lo scheduler non possa mai sceglierlo. Quando il processo viene risvegliato, invece, lo si prende dalla coda del semaforo e lo si inserisce nella coda dei processi ready, in modo che possa così essere scelto dallo scheduler.

Se la coda ready è vuota, si ha una situazione "fisiologica" (e non patologica) se tutti i processi stanno aspettando I/O. I processi infatti in questo caso sono "vivi" ma bloccati. Così, lo scheduler, siccome non avrà processi da scegliere, può per esempio decidere di fare busy waiting in attesa, ad esempio, di un interrupt. Tutta via, il busy waiting, come abbiamo visto, consumano energia e scaldano la CPU

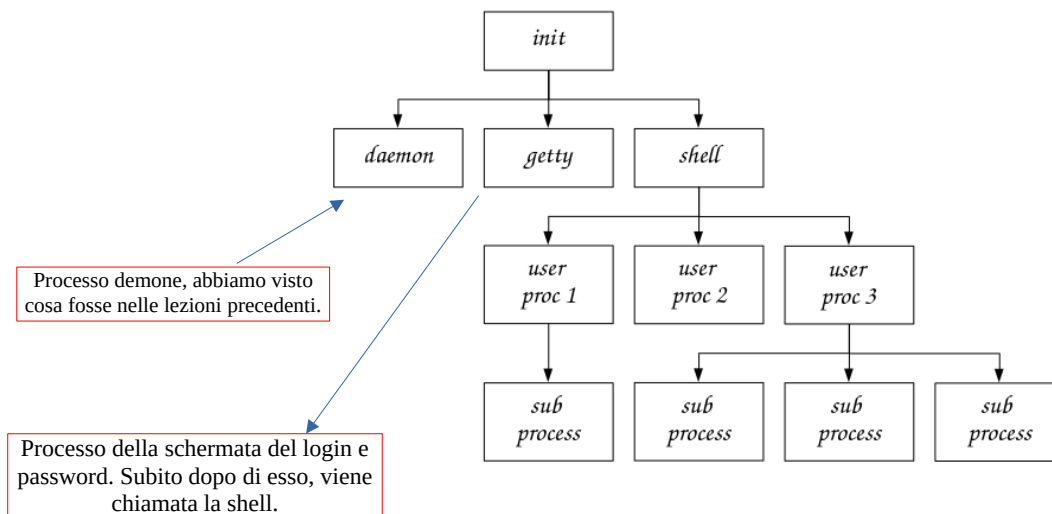
senza fare essenzialmente nulla, dunque non è la cosa più ideale. Dunque, il processore, al posto di essere in stato running, va in uno stato di standby, ovvero lo stato **idle**, in cui aspetta un interrupt e che un processo essenzialmente si sblocchi, senza fare una attesa attiva ma quasi spegnendosi, e aspetta il primo interrupt.

Appena un processo nasce, questo è posto nella readyQueue e quando lo scheduler lo sceglie il processo prende controllo della CPU, e a sto punto può accadere una syscall non bloccante (es. getpid, gettime) oppure il processo può chiamare una syscall bloccante, oppure finisce un time slice o avviene un altro interrupt.

Gerarchia dei processi:

In UNIX, i processi vengono organizzati in **forma gerarchica**. Quando un processo viene creato infatti, il processo creante è chiamato processo padre mentre quello che viene creato è il figlio. In questo modo, si crea un albero di processi. La ragione di questa scelta, sta nel fatto che in questo modo le caratteristiche del processo vengano passate al processo figlio, ed occorra specificare solamente in cosa il processo figlio differisce dal genitore, in modo che così si semplifica il processo. Inoltre, se un processo crea altri processi e occorre terminare il processo padre, con questa astrazione anche i processi figli vengono terminati tutti facilmente, senza dover cercare le dipendenze ed eliminarli tutti uno alla volta.

Questa è una schematizzazione della gerarchia dei processi:



Processi e thread:

Finalmente, chiariamo una volta per tutte la differenza fra processo e thread. Quando si parla di programmazione concorrente, come facevamo nel primo semestre, si parlava esclusivamente di processo; questa nozione di processo assume che ogni processo abbia una singola linea di controllo, e quindi una singola sequenza di istruzioni. Dunque, un singolo processo non può eseguire diverse attività contemporaneamente, ma solo una.

[**METANOTA**: il prof spiega il concetto in modo non troppo accurato, quindi mi sono limitato a copiare ciò che ho trovato su internet in un documento che lo spiegava molto bene.]

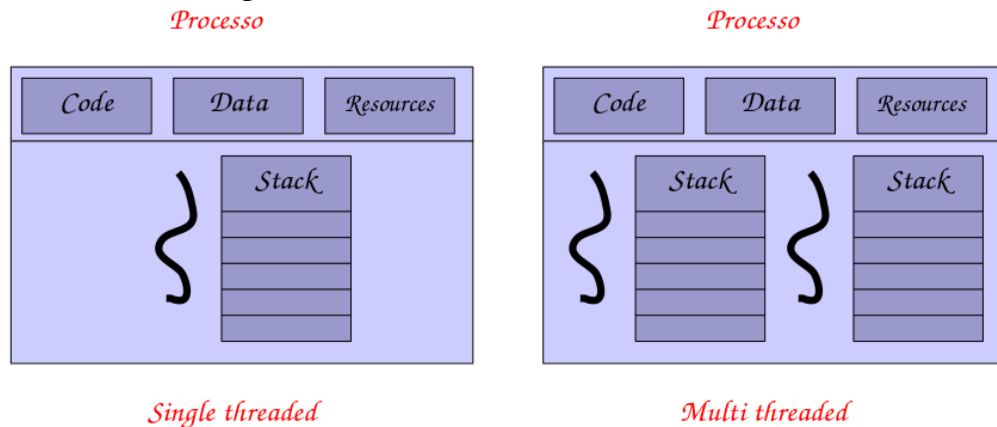
Quando invece si parla di progettazione del sistema operativo, si parla invece di thread, che è un considerato un concetto diverso da quello di processo (un tempo non era così, le definizioni erano più flessibili). L'idea di thread è una idea più "granulare" rispetto a quella di processo, ovvero mentre per **processo** si intende essenzialmente una un'istanza di un programma in esecuzione in modo sequenziale, con **thread** si intende invece l'unità granulare in cui un processo può essere suddiviso (sottoprocesso). In altre parole, un thread è una parte del processo che viene eseguita in maniera concorrente ed indipendente internamente allo stato generale del processo stesso. Il termine inglese rende bene l'idea, in quanto si rifà visivamente al concetto di fune composta da vari fili attorcigliati: se la fune è il processo in esecuzione, allora i singoli fili che la compongono sono i thread.

Una differenza sostanziale fra thread e processi consiste nel modo con cui essi condividono le risorse: mentre i processi sono di solito fra loro indipendenti, utilizzando diverse aree di memoria ed interagendo soltanto mediante appositi meccanismi di comunicazione messi a disposizione dal

sistema, al contrario i thread di un processo tipicamente condividono le medesime informazioni di stato, la memoria ed altre risorse di sistema.

Per come abbiamo visto per ora nel primo semestre, ci siamo comportati come se un processo avesse un solo thread, ma in realtà un processo oggi può avere più thread che appartengono allo stesso processo e condividono le stesse risorse. Questo implica che possiamo avere un grado di concorrenza all'interno del processo stesso. Un esempio di processo multithread potrebbe essere un browser che scarica due o più pagine web contemporaneamente.

Un thread può essere visto come l'unità base di utilizzazione della CPU, infatti ogni thread possiede anch'esso la propria copia dello stato di un processore, il proprio program counter e uno stack separato. I thread che appartengono allo stesso processo condividono fra loro il codice, i dati e le risorse I/O. I thread possono essere forniti direttamente dal kernel, infatti quando abbiamo visto i PCB, possiamo notare che sono divisi in due parti: le parti di identità e di accesso alle risorse del processo fanno parte del processo, mentre invece abbiamo puntatori alla tabella dei thread che contengono lo stato d'avanzamento. Quindi alla fine, in un sistema operativo che supporta multithreading a livello kernel, lo scheduler della CPU non schedula più i processi, ma schedula i thread di ogni processo. Non è più il processo a stare in stato running o blocked, bensì il thread.



Ultimamente, si fa uso soprattutto dei thread che sono forniti dal kernel. In realtà però si può implementare multithread all'interno del codice: anche un kernel che non fornisce multithread a livello kernel si può avere multithread creati da una libreria a livello del processo. In questo modo si possono avere più thread creati da una libreria, ma in realtà il kernel ne vede uno solo e per fare questo si inserisce all'interno del codice una serie di istruzioni che emulano uno scheduler, e che alterna quindi l'esecuzione del codice dei propri thread.

Il benefici del multithreading sono la condivisione dello stesso spazio di memoria e delle risorse che sono allocate degli altri thread dello stesso processo. Inoltre, scrivere codice multithread è più facile: pensiamo all'SSH; questo fa due cose: tutto quello che mandiamo da tastiera lo manda alla macchina remota, e tutto quello che arriva dalla macchina remota deve essere scritto sullo schermo. Questo possiamo implementarlo in due modi: un modo, quello più semplice, è quello di creare due thread, uno per la scrittura dei caratteri da tastiera e l'altro per la ricezione dalla seconda macchina, entrambi indipendenti. L'altro modo, molto più complesso, è quello di usare la programmazione ad eventi (event-driven) che abbiamo visto nel primo semestre e fare uso di syscall come `pause()` e `select()`.

Allo stesso modo, se io facessi due processi al posto che due thread, le cose diventerebbero molto più pesanti in quanto non ci sarebbe una condivisione totale delle risorse, ma la memoria, risorse e codice sarebbero separati. Inoltre, fare context switching fra processi diversi è costoso rispetto a fare context switching fra i thread.

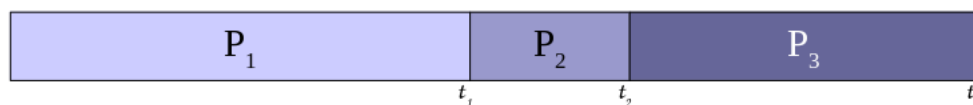
Riprendendo il discorso dei kernel thread, un SO può implementare i thread in due modi diversi:

- A livello utente (**user-thread**). In questo modo vengono implementati con una libreria che crea i thread, e questa libreria è come se che creasse un mini sistema operativo per i suoi thread, con uno scheduler che sceglie quale thread far procedere. È molto efficiente siccome non ci sono context switch fatti dal kernel. Occorre però che i programmi siano scritti in maniera tale da prevenire il blocco, siccome anche solo una syscall bloccante che blocca un thread, bloccherebbe anche l'intero processo in questo caso. Inoltre, sono molto efficienti.
- A livello kernel (**kernel-thread**). Questi sono supportati direttamente dal sistema operativo, e la creazione, gestione e scheduling dei thread sono implementati a livello kernel. Il vantaggio di questi sta nel fatto che poiché è il kernel a gestire lo scheduling dei thread, se un thread esegue una operazione di I/O, il kernel può selezionare un altro thread in attesa di essere eseguito. Tuttavia, i kernel-thread sono piuttosto lenti siccome si ha un context switch vero e proprio fra processi quando si chiama un nuovo thread.

Ci sono kernel che implementano tutto come user thread, dove un certo numero di user thread vengono mappati su un solo kernel thread (**Many-to-One Multithreading**) oppure possiamo fare uso diretto di ogni kernel thread, e quindi ogni user thread viene mappato su un kernel thread (**One-to-One Multithreading**), tuttavia ciò può creare problemi di scalabilità. Si può anche adottare un approccio misto, in cui si fa uso sia di kernel thread che di user thread (**Many-to-Many Multithreading**), anche se ora è poco comune come approccio.

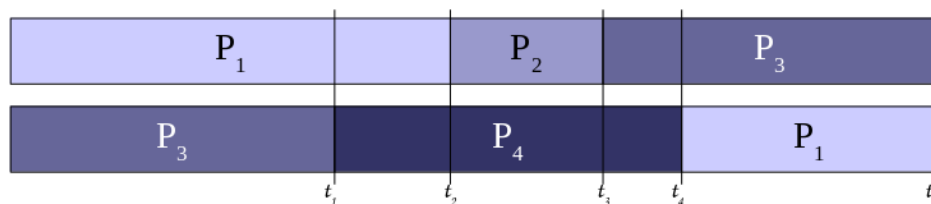
Scheduling:

Per realizzare uno scheduler, si fa uso di un diagramma di Gantt, che indica l'evolversi del possesso della CPU nel tempo:



Il tempo parte da sinistra (dove $t=0$) e va verso destra. In questo caso, il processore dal tempo 0 al tempo t_1 esegue P1, poi succede qualcosa (es. una chiamata bloccante) e lo scheduler al tempo t_1 sceglie P2 che viene eseguito fino al tempo t_2 etc...

Se per caso dobbiamo rappresentare lo schedule di più risorse (e.g., un sistema multiprocessore) il diagramma di Gantt risulta composto da più righe parallele:



Possiamo avere vari eventi che possono causare un context switch, ma in generale si ha un context switch quando:

1. quando un processo passa da stato running a stato waiting (system call bloccante, operazione di I/O, aspetto un segnale...).
2. quando un processo passa dallo stato running allo stato ready (a causa di un interrupt).
3. quando un processo passa dallo stato waiting allo stato ready.
4. quando un processo termina.

Nei casi 2 e 3, lo scheduler PUÒ scegliere di fare un context switch, ma non è obbligato, ovvero può esserci un mode switch senza context switch (es. quando una operazione di I/O finisce, e un processo si sblocca, lo scheduler può decidere di non passare subito il controllo a sto scemo, ma di far continuare il processo corrente.)

Uno scheduler, in base a questo, può essere:

- **non-preemptive** o cooperativo se i context switch avvengono solo nelle condizioni 1 e 4, in altre parole: il controllo della risorsa viene trasferito solo se l'assegnatario attuale lo cede volontariamente (es. Windows 3.1, Mac OS y con y <= 9). Come dice davoli: la CPU si prende il processo e non lo cambia almeno che non sia obbligato. In questi casi, i processi devono essere tali da poter cambiare turno e tenersi tutta la CPU. Questo tipo di scheduler non richiede alcuni meccanismi hardware come ad esempio timer programmabili (es. interval timer).
- **preemptive** se i context switch possono avvenire in ogni condizione. In altre parole: è possibile che il controllo della risorsa venga tolto all'assegnatario attuale a causa di un evento (es. tutti gli scheduler moderni). In questo caso, comanda lo scheduler, e in questo modo il time sharing è implementabile. Permette di utilizzare al meglio le risorse.

In base a cosa dobbiamo scegliere uno scheduler? Abbiamo bisogno di uno scheduler che ottimizzi l'uso della CPU, siccome il tempo in cui la CPU è idle è essenzialmente tempo sprecato, perciò tale periodo deve essere minimizzato. Inoltre, lo scheduler dev'essere tale per cui numero di processi completati per unità di tempo (throughput) sia massimizzato, mentre il tempo che intercorre dalla sottomissione di un processo alla sua terminazione (**tempo di turnaround**) deve essere minimizzato. Altri criteri sono il tempo di attesa di un processo mentre esso si trova nella coda ready, che deve essere minimizzato, e il tempo di risposta ovvero il tempo che intercorre fra la sottomissione di un processo e il tempo di prima risposta, e questo deve essere minimizzato. Quest'ultima caratteristica è di particolare importanza oggi siccome siamo pieni di programmi interattivi.

Un processo, a prescindere dal suo scopo e quindi dal programma che deve eseguire, durante la sua esecuzione si avranno l'alternarsi di calcoli svolti dalla CPU (**CPU Burst**) e delle operazioni di I/O (**I/O Burst**). Sapendo ciò, i processi li possiamo dividere in **CPU Bound** se sono caratterizzati da CPU burst molto lunghi, altrimenti se sono caratterizzati da CPU Burst brevi si dicono **I/O Bound**. Esempi di processi CPU Bound possono essere un processo che fa raytracing, mentre un processo I/O bound potrebbe essere un database management system.

Algoritmi di scheduling:

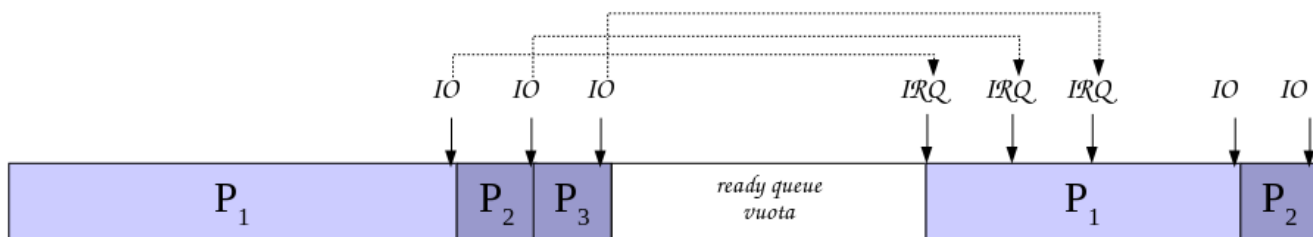
L'algoritmo di scheduling più brutale è il **First Come, First Served (FCFS)**, che consiste essenzialmente in una FIFO, ovvero il processo che arriva per primo, viene servito per primo. Questo tipo di algoritmo crea uno scheduler senza preemption, e può facilmente essere implementato tramite una coda. Tuttavia, presenta elevati tempi medi di attesa e di turnaround, e siccome non c'è la preemptiveness vengono ritardati i processi I/O Bound in favore dei CPU Bound.

Ecco un esempio:

- ordine di arrivo: P_1, P_2, P_3
- lunghezza dei CPU-burst in ms: 32, 2, 2
- Tempo medio di turnaround
 $(32+34+36)/3 = 34 \text{ ms}$
- Tempo medio di attesa:
 $(0+32+34)/3 = 22 \text{ ms}$



Supponiamo ora di avere un processo P_1 tale che sia CPU Bound, finché a un certo punto non viene bloccato da una operazione di I/O. Supponiamo poi che dopo che esso venga bloccato ci sia un certo numero di processi I/O bound tali che anche essi vengano bloccati da una operazione di I/O. In questo modo, la ready queue si svuota e la CPU sta in idle finché non avvengono degli interrupt, che sbloccheranno il processo. In questo caso, sarebbe molto più efficiente se i processi I/O Bound venissero eseguiti prima del processo CPU Bound, siccome mentre i processi I/O Bound aspettano l'interrupt, P_1 potrebbe fare i suoi bei calcoli.



Questo effetto, detto **convoy effect** (effetto convoglio) è uno dei problemi di questo tipo di scheduler, e consiste nel fatto che processi CPU burst brevi tendono a mettersi in coda con processi con CPU burst brevi. Ciò genera attese causate dalla disposizione dei processi, infatti processi brevi sono accodati a processi lunghi.

Per provare a risolvere questo problema, si è ideato l'algoritmo di scheduling **Shortest Job First (SJF)**, che appunto cerca di assegnare la CPU prima ai processi con CPU burst più corti. Anche questo algoritmo è senza preemption.

Ecco un esempio:

- Tempo medio di turnaround: $(0+2+4+36)/3 = 7$ ms
- Tempo medio di attesa: $(0+2+4)/3 = 2$ ms



Questo scheduler però risolve un problema e ne crea altri mille. Nonostante sia ottimale rispetto d'attesa, causa starvation! Infatti se ho solo processi I/O bound continuamente, quelli CPU bound non saranno mai scelti. Ma il vero problema sta nel fatto che questo è IMPOSSIBILE DA IMPLEMENTARE quindi tutto quello che il prof mi ha fatto scrivere adesso è solo puro tempo perso. In realtà, negli scheduler long term possiamo chiedere a chi sottomette un job di predire la durata dei job, ma in quelli short term non possiamo sapere la durata dei CPU burst futuri. In entrambi i casi però si potranno fare solo delle approssimazioni basate sulla durata dei processi passati. Faccio uno screen della slide perché 1) non mi sembra troppo importante 2) non ho voglia di fare quei simboletti uno a uno del c***o.

- **Calcolo approssimato della durata del CPU burst**

- basata su *media esponenziale* dei CPU burst precedenti
- sia t_n il tempo dell' n -esimo CPU burst e τ_n la corrispondente previsione; τ_{n+1} può essere calcolato come segue:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

- **Media esponenziale**

- svolgendo la formula di ricorrenza, si ottiene

$$\tau_{n+1} = \sum_{j=0..n} \alpha(1-\alpha)^j t_{n-j} + (1-\alpha)^{n+1} \tau_0$$

da cui il nome media esponenziale

t_n rappresenta la storia recente,

τ_n rappresenta la storia passata,

α rappresenta il peso relativo di storia passata e recente (compreso fra 0 e 1).

Cosa succede con $\alpha = 0$, 1 oppure $\frac{1}{2}$?

Nel caso in cui $\alpha = 1$, allora significa l'approssimazione che do al prossimo CPU burst è uguale alla durata del precedente. Se invece $\alpha = 0$ allora non do importanza alle ultime cose accadute ($\tau_{n+1} = \tau_0$).

Questa alla fine è l'unica formula che ci interessa

Essendo la media esponenziale, il peso del tempo precedente avrà sempre meno peso.

Esiste anche una versione preemptive della del SJF, secondo cui il processo corrente può essere messo nella coda ready, in caso arrivi un processo con un CPU burst più breve di quanto rimane da eseguire al processo corrente (magari arriva con un interrupt per esempio). Questo viene chiamato "Shortest-Remaining-Time First".

FINE.

Finalmente sta lezione è finita. Era lunghissima e noiosissima. Non ne posso più di avere lezioni teoriche. Se hai finito di leggere sta lezione direi che ti sei meritato una lunga pausa. Per favore, falla, per il bene della tua salute mentale.

Lezione del (9/03/2021) – Presentazione pandos, continuazione scheduling

Nota: mentre il prof parla di interval timer come dispositivo per gestire il timeslice, umps3 ha due livelli di interval timer: l'interval timer così detto e il processor local timer (PLT). Quest'ultimo è anch'esso un interval timer, ma è separato processore per processore, al contrario dell'interval timer che è globale. Quindi, se abbiamo una macchina monoprocesso noi possiamo fare finta che il PLT non ci sia, e possiamo usare l'interval timer per fare tutto.

Round Robin:

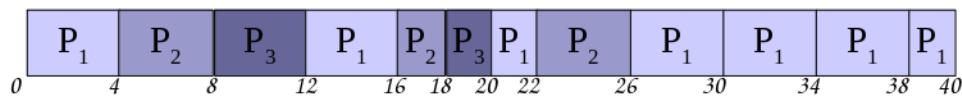
L'algoritmo di scheduling **Round Robin** è un algoritmo preemptive, che prende un processo e gli dà un quanto di tempo (time slice). Dunque, un processo perderà il controllo della CPU nel caso esegua una syscall bloccante o una I/O operation, ma anche se utilizza l'intero quanto tempo. Per implementare il quanto di tempo facciamo uso dell'interval timer (o nel caso di umps3, il processor local timer). Questo meccanismo risolve il problema di monopolizzazione della CPU che fanno molti calcoli, perché anche se ci fossero tanti processi che fanno tanti calcoli, ogni processo verrà comunque accodato nella coda ready quando il suo time slice sarà finito, passando il controllo a quello dopo. Round robin è stata una grande invenzione per far vivere processi in background e processi interattivi, e consente in maniera democratica di dividere la CPU fra tutti i processi in esecuzione.

La durata del quanto di tempo sarà però da tarare, in quanto se il quanto di tempo è breve, il sistema è meno efficiente perché deve cambiare il processo attivo più spesso (inoltre in questo il costo del context switch supera il costo del processo), mentre se il quanto è lungo, in presenza di numerosi processi pronti ci sono lunghi periodi di inattività di ogni singolo processo (cosa molto fastidiosa per gli utenti di sistemi interattivi).

Per implementare questo scheduling, come abbiamo accennato, è necessaria la presenza di un **interval timer**, che essenzialmente è un device I/O che non fa nulla se non aspettare in un tempo determinato esatto, e che si può caricare come un timer da cucina. Quando questo timer scade, viene generato un interrupt, si chiamerà così il gestore degli interrupt che si accorge che l'interval timer è scattato e che quindi un processo dovrà passare da quello in esecuzione a quello successivo. [teniamo a mente che i processi nella loro vita fanno sempre calcolo-I/O-calcolo in modo alternato, chi più e chi meno]

Ecco un esempio:

- tre processi P_1, P_2, P_3
- lunghezza dei CPU-burst in ms ($P_1: 10+14; P_2: 6+4; P_3: 6$)
- lunghezza del quanto di tempo: 4
- Tempo medio di turnaround $(40+26+20)/3 = 28.66$ ms
- Tempo medio di attesa: $(16+16+14)/3 = 15.33$ ms
 - NB (supponiamo attese di I/O brevi, < 2 ms)
- Tempo medio di risposta: 4 ms



Lo svantaggio di Round Robin è che è troppo democratico: infatti i processi non sono tutti uguali, e possono avere esigenze diverse. Per esempio: usando round-robin puro la visualizzazione di un video MPEG potrebbe essere ritardata da un processo che sta smistando la posta.

Un'alternativa a questo, è usare lo **scheduling a priorità**.

Lezione del (12/03/2021) – Scheduling a priorità, grafo di Holt, deadlock prevention

Note prima della lezione:

ogni volta che accendiamo il terminale, avviamo almeno due processi: il primo è il gestore della finestra del terminale (es. xterm, alacritty, konsole...) e il secondo è bash che gestisce il prompt che ha ricevuto il comando.

Il tempo dell'interrupt non viene addebitato al processo corrente, siccome “non è colpa sua”, e in questo periodo l'interval timer viene bloccato. Un discorso diverso è invece il caso per esempio lui voglia eseguire una syscall, allora il tempo di gestione viene addebitato al tempo del processo.

Quando avviene una syscall, si chiama il gestore di trap (che è codice del kernel) che gestisce la trap, ma al contempo si usa una tecnica di fare eseguire al processo codice del kernel in modalità kernel per gestire la syscall, e c'è un cosiddetto PassUp verso il livello di supporto in modo che questa elaborazione venga fatta dal codice kernel in modalità kernel, ma usando il timeslice del processo.

Utilizzando il comando `time`, possiamo vedere quanto tempo (e come) un processo impiega, il campo user indica il tempo impiegato in user mode, mentre il campo sys quello che ha impiegato in modalità kernel.

Scheduling a priorità:

Lo scheduling a priorità si basa sul fatto che viene assegnata una priorità a ciascun processo, e lo scheduler sceglierà per prima il processo pronto con priorità più alta. Le priorità possono essere:

- definite dal sistema operativo: vengono utilizzate una o più quantità misurabili per calcolare la priorità di un processo (sempio: SJF è un sistema basato su priorità).
- definite esternamente: le priorità non vengono definite dal sistema operativo, ma vengono imposte dal livello utente.

A loro volta, si possono avere diverse gestioni delle priorità:

- Priorità statica: la priorità non cambia durante la vita di un processo, questo può causare starvation che colpisce i processi a bassa priorità.
- Priorità dinamica: la priorità può variare durante la vita di un processo. È possibile utilizzare metodologie di priorità dinamica per evitare starvation.

Una delle politiche di priorità dinamica più popolare è la **priorità basata su aging**, che consiste nell'incrementare gradualmente la priorità dei processi in attesa. In particolare, questa politica fa uso di due priorità: *una priorità reale* e *una attuale*. Quando un processo viene inserito nella coda ready, si fa in modo che la priorità reale sia uguale a quella attuale. Quando lo scheduler sceglie il processo di priorità massima, aumenta di 1 la priorità attuale di tutti gli altri processi che non ha scelto. In questo modo, nessun processo rimarrà in attesa per un tempo indefinito perché prima o poi raggiungerà la priorità massima.

Esempio: prendiamo 3 processi, P, Q, R. P ha priorità reale 4, Q ha 3 e infine P ha 1.

Al tempo $t=0$, P, Q e R hanno priorità attuale rispettivamente 4, 3 e 1.

P viene scelto, la priorità attuale di Q ed R aumenta di 1, diventando così 4 e 2 rispettivamente.

P poi viene rimesso nella ready queue, con priorità attuale nuovamente 4 (verrà però posto prima di Q a causa dell'ordine temporale).

Viene così scelto Q, P aumenta a 5 e R aumenta a 3.

Quando Q torna nella ready queue, verrà posto prima di P e R, siccome la sua priorità verrà resettata a 3. Dunque P verrà scelto nuovamente, siccome sta davanti a Q. R aumenta a 4, e Q pure aumenta a 4. P torna, e viene accodato in fondo con priorità 4.

A sto punto viene scelto R, e così via...

Un approccio leggermente diverso è lo **scheduling a classi priorità**, che divide i processi in classi di priorità (duh lol), in base alle loro caratteristiche. La coda ready sarà così scomposta in molteplici "sottocoda", una per ogni classe di processi. Lo scheduler così prima sceglierà la classe, e poi seleziona il processo da eseguire fra quelli di questa sottocoda.

Una ulteriore evoluzione di questo concetto sta nello **scheduling multilivello**, in cui all'interno di ogni classe di processi, è possibile utilizzare una politica specifica adatta alle caratteristiche della classe (es. facciamo una classe per i processi multimediali, poi per i processi batch etc...).

Uno scheduler multilivello cerca prima la classe di priorità massima che ha almeno un processo ready, e sceglie poi il processo da porre in stato running coerentemente con la politica specifica della classe.

Ad esempio, per i processi di sistema possiamo mettere una priorità statica, e nello strato di batch possiamo mettere una politica round robin.

In questo modo, possiamo per esempio implementare lo stato di IDLE del sistema con un processo di priorità bassa, che viene scelto solo quando la readyqueue è vuota, senza nemmeno bisogno di fare un if.

Ecco un altro esempio, direttamente dalle epiche slides del prof:

Quattro classi di processi (priorità decrescente)

- processi server (priorità statica)
- processi utente interattivi (round-robin)
- altri processi utente (FIFO)
- il processo vuoto (FIFO banale)

[**METANOTA**: il prof ha spiegato questa parte giusto per curiosità, ha anche avvisato che non la chiederà mai, quindi dw dude :) a me sembra comunque intrinsecamente interessante però. Se hai ricevuto una copia .odt di questo file, puoi eliminare tutta questa parte che finisce all'inizio dell'headline gestione delle risorse.]

Lo **scheduling real-time** sono una categoria di speciale di scheduler applicata appunto nei **sistemi real-time**. Teniamo a mente che con sistemi real-time si intendono sistemi in cui la correttezza (o il funzionamento corretto) non dipende solamente dal risultato che outputta ma anche dal tempo in cui lo ritorna (es. deve ritornare 42 entro 200 anni). I sistemi hard real-time sono sistemi che richiedono obbligatoriamente che le condizioni vengano rispettate in modo ristretto, ovvero è un questione critica. Le deadline, quindi, non devono essere superate in nessun caso. Nei sistemi soft real-time, gli errori sono più tollerabili.

In questi sistemi abbiamo due tipi di processi:

- I processi periodici, ovvero processi che vengono riattivati con una cadenza regolare, detta periodo (esempi: controllo assetto dei velivoli (fly-by-wire), basato su rilevazione periodica dei parametri di volo rilevati attraverso sensori).
- I processi aperiodici, sono processi che vengono scatenati da un evento sporadico, ad esempio l'allarme di un rilevatore di pericolo.

I sistemi a real-time quindi non sono dei normali pc, sono dei sistemi studiati ad arte, e nei quali si sa a priori quali processi sono attivi in essi. Inoltre, i sistemi a real time hanno due stati di funzionamento: stato critico e stato non critico. Nelle centrali nucleari per esempio, si sa a sistema spento quali processi e come essi devono essere settati (es. con quanto tempo). I programmi di questi sistemi sono scritti con dei linguaggi fatti adhoc, che permettono di avere una alta accuratezza temporale e nel quale non esiste loop infinito.

Alcuni algoritmi di scheduling per questi sistemi così peculiari sono:

- Rate Monotonic: è una politica di scheduling a priorità fissa, valida alle seguenti condizioni:
 - ogni processo periodico deve completare entro il suo periodo
 - tutti i processi sono indipendenti
 - la preemption avviene istantaneamente e senza overhead
 - viene assegnata staticamente una priorità a ogni processo
 - processi con frequenza più alta (i.e. periodo più corto) hanno priorità più alta
 - ad ogni istante, viene eseguito il processo con priorità più alta (facendo preemption se necessario)

Il più famoso di questi scheduler è lo jule heiland (bho non so come si scrive) e in basa a una super formula riesce a capire in base ad una costante dipendente dal numero dei processi, è garantito che lo scheduler soddisfi i requisiti. È molto usato come tipo di scheduler.

- Earliest Deadline First: è una politica di scheduling per processi periodici real-time, in cui viene scelto di volta in volta il processo che ha la deadline più prossima. Viene detto "a priorità dinamica" perchè la priorità relativa di due processi varia in momenti diversi. Anche qui, c'è una specie di super formula: se la somma fra tempo di esecuzione e periodo è minore di 1, lo scheduler funziona.

Gestione delle risorse e deadlock:

[piccola digressione] Per capire initctl bisogna capire init. Init è il primo processo, progenitore di tutti gli altri processi. Init è quello tradizionale di UNIX, che prevede il concetto di livelli di esecuzione, e per funzionare c'era una tabella chiamata `/etc/inithub` che spiegava cosa fare in ogni livello. Man mano le distribuzioni hanno costumizzato il funzionamento di quello che veniva lanciato nei vari livelli operativi (runlevel, ovvero layer di funzionamento del sistema operativo che garantiscono alcuni servizi). Inithub ora non c'è più, ma ora possiamo vedere cosa fa ciascun livello grazie al comando `ls etc/rc*.d`, che mostra quale funzionalità devono essere attivate e quali no per ciascun runlevel (es. nel livello 2 viene attivato il server ssh etc). Initctl permetteva di configurare cosa partiva e cosa no nei vari livelli per init. Tutta questa cosa ora nelle varie distribuzioni è stata superata da un nuovo programma che si usa come processo init, e che si chiama systemd. Questo è diverso da init perché invece di essere un programma di init generale con file di configurazione come script, ha al suo interno molte funzionalità per tanti servizi. Fra init e systemd c'è una grande battaglia online, per esempio esiste debuan, che è una comunità di debian che crea un os basato su debian ma systemd free. Systemd infatti, nonostante sia comodo per alcune applicazioni, secondo i puristi di unix va contro la filosofia di unix stesso, siccome UNIX si basa sul fatto che "bisogna fare applicazioni che facciano una cosa, la facciano bene e che siano minimali per fare quella cosa", siccome cerca di inglobare molte cose diverse. Il prof ha anche litigato con systemd. Systemd spesso automatizza molte cose, ma nel fare ciò crea più danno che beneficio. In un sistema con systemd, al posto di usare initctl, bisogna usare systemctl.

Gestione delle risorse:

Possiamo avere risorse fisiche oppure risorse logiche, possiamo vedere le risorse del sistema come le risorse che usa il cuoco della cucina di Brachetti.

Conosciamo tutti degli esempi di risorse fisiche, ad esempio memorie, stampanti, dischi... mentre degli esempi di risorse logiche possono essere per esempio i descrittori dei processi (process control block). Nella fase 1 per esempio avevamo un massimo di 20 descrittori di processi, e se questi finiscono implica che non possiamo creare dei nuovi processi.

Definizioni sulla gestione di risorse:

Possiamo dividere le risorse in classi, si parla infatti di risorse **fungibili** e **infungibili**. Con questi due termini, usati anche in economia quando si parla di beni fungibili e infungibili. Per esempio, le banconote sono beni *fungibili*, ovvero che se bisogna pagare qualcosa che costa 20€, possiamo usare una qualsiasi banconota da 20€ per pagarla, non c'è bisogno di una banconota con un numero seriale preciso o altro, una banconota vale l'altra.

Le risorse di elaborazione appartengono ad una classe quando sono equivalenti per l'elaborazione (quindi quando tutte le risorse sono fungibili). Se per esempio abbiamo bisogno di un descrittore di processo, un qualsiasi descrittore di processo va bene e quindi tutti i descrittori di processo appartengono alla stessa classe (e quindi sono anche fungibili).

Le risorse di una classe vengono dette **istanze** della classe (es. un descrittore del processo è una istanza della classe dei descrittori di processo). Il numero di risorse in una classe viene detto **molteplicità** del tipo di risorsa.

Le risorse, possono essere ad assegnazione statica o dinamica:

- se sono ad **assegnazione statica** vuol dire che l'assegnazione avviene al momento della creazione del processo e rimane valida fino alla sua terminazione, ovvero per tutta la vita del processo. (es. descrittori di processi, aree di memoria (in alcuni casi, per esempio processi che lavorano ad indirizzamento assoluto, quell'area di memoria deve essere dedicata. Questa casistica è presente per esempio nei sistemi embedded))
- se sono ad **assegnazione dinamica**, invece, i processi richiedono le risorse durante la loro esistenza, le utilizzano una volta ottenute e le rilasciano quando non più necessarie (eventualmente alla terminazione del processo), e che quindi possono essere date e tolte in qualsiasi momento durante l'esecuzione del processo. Alcuni esempi: periferiche di I/O, aree di memoria (in alcuni casi).

Le richieste di una risorsa da parte di un processo possono essere di tre tipi:

- **Richiesta singola**: si riferisce a una singola risorsa di una classe definita, ed è il caso normale e più frequente.
- **Richiesta multipla**: si riferisce a una o più classi, e per ogni classe, ad una o più risorse (Il caso in cui ho una richiesta multipla a una singola classe sarà quella che ci interesserà di più). Questa deve essere soddisfatta integralmente (altrimenti il processo potrebbe semplicemente fare due richieste), quindi in questo caso o gli si vengono date tutte le risorse e può continuare, oppure deve aspettare (viene bloccato).

La richiesta può essere anche **bloccante**, questo avviene quando il processo richiedente non ottiene immediatamente l'assegnazione e quindi viene sospeso finché la risorsa non è disponibile. La richiesta rimane pendente e viene riconsiderata dalla funzione di gestione ad ogni rilascio.

Altrimenti, si può anche fare in modo che la richiesta sia non bloccante nonostante la risorsa non sia disponibile; in questo caso la mancata assegnazione viene notificata al processo richiedente, senza provocarne la sospensione.

Le risorse poi possono essere **non condivisibili** (o seriali) quando una singola risorsa non può essere assegnata a più processi contemporaneamente [es. *processori* (a noi appare che venga usato da più processi contemporaneamente, ma è un'illusione data dalla velocità di avvicendamento di ogni processo, ogni processore infatti viene usato da un processo alla volta), *sezioni critiche* (considerabile una risorsa

logica), stampanti], mentre se sono **condivisibili** possono essere assegnati a più processi (es. i file in sola lettura).

Una risorsa può essere **prerilascibile** (preemptable) se la funzione di gestione può sottrarla ad un processo prima che questo l'abbia effettivamente rilasciata. **Ciò vuol dire che mentre un processo sta usando una risorsa, qualcun altro può prendergliela sospendendo l'esecuzione di tale processo rapinato, che stava usando la risorsa.** La risorsa che è stata rubata verrà restituita in futuro. Una risorsa è prerilascibile quando il suo stato non viene modificato durante il suo utilizzo, oppure il suo stato può essere facilmente salvato e poi ripristinato. Alcuni esempi di risorse prerilascibili sono: il *processore* (infatti quando avviene un interrupt o una trap viene salvato lo stato del processore che poi viene ripristinato per far continuare il processo che era stato sospeso) e *blocchi/partizioni di memoria* (nel caso dell'assegnazione dinamica, oppure se c'è bisogno della memoria centrale posso salvare il contenuto della memoria centrale all'interno della memoria secondaria, fermare il processo che usa il contenuto di quella zona della memoria centrale, per poi farlo ripartire e ripristinare il contenuto della memoria centrale).

Allo stesso modo, le risorse possono anche essere non-prerilascibili, nel caso appunto la funzione di gestione NON possa sottrarla al processo al quale sono assegnate. Le risorse non prerilascibili sono le risorse il cui stato non può essere salvato o ripristinato. Es. stampanti, classi di sezioni critiche (il concetto di sezione critica si basa proprio su questo come abbiamo visto) e partizioni di memoria nel caso della memoria statica (in questo caso infatti non possiamo salvare le cose in memoria secondaria, **[METANOTA:** non sono sicuro di quest'ultima cosa] siccome in questo caso non possiamo spostare la memoria essendo statica).

Deadlock:

Abbiamo visto in programmazione concorrente i problemi di programmazione concorrente, vediamo ora di dare una definizione operativa di deadlock. Il deadlock è una situazione patologica del sistema. Le condizioni per avere un deadlock sono:

1. **Mutua esclusione / risorse non condivisibili**, ovvero le risorse coinvolte devono essere non condivisibili.
2. **Assenza di prerilascio**: ovvero le risorse coinvolte non devono essere prerilascibili, ovvero devono essere rilasciate volontariamente dai processi che la controllano.
3. **Richieste bloccanti** (o "hold and wait"): le richieste devono essere bloccanti, e un processo che ha già ottenuto risorse può chiederne ancora.
4. **Attesa circolare**: ovvero esiste una sequenza di processi P_0, P_1, \dots, P_n , tali per cui P_0 attende una risorsa controllata da P_1 , P_1 attende una risorsa controllata da P_2 , ..., e P_n attende una risorsa controllata da P_0 ... (essenzialmente quello che succedeva durante il problema dei filosofi).

Ognuna di queste condizioni è necessaria, mentre l'insieme delle condizioni è sufficiente per avere deadlock, se una di queste condizioni non c'è, allora non abbiamo deadlock: devono valere tutte contemporaneamente.

Possiamo decidere di prevenire o curare il deadlock. Curare non è possibile, in quanto se dovessimo curare il deadlock dovremmo rompere qualcosa (i.e. uccidere un processo). Dunque non ci rimane che provare a prevenire il deadlock.

Possiamo andare a vedere se c'è deadlock o meno, e provare a fare una vera e propria prevenzione del deadlock oppure evitare che possa succedere. Ma come può fare un sistema a mettere in piedi un meccanismo per notificare al sysadmin che sta per avvenire un deadlock? Qui viene in nostro aiuto il grafo di Holt.

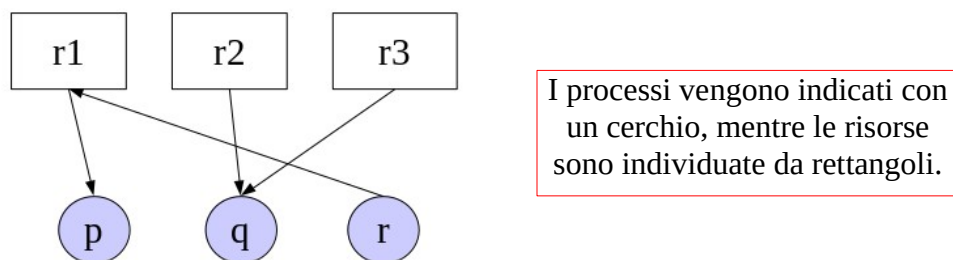
Grafo di Holt:

Il **grafo di Holt** è un grafo diretto (ovvero orientato) e bipartito.

Per capire cosa si intende con bipartito, dobbiamo considerare che noi possiamo definire un grafo come coppia: un insieme e una relazione fra questo insieme. Il primo è detto l'insieme dei nodi, e la relazione fra i nodi è l'insieme degli archi. L'insieme dei nodi è *partizionato* in due sott'insiemi (con partizionato si intende che esistono due sott'insiemi la cui unione dà l'insieme dato e la cui intersezione dà l'insieme vuoto). Tutti gli archi hanno origine e destinazioni in due partizione diverse, ovvero non possono esistere archi che hanno origine e destinazione nella stessa partizione. I due sott'insiemi dell'insieme dei nodi sono *l'insieme delle risorse e l'insieme dei processi*. Avremo quindi:

- archi che vanno da una risorsa a un processo che indicano che la risorsa è assegnata al processo (**risorsa** → **processo**)
- archi vanno da un processo ad una risorsa, che indicano che il processo ha richiesto la risorsa (**processo** → **risorsa**)

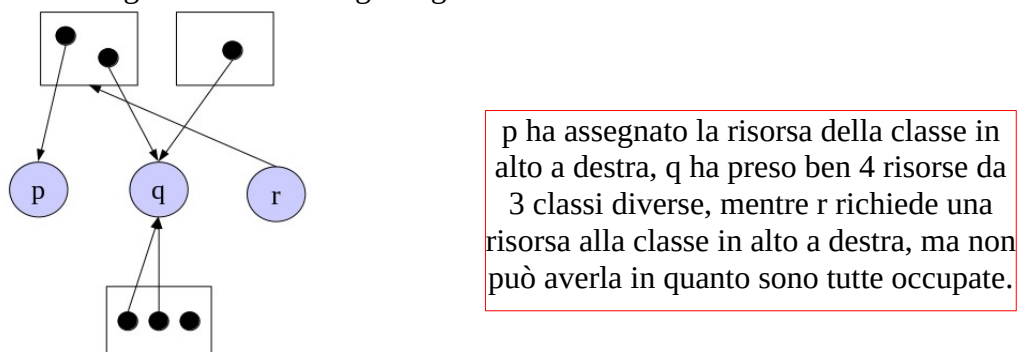
Ecco un esempio di un grafo di Holt:



Nell'esempio qua sopra ipotizziamo che ogni risorsa abbia una sola istanza, ma nella realtà non è esattamente così..

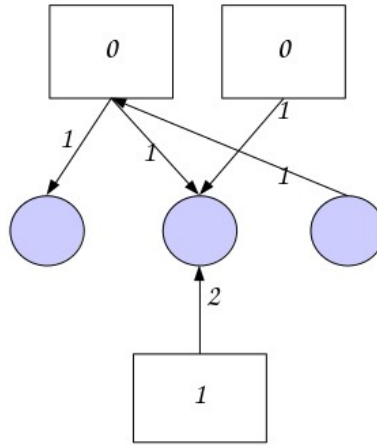
Nel caso di classi contenenti più istanze di una risorsa l'insieme delle risorse è partizionato in classi e gli archi di richiesta sono diretti alla classe e non alla singola risorsa. Graficamente, se abbiamo classi di risorse queste vengono rappresentate come contenitori rettangolari, e le risorse come punti all'interno delle classi (più in particolare, ogni punto rappresenta un'istanza della classe di risorsa). Le frecce andranno o da un processo verso l'intera classe delle risorse, oppure da un'istanza della risorsa verso il processo. Nota importante: se le risorse sono disponibili, i grafi di Holt non riportano mai la freccia dal processo verso la risorsa, bensì immediatamente viene assegnata la risorsa e perciò metto la freccia dalla risorsa verso il processo (non si rappresentano grafi di Holt con archi relativi a richieste che possono essere soddisfatte).

Ecco la rappresentazione grafica del nostro grafo generale:



Il grafo da noi disegnato in questo modo, nella macchina viene memorizzato come un **grafo pesato** (con pesi ai nodi di risorsa e pesi sugli archi). I numeri nel rettangolo della classe delle risorse indicano la quantità di risorse disponibili, mentre il peso sugli archi indica la molteplicità dell'assegnazione o richiesta.

Ecco il grafo di prima con notazione operativa:



Come dicevamo prima, possiamo gestire il deadlock in più modi:

- **Deadlock detection and recovery:** permettere al sistema di entrare in stati di deadlock; utilizzare un algoritmo per rilevare questo stato ed eventualmente eseguire un'azione di recovery. Ma come dicevamo prima, questa è una recovery distruttiva.
- **Deadlock prevention / avoidance:** impediamo al sistema di entrare in uno stato di deadlock, prevenendolo o evitandolo. Previnire ed evitare hanno una leggera sfumatura di differenza, ma questo lo vedremo più avanti (in uno lo evitiamo strutturalmente, nell'altro si usano meccanismi dinamici per evitarlo).
- **Ostrich algorithm (Algoritmo dello struzzo):** ignorare il problema del tutto! Sorprendentemente, questo è applicato da molti sistemi operativi maggiori.

Deadlock detection:

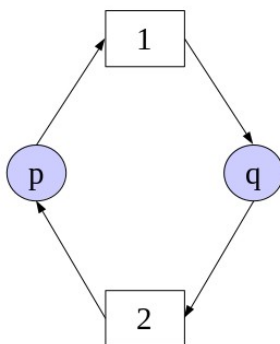
Come facciamo a riconoscere uno stato di deadlock?

Prima di tutto, facciamo in modo che il sistema, tutte le volte che fa una richiesta, aggiorni il grafo di Holt, lasciando la richiesta se non può essere soddisfatta correntemente oppure soddisfacendo le richieste mettendo le varie frecce etc..

In base a come è strutturato il grafo di Holt, possiamo determinare se c'è un deadlock. Esaminiamo i vari casi:

Nel primo caso (grafo monorisorsa), se abbiamo una sola risorsa per classe e se le risorse sono ad accesso mutualmente esclusivo, non condivisibili e non preilasciabili, **lo stato è di deadlock se e solo se il grafo di Holt contiene un ciclo**. Per vedere graficamente se il nostro grafo contiene un ciclo, si utilizza una variante del grafo di Holt, detto **grafo Wait-For**, che si ottiene eliminando i nodi di tipo risorsa e collassando gli archi appropriati. Il grafo di Holt contiene un ciclo se e solo se il grafo Wait-for contiene un ciclo, e se il grafo Wait-for contiene un ciclo, abbiamo attesa circolare.

Ecco una rappresentazione grafica:



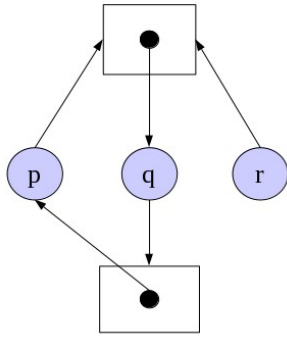
Grafo di Holt



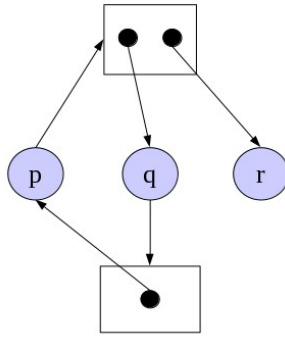
Grafo Wait-For

Se ci incentriamo sul significato simbolico degli elementi del grafo di Holt, potremmo interpretare questo ciclo come p che richiede 1, ma 1 è assegnato a q, e q richiede 2, ma 2 è assegnato a p. Quello che succede è analogo al significato che abbiamo attribuito alla deadlock nel primo semestre.

Nel secondo caso (grafo multirisorsa), quando abbiamo una sola risorsa per classe, la situazione si fa più complessa. Infatti, i cicli non ci vengono (totalmente) in nostro aiuto, siccome **sono condizione necessaria ma non sufficiente per avere deadlock**.



Deadlock



No Deadlock

In entrambi i casi abbiamo dei cicli, ma non abbiamo deadlock in entrambi i casi.

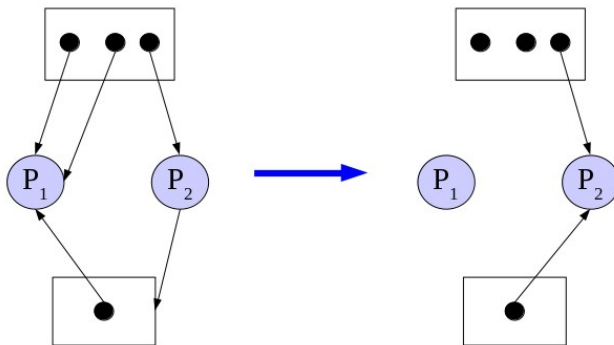
Nel secondo caso il deadlock può essere risolto perché il processo a destra (r) può lasciare la risorsa, che potrà poi essere allocata a p, eliminando la deadlock.

Nel grafo a destra, invece, r richiede la risorsa e basta, e quindi non migliora la situazione.

Come fa quindi a vedere se c'è deadlock avendo un grafo multirisorsa? Per rispondere a questa domanda, dobbiamo introdurre il significato di grafo riducibile. Un grafo di Holt si dice **riducibile** se se esiste almeno un nodo processo con solo archi entranti (ciò implica che esiste un processo che allo stato attuale ha tutte le risorse che gli servono e quindi può completare l'esecuzione).

Per **ridurre** un grafo di Holt, consiste nell'eliminare tutti gli archi di tale nodo e riassegnare le risorse ad altri processi che ne facevano richiesta. Qual è la logica? La logica sta nella "ricerca di una via d'uscita": **se un processo in questo momento ha tutto ciò che gli serve, allora potrà completare la sua operazione**, e la risorsa potrà essere riassegnata.

Ecco un esempio di riduzione:



Riduzione per P_1

Possiamo ridurre il nodo P1 in quanto possiede solo archi entranti. Per P2, la situazione è diversa, siccome ha un arco entrante ed uno uscente.

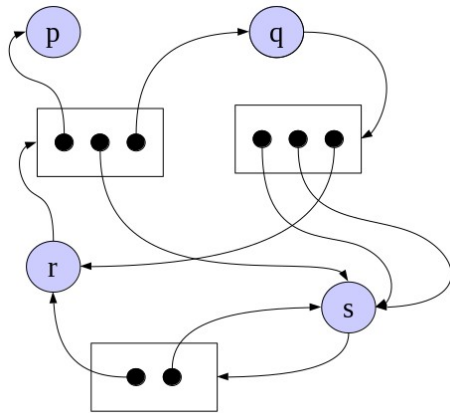
Essenzialmente, supponiamo che P1 prima o poi finirà.

Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili, lo stato non è di deadlock se e solo se il grafo di Holt è completamente riducibile, i.e. esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo.

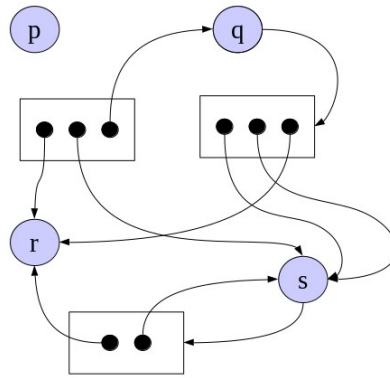
Il teorema appena esposto ci permette quindi di capire quando esiste deadlock in un grafo multirisorsa (o con più classi di risorse).

Ecco un esempio di esercizio passo-passo sui grafi di Holt:

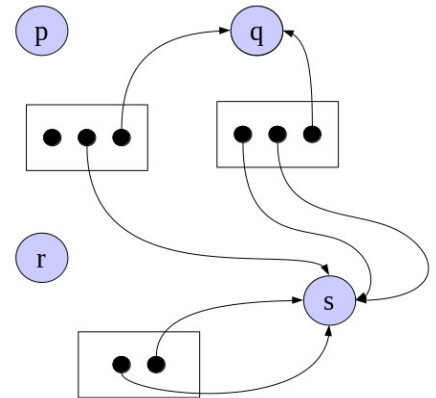
1)



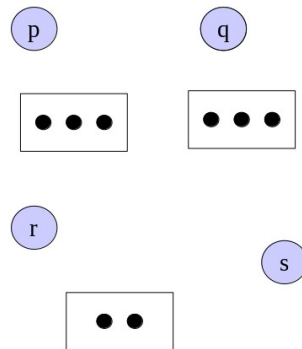
2)



3)



4)



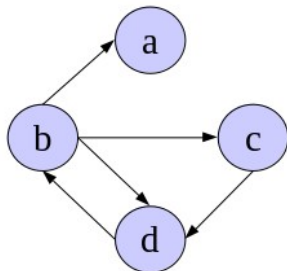
Grafi di Holt, Knot:

Esiste un altro metodo più vicino alla teoria dei grafi per la deadlock detection. Questo metodo consiste nel vedere se nel grafo di Holt è presente un **Knot** (in inglese, nodo. Non il nodo del grafo, il nodo della corda. Per non creare confusione, si usa il termine inglese sempre).

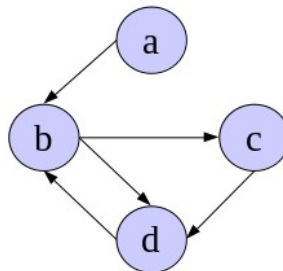
Per capire cosa si intende con Knot, bisogna prima capire cos'è un insieme di raggiungibilità. Dato un nodo n , l'insieme dei nodi (compreso n) raggiungibili da n viene detto **insieme di raggiungibilità di n lungo (o attraverso) un cammino di archi del grafo** (scritto $R(n)$).

Un **knot** del grafo G è il sottoinsieme (non banale) di nodi M tale che per ogni n in M , $R(n)=M$.

In altre parole: partendo da un qualunque nodo di M , si possono raggiungere tutti i nodi di M e nessun nodo all'infuori di esso.



$\{b, c, d\}$ non è un *knot*



$\{b, c, d\}$ è un *knot*

Nel grafo a dx, $\{b, c, d\}$ è un knot. Infatti, $R(b) = \{b, c, d\}$, stessa cosa per c e d , infatti questi sono gli unici nodi che posso raggiungere da b, c o d .

Siccome possiamo trovare un knot attraverso una operazione algoritmica, esiste un teorema che ci dice che dato un grafo di Holt con una sola richiesta sospesa per processo se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili, allora il grafo rappresenta uno stato di deadlock se e solo se esiste un **knot**.

Deadlock recovery:

Dopo aver rilevato un deadlock, bisogna risolvere la situazione. Teniamo a mente, però, che come abbiamo già detto più volte, la cura al deadlock sarà una cura traumatica e distruttiva. In ogni caso, la soluzione può essere:

- **manuale:** l'operatore (sysadmin) viene informato e eseguirà alcune azioni che permettano al sistema di proseguire.
- **automatica:** il sistema operativo è dotato di meccanismi che permettono di risolvere in modo automatico la situazione, in base ad alcune politiche.

Alcuni meccanismi per il deadlock recovery sono:

- **Terminazione totale:** tutti i processi coinvolti vengono terminati. In questo modo evitiamo il deadlock, ma non la situazione non è ottimale.
- **Terminazione parziale:** viene eliminato un processo alla volta, fino a quando il deadlock non scompare. In questo modo si cerca solo di limitare il danno.
- **Checkpoint/rollback:** lo stato dei processi viene periodicamente salvato su disco (*checkpoint*) in caso di deadlock, si ripristina (*rollback*) uno o più processi ad uno stato precedente più stabile, fino a quando il deadlock non scompare. Questo non è sempre possibile, siccome una volta che i processi ripartono, si possono replicare azioni che non vogliamo replicare (ad esempio: se un bancomat ci deve dare dei soldi, ce li dà due volte. CYBERPUNK 2077). Quindi il sistema può avere degli effetti collaterali che peggiorano la situazione se ripetuti.

La deadlock recovery spesso è una cattiva idea, siccome terminare i processi può essere molto costoso: magari possono essere processi che stanno facendone calcoli complessi che impiegano molto tempo, e terminandoli improvvisamente questi dovranno ripartire da capo, e perciò si avrà uno spreco di tempo. In più, se un processo viene terminato nel mezzo di una sezione critica, questo può lasciare le risorse in uno stato incoerente (inconsistent == incoerente in italiano), e in questo modo la sezione critica viene violata.

Deadlock prevention e avoidance:

Prevention e avoidance, come già accennato prima, indicano due concetti differenti.

Con **prevention** si intende che il deadlock viene eliminato *strutturalmente*, eliminando una delle quattro condizioni della deadlock in modo che una di esse non si possa mai attuare.

Con **avoidance**, invece, si intende un metodo diverso: prima di assegnare una risorsa ad un processo, si controlla se l'operazione può portare al pericolo di deadlock. Se è così, l'operazione viene ritardata.

Deadlock Prevention:

Usando la deadlock prevention, "attacchiamo" una delle 4 condizioni di deadlock:

- Una di queste è la condizione di "**Mutua esclusione**", perciò dovremo permettere la condivisione di risorse (i.e. spool di stampa, tutti i processi "pensano" di usare contemporaneamente la stampante)
- Per attaccare la condizione di "**Richiesta bloccante**", un modo per risolvere il problema sarebbe l'allocazione totale, che consiste nel fare in modo che un processo richieda tutte le risorse a lui necessarie all'inizio della computazione. Nel caso riesca a prendersela, allora il processo prima di partire saprà se potrà finire la sua esecuzione oppure no, e nel caso appunto non riesca ad appropriarsi di tutte le risorse, allora gli viene negato l'accesso. In questo modo però si riduce il parallelismo, siccome un processo in questo modo mantiene per sé risorse che magari erano necessarie solo per qualche microsecondo.
- Attaccare la condizione di "**Assenza di prerilascio**": questa non è sempre possibile, e può richiedere interventi manuali.
- Attaccare la condizione di "**Attesa Circolare**": può essere risolta da un meccanismo che è più debole dell'allocazione totale, ovvero l'allocazione gerarchica, che consiste nelle associare alle classi di risorse dei valori di priorità. Ogni processo in ogni istante può allocare solamente risorse di priorità superiore a quelle che già possiede, e se un processo vuole allocare una

risorsa a priorità inferiore, deve prima rilasciare tutte le risorse con priorità uguale o superiore a quella desiderata.

(es. supponendo che $a < b < c$, se ho 3a e 5c e voglio allocare 2b, devo rilasciare le 5 di classe c e poi prendere quelle di classe b. Se ora vuole 1 risorsa in più di classe c rilascia le 5 e ne prende 6.)

Sia allocazione gerarchica che allocazione totale, anche se prevengono il verificarsi di deadlock, ma sono altamente inefficienti. Entrambi chiedono di allocare risorse anche quando non sono utili per evitare il deadlock.

Nell'allocazione gerarchica, infatti, l'indisponibilità di una risorsa ad alta priorità ritarda processi che già detengono risorse ad alta priorità (Ovvero, quando uno prende una risorsa ad alta priorità, blocca per tutto il tempo quelli a priorità inferiore).

Invece, direi che abbiamo già dissato l'allocazione totale abbastanza.

Lezione del (16/03/2021) – algoritmo del Banchiere, gestione della memoria, binding e linking.

Osservazioni sulla lezione precedente:

Nell'allocazione gerarchica, un processo può chiedere solo risorse di classe superiore di quello che ha già'. In questo modo, se creiamo un grafo di Holt e posiziamo il processo subito davanti alla risorsa di classe massima che possiede, e frecce di avanzamento saranno tutte verso destra, in questo modo non ci saranno cicli nel grafo di Holt, e si evita la attesa circolare (che è una delle condizioni necessarie per la deadlock), e quindi anche la deadlock.

La classificazione delle risorse viene fatta da chi scrive il sistema operativo.

Deadlock avoidance:

Nella deadlock avoidance non si negano le 4 condizioni, ma bensì si cerca di mantenere uno stato "sicuro", in modo che dato lo stato attuale non sia possibile evolvere in deadlock, limitiamo le possibili evoluzioni del sistema a quelli che hanno almeno una via d'uscita dal deadlock. Uno degli algoritmi che permette ciò è l'algoritmo del banchiere.

[METANOTA: il concetto qui è molto confuso, viene spiegato meglio subito sotto (vai a "Per capire meglio...")]

Dice essenzialmente che un banchiere ha una certa quantità di denaro da amministrare e deve soddisfare le richieste di prestito dei clienti.

Se il fido (ovvero la quantità massima di denaro che qualcuno può chiedere) per tutti i clienti è minore di 100, allora siamo apposto, chiunque potrà chiedere soldi... Tuttavia, questa banca (come quelle vere) presta più soldi di quanti ne ha effettivamente

Per fare in modo che ogni richiesta di un cliente possa essere soddisfatta, essenzialmente la banca "promette" che appunto essa soddisfatta, ma non subito, infatti dirà di aspettare finché non sarà disponibile.

Per capire meglio il ragionamento, occorre definire il problema:

un banchiere desidera condividere un capitale (fisso) con un numero (prefissato) di clienti, ogni cliente specifica in anticipo la sua necessità massima di denaro (che ovviamente non deve superare il capitale del banchiere). I clienti fanno due tipi di transazioni: richieste di prestito e restituzioni.

Il denaro prestato ad ogni cliente non può mai eccedere la necessità massima specificata a priori dal cliente. Ogni cliente può fare richieste multiple, fino al massimo importo specificato. Una volta che le richieste sono state accolte e il denaro è stato ottenuto, il cliente deve garantire la restituzione in un tempo finito.

Il banchiere deve essere in ogni istante in grado di soddisfare tutte le richieste dei clienti, o concedendo immediatamente il prestito oppure comunque facendo loro aspettare la disponibilità del denaro in un tempo finito.

Usando questa metafora, le monete sono le risorse, e il banchiere è il SO e i processi sono i clienti. Ovviamente se un processo chiede più delle risorse totali disponibili, lo si manda a fanculo e non gliene si dà.

Esaminiamo un esempio:

Il banchiere ha un 1 GB, e un processo A ha bisogno 600 MB e un processo B pure.

Il processo A chiede 200 MB, allora al banchiere ne rimangono 800 MB, il B ne chiede 500 MB allora ne rimangono 300 MB. A poi ne ha bisogno di 300 MB, e ne rimangono 0. Né A né B potranno così chiedere altre risorse, e perciò in questo caso la situazione diventa pericolosa. Potrebbe anche non andare in deadlock, perché magari il processo A riesce a fare tutto anche solo con 500 MB, però in questa situazione rimane pericolosa. Questo ragionamento è alla base di ciò che usa l'algoritmo del banchiere.

Ovviamente questo funziona solamente se un processo dichiara esattamente la quantità di risorse di cui ha bisogno a priori, e nel caso chiedesse più di quanto esso ha dichiarato, semplicemente non gli si viene dato.

Nel nostro algoritmo, avremo che:

- N : numero clienti
- IC : capitale iniziale
- c_i : limite credito del cliente i (o il fido appunto) ($c_i < IC$).
- p_i : denaro prestato al cliente i ($p_i \leq c_i$)
- $n_i = c_i - p_i$: credito residuo del cliente i
- $COH = IC - \sum_{i=1..N} P_i$: saldo di cassa (cash on hand, quanti soldi ha il banchiere in questo momento)

Con l'algoritmo del banchiere, lo stato è **safe** quando esiste una sequenza di processi (una permutazione s dei valore $1..N$) es. $N=4$, $s = 1, 3, 4, 2$.

Questa sequenza è in grado di soddisfare almeno un processo (ovvero il primo), quindi quello che è stato prestato verrà restituito. Si calcola il vettore avail, ovvero le risorse disponibili a ciascun punto j , come segue:

$$\begin{aligned} \text{avail}[1] &= COH \\ \text{avail}[j+1] &= \text{avail}[j] + p_{s(j)} \text{ con } j = 1..N-1 \end{aligned}$$

Con $s(j)$ si intende il processo $s(j)$ in posizione j della sequenza s

Se ad ogni passo della sequenza vale la condizione:

$$n_{s(j)} \leq \text{avail}[j], \text{ con } j=1..N$$

Allora lo stato del sistema si dice SAFE, se invece questa condizione non vale siamo nello stato UNSAFE. Lo stato UNSAFE è condizione necessaria ma non sufficiente per avere deadlock (i.e., un sistema in uno stato UNSAFE può evolvere senza procurare alcun deadlock).

Es. con $IC = 150$.

<i>Cliente</i>	<i>MAX</i>	<i>Prestito Attuale</i>	<i>Credito residuo</i>
i	c_i	p_i	n_i
1	100	0	100
2	20	0	20
3	30	0	30
4	50	0	50
5	70	0	70

→

<i>Cliente</i>	<i>MAX</i>	<i>Prestito Attuale</i>	<i>Credito residuo</i>
i	c_i	p_i	n_i
1	100	70	30
2	20	10	10
3	30	30	0
4	50	10	40
5	70	30	40

Questo stato che si ottiene, anche se il COH del banchiere è 0, è comunque uno stato SAFE.

Esiste infatti una sequenza che consiste di soddisfare tutte le richieste. Il processo 3 può concludere, rilasciando 30 €, e con 30 in cassa poi possiamo soddisfare 1 e 2 a scelta (la sequenza che possiamo scegliere non è unica).

Per questo algoritmo c'è inoltre una regola pratica: **lo stato SAFE può essere verificato usando la sequenza che ordina in modo crescente i valori di n_i** (ovvero del credito residuo di ciascun cliente).

Es:

	n_i			avail[i]
3	30	30	0	0
2	20	10	10	30
1	100	70	30	40
4	50	10	40	110
5	70	30	40	120

$n[i]$ è sempre minore di avail[i], quindi lo stato è safe.

Facciamo un esempio di stato unsafe:

In questa situazione, il IC è sempre 150€, e il COH è addirittura > 0 (10 per l'esattezza), ma nonostante tutto, si ha comunque stato UNSAFE:

Cliente	MAX	Prestito Attuale	Credito residuo
i	c_i	p_i	n_i
1	100	65	35
2	20	10	10
3	30	5	25
4	50	15	35
5	70	35	35

Usiamo adesso la regola pratica vista prima, ordinando tutti i processi in base ad n_i :

2	20	10	10	10
3	30	5	25	20
1	100	65	35	25
5	70	35	35	100
4	50	15	35	135

avail[i] NON è sempre minore di $n[i]$, quindi lo stato è UNSAFE. Nelle celle in blu, vengono mostrati gli stati in cui la safety fallisce.

UNSAFE non implica necessariamente deadlock: se il cliente 5 restituisce il suo prestito di 35 euro la situazione ritorna SAFE. (Dunque, lo stato UNSAFE è condizione necessaria ma non sufficiente per deadlock.)

Banchiere multivaluta:

L'algoritmo del banchiere per ora ha due punti deboli: funziona solo se il processo specifica la quantità di risorse richieste a priori e solo se abbiamo una sola classe di risorsa.

Purtroppo però nella vita reale le classi di risorse sono più di una, e quindi occorre estendere l'algoritmo del banchiere trasformandolo nell'algoritmo del **banchiere multivaluta**, dove le diverse valute rappresentano le varie classi di risorse. Supponendo di aver capito l'algoritmo del banchiere nel caso monovaluta, capire quello multivaluta non è così difficile: infatti basta passare dallo scalare al vettoriale.

La nostra notazione sarà infatti abbastanza simile a quella iniziale, se non appunto per l'aggiunta dei vettori.

[NOTA SULLA SINTASSI: adesso i simboli/sigle specificate indicano vettori e non più un valore singolo]

Nel nostro algoritmo, avremo che:

- N : numero clienti (sarà il nostro unico scalare)
- \underline{IC} : capitale iniziale (vettore, un elemento per ogni valuta)
- \underline{c}_i : limite credito del cliente i (o il fido appunto) ($\underline{c}_i < \underline{IC}$).
- \underline{p}_i : denaro prestato al cliente i ($\underline{p}_i \leq \underline{c}_i$)
- $\underline{n}_i = \underline{c}_i - \underline{p}_i$: credito residuo del cliente i
- $\underline{COH} = \underline{IC} - \sum_{i=1..N} \underline{p}_i$: saldo di cassa (cash on hand) (si fa la sottrazione elemento per elemento)

Semplicemente ci basterà fare i calcoli elemento per elemento del vettore, senza prodotti scalari strani grazie a Dio. Ciò che invece cambierà saranno le disequazioni, e quindi le condizioni di stato.

Lo stato è SAFE quando esiste una sequenza di processi (una permutazione s dei valori $1..N$) es. $N=4$, $s = \langle 1, 3, 4, 2 \rangle$. (indichiamo con $s(i)$ l' i -esima posizione della sequenza) si calcoli il vettore $avail_k$ come segue

$$avail[1] = COH$$

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

uno stato del sistema si dice safe se vale la seguente condizione:

$$n_{s(j)} \leq avail[j], \text{ con } j=1..N$$

La cosa però si complica. Infatti, se abbiamo due vettori A e B con $A = (1,2)$ e $B = (2, 1)$, allora A e B non sono comparabili. Quindi, non possiamo fare un ordinamento secondo un ordine parziale, e allora la metodologia di usare l'ordinamento dei valori di n_i non è applicabile. **Dunque, ci tocca usare un nuovo metodo, è questo metodo consiste nel la sequenza procedendo passo passo scegliendo un processo a caso fra quelli completamente soddisfacibili** (in modo simile alla riduzione nel grafo di Holt).

C'è poi questo teorema che ci aiuta: se durante la costruzione della sequenza s si giunge ad un punto in cui nessun processo risulta soddisfacibile, **lo stato non è SAFE** (i.e. non esiste alcuna sequenza che consenta di soddisfare tutti i processi)

Ciò vuol dire che ce ne deve essere sempre almeno uno soddisfacibile, e ad ogni passo devo scegliere il processo soddisfacibile.

[METANOTA: il prof ha spiegato la dimostrazione in modo un po' confuso, in ogni caso consiglio di riguardare questa parte della lezione, che si trova a 1.10h]

Il funzionamento di questo teorema si può dimostrare per assurdo: supponiamo che lo stato sia SAFE, ovvero che esista la sequenza che consente di soddisfare tutti i processi. Sia C la sequenza interrotta e C' la sequenza che porta allo stato SAFE:

Siano C e C' le sequenze di processi come in figura

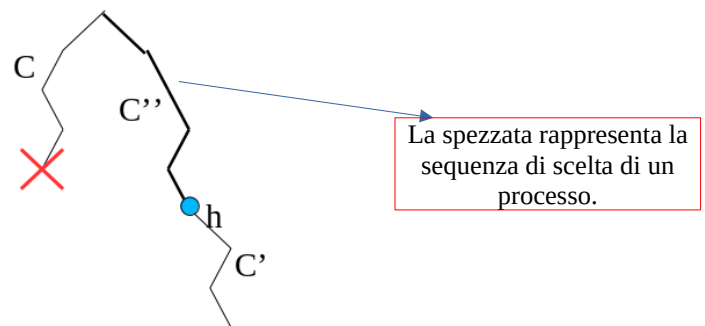
Sia $H = \{p \in \text{processi} \mid p \notin C\}$

sia h il primo elemento di H che compare in C'

tutti gli elementi di C' prima di h compaiono in C . Chiamiamo C'' il segmento iniziale di C' fino al punto precedente l'applicazione di h

le risorse disponibili all'applicazione di h in C' sono $avail(C'') = COH + \sum_{j \in C''} p_j$; h è applicabile quindi $n \leq avail(C'')$

ma $avail(C) = COH + \sum_{j \in C} p_j$, quindi se $avail(C) \leq avail(C'')$ allora h è applicabile alla fine di C contro l'ipotesi che C non fosse ulteriormente estendibile



Tutte le volte che un processo ci fa una richiesta, essenzialmente noi prendiamo la tabella e ne facciamo una copia. Facciamo finta di aggiungere una richiesta e andiamo a vedere se lo stato è safe o unsafe. Se lo stato è safe, aggiorniamo la tabella nella sua copia vera e gestiamo la richiesta, se invece è unsafe, la richiesta rimane pendente. Quando i processi restituiscono delle risorse, andiamo a vedere la prima richiesta in sospeso e si va a vedere se nella tabella se, date le ulteriori risorse disponibili, lo stato è safe, allora la prima richiesta in coda rimane soddisfatta (senza soddisfare nuove richieste, che invece verranno accodate). Se la coda non è ancora vuota, controllo se lo stato sarebbe safe gestendo il processo nella coda, e se lo è, lo gestisco, altrimenti no.

L'algoritmo del banchiere multivaluta è fighissimo, ma rimane comunque un problema: è molto difficile per i processi dichiarare a priori le risorse che gli saranno necessarie. Perciò, molti sistemi operativi fanno uso dell'algoritmo dello struzzo, ovvero si fa finta che i deadlock non si possano mai verificare. Si usa questo algoritmo siccome il costo della deadlock detection/avoidance può essere troppo elevato. Questa è la soluzione usata nei sistemi Unix (e anche nelle JVM). In questo, la gestione delle deadlock viene affidato all'amministratore di sistema.

Precisazioni sul livelock:

Possiamo distinguere il livelock dal deadlock con questa metafora: pensiamo al deadlock come dei lavoratori, che stanno lì seduti senza fare nulla, e invece al livelock come dei lavoratori che effettivamente fanno cose e lavorano, però il loro lavoro non porta effettivamente a nulla. È difficile accorgersi di un livelock, ed è ancora un problema molto aperto. Il livelock più banale è il ciclo infinito, ma non è l'unico modo, infatti potrebbe esserci anche un cambiamento di stato durante una livelock per esempio. Questo è un problema arduo e molto costoso da rintracciare.

Gestione della memoria:

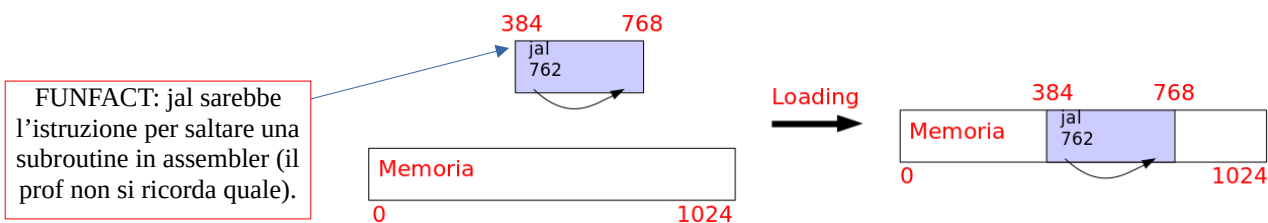
La gestione della memoria nei sistemi la parte del sistema operativo che gestisce la memoria centrale si chiama **memory manager**. In alcuni casi, il memory manager può gestire anche parte della memoria secondaria, al fine di emulare memoria centrale in modo di dare l'illusione di avere più memoria centrale (memoria virtuale). Il memory manager deve tenere traccia della memoria libera e occupata, e deve operare ogni qualvolta ci sia bisogno di allocare memoria ai processi o di deallocarla quando non più necessaria, perciò opera ogni volta che c'è un cambiamento dello stato delle aree di memoria. Il memory manager NON ENTRA IN FUNZIONE ogni volta che c'è un accesso in memoria, altrimenti ci sarebbero rallentamenti.

È importante però capire la differenza fra MMU e memory manager: il memory manager è software (componente del S.O), mentre la Memory Management Unit (MMU) è hardware che viene configurato dal memory manager, ed entra in funzione ogni qualvolta un processo deve accedere a un dato in memoria, che lavora in tempo di hardware. Es. se la MMU deve gestire un indirizzo per il quale non ha regola di trasformazione, genera una trap che verrà poi gestita dal kernel che chiama il memory manager.

Con **Binding** si indica l'associazione di indirizzi logici di memoria (e.g. nomi di variabili, label) ai corrispondenti indirizzi fisici, e questa associazione può avvenire durante la compilazione (es. il compilatore dice "i" sta ad indirizzo 42), durante il caricamento (il compilatore può mantenere traccia dicendo che occorre una "i" perché funzioni, e durante il caricamento mi fa corrispondere al nome "i" l'indirizzo 44), oppure durante l'esecuzione.

Binding durante la compilazione:

Durante la compilazione (binding statico) si crea codice assoluto, ciò significa che il compilatore decide a che indirizzo si troverà la variabile. Quindi, tutte le volte che lanciamo il programma, la variabile sarà a quell'indirizzo (gli indirizzi vengono calcolati al momento della compilazione e resteranno gli stessi ad ogni esecuzione del programma). Alcuni esempi sono i codici per microcontrollori, per il kernel, file COM in MS-DOS.



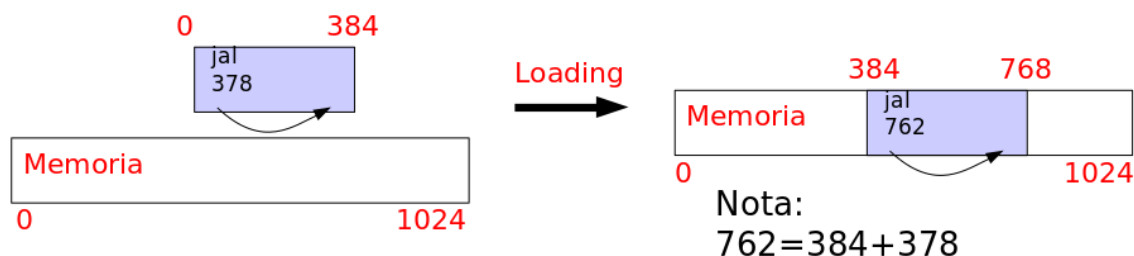
Il codice assoluto non ammette la creazione di più istanze dello stesso programma siccome altrimenti ci sarebbe un conflitto fra gli indirizzi (es. kernel, che deve avere una sola istanza).

I vantaggi di questo approccio sono che non richiede hardware speciale, e la compilazione è molto semplice e veloce. Lo svantaggio invece sta nel fatto che non funziona con la multiprogrammazione.

Binding durante il caricamento:

Il codice generato dal compilatore non contiene indirizzi assoluti ma relativi (al PC oppure ad un indirizzo base), per questo di codice generato viene detto codice rilocabile.

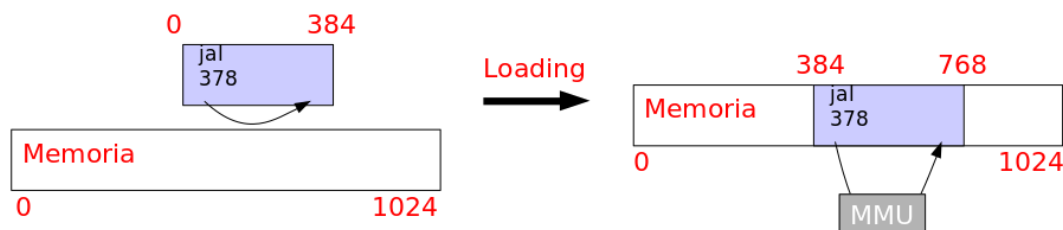
Durante il caricamento, il loader si preoccupa di aggiornare tutti i riferimenti agli indirizzi di memoria coerentemente al punto iniziale di caricamento.



Il binding durante il caricamento permette di gestire multiprogrammazione e anch'esso non richiede uso di hardware particolare (si può adottare anche senza MMU), tuttavia richiede una traduzione degli indirizzi (es. deve anche aggiornare i comandi assembly correttamente, come jmp etc) da parte del loader, e quindi formati particolare dei file eseguibili.

Binding durante l'esecuzione:

L'individuazione dell'indirizzo di memoria effettivo viene effettuata durante l'esecuzione da un componente hardware apposito: la memory management unit (MMU) [da non confondere con il memory manager (MM), ancora, per la millesima volta].



La MMU consente anche di fare allocazioni di memoria ad indirizzi fisicamente non contigui, quindi è molto flessibile.

Se usiamo un debugger normale tipo gdb, vedremo solamente indirizzi logici della memoria. Per vedere quelli fisici, dobbiamo usare un debugger del kernel.

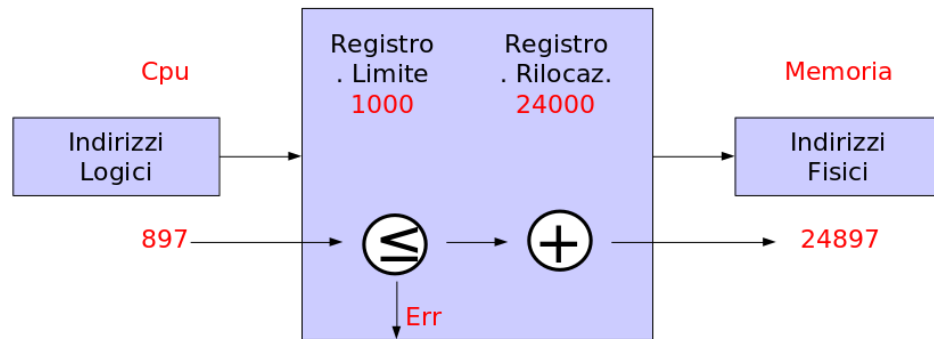
Indirizzi logici ed indirizzi fisici:

La CPU, quando genera indirizzi di memoria, genera indirizzi logici, e solo la MMU sa, dato l'indirizzo logico, la locazione di quello fisico. Il memory manager configura la MMU, e tiene la contabilità di come questo è configurato, ma a run-time è la MMU che trasforma gli indirizzi logici in fisici.

La funzione di indirizzamento può essere parziale (ovvero non tutti gli indirizzi logici sono validi) e in questo modo si crea implicitamente anche protezione di memoria.

Un esempio di MMU è il **registro di rilocalizzazione**, questa era una tecnica molto vecchia usata nel processore 80x86, che traduce l'indirizzo logico richiesto dalla CPU in un nuovo indirizzo logico sommato di un valore che si trova nel registro di rilocalizzazione. [Aggiunta dal 19/03: questo tipo di MMU può gestire solo allocazione contigua.]

Un altro esempio di MMU sta nell'aggiungere, oltre che un registro di rilocalizzazione, anche un **registro limite**, per implementare meccanismi di protezione della memoria, facendo in modo che non si potesse "tracimare" e andare nella memoria del processo a fianco. Nel caso la Mmu ricevesse un indirizzo logico non legale, si genera una trap, e il kernel deciderà cosa farà al processo che ha provato ad accedere a un indirizzo illegale.



Meccanismi speciali di protezione:

Il **loading dinamico** è un tipo speciale di protezione della memoria, che consente di poter caricare alcune parti del programma (es. routine di libreria, routine di errore) solo quando vengono richiamate.

Questa è una funzionalità che non è per nulla trasparente al programmatore, che per usarlo dovrà richiamare le funzioni specifiche che permettono di fare ciò.

Infatti, ciò viene implementato in modo che tutte le routine a caricamento dinamico risiedono su un disco (codice rilocabile), e solo quando servono quando servono vengono caricate.

Il linking consiste nel collegare più file oggetto fra loro risolvendo i simboli, e alcuni di essi possono trovarsi anche in delle librerie.

Il linking statico consiste nel copiare le routine di libreria nell'eseguibile, e in questo modo l'eseguibile è un codice completo, e contiene tutti i codici necessari (e.g. printf in tutti i programmi C).

Con il **linking dinamico**, invece, viene posticipato il vero linking delle routine di libreria al momento del primo riferimento durante l'esecuzione. In questo modo, gli eseguibili sono più compatti, e le librerie vengono implementate come codice reentrant: esiste una sola istanza della libreria in memoria e tutti i processi eseguono il codice di questa istanza. In questo, si consente di risparmiare memoria e consente l'aggiornamento automatico delle librerie.

Ci possono però essere problemi di versioning, siccome se aggiungiamo un parametro, potrebbe risultare incompatibile con alcuni programmi. Per questo, bisognerà cambiare numeri di revisione e non di release (che invece indica la nuova versione, possibilmente incompatibile).

Oggi si usa un misto di loading dinamico e linking dinamico, e in questo modo (attraverso per esempio la funzione dlopen()) le librerie dinamiche vengono linkate a run-time.

Lezione del (19/03/2021) – Allocazione statica e dinamica, paging, memoria virtuale.

Allocazione:

L'**allocazione** (dal latino, "fare spazio per") è una delle funzioni principali del gestore di memoria, e consiste nel reperire ed assegnare uno spazio di memoria fisica a un programma che viene attivato,

oppure per soddisfare ulteriori richieste effettuate dai programmi durante la loro esecuzione. Ma come facciamo a capire quale area di memoria assegnare a un processo?

Si parla di **assegnazione contigua** quando tutto lo spazio assegnato ad un processo deve essere formato da celle consecutive (ovvero quindi celle tutte attaccate). Altrimenti, si parla **assegnazione non contigua** quando le aree di memoria che si assegnano sono separate. La continua è più facile da gestire, ma può provocare problemi di spazio, mentre per quella non contigua è l'inverso (infatti la MMU deve essere in grado di gestire la conversione in modo coerente).

L'allocazione inoltre può essere statica o dinamica: con **allocazione statica** un processo deve mantenere la propria area di memoria dal caricamento alla terminazione, non è possibile rilocalizzare il processo durante l'esecuzione, mentre con **allocazione dinamica** un processo può essere spostato all'interno della memoria durante la sua esecuzione.

Allocazione a parti fisse:

Per fare in modo che non abbiamo spostarsi, si creano delle partizioni fisse di memoria che verranno assegnate ai vari processi (questo ad esempio nei sistemi embedded). Per esempio, se so già di avere un massimo di 3 processi e che ciascuno di questi processi non avrà mai bisogno di un tot di memoria, potrò anche compilare il codice con binding statico (ovvero, come abbiamo visto, si crea codice assoluto), in modo che il primo processo vada in una partizione, e un'altro nell'altra etc... [in teoria si può fare anche allocazione non contigua con binding statico (siccome basta riferire le informazioni delle celle non contigue al compilatore), quello che sicuramente non si potrà fare però sarà allocazione dinamica con binding statico.]

In poche parole, la memoria disponibile (NON occupata dal s.o.) viene divisa in partizioni, e ogni processo viene caricato in una di queste partizioni libere. Solitamente questa strategia fa uso di allocazioni statiche e contigue, e nonostante sia molto semplice (non richiede un MMU), causa un certo spreco di memoria e limita il parallelismo. [Es. immaginiamo di avere 3 partizioni di grandezza decrescente, se ho un processo "piccolo" da assegnare quando gli altri due sono occupati ed ho solo la partizione grande libera, allora sarò costretto a mettere il processo in quella grande e avrò uno spreco di spazio] Posso avere una sola coda dei processi per tutte e tre le partizioni, oppure una coda a partizione. Nei sistemi monoprogrammati (es. arduino,) abbiamo una sola partizione in cui viene caricato un singolo programma (nel caso di FreeDOS o MS DOS abbiamo una singola partizione utente dedicata al caricamento dei programmi e una per il sistema operativo).

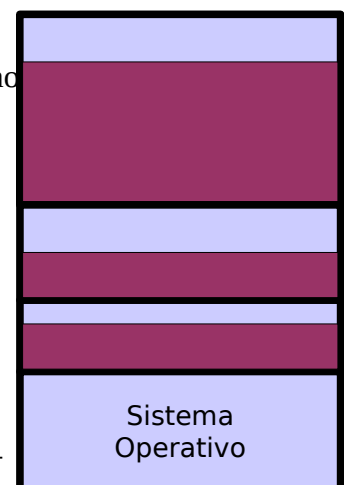
[Piccola curiosità: esiste anche debina con kernelBSD (GNU/kFreeBSD). BSD è un sistema UNIX sviluppato nell'università di Berkley. È diverso da Linux, ma molte parti possono comunque funzionare.]

Frammentazione interna:

Questo è un concetto chiave: il concetto di frammentazione (esterna o interna che sia), indica che ci sono dei frammenti di memoria che rimangono inutilizzati.

Nell'allocazione a partizione fisse, se un processo occupa una dimensione inferiore a quella della partizione che lo contiene, lo spazio non utilizzato è sprecato (come dicevamo nell'esempio). La presenza di spazio inutilizzato all'interno di un'unità di allocazione si chiama **frammentazione interna**. In altre parole, si ha frammentazione interna se all'interno di una unità di allocazione di dimensione predeterminata, parte della memoria rimane inutilizzata.

Il fenomeno della frammentazione interna non è limitata all'allocazione a partizioni fisse, ma è generale a tutti gli approcci in cui è possibile allocare più memoria di quanto richiesto (per motivi di organizzazione)

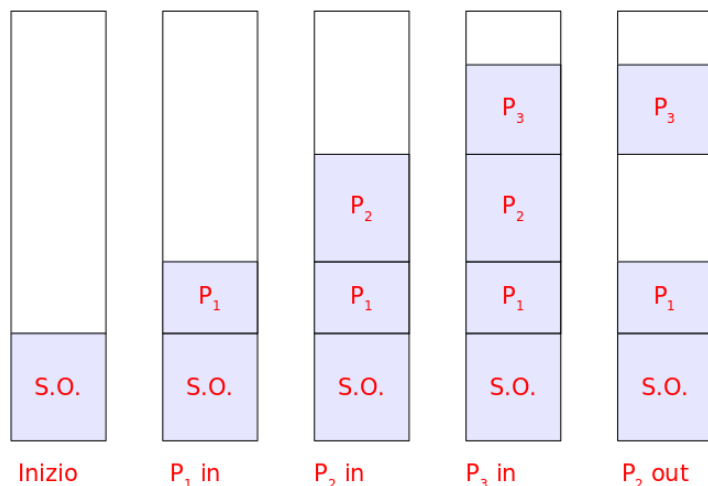


Questo sopra è un esempio epico di frammentazione interna. Notiamo infatti che la memoria è divisa in blocchi.

Allocazione a partizione dinamica:

Siccome l'allocazione a partizioni fisse genera frammentazione interna, allora ci affidiamo alla allocazione a partizioni dinamiche. La memoria disponibile (nella quantità richiesta) viene assegnata ai processi che ne fanno richiesta in modo dinamico, senza dover assegnare una partizione predefinita. Nonostante ciò, nella memoria possono essere presenti diverse zone inutilizzate per effetto della terminazione di processi, oppure per non completo utilizzo dell'area disponibile da parte dei processi attivi. Anche qui, l'allocazione è contigua e comunque statica.

Esempio di allocazione a partizione dinamica:



P₄: non c'è spazio

Ad ogni passo, allochiamo spazio della memoria per ogni processo diverso. È necessario comunque che il binding sia al momento del caricamento, siccome se per esempio P3 fosse stato allocato prima di P2, le locazioni di memoria sarebbero cambiate.

Nonostante ci sia spazio libero, non possiamo allocare P4 siccome lo spazio non è contiguo, e in più si verifica frammentazione esterna.

Siccome in questo caso abbiamo aree di memoria libere che non si trovano dentro a un contenitore di grandezza predefinita (come nelle partizioni statiche), ma sono “all'esterno”, nella memoria generale, e quindi si parlerà di **frammentazione esterna**. [Dalle slides: la frammentazione interna dipende dall'uso di unità di allocazione di dimensione diversa da quella richiesta, mentre la frammentazione esterna deriva dal susseguirsi di allocazioni e deallocazioni].

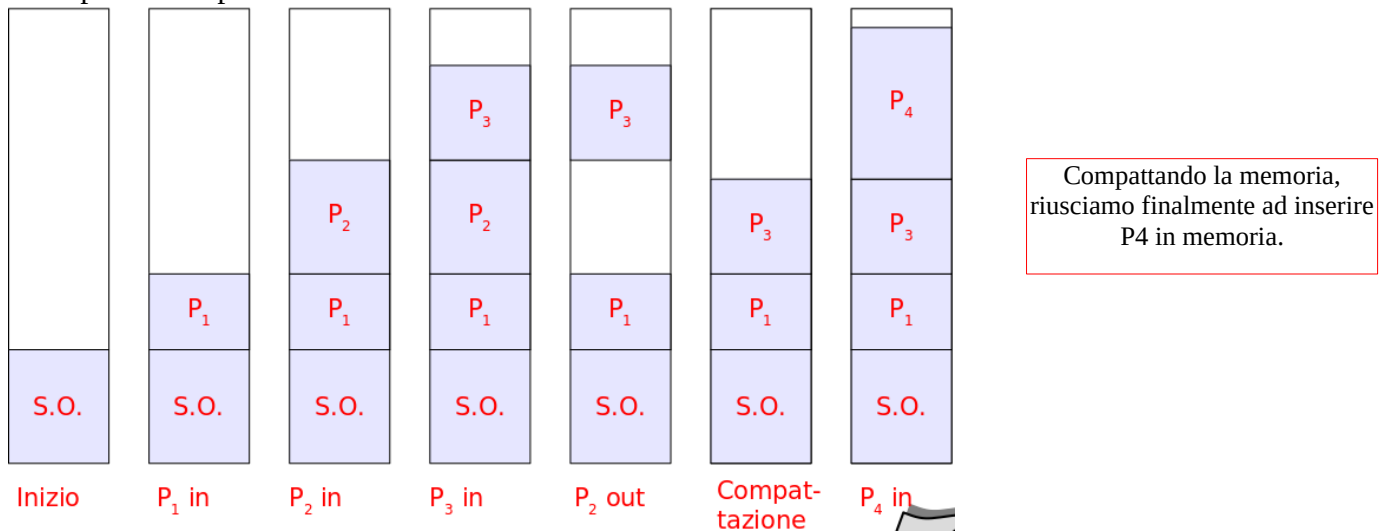
Per ovviare alla frammentazione esterna, un modo è quello di usare la **compattazione**. [[DUBBIO? Allocazioni a partizioni dinamica, ma comunque allocazione statica?]] Questa volta sarà però necessario un binding dinamico siccome durante l'esecuzione del processo, gli indirizzi di memoria cambieranno. Durante la compactazione, si fermano i processi, e si copiano le aree di memoria in modo da non lasciare buchi (copiamo letteralmente byte per byte). Quando poi facciamo ripartire i processi, questi useranno degli indirizzi diversi, e perciò dovremo fare in modo che gli indirizzi vengano ricalcolati gli indirizzi giusti modificando, per esempio, i registri base della MMU.

In altre parole, compactare la memoria significa spostare in memoria tutti i processi in modo da riunire tutte le aree inutilizzate. La compactazione può essere usata solo se è possibile relocare i programmi durante la loro esecuzione. Il problema sta nel fatto che questa è una operazione costosissima, infatti dobbiamo copiare un sacco di dati. Potremmo migliorare la performance fermando solo i processi in corso di copiatura e mandando gli altri avanti. Inoltre, non può essere utilizzata in sistemi interattivi, siccome i processi devono essere fermi durante la compactazione (può essere usata solo nel batch processing, ovvero programmi che lanciamo e aspettiamo il risultato senza interazione).

[La MMU sa quale processo ha occupato quale area di memoria, in quanto la MMU fa da contabile, quindi quando viene fatta la copia, la MMU sa già il punto di inizio e il punto di fine dell'area di memoria di un dato processo e perciò la rilocalizzazione viene fatta facilmente.]

[Frammentazione interna e frammentazione esterna non sono mutualmente esclusivi, infatti entrambi possono avvenire contemporaneamente. Ad esempio, possiamo creare dei blocchi che contengono dei processi e alcuni possono essere occupati (magari appunto parzialmente) ed altri no.]

Esempio di compattazione:



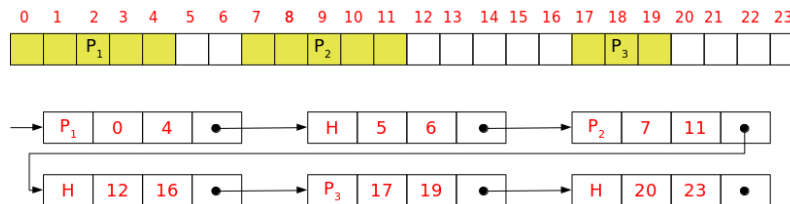
Ma come fa la memory manager a tenere traccia delle zone usate dai processi? Ci sono due modi:

1. mappe di bit
2. liste con puntatori

Nelle **mappe di bit** si suddivide la memoria in unità di allocazione, e quindi si potrebbe dare luogo a frammentazione interna, e ad ogni unità di allocazione corrisponde un bit in una bitmap. Alle unità libere si associa il valore 0, mentre alle unità occupate si associa 1. (Questo metodo di solito non viene usato molto nell'ambito della memoria, ma verrà usato in molti altri ambiti, tipo il file system.)

La mappa di bit si usa in quanto esistono delle istruzioni nei processori CISC che permettono di vedere se ci sono i bit tutti a 0 oppure tutti a 1 all'interno di una voca. La dimensione di una unità di allocazione è un parametro molto importante per questo algoritmo, siccome più sono piccoli e più il costo di ispezione (che è sempre una scansione lineare) è alto e la frammentazione interna è bassa, mentre più si ha un'unità ampia di allocazione e più la frammentazione interna diventa alta. Per individuare uno spazio di memoria di dimensione k unità, è necessario cercare una sequenza di k bit 0 consecutivi, e in generale, tale operazione è $O(m)$, dove m rappresenta il numero di unità di allocazione. Il vantaggio principale sta nel fatto che la struttura dati ha una dimensione fissa e calcolabile a priori.

Nelle **liste con puntatori**, invece, si mantiene una lista dei blocchi allocati e liberi di memoria. Ogni elemento della lista specifica se si tratta di un processo (P) o di un blocco libero (hole, H) e la dimensione (inizio/fine) del segmento.

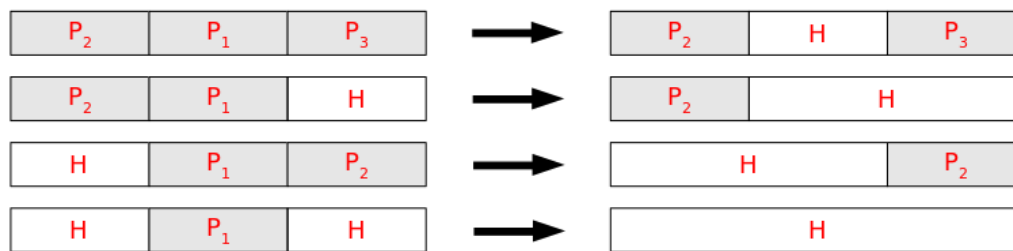


Per allocare la memoria usando la lista di puntatori, viene selezionato un blocco libero, il quale verrà suddiviso in due parti: un blocco processo (P_i) della dimensione desiderata ed un blocco libero (H) con quanto rimane del blocco iniziale. Nel caso la dimensione del processo fosse uguale a quella del blocco scelto, si crea solo un nuovo blocco processo.

La deallocazione è una questione leggermente più complessa: a seconda dei blocchi vicini, lo spazio liberato può creare un nuovo blocco libero, oppure essere accorpato ai blocchi vicini. L'operazione può essere fatta in tempo $O(1)$.

Abbiamo 4 casi possibili di rimozione:

Rimozione P_1 , quattro casi possibili:



Usando le liste con puntatori, il costo della deallocazione è $O(1)$, ma è possibile ottimizzare il costo di allocazione, per esempio mantenendo una lista di blocchi liberi separata e, eventualmente, ordinando tale lista per dimensione. Possiamo quindi mantenere la lista dei blocchi occupati nella tabella dei processi (es. la lista dei blocchi occupati dal processo p devono essere mantenuti dentro il PCB di p) e mantenere le informazioni sui blocchi liberi nei blocchi stessi.

Il MM, quando un nuovo processo da far partire, deve comunque decidere quale area da utilizzare fra quelle libere. Esistono vari algoritmi di scelta, e sono tutti indipendenti dalla struttura dati utilizzata. Fra questi, abbiamo:

1. **First Fit:** scorre la lista dei blocchi liberi fino a quando non trova il primo segmento vuoto grande abbastanza da contenere il processo.
2. **Next Fit:** come First Fit, ma invece di ripartire sempre dall'inizio, parte dal punto dove si era fermato all'ultima allocazione. Next Fit è stato progettato per evitare di frammentare continuamente l'inizio della memoria, ma sorprendentemente, ha performance peggiori di First Fit (LOL).
3. **Best Fit:** seleziona il più piccolo fra i blocchi liberi presenti in memoria, ovvero cerca fra gli spazi liberi quello che si meglio adatta al processo che dobbiamo far partire (quindi quello che crea la frammentazione esterna minore). È più lento di First Fit, in quanto richiede di esaminare tutti i blocchi liberi presenti in memoria, e genera più frammentazione di First Fit, in quanto tende a riempire la memoria di blocchi liberi troppo piccoli.
4. **Worst Fit:** seleziona il più grande fra i blocchi liberi presenti in memoria. È stato proposto per evitare i problemi di frammentazione di First/Best Fit, ma rende difficile l'allocazione di processi di grosse dimensioni

Paginazione:

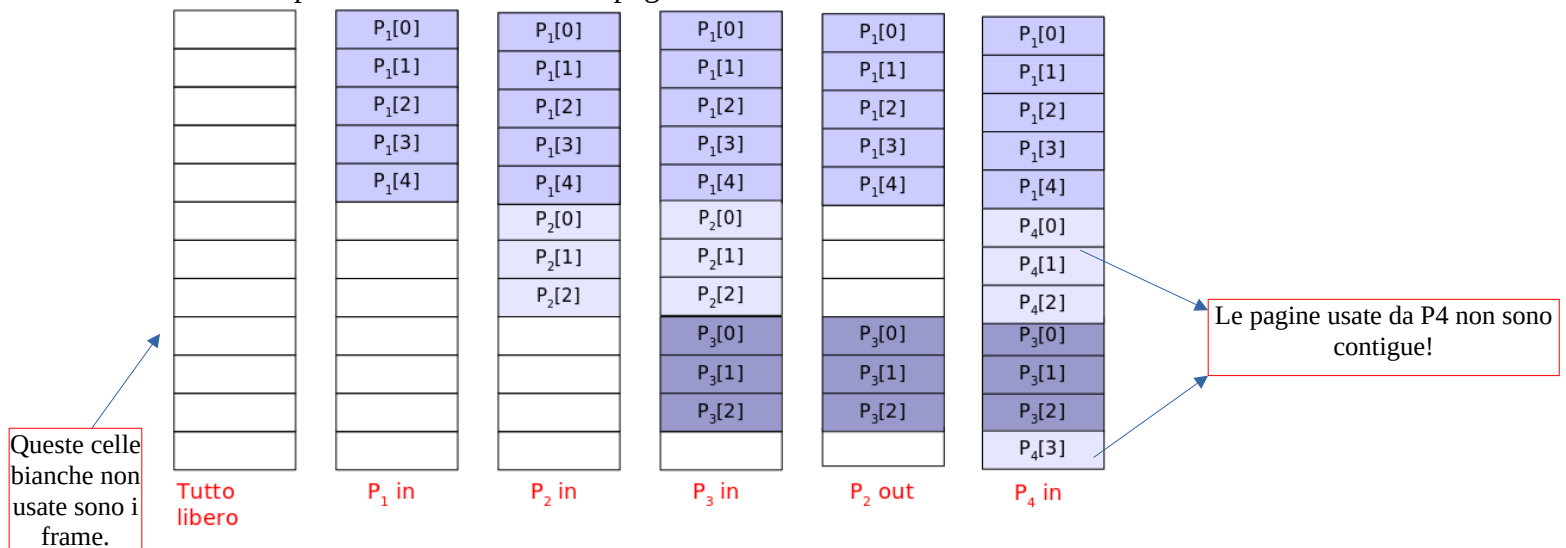
Il meccanismo della paginazione è usato da tutti i sistemi operativi moderni. Le partizioni fisse e dinamiche non sono efficienti nell'uso della memoria, e possono causare frammentazione interna e/o esterna. La **paginazione** quindi ci viene in nostro aiuto, siccome riduce il fenomeno della frammentazione interna e minimizza (lo elimina quasi) quello della frammentazione esterna. Tuttavia, questo approccio necessita di hardware dedicato.

Ogni processo ha uno spazio di indirizzamento logico, e questo viene diviso in un insieme di blocchi di dimensione fissa chiamati *pagine* (es. umips3 usa pagine di 4kB di dimensioni). Supponendo che la grandezza di una pagina sia di 4 kB, se un programma ha bisogno di 12 kB, verranno allocate 3 pagine. La memoria fisica viene suddivisa in un insieme di blocchi della stessa dimensione delle pagine, chiamati **frame**.

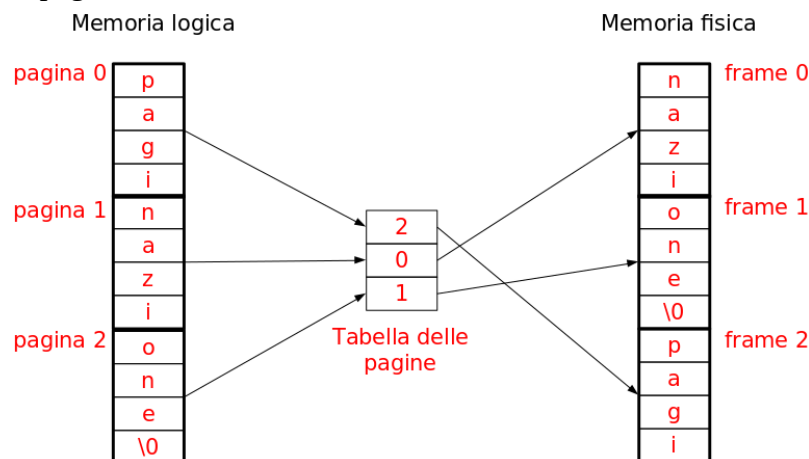
Quando un processo viene allocato in memoria vengono reperiti ovunque in memoria un numero sufficiente di frame per contenere le pagine del processo (con quell'ovunque si intende che non ci interessa della locazione del frame che viene allocato: se un processo ha bisogno di 3 pagine, alloco un frame, e poi un altro frame che può essere da tutta altra parte nella memoria fisica etc...). Ciò implica che le pagine assegnate a questo processo non devono essere necessariamente contigue.

[I frame contengono le pagine!!!]

Ecco un esempio di funzionamento della paginazione:



Per permettere ciò, occorre una MMU più complessa, capace di gestire questa situazione. Il processo vede solo gli indirizzi logici, e quindi ogni processo (per ora lo visualizziamo in questo modo, ma vedremo che ci saranno delle complicazioni) ha una tabella che permette di riordinare in modo corretto le varie pagine, rilocandole correttamente.



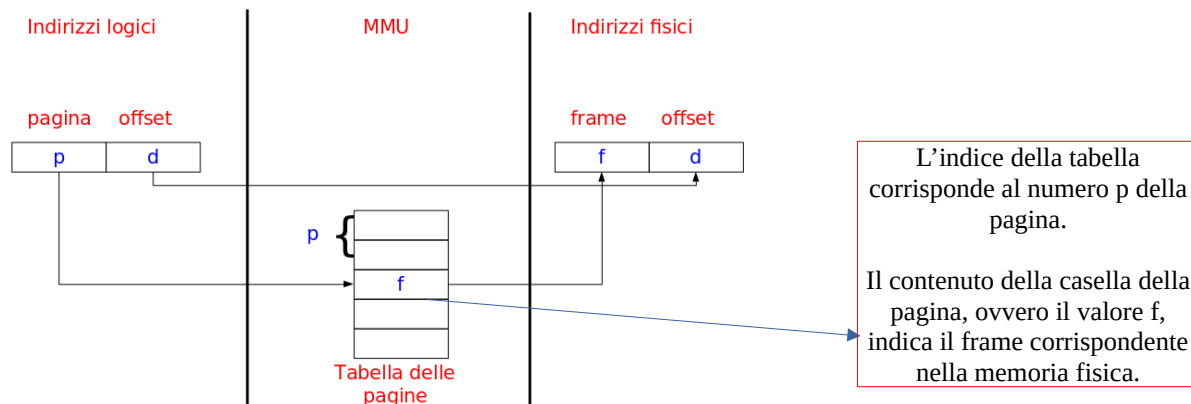
L'ampiezza di una pagina deve necessariamente essere una potenza del 2, perché la trasformazione degli indirizzi diventa troppo onerosa. Dobbiamo scegliere l'ampiezza delle pagine tenendo a mente che con pagine troppo piccole, la tabella delle pagine cresce di dimensioni, mentre con pagine troppo grandi, lo spazio di memoria perso per frammentazione interna può essere considerevole.

Alcuni valori tipici di ampiezza di pagina sono: 1KB, 2KB, 4KB.

Col comando `getconf PAGE_SIZE`, possiamo vedere quanto è grande la pagina nel nostro sistema.

[digressione] Il buddy system consiste nell'avere un'area, separarla dimezzandola fino a trovare la minima pagina capace di contenere una data cosa. In Linux, questa tecnica viene usata per il caricamento dei moduli. **[fine digressione]**

Vediamo ora una visione semplificata del funzionamento dell'MMU [Noi ora immaginiamo che ogni processo ha una propria tabella delle pagine, ma ciò non è ottimale e vedremo la cosa meglio poche righe dopo.]:



Per capire meglio, bisogna considerare i numeri 4096, 8192, 12288 etc.. sommando ogni volta 4096 e prendiamo la loro rappresentazione binaria, noteremo una cosa fighissima:

0 ->	00000000000000000000	000000000000
...		
4095 ->	00000000000000000000	111111111111
4096 ->	00000000000000000001	000000000000
8191 ->	00000000000000000001	111111111111
8192 ->	00000000000000000010	000000000000
12287 ->	00000000000000000010	111111111111
12288 ->	00000000000000000011	000000000000
16383 ->	00000000000000000011	111111111111
16384 ->	00000000000000000100	000000000000

Questo rappresenta il numero della pagina (numero a sinistra)

Questo rappresenta la posizione all'interno della pagina stessa. (parte a destra)

I frame sono larghi quanto le pagine, ma non possono essere posizionati a caso! Ogni frame dovrà essere posto partendo dal primo indirizzo che ha i primi 12 bit (nel caso delle pagine grandi 4kb) uguali a 0.

Dove mettiamo la tabella delle pagine? In tal caso abbiamo due soluzioni. Possiamo fare uso di:

1. **registri dedicati:** la tabella può essere contenuta in un insieme di registri ad alta velocità all'interno del modulo MMU (o della CPU). Questo approccio sarebbe però troppo costoso (ad esempio: con pagine di 4K, e un processore a 32 bit, il numero di pagine nella page table sarebbe 1M (1.048.576), dunque occorrerebbe un 1 MB di registri, ovvero dei registri davvero troppo capienti).
2. **totalmente in memoria:** in questo modo però il numero di accessi in memoria verrebbe raddoppiato; ad ogni riferimento, bisognerebbe prima accedere alla tabella delle pagine, poi al dato, e il tutto sarebbe completamente inefficiente (si usa di solito questo metodo + TLB).

La vera soluzione sarà l'uso di un TLB, che sta per Translation Lookaside Buffer.

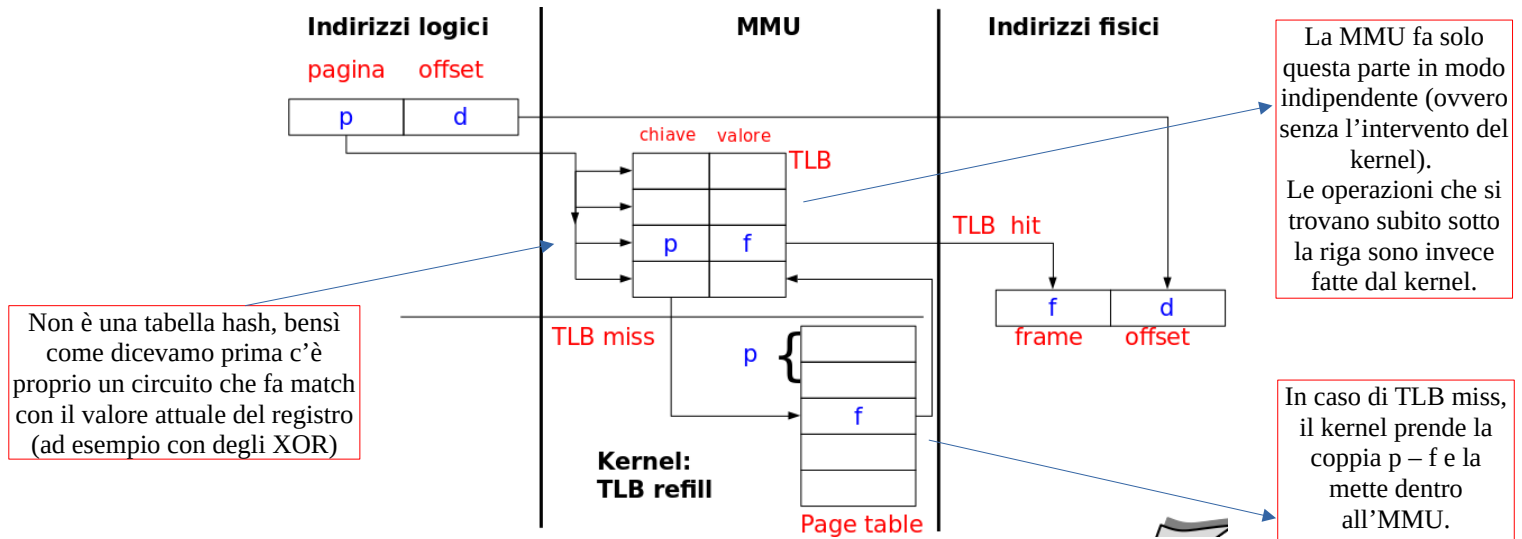
TLB, Translation Lookaside Buffer:

Un **TLB** viene usato per tradurre gli indirizzi logici in indirizzi fisici, ed è costituito da un insieme di registri associativi ad alta velocità. Possiamo vedere il TLB come una cache della tabella delle pagine, e mantiene alcuni elementi della tabella delle pagine. [METANOTA: queste 3 righe(fino al punto) secondo me non sono esatte, ma mi sono limitato a scrivere le parole esatte del prof per completezza] È una matrice associativa (esiste infatti un circuito nella CPU che permette proprio di creare matrici associative velocemente), e fa in modo che quando arriva un indirizzo logico, viene cercato un elemento che fa match con l'indirizzo fisico leggendo tutte le righe della tabella contemporaneamente.

Ogni registro è suddiviso in due parti, una chiave e un valore.

L'operazione di lookup avviene seguendo questi passi:

1. viene richiesta la ricerca di una chiave
2. la chiave viene confrontata simultaneamente con tutte le chiavi presenti nel buffer,
3. se la chiave è presente (TLB hit), si ritorna il valore corrispondente, se la chiave non è presente (TLB miss), si utilizza la tabella in memoria (il sistema operativo fa TLB refill).



La TLB, quindi (come abbiamo specificato prima), agisce come memoria cache per le tabelle delle pagine. Il meccanismo della TLB (come tutti i meccanismi di caching) si basa sul principio di località. L'hardware per la TLB è costoso, e contiene una quantità di registri dell'ordine di 8-2048 registri.

La page table "vera" è localizzata in memoria.

Segmentazione:

In un sistema con segmentazione, dividiamo la memoria di un processo in segmenti, il che significa riconosciamo che un processo ha delle aree distinte che possono essere gestite in modo diverso e che hanno funzionalità diverse. In altre parole, la memoria associata ad un processo è suddivisa in aree differenti dal punto di vista funzionale [Esempio: un processo possiede delle aree "text" che contengono il codice eseguibile e sono normalmente in sola lettura (solo i virus cambiano il codice). In più, queste possono essere condivise tra più processi (codice reentrant). Dopodiché, ci sono le aree dati possono essere condivise oppure no, l'area stack che invece non può essere condivisa etc...].

In un sistema basato su segmentazione uno spazio di indirizzamento logico è dato da un insieme di segmenti.

Un segmento è un'area di memoria (logicamente continua) contenente elementi tra loro affini.

Ogni segmento è caratterizzato da un **nome** (normalmente un indice) e da una **lunghezza**, che può essere variabile ed è la cui ampiezza è dipendente dal programma che si esegue.

Ogni riferimento di memoria è dato da una coppia <nome segmento, offset> (molto simile all'indirizzamento logico nelle pagine quindi).

Spetta al programmatore o al compilatore la suddivisione di un programma in segmenti.

Paginazione e segmentazione sono due cose ortogonali, infatti:

Nella paginazione abbiamo che:

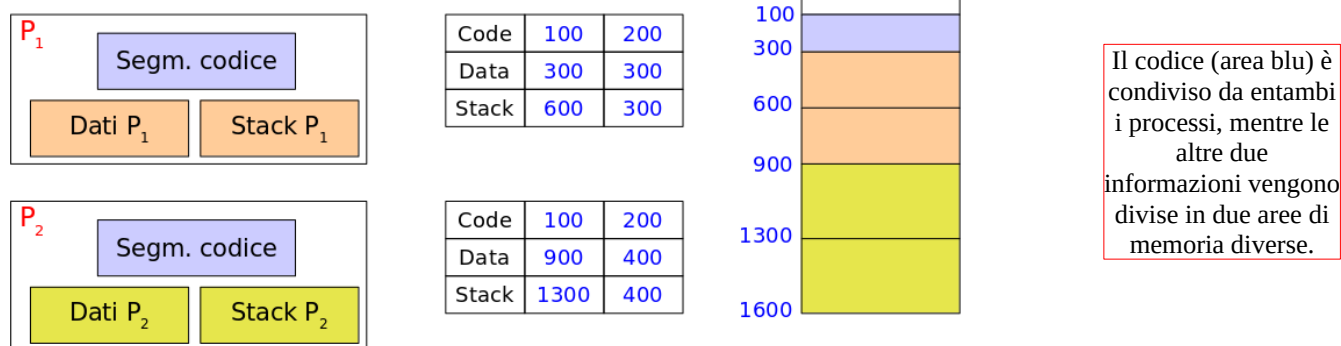
1. la divisione in pagine è automatica.
2. le pagine hanno dimensione fissa
3. le pagine possono contenere informazioni disomogenee (ad es. sia codice sia dati)
4. una pagina ha un indirizzo
5. dimensione tipica della pagina: 1-4 KB

Nella segmentazione abbiamo che:

1. la divisione in segmenti spetta al programmatore.
2. i segmenti hanno dimensione variabile
3. un segmento contiene informazioni omogenee per tipo di accesso e permessi di condivisione
4. un segmento ha un nome.
5. dimensione tipica di un segmento: 64KB – molti MB

[digressione] quando si mappano in memoria dei blocchi (ad esempio usando il comando `mmap`), bisogna dire come proteggere l'accesso a tali blocchi, usando gli argomenti adatti. [fine digressione]

Se abbiamo due processi che stanno eseguendo lo stesso codice (per esempio un editor), la segmentazione consente la condivisione di codice e dati.



La segmentazione aiuta ad avere le aree diverse separate, ma non fa nulla per risolvere la frammentazione. La soluzione sta nel combinare la segmentazione e la paginazione assieme. Infatti, è possibile utilizzare il metodo della paginazione combinato al metodo della segmentazione. Ogni segmento viene suddiviso in pagine che vengono allocate in frame liberi della memoria (non necessariamente contigui). Per utilizzare questo approccio, la MMU deve avere sia il supporto per la segmentazione sia il supporto per la paginazione. In questo modo, abbiamo i benefici di condivisione e protezione forniti dalla segmentazione, e la riduzione della frammentazione esterna grazie alla paginazione.

Mentre la paginazione è utilitatissima, la segmentazione è poco usata. Si preferisce infatti usare la protezione di gruppi di pagine (che possono essere condivise) non attraverso segmenti, ma bensì usando `mmap`.

Memoria Virtuale:

È risaputo che un processo, per lavorare, ha bisogno di memoria. Un processo, di solito, utilizza solo una piccola parte della effettiva memoria di cui ha bisogno (ad esempio, ci sono processi che hanno routine della gestione degli errori, e se non avviene un errore, quelle aree di memoria dedicate alla gestione degli errori non vengono utilizzate. Oppure, ci sono processi che allocano vettori per l'ampiezza massima dei dati da gestire, ma magari il processo ne gestisce meno, oppure moltissime strutture dati sono allocate con la massima capienza ma solo in maniera preventiva, oppure abbiamo una esecuzione a fasi...).

Il concetto di **memoria virtuale** si basa sul pensiero che io, essenzialmente, per risparmiare memoria posso caricare in memoria centrale i dati "utili" che mi servono in quel momento, e tutti gli altri invece

posso anche lasciarli nella memoria secondaria. In questo modo, siccome ho più memoria centrale a mia disposizione, posso anche avviare più processi rispetto a prima. Dunque, si usa un pezzo di memoria secondaria come un'estensione della memoria primaria. Inoltre, questo concetto rispetta anche l'architettura di Von Neumann, che ci dice che le istruzioni da eseguire e i dati su cui operano devono essere in memoria, ma non specifica appunto quale memoria.

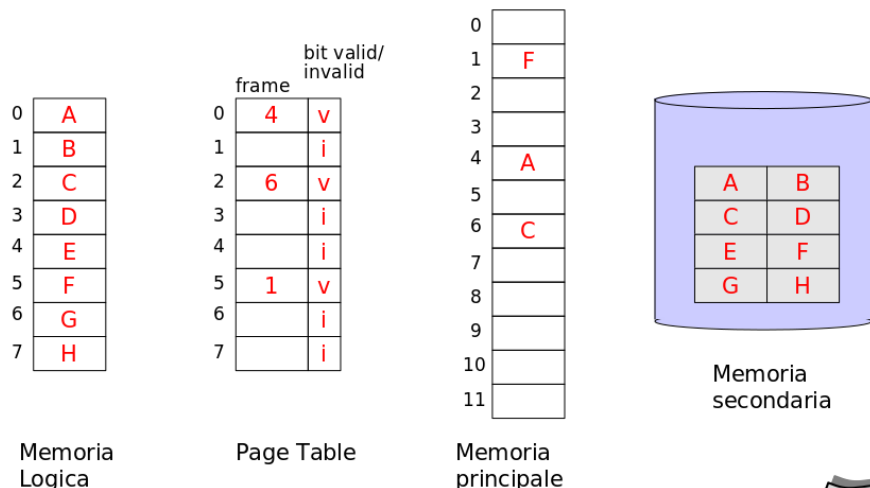
Ogni processo ha accesso ad uno spazio di indirizzamento virtuale che può essere più grande di quello fisico, e gli indirizzi virtuali (logici) possono essere mappati su indirizzi fisici della memoria principale oppure, possono essere mappati su memoria secondaria. In caso di accesso ad indirizzi virtuali mappati in memoria secondaria, i dati associati vengono trasferiti in memoria principale. Se la memoria è piena, si sposta in memoria secondaria i dati contenuti in memoria principale che sono considerati meno utili. [Disclaimer: ricordiamo che la paginazione serve solamente per limitare la frammentazione, **e si può fare paging senza memoria virtuale**, semplicemente le pagine sono sparpagliate nei frame in modo che possiamo allocare in maniera non contigua. Però, il meccanismo della paginazione può essere usato anche per supportare la memoria virtuale.]

Il meccanismo della paginazione può essere usato anche per supportare la memoria virtuale, e in questo caso si parla di **paginazione a richiesta** (demand paging). In poche parole, nella tabella delle pagine si aggiunge un bit v, che sta per *valid*, che indica se la pagina è presente in memoria centrale oppure no. Quando un processo tenta di accedere ad un pagina non in memoria, il processore genera un trap (page fault) (o diventa un caso particolare del TLB refill) [infatti, se abbiamo una macchina con TLB, diventa un caso particolare del TLB miss... ECCO COSA SUCCEDDE IN UNA MACCHINA CON TLB:

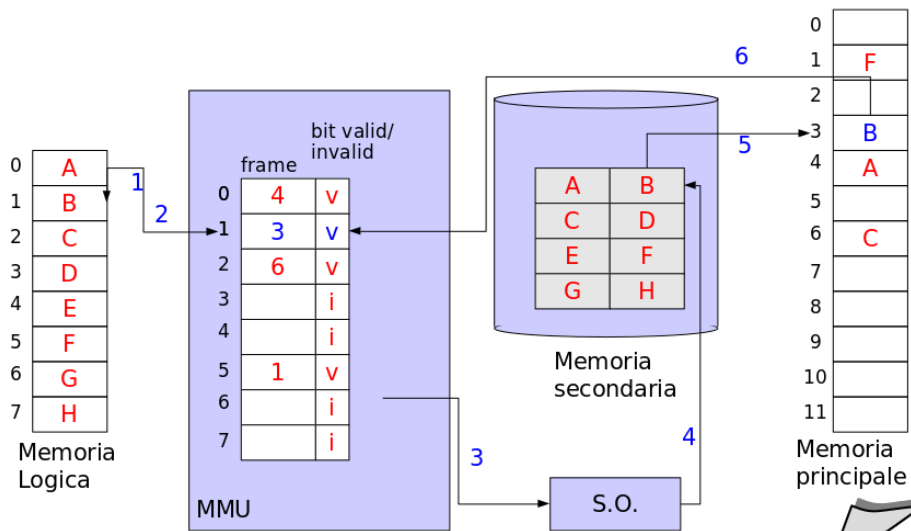
Quando un processo tenta di accedere ad un indirizzo che non è presente in memoria centrale, siccome non sarà presente nemmeno nel TLB, il kernel (che viene chiamato dopo una trap) tenterà di trovare quell'indirizzo nella memoria centrale (siccome non è in memoria centrale non c'è nell'MMU l'elemento di trasformazione) e si accorgerà che non è presente...

Quindi la routine di TLB refill genera un'altra trap in maniera tale che il sistema si accorga che manca una pagina].

Un componente del s.o. (**pager**) si occupa di caricare la pagina mancante in memoria, e di aggiornare di conseguenza la tabella delle pagine.



Ecco un esempio di ciò che avviene quando si gestisce un page-fault:



Seguiamo ciò che avviene basandoci sui numeri scritti:
Il codice nella pagina A del processo fa riferimento alla pagina B (1), che non è presente (inizialmente, quindi per ora facciamo finta che le scritte in blu non ci siano) nella page table della MMU (2). Allora si genera una trap “page-fault”, il kernel viene chiamato (3) e va a cercare quindi la pagina in memoria secondaria (4). Quando questa viene trovata, la si carica in memoria centrale (5) e si aggiorna la page table (6) (e anche la TLB se abbiamo un sistema che supporta il TLB).

In alcuni sistemi operativi (Linux incluso, infatti io sta cosa l’ho trovata anche in Manjaro), vediamo che vengono usati i termini di paging e di swapping come sinonimi, ma ciò è errato: con **swapping**, infatti, intendiamo il caricamento/scaricamento dell’intero processo (in particolare, dell’intera area di memoria di un processo) dalla memoria secondaria (dalla memoria secondaria alla memoria principale si dice swap-in, mentre dalla memoria principale alla memoria secondaria swap-out).

L’area di swap, infatti sarebbe da chiamare l’area di paging.

[METANOTA: il prof non ha detto ste cose, ma c’erano sulle slides quindi le aggiungo]

La paginazione su richiesta può essere vista come una tecnica di swap di tipo lazy (pigro), infatti viene caricato solo ciò che serve. Se abbiamo un processo che non fa nulla per un po’, viene swappato fuori siccome non viene considerato utile.

Torniamo alla page-fault. Cosa succede in mancanza di frame liberi? In tal caso, dobbiamo liberarne uno, e quindi occorre trovare la pagina in memoria meno “utile”, e per stabilir quale sia la pagina meno utile, si fa uso degli **algoritmi di rimpiazzamento**.

[Questa parte è stata aggiunta dal prof alla fine, e non è presente nelle slides]

Il dirty bit serve per sapere se una pagina è stata modificata oppure no, non è necessaria, ma è bene che ci sia, perché se una pagina è stata modificata, quando questa viene liberata, dobbiamo salvarla, altrimenti possiamo non fare nulla. Questo può essere implementato in vari modi, o la MMU lo prevede, quindi se accediamo ad una pagina in scrittura la MMU accende il bit, oppure ci sono tecniche per realizzarlo, per esempio si pone la pagina in divieto di scrittura quando viene caricata, se accendiamo alla pagina in scrittura avviene un errore d’accesso che però è un falso errore d’accesso, siccome questo errore di scrittura viene usato per vedere se effettivamente c’è stato un accesso in scrittura, e allora la routine del memory manager quando “prende” l’errore in scrittura, gestisce il dirty bit e dà il permesso al processo di accedere in scrittura.

Lezione del (23/03/2021) – Algoritmo di rimpiazzamento, LRU, LFU, Trashing

Recap della lezione precedente:

Quando non c’è un elemento nella TLB, come abbiamo visto, il kernel chiama una route di TLB refill in cui carica i campi corrispondenti alle informazioni della pagina dalla page table alla TLB. La paginazione si può avere con o senza memoria virtuale, e nel caso ci sia solo paginazione, i casi sono due: o la pagina c’è nella tabella delle pagine, e quindi possiamo caricare la corrispondenza pagina-frame nella TLB, oppure c’è un errore! [METANOTA: i concetti di queste 3 righe sono abbastanza

confusi, e ancora mi sono limitato a scrivere esattamente le parole del prof.] Questa idea viene usata ancora in maniera più proficua con la memoria virtuale, quindi quando c'è un TLB miss si può scoprire, attraverso la page table, che la pagina non c'è in memoria, e si scatena in questo modo il caricamento della pagina.

Quindi la paginazione serve più che altro serve per ridurre la frammentazione, poi si può usare anche per la memoria virtuale.

La segmentazione dobbiamo vederla come una idea del passato, e delle funzionalità simili si possono ottenere con le page-table gerarchiche, in cui c'è un primo livello di page table, con degli indici di page-table, e un secondo livello di page table che punta alle pagine.

Algoritmi di rimpiazzamento:

Un algoritmo che il pager può adottare è l'algoritmo di **demand paging**, che consiste in questi passi:

1. Individua la pagina in memoria secondaria
2. Individua un frame libero
3. Se non esiste un frame libero
 - I. richiama algoritmo di rimpiazzamento
 - II. aggiorna la tabella delle pagine (invalida pagina "vittima")
 - III. se la pagina "vittima" è stata variata, scrive la pagina sul disco (quindi viene salvata)
 - IV. aggiorna la tabella dei frame (frame libero)
4. Aggiorna la tabella dei frame (frame occupato)
5. Leggi la pagina da disco (quella che ha provocato il fault)
6. Aggiorna la tabella delle pagine (caricato il TLB)
7. Riattiva il processo

Queste due operazioni (5 e 3.3) sono lente siccome nascondono delle operazioni di I/O. Quindi, nel 3.3 in particolare, c'è un problema di programmazione concorrente, e perciò, ancora prima di scaricare la pagina, dovrà "invalidarla" rendendola non disponibile. In questo modo, se un altro processo prova ad accedervi, genererà un page fault. In oltre, questo frame non è considerato fra quelli liberi. Nel 3.3 la pagina viene dichiarata invalida, mentre nel 5 viene dichiarata valida.

L'**algoritmo di rimpiazzamento** si occupa di trovare la pagina "vittima", entra in funzione quando un processo ha bisogno di un frame siccome c'è una pagina che manca, ma tutti i frame sono occupati.

L'obiettivo di un algoritmo di rimpiazzamento è di minimizzare il numero di page fault. Per valutare l'algoritmo di rimpiazzamento si esamina come essi si comportano applicati ad una **stringa di riferimenti in memoria** (ovvero la sequenza degli indirizzi che uno o più processi richiedono, ad esempio se legge in A e scrive in B, richiederà/accederà agli indirizzi A e B), che possono essere generate esaminando il funzionamento di programmi reali o con un generatore di numeri random. La stringa di riferimenti può essere limitata ai numeri di pagina, in quanto non siamo interessati agli offset (ovvero al "cursore" all'interno della pagina).

Facciamo un esempio. Prendiamo una pagina da 16 byte, ovvero 4 bit di offset e il resto di indirizzo di pagina: sia la stringa di riferimento completa (in esadecimale)

71,0a,13,25,0a,3f,0c,4f,21,30,00,31,21,1a,2b,03,1a,77,11, la stringa di riferimento delle pagine sarà (in esadecimale, con pagine di 16 byte) 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,1.

Torniamo ai nostri algoritmi di rimpiazzamento. Quello più semplice è l'**algoritmo FIFO**, che sta per first-in-first-out. Essenzialmente, quando il pager si accorge che non c'è spazio in memoria e che occorre caricare un'altra pagina ma che non ci sono frame liberi, va a vedere quello che aveva caricato prima di tutti gli altri. Quindi, viene individuata come vittima il frame che è stato caricato in memoria per primo. Il vantaggio di questo approccio è che è molto semplice da implementare e che non richiede particolari supporti hardware. Tuttavia, c'è il rischio che vengano scaricate delle pagine che magari sono comunque utilizzate da molti processi nonostante siano state caricate per prime (il tempo in coda non ci dice effettivamente quanto essa sia utilizzata), e perciò magari le buttiamo via e poi ci tocca ricaricarle.

Facciamo un esempio:

- numero di frame in memoria: 3
- numero di page fault: 15 (su 20 accessi in memoria)

tempo	1 20																			
stringa riferim.	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
pagine in memoria.	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1

Ricordiamo che i numeri si riferiscono al numero della pagine. Es. nella casella numero 2/5/7 si fa rimento alla pagina 0, anche magari si svolge un accesso a indirizzi diversi all'interno della pagina.

Ogni colonna rappresenta un unità di tempo. Vediamo che al $t=4$, si attua effettivamente il rimpiazzamento FIFO: si sostituisce 7, la pagina più vecchia, con 2.

Se facciamo lo stesso esempio con una memoria di 4 frame, si ha un aumento del numero di pagefault! Normalmente, infatti, quello che ci aspettiamo è che aumentando il numero di frame, il numero di page fault diminuisca. Tuttavia non è così: non è detto che aumentando la quantità di memoria, la memoria virtuale lavori meglio. Questo fenomeno si verifica in particolare usando alcuni algoritmi di rimpiazzamento, e viene chiamato **anomalia di Belady**.



Perciò, noi vogliamo trovare algoritmi che siano privi di questa anomalia.

Grazie a Dio, esiste un algoritmo ottimale. Usando infatti l'**algoritmo MIN**, si ottiene il numero minimo di page fault. Questo algoritmo seleziona come pagina vittima una pagina che non sarà più acceduta o la pagina che verrà acceduta nel futuro più lontano. Sfortunatamente però, questo è un algoritmo esclusivamente teorico, perché richiederebbe la conoscenza a priori della stringa dei riferimenti futuri del programma, e questa è una informazione che semplicemente noi non abbiamo. Tuttavia, ha comunque una certa utilità, in quanto viene utilizzato a posteriori come paragone per confrontare le performance degli algoritmi di rimpiazzamento reali, e vedere la loro "distanza" con l'esempio ottimale.

Ecco un esempio della sua esecuzione:

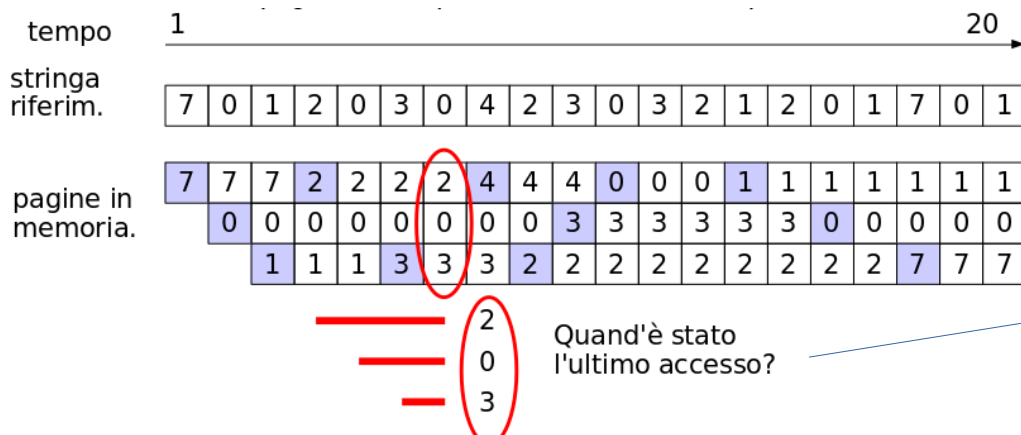
tempo	1																				20									
stringa riferim.	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1										
pagine in memoria.	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7										
		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0										
			1	1	1	3	3	3	3	3	3	3	3	3	1	1	1	1	1	1										
Quand'è il prossimo accesso?			7																											
			0																											
			1																											

La lunghezza della riga indica a quale distanza di tempo verrà effettuato il prossimo accesso. Come si può vedere, al passo 4 viene scelto 7 come pagina da sostituire. Possiamo anche misurare la distanza osservando la stringa di riferimento.

Algoritmo Least Recently Used:

Un algoritmo di rimpiazzamento famoso è il **Least Recently Used (LRU)**. Questa politica consiste nel selezionare come pagina vittima la pagina che è stata usata meno recentemente nel passato.

È basato sul presupposto che la distanza tra due riferimenti successivi alla stessa pagina non vari eccessivamente, perciò stima la distanza nel futuro utilizzando la distanza nel passato.



La lunghezza della riga, sta volta, indica la distanza di tempo dall'ultimo accesso. Come si può vedere, al passo 7 viene rimpiazzata la pagina 2, siccome è quella che ha avuto l'accesso più lontano. Anche qui, possiamo anche misurare la distanza osservando la stringa di riferimento.

LRU è implementabile, ma richiede un supporto hardware, ovvero l'MMU. La MMU deve registrare nella tabella delle pagine un time-stamp quando accede ad una pagina, e il time-stamp può essere implementato come un contatore che viene incrementato ad ogni accesso in memoria. **Questo processo è pesantissimo per la MMU**, e la quantità di accessi in memoria è altissima, talmente alta che forse non basterebbe nemmeno un contatore a 64 bit. Questo perché:

- L'MMU dovrà mantenere anche 8 byte per ogni pagina in memoria.
- Bisogna gestire l'overflow dei contatori (wraparound), che si verificherà molto spesso se usiamo contatori piccoli.
- i contatori devono essere memorizzati in memoria e questo richiede accessi aggiuntivi alla memoria.
- In più, la tabella deve essere scandita totalmente per trovare la pagina LRU (e a volte può contenere anche un milione di elementi).

Queste sono richieste troppo complesse per un MMU.

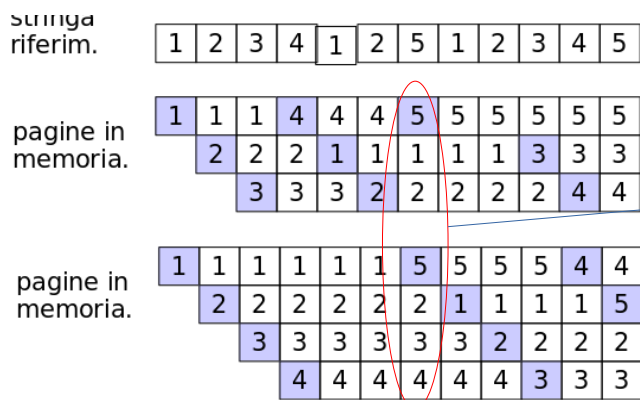
Algoritmi a stack

Facciamo un piccolo detour per spiegare un concetto importante che vedremo subito dopo: dobbiamo prima vedere che cos'è un algoritmo a stack. Data una stringa di riferimenti s e indicato con $S_t(s, A, m)$ l'insieme delle pagine mantenute in memoria centrale al tempo t dell'algoritmo A , e infine data una memoria di m frame relativamente alla stringa s , **un algoritmo di rimpiazzamento viene detto "a stack" (stack algorithm) se per ogni istante t e per ogni stringa s si ha:**

$$S_t(s, A, m) \subseteq S_t(s, A, m+1)$$

In altre parole, se l'insieme delle pagine in memoria con m frame è sempre un sottoinsieme delle pagine in memoria con $m+1$ frame, per ogni stringa per ogni tempo per ogni ampiezza di memoria.

COTRO-Esempio:



In questo contro esempio, in cui usiamo un algoritmo di rimpiazzamento FIFO vediamo che all'istante cerchiato, l'insieme delle pagine dello schema più in alto (5,1,2) NON è sott'insieme di quello più in basso (5,2,3,4). Dunque, il FIFO non è a stack.

La caratteristica più fida di un algoritmo a stack è che **non genera casi di Anomalia di Belady**.

(Esatto! Proprio non li genera!) Dimostriamo questo con un esempio:

Prendiamo due insiemi $St1(s, A, m)$ e $St2(s, A, m+1)$, il primo sott'insieme del secondo.

Quindi, supponiamo che il primo abbia (p_1, p_2, p_3, \dots) elementi, mentre il secondo abbia $(p_1, p_2, p_3, \dots, p_0)$ elementi. Per avere anomalia di Belady, occorrerebbe che ci sia page fault con la memoria più grande ($St2$) e non con la memoria più piccola ($St1$). Analizziamo quindi i vari casi:

- caso 1: la pagina sta in $p_1, p_2, p_3 \rightarrow$ non c'è page-fault né con ampiezza m né con ampiezza $m+1$
- caso 2: la pagina è in $p_0 \rightarrow$ c'è page-fault con m , ma non con $m+1$
- caso 3: la pagina non sta in nessuno dei precedenti \rightarrow page-fault sia in m che in $m+1$

Non è quindi possibile che ci sia page-fault in $m+1$ e non in m , il risultato è quindi che l'algoritmo a stack non soffre di anomalia di Belady.

L'algoritmo di LRU è a stack, [METANOTA: il prof ha spiegato un po' male questa parte, mi limito a scrivere le esatte parole che ha detto] siccome se prendiamo lo stato in memoria in qualsiasi momento, noi andiamo a scegliere su un ordine totale. Detto brutalmente: se prendiamo un unico istante di rilevamento, la hit-parade dei 7 brani più gettonati questa settimana è un sott'insieme degli 8 più sentiti questa settimana, siccome stiamo prendendo quelli di un ordine totale. Data una stringa di riferimenti statica, abbiamo in ogni istante che lo stato della memoria m frame è un sott'insieme dello stato $m+1$ frame, e andiamo ad esaminare quale fra gli m frame è quello accaduto in un passato più remoto, m o $m+1$. [fine METANOTA]

[**NOTA**: il prof non ha visto questa parte delle slides ma io la riporto comunque. Riprende la spiegazione normale dopo queste 5 righe.]

Una implementazione diversa consiste nel mantenere uno stack di pagine (implementazione **basata su Stack**) tutte le volte che una pagina viene acceduta, viene rimossa dallo stack (se presente) e posta in cima allo stack stesso. In questo modo in cima si trova la pagina utilizzata più di recente e in fondo si trova la pagina utilizzata meno di recente. Da notare: l'aggiornamento di uno stack organizzato come double-linked list richiede l'aggiornamento di 6 puntatori, la pagina LRU viene individuata con un accesso alla memoria, esistono implementazioni hardware di questo meccanismo. [**fine nota**]

LRU è troppo costoso per gestire e mantenere le informazioni del contatore. Cosa si può fare per migliorare questa situazione, in modo da assicurare una spesa della MMU che sia gestibile? Facciamo il minimo, ovvero facciamo che la MMU abbia un bit (detto **reference bit**) nella TLB che metterà a 1 se la pagina viene acceduta, che è un'operazione semplice per la MMU.

Inizialmente tutti i reference bit vengono posti a 0, e durante l'esecuzione dei processi, le pagine in memoria vengono accedute e i reference bit vengono posti a 1 dall'MMU. Periodicamente, è possibile osservare quali pagine sono state accedute e quali no osservando i reference bit.

[Il prof non ha spiegato questo, ma ha fatto un esempio usando questa tecnica, quindi io la riporto per completezza].

Io però in questo modo non so in che ordine questi vengano acceduti. A questo risponde il Additional-Reference-Bit-Algorithm, in cui possiamo aumentare le informazioni di ordine "salvando" i reference bit ad intervalli regolari (ad es., ogni 100 ms), ad esempio: manteniamo 8 bit di "storia" per ogni pagina.

Il nuovo valore del reference bit viene salvato tramite shift a destra della storia ed inserimento del bit come most signif. bit. La pagina vittima è quella con valore minore; in caso di parità, si utilizza una disciplina FIFO.

Es: abbiamo una area di memoria nella TLB dedicata alla storia di una pagina, in cui i bit vengono posti ad 1 ogni volta che si fa un accesso. Ad ogni accesso, tutti i bit vengono shiftati di 1, aggiungendo uno 0 se non stati acceduti e un 1 in caso contrario. In questo modo, mi basta scegliere la pagina con la storia di valore minimo e buttare via quella.

LRU Second-chance:

L'algoritmo di **second-chance** è un algoritmo applicato spesso nei SO moderni. È conosciuto anche come algoritmo dell'orologio. Corrisponde ad un caso particolare dell'algoritmo precedente, dove la dimensione della storia è uguale a 1, e quindi la MMU ha bisogno solo di un bit di riferimento.

Le pagine di memoria vengono gestite come una lista circolare. Basti pensare alla tabella delle pagine, che mantiene l'informazione riguardo a quale frame corrisponde a quale pagina, sia un vettore pensato come circolare, ovvero che a partire dalla posizione successiva all'ultima pagina caricata, si scandisce la lista con la seguente regola:

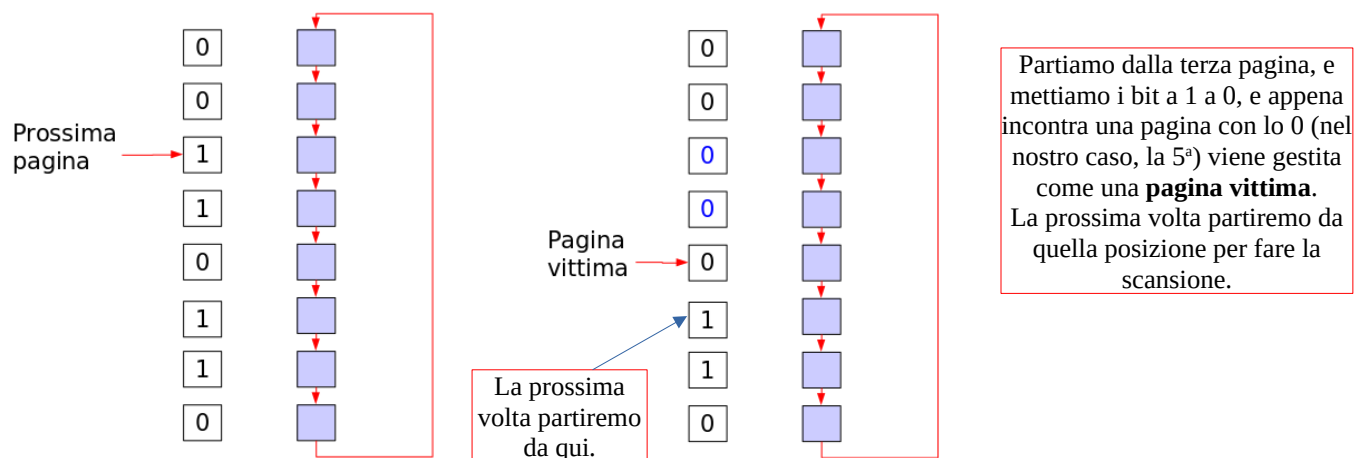
- se la pagina è stata acceduta (reference bit a 1), **il reference bit viene messo a 0.**
- se la pagina non è stata acceduta (reference bit a 0), **la pagina selezionata è la vittima.**

Ciò avviene durante un page fault o un TLB refill. In poche parole, dall'ultima volta che c'è stato un "accesso" alla tabella da parte dell'MMU, si controlla il reference bit.

La scansione della pagina "riparte" dall'ultima che è stata controllata durante l'ultima scansione, e prosegue, appunto, in modo circolare.

Nel caso in cui il reference bit di tutte le pagine siano a 1, proseguirà a mettere il bit a 0 a tutte le pagine, e siccome si muove in modo circolare, quando si trova nell'ultima posizione del vettore, ritornerà alla prima pagina e siccome il suo bit sicuramente sarà a 0, la toglierà.

[Piccola precisazione: il bit di riferimento è situato nella tabella delle pagine, che a sua volta è situato in memoria centrale.]



Questo algoritmo è molto semplice da implementare e non richiede capacità complesse da parte della MMU. Inoltre, è molto utilizzato.

Si chiama second chance siccome se però la pagina è stata acceduta, gli si dà una "seconda possibilità" (second chance).

Least frequently used (LFU):

in sostanza, si mantiene un contatore del numero di accessi ad una pagina, la frequenza e' il valore del contatore diviso per il "tempo" di permanenza in memoria (i.e. Il numero di accessi con quella pagina presente). La pagina con il valore minore viene scelta come vittima.

È un algoritmo **teorico**, siccome mantenere un contatore e, allo stesso tempo, ricordarsi di quando una pagina sia stata caricata è costosissimo. Inoltre, è meno attendibile di LRU, perché una pagina può essere stata acceduta frequentemente per un certo periodo di tempo, poi entra in dimenticanza. Quindi, se una pagina viene utilizzata frequentemente all'inizio, e poi non viene più usata, non viene rimossa per lunghi periodi.

Un altro algoritmo puramente teorico è MFU, ma a noi questo non ce ne frega un cazzo. Il prof infatti l'ha barrato. LOL.

Trashing:

Con **algoritmo di allocazione** (per memoria virtuale) si intende l'algoritmo utilizzato per scegliere quanti frame assegnare ad ogni singolo processo. Esistono due tipi di allocazione:

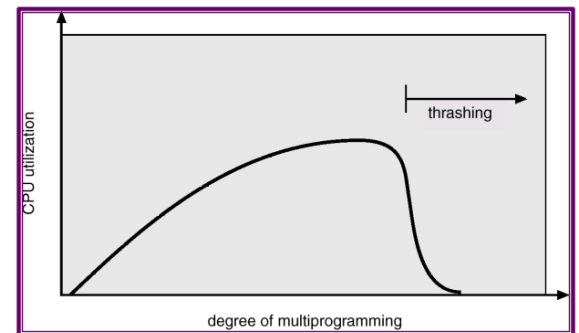
- **Allocazione locale:** ogni processo ha un insieme proprio di frame, ovvero si ha un pool di frame riservato ad un singolo processo e il processo caricherà o scaricherà le sue pagine. È poco flessibile, siccome è una allocazione molto statica.
- **Allocazione globale:** tutti i processi possono allocare tutti i frame presenti nel sistema (sono in competizione). In questo modo, posso allocare più frame a un processo nel momento in cui abbia bisogno di più risorse, e poi nel momento successivo invertito la situazione. Il problema è che se tutti i processi competono per allocare frame, può portare a **trashing**.

Un processo (o un sistema) si dice che è in **trashing** quando spende più tempo per la paginazione che per l'esecuzione. Come specificato prima, in un sistema con allocazione globale, si ha trashing se i processi tendono a "rubarsi i frame a vicenda", i.e. **non riescono a tenere in memoria i frame utili a breve termine (perché altri processi chiedono frame liberi) e quindi generano page fault ogni pochi passi di avanzamento**. Succede molto spesso quando si hanno molti processi in esecuzione che richiedono molta memoria, e poca memoria disponibile. Ad esempio, se un processo richiede una pagina, avverrà page fault, e succede che questo processo prima o poi otterrà la pagina, ma magari nel mentre un altro processo gliene ruba un'altra, e così dopo avverrà un altro page fault che dovrà gestire e così via. Inoltre, caricare e scaricare una pagina impiega tempo. **Per uscire da una situazione di trashing, ci basta sospendere (senza necessariamente arrestare) alcuni processi.**

Ci sono delle metodologie algoritmiche per calcolare il trashing. Tra l'altro, il trashing è stato scoperto quando si creò una macchina in cui appena il carico di CPU era basso, si aggiungeva un programma da eseguire. Questo però è stato deleterio, infatti quando la CPU va in trashing, non va ad un 1%, bensì viene usata allo 0%, va letteralmente in idle, la coda dei processi ready è vuota e il caricamento della pagina (che è un I/O involontario, per questo sta in idle). Questo dunque causò un effetto valanga, siccome dato che la CPU era a 0% si aggiungevano continuamente tanti processi.

[Stessa cosa della parte poco importante, ma detta meglio] esaminiamo un sistema che accetti nuovi processi quando il grado di utilizzazione della CPU è basso. Se per qualche motivo gran parte dei processi entrano in page fault: la ready queue si riduce, e il sistema sarebbe indotto ad accettare nuovi processi... **E' UN ERRORE!**

Il sistema così genererà un maggior numero di page fault, e di conseguenza diminuirà il livello della multiprogrammazione.



Working Set:

Per gestire il trashing, si usa il concetto di **working set**, ed è un modo di stimare la quantità di pagine di cui avrà bisogno un processo.

Si definisce **working set di finestra Δ** l'insieme delle pagine accedute nei più recenti Δ riferimenti. [QUESTO NON C'ENTRA NULLA CON GLI ALGORITMI DI RIMPIAZZAMENTO]

È una rappresentazione approssimata del concetto di località, se una pagina non compare in Δ riferimenti successivi in memoria, allora esce dal working set; non è più una pagina su cui si lavora attivamente

Esempio: $\Delta = 5$ (qui sotto è rappresentata una stringa dei riferimenti di un processo)

La cosa che dobbiamo considerare è che in questo span, al processo servono 4 pagine su 5 della finestra

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

{ 0,1,2,7 }

{ 0,1,2 }

Questa volta, accediamo a 3 pagine diverse in una finestra di 5 pagine.

Come dicevamo prima, se l'ampiezza della finestra è ben calcolata, il working set è una buona approssimazione dell'insieme delle pagine "utili"... sommando quindi l'ampiezza di tutti i working set dei processi attivi, questo valore **deve essere sempre minore del numero di frame in memoria disponibili, altrimenti il sistema è in trashing**. Nel caso ci sia trashing, tenendo a mente che **IL TRASHING NON È DEADLOCK**, possiamo uscirne sospendendo temporaneamente uno o più processi (senza necessariamente arrestarli). Quando ci sono sufficienti frame disponibili non occupati dai working set dei processi attivi, allora si può attivare un nuovo processo. Se al contrario la somma totale dell'ampiezza dei working set supera il numero totale dei frame, si può decidere di sospendere l'esecuzione di un processo.

Se si sceglie **Δ troppo piccolo**, si considera non più utile ciò che in realtà serve, quindi si sottovaluta il numero di pagine necessarie per il processo. In questo caso, ci saranno falsi negativi di trashing. Se si sceglie **Δ troppo grande** si considera utile anche ciò che non serve più, quindi si sopravvaluta il numero di pagine necessarie. In questo caso, ci saranno falsi positivi di trashing.

Lezione del (26/03/2021) – Gestione I/O, SSD, HDD e RAID

Accenni di pre-lezione: **[METANOTA: riguardare questa parte che va 18:00–35:00]**

Dimostrazione di un algoritmo a stack: un algoritmo è a stack quando $S(A, s, m, t)$ è sott'insieme di $S(A, s, m+1, t)$, dove A è l'algoritmo, s è la stringa dei riferimenti, m e $m+1$ sono le capienze e t è l'istante di tempo al quale si trova la memoria e con S si intende lo stato della memoria.

Facciamo una dimostrazione per induzione matematica. Con induzione si intende che noi verifichiamo che la proprietà sia vera per il valore 0 (o lo stato iniziale), e poi dimostriamo che se $P(n)$ allora $P(n+1)$. In base all'ipotesi e alla definizione di algoritmo a stack, sappiamo che nella sottosequenza iniziale che contiene un numero di pagine distinte minore o uguale di m lo stato della memoria con m e $m+1$ è identico. Quando la sequenza include l' $(m+1)$ esimo riferimento a pagina distinta, mentre con $m+1$ frame rimarranno in memoria, con m frame uno verrà sostituito (non si sa quale e questo dipende da A) ... In ogni caso, fino a questo punto l'ipotesi è valida, ovvero $S(A, s, m, t)$ is subset of $S(A, s, m+1, t)$ (possiamo dunque considerare questo come il caso base). Dobbiamo ora fare il caso induttivo: vogliamo dimostrare che $S(A, s, m, t+1)$ is subset of $S(A, s, m+1, t+1)$ (ovvero se l'ipotesi è vera anche nell'istante successivo).

Prendiamo ora due set di pagine, in modo che $S(A, s, m, t)$ sia composto dai frame p_0, p_1 e p_2 e che $S(A, s, m+1, t)$ sia composto dai frame p_0, p_1 e p_2

Gestione I/O:

Durante la spiegazione di questa parte, dobbiamo tenere a mente che quando si parla di driver stiamo facendo riferimento a una parte software, quando invece parliamo di controller stiamo facendo riferimento a una componente hardware.

I **controller** sono delle interfacce fisiche che consentono di accedere dal BUS all'unità collegata, e tutti questi controller fisici hanno delle loro originalità. Quella parte del sistema operativo che deve prendersi carico di tener buoni tutti i controller comunicando con essi sono i driver.

Le modalità d'accesso possono essere di tipo *sequenziale* (come le linee seriale o i modem) oppure *random* (come nei CD-ROM, anche se in tal caso è pseudo-random).

Il trasferimento poi può essere *sincrono* (come nei masterizzatori), ovvero in cui se si fa una azione bisogna completarla prima di farne una nuova, e quindi ci sono dei vincoli temporali molto stretti. o *asincrono* (come nei mouse).

Le modalità di trasferimento possono essere a blocchi o caratteri. Questa particolare caratteristica l'abbiamo esplorata anche nel primo semestre, in cui abbiamo visto che nella cartella dev, se guardavamo i permessi di questi file, a seconda della presenza del carattere c o b , capivamo se si trattava di dispositivi a carattere o a blocchi rispettivamente.

La **comunicazione a blocchi** è particolarmente in uso nei sistemi moderni, ed è caratterizzato dal fatto che i dati vengono letti/scritti a blocchi, tipicamente di 512-1024 byte. Il driver non gestisce l'accesso al filesystem, bensì al dispositivo vero e proprio es. SSD, e fornisce quindi fornisce solo l'accesso raw. L'accesso al file system è quindi uno strato in più che si può mettere, ma non c'entra con le operazioni di I/O. L'interfaccia a blocchi permette anche l'accesso tramite memory mapped I/O, in cui il contenuto di un file viene mappato in memoria, e si esegue l'accesso tramite operazioni di load e store del processore. Le operazioni di raw I/O includono operazioni di read, write e seek per i blocchi.

La **comunicazione a caratteri** consiste nel fatto che i dati vengono letti/scritti un carattere alla volta (es. terminale UNIX). Le operazioni di raw I/O includono operazioni di get/put di un singolo carattere. Si può anche fare *bufferizzazione*, attraverso la lettura/scrittura di "una linea alla volta" (es. vim bufferizzazione siccome la digitazione di ogni carattere "perturba" il funzionamento del programma, mentre invece nel terminale UNIX solo dopo che inviamo la riga abbiamo un feedback).

Esistono per fare accessi raw ed esistono driver per fare memory mapped, e quando si configura un sistema, a seconda del servizio, si dice se è meglio usare RAW o memory mapped. Per esempio, l'area di swap si può vedere come un caso particolare di memory mapped, perché il sistema va a cercare sul disco facendo uso di indirizzi. Altro esempio di memory mapped è il framebuffer

Progettazione di dispositivi I/O:

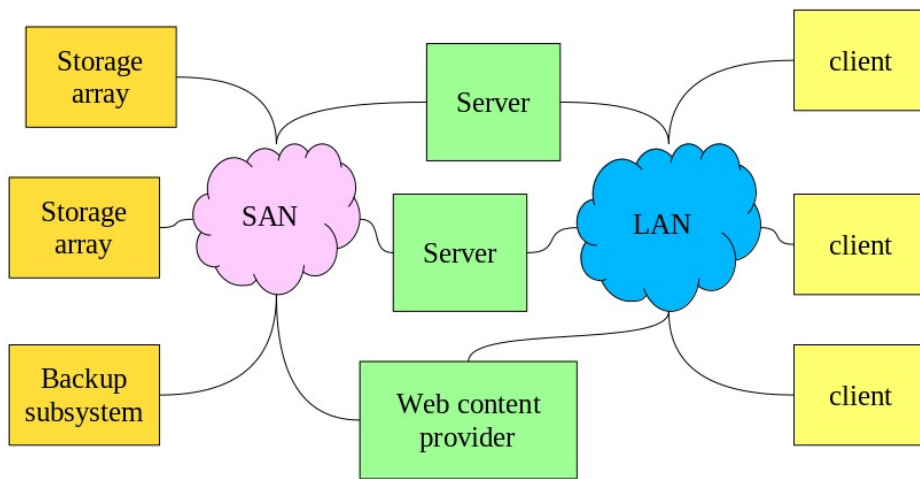
Alcune tecniche usate nella gestione dei dispositivi di I/O sono:

- **Buffering**, è utile per disaccoppiare la produzione dei dati con il loro consumo. Può essere usato per gestire una differenza di velocità tra il produttore e il consumatore di un certo flusso di dati, oppure per gestire la differenza di dimensioni nell'unità di trasferimento.
- **Caching**, permette di mantenere parte del contenuto di memorie più lente all'interno di memorie più veloci.
- **Spooling**, serve per rendere condivisibili delle unità non condivisibili, l'esempio di spooling più comune sono le stampanti. In altre parole, è un buffer che mantiene output per un dispositivo che non può accettare flussi di dati distinti. Quando pensiamo a una spool, pensiamo a una coda di lavori da fare.
- I/O scheduling

Memoria secondaria:

Al giorno d'oggi, possiamo condividere le memorie secondarie (pensiamo a un NAS, Network Attached Storage). Nei sistemi UNIX-like, si fa mount remoto di queste unità secondarie (per esempio usando protocolli come NFS), ma dal punto di vista della macchina queste appaiono come normali memorie secondarie, ai quali si accede con modalità di filesystem.

Questo concetto evolve nel concetto di SAN (Storage Area Network). Un esempio di questi potrebbe essere la macchina virtuale che il prof usare per streammare la lezione.



Prendendo in considerazione sempre l'esempio della macchina virtuale usata dal prof, la macchina virtuale si trova dentro ai server (che sono ridondati), ma il disco contenente l'immagine del sistema usato da so, non collegato direttamente al server della macchina virtuale, ma usando una SAN questa si trova su una batteria di dischi (ovvero gli Storage Array) che sono accessibili a tutti i server.

Le apparecchiature che servono per la SAN possono essere dedicate con protocolli (anche fisici) specifici (tipo Hyper-Channel), ma possono essere anche comuni apparati di rete, quindi si può fare una SAN usando una comune gigabit ethernet per collegare fra loro gli storage arrays. Questa architettura con gli storage array permette di essere meno soggetto a guasti. Invece di avere un elemento che gestisce tutto, nella SAN ogni elemento svolge la propria funzionalità.

SSD:

La memoria secondaria, sia che sia sui sistemi, sia che sia in NAS, può essere fatta tramite dischi (detti anche dischi rotazionali) o tramite SSD (solid state disk).

I **dischi allo stato solido**, rispetto ai dischi, non hanno fragilità meccaniche (anche qualcosa come una scossa di terremoto può danneggiare dei dischi rotazionali, e anche la variazione di pressione atmosferica) e consumano meno energia dei dischi rotazionali.

Hanno un numero massimo di cicli di scrittura, quindi occorre centellinare (ovvero fare a piccole dosi) le operazioni di scrittura, e questo implicherà delle scelte sia per i device driver, ma anche per i file system (ci sono infatti configurazioni di FS più adatti agli ssd rispetto che altri).

In questi dischi, la velocità lettura è maggiore della velocità di scrittura, cosa che invece non accade nei dischi rotazionali/hard drive meccanici, in cui la velocità di lettura si avvicina molto a quella di scrittura. Si legge a blocchi (quindi, per esempio, 4kb alla volta), si scrive a "banchi" (molti blocchi insieme, siamo nell'ordine dei megabyte), questo deriva da un problema fisico di organizzazione di questi elementi, infatti il problema sta nel fatto che non si può fare una "vera" operazione di scrittura, bisognerà infatti prima cancellare e poi scrivere, e l'operazione di cancellazione è una operazione che prevede dei processi fisici, per questo l'intero banco viene cancellato e poi riscritto.

Un'altra caratteristica è l'accesso uniforme su tutto lo spazio di memoria, ovvero che la velocità di lettura, per esempio, non dipende dalla posizione del dato sul disco: se leggo il byte 0 o il byte 1000, la velocità rimane la stessa, e questa cosa invece non sarà vera per i dischi meccanici.

HDD (dischi rotazionali):

I **dischi rotazionali** sono composti da un insieme di piatti, suddivisi in tracce, le quali sono suddivise in settori. I dischi sono caratterizzati da tre parametri fondamentali: la velocità di rotazione, espressa in *rpm* (revolutions per minute), che indichiamo con r . Il tempo di seek, ovvero il tempo medio necessario affinché la testina si sposti sulla traccia desiderata, che indichiamo con T_s . La velocità di trasferimento, espressa in byte al secondo, che indichiamo con V_r .

Il **tempo di accesso**, ovvero il tempo necessario per leggere un settore del disco, dipende dal tempo di seek, ritardo rotazionale e tempo di trasferimento.

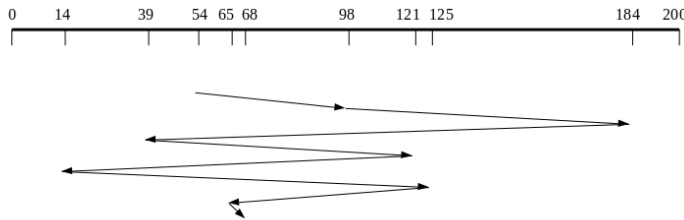
- Con **ritardo rotazionale** si intende il tempo medio necessario affinché il settore desiderato arrivi sotto la testina, ed è uguale a $1 / 2r$ (ovvero il tempo che ci mette per fare mezzo giro), dove r è in rpm. Le velocità rotazionali possono essere di 54000, 72000 e 10000 rpm [Non è possibile avere un tempo di accesso minore di 4 ms]

- Con **transfer time** si intende il tempo di trasferimento, ovvero se dobbiamo leggere dei dati dal disco dobbiamo poi trasferirli al sistema via bus al controller, dipende dalla quantità di dati b da leggere (supponendo che siano contigui sulla stessa traccia), ed è uguale a b / V_r .
- Il **tempo di seek** è un ritardo non banale siccome è meccanico, e consiste nel tempo che impiega la testina a saltare da un cilindro all'altro. Normalmente dura 8-10 ms.

Il gestore del disco può avere numerose richieste pendenti, da parte dei vari processi presenti nel sistema, e il sistema sarà più efficiente se le richieste pendenti verranno evase seguendo un ordine che minimizza il numero di operazioni che richiedono molto tempo (e.g. seek). Tra le politiche di organizzazione delle richieste abbiamo la **First Come, First Served** (FCFS), altrimenti detta FIFO, che è una politica di gestione fair in quanto non può mai generare starvation, ma non minimizza il numero di seek.

Facciamo un esempio:

- **Coda delle richieste:** 98, 184, 39, 121, 14, 125, 65, 68
- **Posizione iniziale:** 54

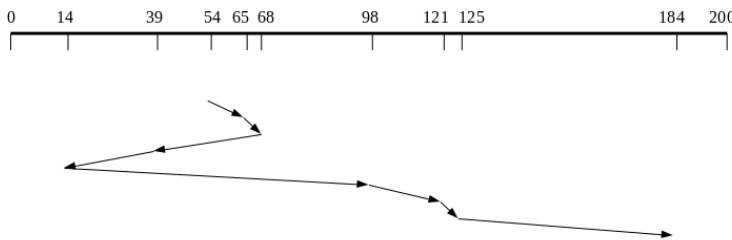


Questo effetto "zig zag", in cui spostiamo continuamente la testina avanti e indietro, deriva proprio dal fatto che non ottimizziamo con l'ordine con cui soddisfare le richieste, e perciò non minimizziamo il numero di seek. Se avessimo posto le richieste in posizione vicine in modo ordinato, la situazione sarebbe diversa.

Lunghezza di seek totale: 639
media: 79.88

Un'alternativa è usare, al posto di FCFS, l'algoritmo **Shortest Seek Time First** (SSTF), che seleziona la richieste che prevede il minor spostamento della testina dalla posizione corrente. Nel caso di equidistanza, la direzione viene scelta casualmente. La cosa brutta di questo algoritmo è che può provocare starvation, per questo motivo non è molto usato.

- **Coda delle richieste:** 98, 184, 39, 121, 14, 125, 65, 68
- **Posizione iniziale:** 54



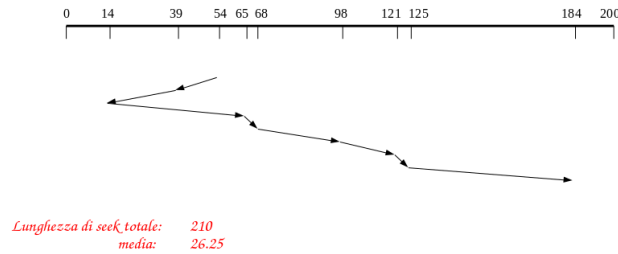
L'effetto zig zag è ridotto, ma ora abbiamo il problema della starvation, infatti potrebbe continuare a soddisfare le richieste che arrivano poste a distanza libera, senza gestire quelle di distanza più elevata.

Lunghezza di seek totale: 238
media: 29.75

L'algoritmo usato di più è l'**algoritmo dell'ascensore**, detto anche **LOOK**. In pratica, ad ogni istante, la testina è associata ad una direzione, e la testina si sposta di richiesta in richiesta, seguendo la direzione scelta. Quando si raggiunge l'ultima richiesta nella direzione scelta, la direzione viene invertita e si eseguono le richieste nella direzione opposta (si comporta essenzialmente come un ascensore).

Questo algoritmo è molto efficiente, è esente da starvation (anche se parzialmente).

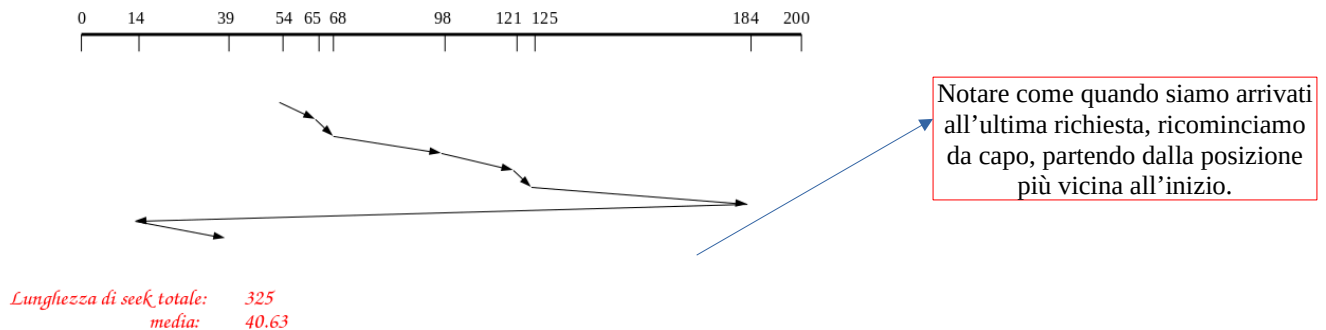
- **Coda delle richieste:** 98, 184, 39, 121, 14, 125, 65, 68
- **Posizione iniziale:** 54



Il problema più grande, però, è che il tempo medio di accesso non è omogeneo, siccome nei cilindri centrali sono privilegiati e le prestazioni sono peggiori nei cilindri periferici. Se abbiamo fortemente bisogno di avere un accesso omogeneo, al posto di LOOK si può usare il **C-LOOK**, che si basa sullo stesso principio, C-LOOK, ma la scansione del disco avviene in una sola direzione.

Quando si raggiunge l'ultima richiesta in una direzione, la testina si sposta direttamente alla prima richiesta. Questo algoritmo viene anche chiamato algoritmo della macchina da scrivere, perché essenzialmente arrivati in fondo, torniamo all'inizio della riga.

- **Coda delle richieste:** 98, 184, 39, 121, 14, 125, 65, 68
- **Posizione iniziale:** 54



[DISCLAIMER: quando diciamo che privilegiamo le aree centrali, indichiamo NON che privilegiamo il cerchio più vicino al centro, ma la “sezione” che si trova tra il cerchio più esterno e quello più interno del disco.]

Con LOOK e C-LOOK, è possibile che il braccio della testina non si muova per un periodo di tempo considerevole, ad esempio quando un certo numero di processi continua a leggere sullo stesso cilindro (era essenzialmente quel “parzialmente” che avevamo aggiunto parlando di starvation nel LOOK e C-LOOK). Se infatti stiamo lavorando al cilindro 10, e continuano ad arrivare richieste per il cilindro 10, allora abbiamo starvation. Perciò, la coda delle richieste può essere suddivisa in due sottocode separate. In LOOK quindi abbiamo la coda della direzione corrente e la coda della direzione successiva, C-LOOK invece ha scansione corrente e scansione successiva. Quando tutte le richieste della prima coda sono state esaurite, si scambiano le due code.

Se abbiamo richieste che sono nei cilindri della stessa direzione, ma non nel cilindro attuale, le inseriamo nella coda corrente, mentre se abbiamo delle richieste nel cilindro attuale o in un cilindro della direzione opposta, le inseriamo nella coda della direzione successiva.

RAID:

La tecnica RAID, che sta per **Redundant Array of Independent Disks** serve per aumentare la velocità d'accesso dei dischi. In pratica, col passare del tempo la velocità dei processori è andata crescendo secondo la legge di Moore, mentre la velocità dei dispositivi di memoria secondaria molto più lentamente, e questo era un grosso problema... Dunque, per aumentare la velocità di un componente, una delle possibilità è quella di utilizzare il parallelismo.

L'idea del RAID è quella di utilizzare un array di dischi indipendenti, che possano gestire più richieste di I/O in parallelo (tutto ciò però funziona se garantiamo che i dati letti in parallelo risiedano su dischi indipendenti).

Il RAID è diventato uno standard industriale per l'utilizzo di più dischi in parallelo, e consiste di 7 schemi diversi (0-6) che rappresentano diverse architetture di distribuzione dei dati.

Caratteristiche comuni ai sette schemi:

- un array di dischi visti dal s.o. come un singolo disco logico.
- i dati sono distribuiti fra i vari dischi dell'array
- la capacità ridondante dei dischi può essere utilizzata per memorizzare informazioni di parità, che garantiscono il recovery dei dati in caso di guasti. Questo è dovuto al fatto che l'utilizzo di più dischi aumenta le probabilità di guasto nel sistema, e quindi per compensare questa riduzione di affidabilità, RAID utilizza meccanismi di parità.

Per fare in modo che ci sia effettivamente un aumento della performance, il s.o. deve presentare al disco richieste che possano essere soddisfatte in modo efficiente. Ad esempio, richieste di lettura di grandi quantità di dati sequenziali e un gran numero di richieste indipendenti (una singola richiesta ci metterà lo stesso tempo di un sistema non parallelo, siccome viene comunque gestita da un singolo disco).

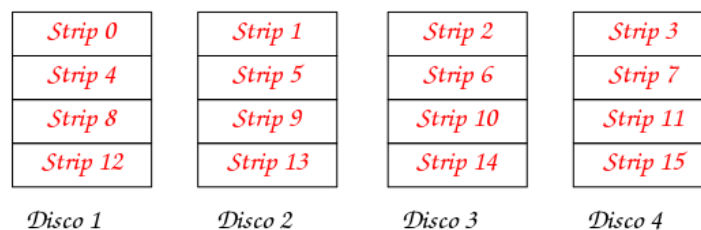
Queste richieste ovviamente generano un maggiore traffico, e quindi il data path che va dai dischi alla memoria (controller, bus, etc) sia progettato per gestire le maggiori performance.

RAID 0:

Questo tipo particolare di RAID non ha ridondanza [dovrebbe essere chiamato AID, lol]. Questo tipo di RAID fornisce la velocità, riducendo l'affidabilità. È anche il meno costoso, siccome non deve gestire meccanismi di recovery etc... È comunque molto utilizzato, in particolare per applicazioni in cui l'affidabilità non è un grosso problema, ma lo sono la velocità e il basso costo. I dati vengono comunque distribuiti su più dischi, solo non si ripetono.

Se due richieste di I/O riguardano blocchi indipendenti di dati, c'è la possibilità che i blocchi siano su dischi differenti, e quindi le due richieste possono essere servite in parallelo.

RAID 0 viene anche chiamato **striping**, siccome i dati nel disco logico vengono suddivisi in strip (e.g., settori, blocchi, oppure qualche altro multiplo). Strip consecutivi sono distribuiti su dischi diversi, aumentando le performance della lettura dei dati sequenziali.



RAID 0, per grandi trasferimenti di dati, è efficiente, in particolare se la quantità di dati richiesta è relativamente grande rispetto alla dimensione degli strip.

È efficiente anche per un gran numero di richieste indipendenti, in particolare se la quantità di dati richiesta è paragonabile alla dimensione degli strip.

Quanto più è omogenea la richiesta sui vari dischi, tanto più è efficiente.

RAID 1:

Il RAID di livello 1 viene anche detto **mirroring**, siccome si ottiene ridondanza duplicando tutti i dati su due insiemi indipendenti di dischi.

Come prima, il sistema è basato su striping, ma questa volta uno strip viene scritto su due dischi diversi, e in questo modo il costo per unità di memorizzazione raddoppia (vediamo essenzialmente uno spazio la cui grandezza reale sarebbe di 4 dischi come ampio 2 dischi).

Strip 0	Strip 1	Strip 0	Strip 1
Strip 2	Strip 3	Strip 2	Strip 3
Strip 4	Strip 5	Strip 4	Strip 5
Strip 6	Strip 7	Strip 6	Strip 7
Disco 1	Disco 2	Disco 3	Disco 4

Il vantaggio è che una richiesta di lettura può essere servita da uno qualsiasi dei dischi che ospitano il dato, e possiamo scegliere quello con tempo di seek minore.

Tuttavia, le cose cambiano se abbiamo una richiesta di scrittura, che invece dovrà essere servita da tutti i dischi che ospitano il dato (siccome essenzialmente dobbiamo fare più copie dello stesso dato), dunque questa velocità sarà dipendente dal disco con tempo di seek maggiore.

Inoltre, il recovery è molto semplice: se un disco si guasta, i dati sono accessibili dall'altro disco, e basterà sostituire il disco guasto e fare una copia del disco funzionante (perciò vengono detti Hot Pluggable, siccome posso togliere uno dei dischi e il sistema non si guasta).

[RAID 2,3,4 non sono molto utilizzati. Stessa cosa per 6 e 7.]

RAID 5 e RAID 6:

Per capire il RAID 5, pensiamo al RAID 1: in quel caso, siamo protetti solo dalla rottura di un 1 disco (nel caso ne avessimo 2), e perciò si possono avere metodi più efficienti del RAID 1, e perciò esiste il RAID 5.

Nel **RAID 5** essenzialmente i dischi memorizzano i dati come se avessimo $n-1$ dischi (quindi se abbiamo 5 dischi, ne memorizzano come se ne avessimo 4, ovvero abbiamo la capacità di 4 dischi), in questo modo si tollera la rottura di un disco. Per ogni strip corrispondente dei vari dischi, viene creato uno **strip di parità** in cui verrà memorizzato il risultato dell'operazione $Strip\ 0\ XOR\ Strip\ 1\ XOR\ Strip\ 2\ XOR\ Strip\ 3$ (da 0 a 3 se supponiamo di avere 5 dischi, quindi 4 dischi utilizzabili veramente), stessa cosa per gli strip da 4 a 7 etc... Ma invece di usare un disco singolo per la parità, questa viene distribuita ciclicamente fra i vari dischi. Al contrario del RAID 4, in cui abbiamo un singolo disco dedicato alla parità, le informazioni di parità vengono distribuiti fra i vari dischi, in questo modo l'uso è uniforme e gli accessi in lettura sono distribuiti fra i vari dischi.

Strip 0	Strip 1	Strip 2	Strip 3	P(0-3)
Strip 4	Strip 5	Strip 6	P(4-7)	Strip 7
Strip 8	Strip 9	P(8-11)	Strip 10	Strip 11
Strip 12	P(12-16)	Strip 13	Strip 14	Strip 15
Disco 1	Disco 2	Disco 3	Disco 4	Disco 5

Da notare come, appunto, la disposizione dei P sia ciclica.

Per gli accessi in scrittura, in teoria dovrei leggere, facendo sempre riferimento all'esempio qui sopra, Strip 0, Strip 1, Strip 2 e Strip 3 e poi cambiare P(0-3), ma così non è!

Infatti, sapendo che $s0 \wedge s1 \wedge s2 \wedge s3 = P(0-3)$ (dove \wedge è lo XOR) e che $X \wedge X = 0$, ho che, se cambio $s0$, $P'(0-3) = P(0-3) \wedge s0 \wedge s0'$, da cui $P'(0-3) = s0 \wedge s1 \wedge s2 \wedge s3 \wedge s0 \wedge s0'$, e siccome lo XOR è commutativo e associativo, ho che

$$P'(0-3) = s1 \wedge s2 \wedge s3 \wedge s0' \text{ se sto cambiando } s0.$$

[**DISCLAIMER**: la formula scritta qua sopra in realtà non è troppo utile, la cosa fondamentale da sapere è che $P'(0-3) = P(0-3) \wedge s0 \wedge s0'$]

Dunque, l'accesso in scrittura, non coinvolge 5 dischi o $n-1$ dischi, ma ne coinvolge sempre e solo 2.

RAID 6 è come RAID 5, ma si utilizzano due strip di parità invece di uno, e ciò aumenta l'affidabilità (è necessario il guasto di tre dischi affinché i dati non siano utilizzabili).

Spesso in giro si trovano scritture come RAID 10, o RAID 50. Queste si leggono come RAID 5-0 o RAID 1-0, e indicano un “misto” di due tecniche di RAID. Ad esempio, RAID 1-0 indica fare striping e mirroring, RAID 5-1 vorrebbe dire disporre più unità RAID 5 in mirroring fra loro.

File System e Memoria secondaria sono due strati diversi dell'architettura: il gestore della memoria secondaria ci fa vedere la memoria secondaria come una sequenza di blocchi, mentre lo strato di file system ci fa apparire sopra questo array di blocchi una “metafora”, usando archivi e file.

Lezione del (30/03/2021) – File System: Allocazione e Gestione spazio libero

File system:

Il **file system** è un'astrazione, che sostanzialmente permette di nascondere ancora di più (e in un modo pratico) quella che è la struttura fisica di memorizzazione. Se usiamo un supporto di memorizzazione, sia che sia una chiavetta USB, un cd-rom o un disco rigido, il livello della memoria secondaria lo fa apparire come una sequenza di blocchi. Usare una sequenza di blocchi da 4kB, però, non è esattamente comodo, soprattutto se abbiamo un hard disk da 4 TB. Per esempio, dobbiamo ricordarci che la lettera/dato che vogliamo sta, per esempio, al blocco 45. In più, se il dato è più grande di un blocco, dovremo ricordarci che la prima parte del blocco sta nella zona 45 e che la parte successiva sta, per esempio, al blocco 52. Dunque, si è pensato di paragonare questi blocchi ad un archivio d'ufficio, dove con files indichiamo le “pratiche” d'ufficio.

In altre parole, il compito del file system è quello di astrarre la complessità di utilizzo dei diversi media proponendo una interfaccia per i sistemi di memorizzazione.

L'entità atomica nel file system è il **file**, che è l'unità logica di memorizzazione. In questi file, vengono appunto inseriti qualsiasi tipo di dato, e il file system si preoccupa di rendere disponibile il file e mantenerlo, ma di ciò che si trova effettivamente dentro al file se ne occupa l'applicazione. Le **directory** servono per organizzare e fornire informazioni sui file che compongono un file system. Il file system, in Linux, non è un programma, bensì un servizio del kernel.

Un file contiene i seguenti *attributi*:

- **Nome**: stringa di caratteri che permette agli utenti ed al sistema operativo di identificare un particolare file nel file system. Alcuni sistemi differenziano fra caratteri maiusc./minusc., altri no.
- **Tipo**: necessario in alcuni sistemi per identificare il tipo di file (ovvero l'estensione).
- **Locazione e dimensione**: informazioni sul posizionamento del file in memoria secondaria.
- **Data e ora**: informazioni relative al tempo di creazione ed ultima modifica del file.
- **Informazioni sulla proprietà**: utenti, gruppi, etc., utilizzato per accounting (ovvero per tenere traccia di quanti archivi sta usando un determinato utente e magari evitare che un utente si impossessi di tutte le risorse del sistema) e autorizzazione (es. chi è il proprietario, chi è il gruppo di proprietà...).
- **Attributi di protezione**: informazioni di accesso per verificare chi è autorizzato a eseguire operazioni sui file.
- **Altri attributi**: flag (sistema, archivio, hidden, etc.), informazioni di locking (per esempio, per prevenire problemi di accesso concorrente ai file, che possono perdere informazioni se vengono aggiornati contemporaneamente da più utenti), etc...

Possiamo classificare i file a seconda della struttura interna:

- senza formato (stringa di byte): file testo
- con formato: file di record, file di database, a.out,...

Oppure a seconda del contenuto:

- ASCII/binario (visualizzabile o no, 7/8 bit) (oggi, per permettere la codifica anche di caratteri di altre lingue straniere, si usano anche l'UTF-8 e l'UTF-16. Un file ASCII è anche UTF-8).
- sorgente, oggetto,
- eseguibile (oggetto attivo)

[DA QUESTO PUNTO IN POI, per un po', le cose che dice davoli sono le stesse delle slide, quindi gli appunti sembrano un po' messi peggio, but dw sono comunque comprensibili e contengono degli insights interessanti.]

Per vedere l'hexdump di un file, possiamo usare il comando `bvi`.

Alcuni S.O. supportano e riconoscono diversi tipi di file, e conoscendo il tipo del file, il s.o. può evitare alcuni errori comuni, quali ad esempio stampare un file eseguibile.

Esistono tre tecniche principali per identificare il tipo di un file

- **meccanismo delle estensioni**, se un file ha estensione .pdf sospettiamo che sia un .pdf, ma volendo possiamo creare dei file con estensione .pdf che però non sono pdf, e quindi se un programma li legge si incasina.
- utilizzo di un **attributo "tipo"** associato al file nella directory. Le directory elencheranno i file con delle informazioni di corredo, e fra queste informazioni di corredo ci può essere il tipo.
- **magic number**, abbiamo nel contenuto del file dei particolari valori che vengono usati come "indizi" per stabilire il tipo del file.

Queste 3 tecniche si differenziano per dove si trova l'informazione del tipo.

Vediamo ora alcune implementazioni negli O.S. più famosi.

In MS-DOS e free-DOS, il nome del file poteva essere di massimo 11 caratteri (nome (8) + estensione (3)). Il tipo era contenuto nei suffissi: tra le estensioni, abbiamo .COM (libreria, e file load and stay resident), .EXE (per gli eseguibili), .BAT (per gli script).

In Windows 9x / NT / 7/8/9/10, le cose sono leggermente cambiate: abbiamo nomi/estensioni di lunghezza variabile, e il riconoscimento del tipo avviene anche qui attraverso le estensioni .COM, .EXE, .BAT, che formano una associazione estensione / programma.

In Mac OS, abbiamo come attributo il programma creatore del file. MacOS inoltre fu il primo ad associare automaticamente il programma dato il tipo di file (prima bisognava lanciare il programma e aprire il file).

Infine, Unix/Linux usa un mix di magic number + estensione + euristica (UNIX). Per esempio, quando lanciamo il comando `file`, a volte si ha una scansione con un risultato "certo", altre volte invece abbiamo un risultato euristico.

ELF (ovvero Executable and Linkable File) è un tipo di file eseguibile, e rappresenta lo standard di file binario dei sistemi UNIX con architettura x86.

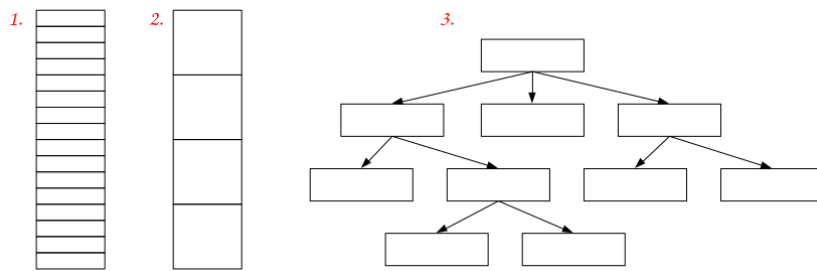
In un file system nei sistemi UNIX, sono presenti:

- file regolari, che sono sequenze di byte
- directory, che sono file di sistema per mantenere la struttura del file system
- file speciali a blocchi, utilizzati per modellare dispositivi di I/O come i dischi (*)
- file speciali a caratteri, utilizzati per modellare device di I/O seriali come terminali, stampanti e reti (*)
- altri file speciali, come ad es., pipe (*)

Le PIPE, i file speciali a caratteri e i file speciali a blocchi NON sono veri file, infatti l'unico contenuto che esiste di questi file è la entry nella directory. Come dicevamo nel primo semestre, infatti, una scelta chiave di UNIX è che il file system, oltre che essere appunto un file system, è anche lo strumento principale di naming di ogni cosa all'interno del sistema.

I file possono essere strutturati in molti modi:

1. sequenze di byte, è la gestione più semplice
2. sequenze di record logici
3. file indicizzati (struttura ad albero), è usato per esempio nella gestione dei database



I sistemi operativi possono attuare diverse scelte nella gestione della struttura dei file:

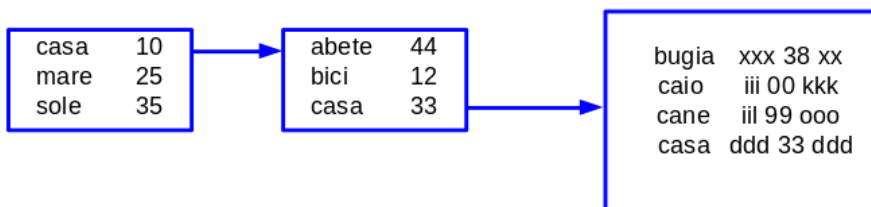
- **scelta minimale**, in cui i file sono considerati semplici stringhe di byte, a parte i file eseguibili il cui formato è dettato dal s.o. (e.g., UNIX e MS-DOS).
- **parte strutturata/parte a scelta dell'utente**, e.g. Macintosh (resource fork / data fork), ovvero i file hanno due parti: la parte delle risorse e la parte dei dati. La parte dei dati è una sequenza dei byte, mentre la parte delle risorse contiene delle informazioni con un formato predefinito (es. stringhe), questo è stato permesso dal Macintosh per creare traduzioni dei programmi senza averne i sorgenti.
- **diversi tipi di file predefiniti**, e.g., VMS, MVS.

La struttura interna dei file viene lasciata ai programmi applicativi, che stanno fuori dal kernel in modo da rendere il kernel snello.

Se un kernel supporta più formati di filesystem, il codice di sistema diventa più ingombrante e ci potrebbero essere delle incompatibilità di programmi (accesso a file di formato differente), tuttavia la gestione dei file diventa più efficiente e non duplicata per i formati speciali. Se invece un kernel supporta meno formati, abbiamo il vantaggio di avere un codice di sistema più snello.

I metodi d'accesso di memoria si differenziano in **sequenziale**, ovvero accediamo "camminando" nel file, scandendo il file sequenzialmente. Oppure ci si può accedere ad **accesso diretto**, ovvero accediamo al file partendo da una posizione precisa che specifichiamo (es. accediamo partendo da un certo indice del record). Infine, abbiamo l'**accesso indicizzato** (usato molto nei database), che si ha quando la lettura e la scrittura avvengono tramite una chiave associata all'elemento, e questa chiave sarà l'identificativo del record all'interno del file.

Nell'accesso indicizzato abbiamo una tabella, salvata in memoria primaria o su disco, che contiene delle associazioni di corrispondenza chiave-posizione.



In pratica, ordiniamo in ordine alfabetico le entry, e ciascuna di queste entry indicherà che fino a "abete" sta il blocco 44, fino a "casa" sta il blocco 33 etc...

Se prendessimo per esempio il blocco arnia, siccome è più "grande" di abete e più piccolo di "bici", sarà sicuramente nel blocco 12.

In poche parole, il nome ci dice a che blocco finisce l'area.

Possiamo vedere questo come un btree visto ad algorithmi.

[Piccola curiosità] Posso accedere direttamente a un file senza passare da un file system, ma in questo modo se modifico qualcosa mentre il kernel sta eseguendo una operazione di lettura/scrittura nella stessa area, potrei potenzialmente rovinare tutto.

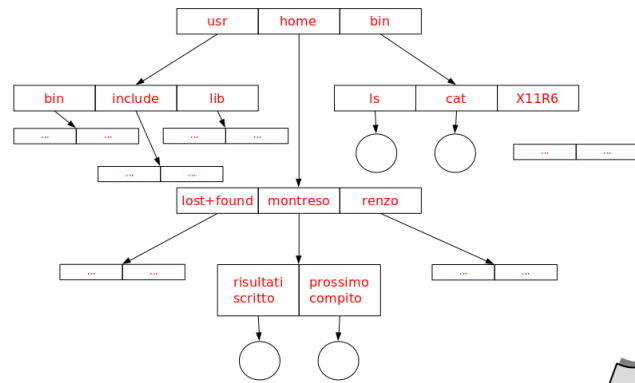
Come abbiamo notato in passato, l'interfaccia per la programmazione relativa alle operazioni su file è basata sulle operazioni open/close, infatti i file devono essere "aperti" prima di effettuare operazioni e "chiusi" al termine. Questo è dovuto al fatto che l'astrazione relativa all'apertura/chiusura dei file è utile per mantenere le strutture dati di accesso al file, controllare le modalità di accesso e gestire gli accessi concorrenti e definire un descrittore per le operazioni di accesso ai dati (ovvero trovare un file descriptor). Se ogni volta che accedo a un file dovessi recuperare le strutture necessarie al suo accesso, allora il sistema sarebbe davvero inefficiente.

Infatti, volendo, potrei comunque accedere a dei file senza aprirli, ma ciò implicherebbe che io ogni volta che ci accedo devo cercare il file, controllare i permessi e poi ripristinare il file in modo che non interferisca con gli altri blocchi, e ciò non è esattamente efficiente. Invece, se io apro un file, è come se mantenessi una cache di tutte le informazioni che mi servono per fare le operazioni sul file.

Rimandando alla metafora dell'ufficio: se io accedessi un file senza aprirlo o chiuderlo, sarebbe come prendere una procedura da un cassetto, leggere un paragrafo, e poi rimetterla dentro al cassetto per poi riprenderla ogni volta che devo leggere un paragrafo.

[Infatti, noi accediamo ad un file in Linux con le syscall open e read, dove open prende il file dalla sua locazione nel fs, mentre read prende il file descriptor del file aperto.]

L'organizzazione dei file system è basata sul concetto di **directory**, che fornisce un'astrazione per un'insieme di file. In molti sistemi, le directory sono file (speciali).



E' anche possibile considerare grafi diversi dagli alberi, nel caso in cui un file possa essere contenuto in due o più directory (è il caso dei link fisici su Linux per esempio). In questo modo, ogni modifica al file è visibile in entrambe le directory. La struttura risultante diventa quindi un grafo diretto aciclico (DAG).

Semantica della coerenza

In un sistema operativo multitasking, i processi possono accedere ai file indipendentemente. Ma la vera domanda è: quando e in che modo vengono viste le modifiche ai file da parte dei vari processi? Uno può pensare che ciò che avvenga sia un caso di semantica immediata, in cui se un file viene modificato da un processo, immediatamente tutti questi altri processi possono osservare questa modifica.

In UNIX la situazione è esattamente questa, ovvero le modifiche al contenuto di un file aperto vengono rese visibili agli altri processi immediatamente.

Tuttavia, non è sempre così. Abbiamo per esempio dei file system (abbastanza specifici) in cui vige la **semantica delle sessioni** (tra cui AFS). L'**AFS** (che sta per Andrew's File System) è un filesystem che consente di avere directory diverse in server diversi che sono geograficamente distribuiti (per esempio, si passa da una directory memorizzata in Italia ad una memorizzata in California). **In questo tipo di file system, qualcosa come una semantica immediata non sarebbe viabile**, siccome per l'accesso sarebbe dovuto essere svolto in mutua esclusione, e per capire il turno di ciascun processo in ciascun server ci sarebbe voluto molto tempo (3 pacchetti RTT). Nella **semantica delle sessioni**, invece, ogni processo accede alla fotografia del file che si aveva al momento dell'apertura, e lo può modificare, ma gli altri processi vedranno queste modifiche fatte dal processo iniziale solo al momento della chiusura.

Implementazione del file-system:

Nella implementazione di un file system, dovremo tenere in considerazione una serie di fattori, tra cui:

- organizzazione di un disco
- allocazione dello spazio in blocchi (come organizzare l'astrazione di file dati i blocchi)
- gestione spazio libero (i blocchi liberi dovranno essere posti in una struttura dati per poi essere riutilizzati quando si crea un altro file)
- implementazione delle directory
- tecniche per ottimizzare le prestazioni
- tecniche per garantire e ripristinare la coerenza (dobbiamo trovare il modo di non perdere i dati).

Organizzazione di un disco

Un disco (o in generale una unità di memoria) può essere diviso in una o più **partizioni**, porzioni indipendenti del disco che possono ospitare file system distinti.

Per convenzione, il primo settore dei dischi è il cosiddetto **master boot record (MBR)**, ed è utilizzato per fare il boot del sistema. Questo contiene le informazioni della struttura delle partizioni attraverso la **partition table** (tabella delle partizioni) e contiene l'indicazione della partizione attiva.

Al boot, il **MBR** viene letto ed eseguito dal bootloader. Il **GPT** è un tipo moderno di MBR.

Il partizionamento può causare frammentazioni interne. Noi usiamo la tecnica del partizionamento sia per ospitare più tipi di file system diversi, che per ospitare anche dati/parti senza file system : è il caso ad esempio della swap partition (o **partizione di paging** come nome più appropriato) che appunto non possiede un file system, e viene gestito dal pager. Il partizionamento è anche usato per ospitare sistemi operativi diversi sulla stessa macchina. Il partizionamento può essere usato anche nei server, dove i log e le informazioni variabili possono aumentare smisuratamente riempiendo il file system (anche in caso di un attacco), e quindi essenzialmente viene dedicata una partizione per questo tipo di dati, in modo che se questa viene intasata non ci siano complicazioni all'interno del server.

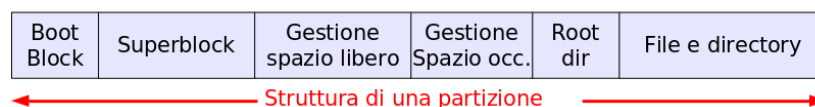
Struttura di una partizione

Ogni partizione inizia con un **boot block**, e dunque il MBR carica il boot block della partizione attiva e lo esegue. Dopodiché, il boot block carica il sistema operativo e lo esegue. [NOTA: il MBR, o master boot record, contiene le informazioni sulle varie partizioni e permette di distinguerle. Il boot block all'interno delle].

L'organizzazione del resto della partizione dipende dal file system.

Normalmente, in ogni partizioni abbiamo anche:

- un **superblock**, che contiene informazioni sul tipo di file system e sui parametri fondamentali della sua organizzazione, quindi è l'indice dell'intero file system. (Il prof nel ripasso della lezione dopo ha specificato che viene quasi sempre duplicato).
- **tabelle per la gestione dello spazio libero**: struttura dati contenente informazioni sui blocchi liberi, in questo modo si può sapere facilmente dove aggiungere blocchi di un file.
- **tabelle per la gestione dello spazio occupato**: contiene informazioni sui file presenti nel sistema (non presente in tutti i file system), e consentono di ritrovare le parti che compongono un file.
- **root dir**, ovvero la directory radice del file system.
- file e directory



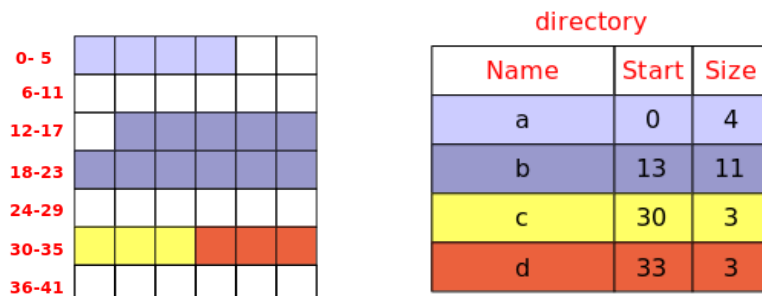
Non è detto che le componenti di una partizione siano fatte in questo ordine o in questo modo preciso (abbiamo per esempio file system che permettono di usare la stessa tabella sia per la gestione dello spazio libero che per la gestione dello spazio occupato, altri in cui il superblock è critico e viene copiato in più punti diversi della partizione.)

Allocazione

Consiste nel problema di creare l'idea di file, ovvero *come vengono scelti i blocchi dati da utilizzare per un file?* e *come vengono collegati assieme questi blocchi dati per formare una struttura unica?*

Un modo per gestire l'allocazione sta nella **allocazione contigua**, in cui i file vengono memorizzati in sequenze continue di blocchi di dischi. In questo modo, non è necessario usare strutture dati per collegare i vari blocchi, e l'accesso sequenziale è efficiente, siccome non necessitano operazioni di seek, ma soprattutto l'accesso diretto è efficiente, in quando per accedere a un dato byte di un file posso andarlo a cercare nel blocco corrispondente usando la formula $pos = offset \% blocksize$, dove l'offset è il numero del byte che cerchiamo.

Ecco un esempio:



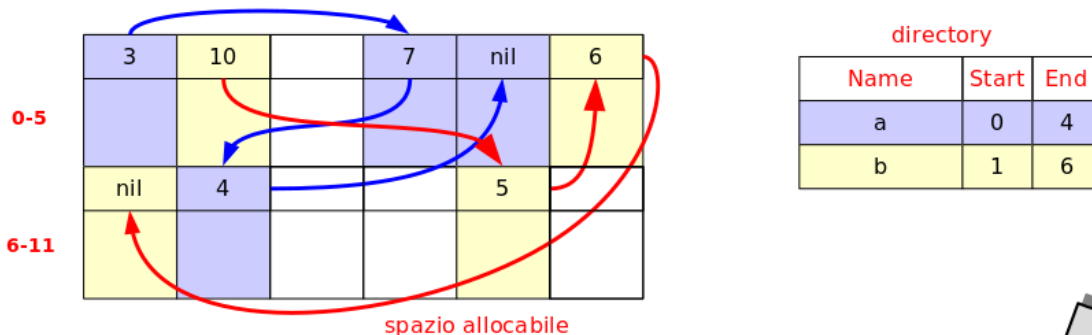
Il problema di questo approccio è che se abbiamo per esempio il file c dell'esempio, che occupa i blocchi 30, 31, 32, e lo vogliamo ampliare, dobbiamo copiarlo completamente in un'altra zona del disco, perché sennò andremo a scrivere i contenuti del file d.

Viene però tutt'ora usato nei file system dei DVD e CDROM, che necessitano file system **WORM** (Write Once, Read Many), in cui i dati vengono scritti una sola volta.

Un altro approccio è l'uso di un'**allocazione concatenata**.

Con questo metodo, ogni file è costituito da una lista concatenata di blocchi, ed ogni blocco contiene un puntatore al blocco successivo.

Il descrittore del file contiene i puntatori al primo e all'ultimo elemento della lista (anche se in realtà è possibile anche unicamente usare un puntatore al primo blocco, siccome dal primo si riescono a raggiungere tutti i blocchi collegati. Tuttavia, avere un riferimento all'ultimo elemento del file è sempre utile quando dobbiamo espanderlo).



In questo modo si risolve il problema dell'allocazione contigua, siccome possiamo spaziare lungo tutta la memoria, e basta trovare così un solo blocco libero.

Rimane però il problema che l'accesso diretto è inefficiente. Infatti se abbiamo blocchi da un 1kB e vogliamo cercare quello che contiene il byte 15223, dobbiamo scandire tutti i blocchi finché non troviamo il byte che vogliamo. Ad ogni accesso ad un blocco si avrà un accesso al disco, e tutti gli accessi al disco costano. In più, la località ora è assente, quindi se abbiamo un HDD a dischi, sarà necessario fare una seek() se i blocchi si trovano in sezioni molto lontane (in poche parole, si possono spargere i blocchi nella memoria).

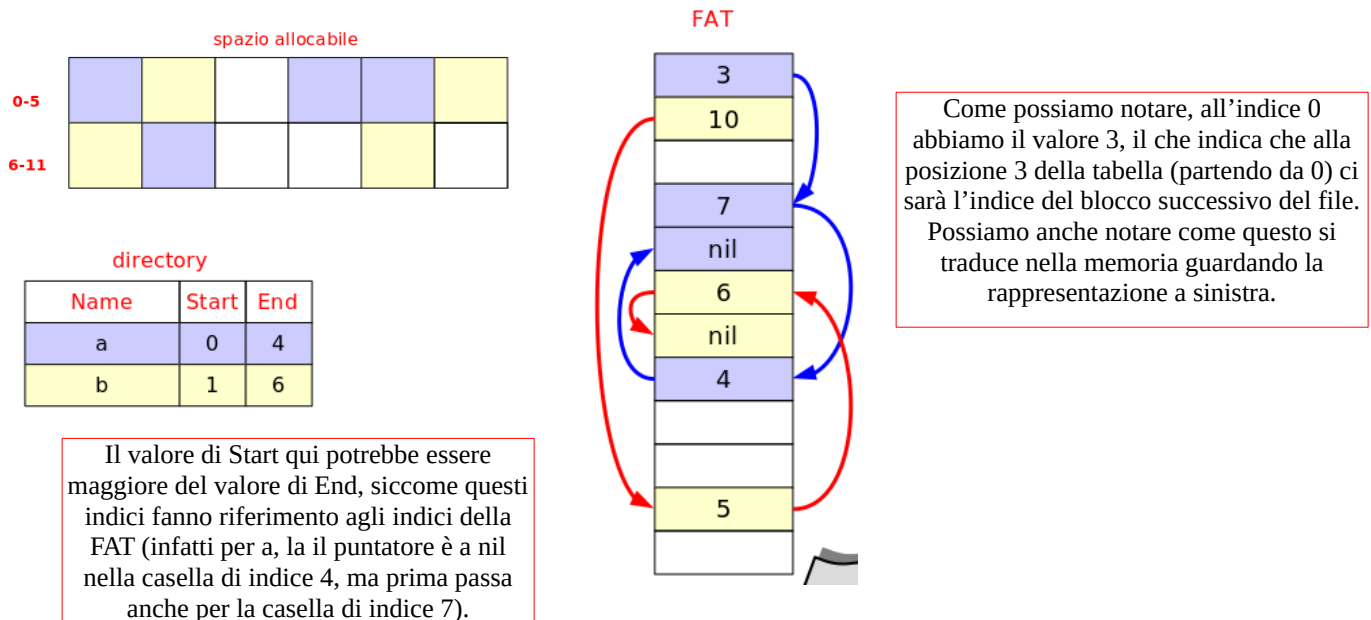
Abbiamo anche un altro problema più sottile: se noi abbiamo blocchi da 1kB, e usiamo 4 o 8 byte come puntatore al prossimo blocco, rimaniamo con un'area dati da 1020 o 1016 byte, che non è più una potenza di due! Questo ci dà molto fastidio, infatti come abbiamo visto con la gestione della memoria quando si divideva la parte del numero di pagine dall'offset [qui], se noi abbiamo o dividiamo i contenuti in unità/blocchi che hanno ampiezza che è una potenza di 2, troviamo facilmente l'indice del

blocco e l'offset all'interno del blocco facendo semplicemente una operazione di mascheramento e shift, altrimenti dobbiamo fare delle divisioni, che sono molto più costose per la CPU!

Per ridurre l'overhead dovuto ai puntatori, possiamo riunire i blocchi in cluster da 4, 8 o 16 blocchi, che vengono allocati in modo indivisibile. In questo modo, la percentuale di spazio di usata per i puntatori diminuisce (si allocano aree più grandi), ma si ha più frammentazione interna.

Allocazione mediante FAT

Questo è il più usato. In pratica, invece di mettere il puntatore al prossimo blocco all'interno del blocco attuale, si crea una tabella unica con un elemento per blocco (o cluster di blocchi). Questa tabella viene chiamata **File Allocation Table** (da qui il nome FAT). In ciascun elemento della tabella, per ciascun blocco viene definito un blocco successivo. QUESTA TABELLA DUNQUE CONTIENE SOLAMENTE INDICI (può essere benissimo implementata come un array di interi).



Grazie a questo sistema, i blocchi sono interamente dedicati ai dati, ma la scansione richiede anche la lettura della FAT, aumentando così il numero di accessi al disco.

Si può anche fare caching in memoria dei blocchi FAT, e in questo modo l'accesso diventa un po' più efficiente.

[**DISCLAIMER**: il prof non ha spiegato questa parte, ma l'ha chiesta in una domanda.]

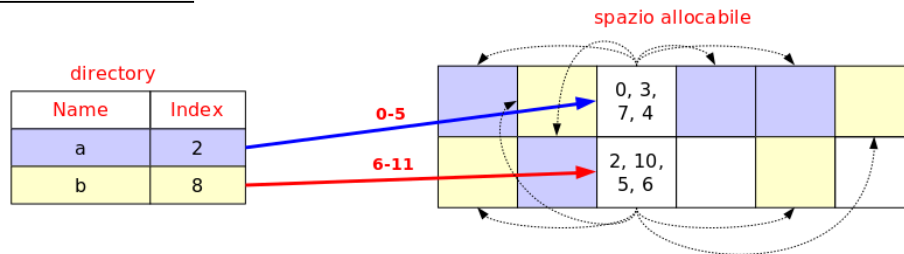
La lunghezza massima di un file che può essere memorizzato su un file system di tipo FAT viene calcolata moltiplicando il massimo numero di cluster (dimensione dell'unità di allocazione) identificabili per la loro dimensione. Il massimo numero di cluster identificabili dipende da quanti bit sono allocati per numerarli/indicizzarli, e per questa distinzione esistono varie versioni di FAT (FAT12, FAT16, FAT32).

- FAT32 memorizza la dimensione dei file in un intero unsigned da 32bit, e non può quindi gestire un file di dimensione superiore ai 2^{32} bit = ~4GB.
- FAT16, allocando 16bit per identificare i cluster, può contarne al massimo 2^{16} (65536). Prendendo quindi, ad esempio, un file system FAT16 con cluster da 32KB, possiamo avere file grandi al massimo $32KB * 65536 = \sim 2GB$.
- FAT12 di conseguenza può numerare al massimo 2^{12} (4096) cluster, ma dato che il numero dei settori del disco viene memorizzato in un intero signed a 16bit, la massima dimensione del disco è $2^{15} = 32MB$. Ovviamente, non si può memorizzare un file di grandezza superiore alla

dimensione dell'unità di memorizzazione stessa, e quindi anche i file in FAT12 possono avere dimensione massima di 32MB.

Allocazione indicizzata

In questo tipo d'allocazione, l'elenco dei blocchi che compongono un file viene memorizzato in un blocco (o area) indice. Quindi, per accedere ad un file, si carica in memoria la sua area indice e si utilizzano i puntatori contenuti.



I vantaggi di questo approccio sono che le aree dati vengono usati solo come aree dati, e che l'accesso sequenziale, l'accesso diretto, l'accesso in extend mode (o append mode) è rapido.

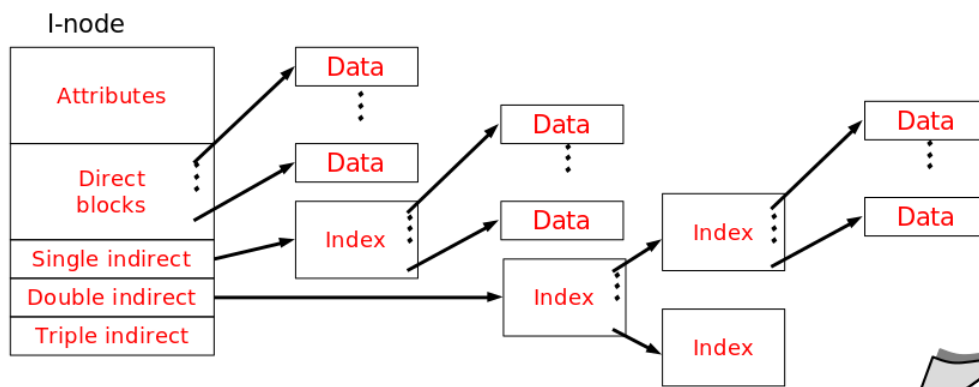
Tuttavia, vi è uno svantaggio molto grande: il blocco dati che contiene gli indici avrà una sua dimensione, quindi non potrà contenere più di un certo numero di indici (es. se ho un blocco da 1KB e metto degli indici da 4 byte, potrò memorizzare solo fino a 256 KB per file).

Il modo migliore per risolvere questo problema è l'**indice multilivello**. Quando un blocco indice non basta più, si fa un blocco indice del blocco indice. In questo modo le prestazioni si riducono leggermente, però non c'è più il problema di ampiezza massima del file.

In UNIX, tutti i file system (che siano ext4, HPFS, ext2 ...) sono tutti ad allocazione indicizzata, e nella struttura dati che si chiama **inode** che tiene gli attributi di un file [l'abbiamo visto qui] ci sono anche i puntatori ai reali dati. Alcuni di questi puntatori sono diretti, che puntano direttamente alle aree dati, se non bastano c'è un puntatore doppio indiretto e un puntatore triplo indiretto.

Il motivo di questa struttura è dovuto al fatto che, dopo alcune analisi statistiche, si è stabilito che i file di piccole dimensioni sono solitamente molto di più dei file di grandi dimensioni.

I puntatori diretti si usano solamente per file di piccole dimensioni, mentre quelli indiretti per file di dimensioni molto grandi. In questo modo, c'è una ampiezza massima per un file, ma è davvero molto grande (e con al massimo 4 accessi si accede in qualsiasi punto del file, anche di file enormi [gli accessi sono 4 siccome accedo prima all'inode, poi al primo index, poi al secondo, e infine al dato]).



Si possono anche aggiungere ulteriori miglioramenti, tra cui:

- pre-caricamento (per esempio nell'allocazione concatenata fornisce buone prestazioni per l'accesso sequenziale).
- combinazione dell'allocazione contigua e indicizzata:
 - contigua per piccoli file ove possibile
 - indicizzata per grandi file
 - Indicizzata di chunk (sottosequenze) contigue (è il caso di **ext4**).

Gestione dello spazio libero

Un modo per gestire lo spazio libero può essere la creazione di **una mappa di bit**, all'interno della quale ogni bit corrisponde ad un blocco, e i blocchi liberi sono indicati col valore 0, mentre quelli occupati col valore 1.

Tuttavia, se usiamo dischi molto grandi, allora la memorizzazione della bitmap potrebbe richiedere molto spazio (pensiamo ad una memoria da 1 TB, avremo una bitmap da 128 MB se usiamo blocchi da 1KB).

Se usiamo un **filesystem di tipo FAT**, allora la gestione delle aree libere può essere fatta direttamente sulla FAT stessa: **possiamo concatenare gli elementi della FAT non usati come una lista libera**. Se però togliamo e inseriamo elementi in una memoria FAT, potrebbe accadere un evento che viene chiamato (in modo improprio) frammentazione (per questo a volte dobbiamo usare un defragger).

File system come ext4, ext2 non hanno bisogno di defrag, mentre FAT e NTFS sì.

Lezione del 9/04/2021 – FS: Fsck e log/journaling, introduzione breve alla sicurezza

[**Digressione interessante**] possiamo mettere in un dato disco un qualsiasi tipo di file system, siccome viene visto come un array di blocchi a livello macchina, ed è poi attraverso la formattazione che creiamo un nuovo FS. A volte però usare un dato file system su alcune periferiche potrebbe risultare uno spreco (es. se usiamo un FS dinamico come FAT su un CDROM).

[Il prof ha praticamente saltato questo]

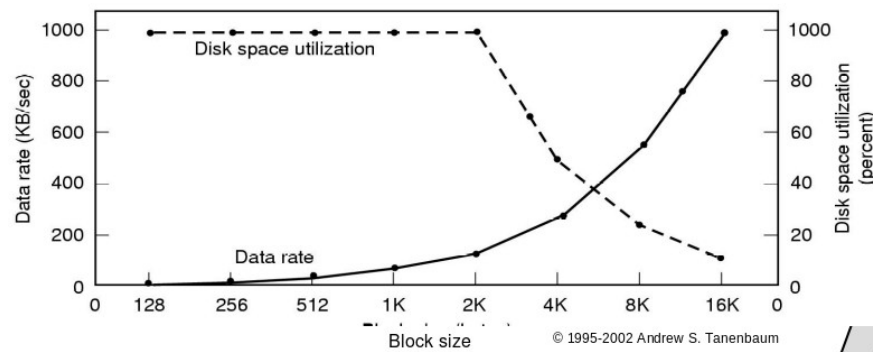
Gestione dello spazio libero con lista concatenata di blocchi

Un modo per gestire lo spazio libero può essere usando una lista concatenata di blocchi che contengono dei puntatori a dei blocchi liberi. In questo modo è sufficiente mantenere in memoria un blocco che contiene gli elementi liberi, e non è necessaria una struttura dati a parte, siccome i blocchi contenenti elenchi di blocchi liberi possono essere mantenuti all'interno dei blocchi liberi stessi.

Gli svantaggi sono che l'allocazione di un'area di ampie dimensioni diventa molto costosa, e l'allocazione di aree libere contigue è molto costosa.

Come scegliere la dimensione di un cluster?

Se si aumenta la dimensione di un cluster, allora abbiamo una velocità di lettura alta, a scapito però del verificarsi di frammentazione interna. Se invece scegliamo di avere dei cluster piccoli, avremo una



minore frammentazione, ma la velocità di lettura sarà più bassa.

Directory

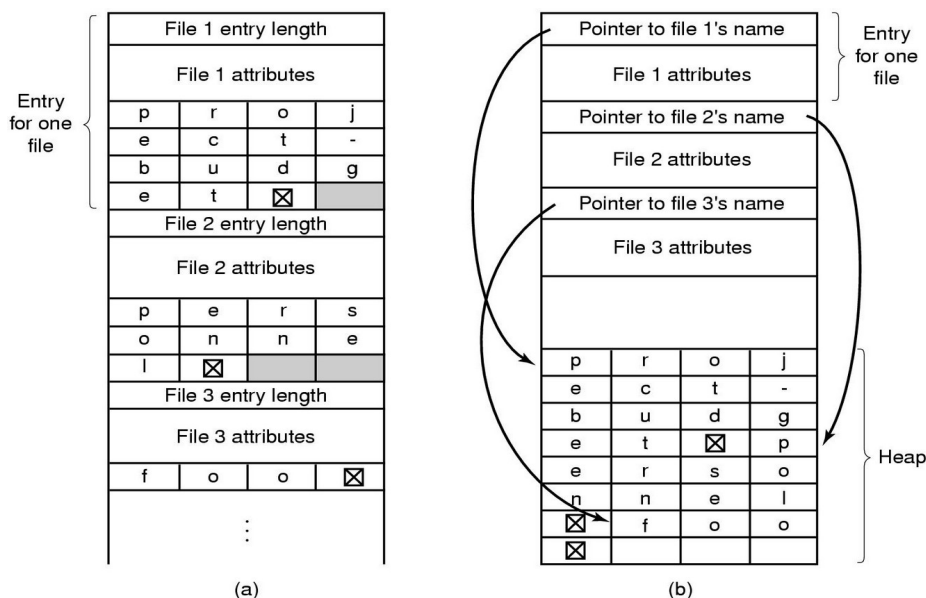
La **directory** è un file speciale contenente le informazioni della “directory”, ha un formato anch’esso determinato dallo standard del file system. Ogni elemento della directory è detto “**directory entry**”, e deve permettere di accedere a tutte le informazioni necessarie per gestire il file (tra cui nomi, attributi, informazioni di allocazione...). Notare che appunto la directory non deve necessariamente contenere queste informazioni, ma appunto deve permettere di accedere a tali informazioni. In questo modo possiamo decidere appunto se contenere le informazioni dei file all’interno della directory, oppure possiamo associare a ciascun nome un puntatore che consenta di recuperare queste informazioni altrove.

Alcuni esempi di implementazioni delle directory sono:

- In MS-DOS, la directory entry contiene tutte le informazioni necessarie associate ad un file.
- In UNIX, le informazioni sono contenute negli inode; e quindi una directory entry contiene un indice di inode. (quindi file system ext2, ext3, ext4, HPFS).

Tale implementazione in UNIX è dovuta al fatto che in UNIX possiamo fare i link fisici, e dunque se abbiamo due nomi che hanno lo stesso inode (anche in due cartelle diverse) punteranno entrambi allo stesso file (o ancora meglio, **saranno due nomi per lo stesso file**).

Un file system determina anche la lunghezza massima dei nomi, che può essere fissa (es. la prmissa FAT di MS DOS, in cui il massimo era 11 caratteri), oppure di lunghezza variabile. Quest’ultima ovviamente sarà più difficile da implementare.



(a) abbiamo un metodo per memorizzare i nomi, in cui essenzialmente il nome è memorizzato localmente alla entry. Nel nome possiamo notare che ci sono delle caselle grigie, queste sono dovute all’allineamento alla parola. Se non ci fosse l’allineamento alla parola, la memoria sarebbe molto incasinata.

(b) in questo altro metodo, il nome del file viene posto in una zona posta alla fine degli attributi, e viene puntato da una delle entry.

Possiamo implementare le directory come una lista lineare, che comunque anche se sarà semplice da implementare, sarà inefficiente nel caso di directory di grandi dimensioni, oppure attraverso una tabella hash, che però necessita di predeterminare le dimensioni della tabella hash e può risultare inefficiente in caso di collisioni.

Link simbolici e hard link (o link fisici)

Quando creiamo un **link simbolico**, viene creato un tipo speciale di directory entry, che contiene un riferimento (sottoforma di cammino assoluto) al file a cui esso punta.

Quando si apre un link simbolico, dopo averlo cercato nella directory, quando si scopre che si tratta di un link simbolico, il link viene risolto, ovvero si usa il cammino assoluto contenuto nel link per recuperare il file.

Quando invece creiamo un **hard link** in una directory diversa rispetto al file a cui punta, le informazioni relative al file sono presenti, uguali, in entrambe le directory.

Dunque, non è necessaria una doppia ricerca nel file system e sarà impossibile distinguere la copia dall'originale.

L'hard link permette di struttura il filesystem come un grafo aciclico, tuttavia ciò crea una serie di problemi nuovi: infatti non tutti i file system sono basati su DAG; esempio MS-DOS non li supporta.

Implementazione di un hard link

Per implementare un hard link, è necessario utilizzare la tecnica degli inode. In FS come quello di UNIX, ogni file ha un record, detto **inode**, che contiene tutte le informazioni su tale file. Se noi creiamo un hard link ad un file, usando il comando `ls -i` possiamo notare che l'hard link creato contiene lo stesso numero di inode del file che abbiamo linkato. Nell'inode tra l'altro c'è un contatore del numero dei riferimenti. Creando un hardlink, se noi eliminiamo il file di cui abbiamo creato l'hardlink, il file sarà comunque presente nel sistema, infatti il file esisterà finché il suo contatore del numero di riferimenti non sarà a 0. [abbiamo visto ste cose anche nel primo semestre qui].

Tecniche di miglioramento

Per migliorare le performance, possiamo avere delle ottimizzazioni sia a livello di gestione della memoria centrale (quindi all'interno del kernel), che a livello di un controller del device. Tra le varie cose che possiamo fare per migliorare la performance, possiamo:

- In memoria centrale
 - mantenere una cache dei blocchi usati recentemente
 - avere una tabella dei FD aperti
 - avere un buffer di blocco per i device basati su DMA
- Nei controller, invece:
 - possiamo implementare un buffer di traccia (si intende traccia nel HDD), ovvero ogni volta che si fa una seek(), può mantenere un buffer delle tracce precedenti che funziona quasi come una cache.

Il problema più importante dobbiamo risolvere, però, è il **problema della coerenza**: se una cache non viene aggiornata, ci potrebbero essere delle incoerenze (inconsistent in inglese si traduce con incoerente in italiano) nel filesystem. Pensiamo per esempio ad uno spegnimento forzato, magari a causa di una interruzione di corrente: se un file mentre è caduta la corrente, si stavano facendo delle operazioni in RAM (che è volatile) e si creavano strutture dati che poi in secondo momento andavano trasferite sul disco principale. Se ciò non è stato fatto, ovvero se le informazioni non state trasferite con successo da RAM a memoria secondaria, si possono perdere delle informazioni! In particolare, se vengono danneggiate o non aggiornate correttamente **zone critiche del disco come FAT, bitmap, i-node o directory**, che sono delle informazioni di struttura, **il sistema degenera progressivamente**. Se invece ciò avviene con dati non critici, allora semplicemente questi dati conterranno delle informazioni obsolete.

Possiamo difenderci dall'inconsistency:

- **curando**, attraverso l'uso di filesystem checker come scandisk, fsck...
Questi programmi analizzano il disco per trovare delle incoerenze nel filesystem, e nonostante sistemino il problema riparando il fs, non possono fare nulla per i dati che potrebbero essere andati persi.
- **prevenendo**, usano un journaling file system come ext3, reiserfs.

Controlli di coerenza: fsck in UNIX

fsck in UNIX lavora in questo modo:

1. *Scandisce la tabella degli inode*, e controlla che l'elemento di inode sia coerente, per esempio se c'è un file da 10-15 caratteri, non siano allocati più di un blocco...
2. *Controlla le directory*: guarda che il formato delle directory sia coerente.
3. Dopo che ha trovato delle incoerenze, *scandisce l'albero e inizia a provare a raggiungere tutti i file delle directory che hanno degli inode validi*. Se trova dei file non raggiungibili o che hanno un inode corrotto, li mette in una cartella L+F (**lost+found**). In questa cartella, se ci sono problemi col filesystem troveremo dei file che hanno per nome l'inode (non avranno il loro vero nome perché appunto abbiamo trovato l'inode, ma non c'è l'elemento di directory).
4. *Controllo del reference count*, ovvero controlla che il numero di riferimenti a ciascun file sia coerente. Questo è molto importante, siccome se un file ha due riferimenti di cui uno non risolto, non potrà essere cancellato.
5. A questo punto, ha ricreato la tabella degli inode liberi e occupati, e nella fase 5 *aggiorna la tabella dei blocchi liberi-occupati* (detto anche cylinder group).
6. Si aggiornano le tabelle liberi-occupati per salvare i cambiamenti [poco importante].

File system basati su log (o di tipo journaling)

Questo meccanismo consente di ripristinare la coerenza del fs in meno tempo, in particolare i casi di incoerenza basata su errore. I log non possono però risolvere gli errori "fisici" che si creano in un hard disk (per esempio se la testina graffia uno dei dischi).

Quando facciamo delle operazioni su un file system, queste possono essere divise in una sequenza logica di operazioni, e si può fare questa operazione in modo che sia idempotente (ovvero che fare mille volte questa operazione diventa come farlo una volta sola), per esempio una operazione di aggiunta di 10 caratteri a un file diventa "aggiungi 10 caratteri, cambia le informazioni sulla grandezza del file nell'inode".

I **file system basati su log** operano in questo modo: al posto di fare ogni operazione direttamente su disco, si registrano questi passi da fare come se fosse un copione di operazioni che devono essere fatte in seguito alla richiesta (o un diario, da qui journal), e si mettono in una struttura che abbia transazioni, ovvero che abbia la maniera di registrare le operazioni uno dopo l'altra, come se avesse un diario di bordo (o journal) che è una struttura sequenziale. La transazione fa in modo che quando si elenca la sequenza di operazioni da fare, o viene registrata tutta la sequenza di operazioni, oppure non viene registrato nulla. Tuttociò ogni volta che stiamo facendo una **operazione di modifica/aggiornamento del FS**.

Quando si è certi che tutte le modifiche sono state effettuate, si può cancellare quella operazione sul log.

Se non abbiamo un FS basato su log, noi dobbiamo fare tante operazioni in punti diversi del disco, mentre con il log abbiamo la sequenza delle operazioni da fare in un'unica struttura lineare.

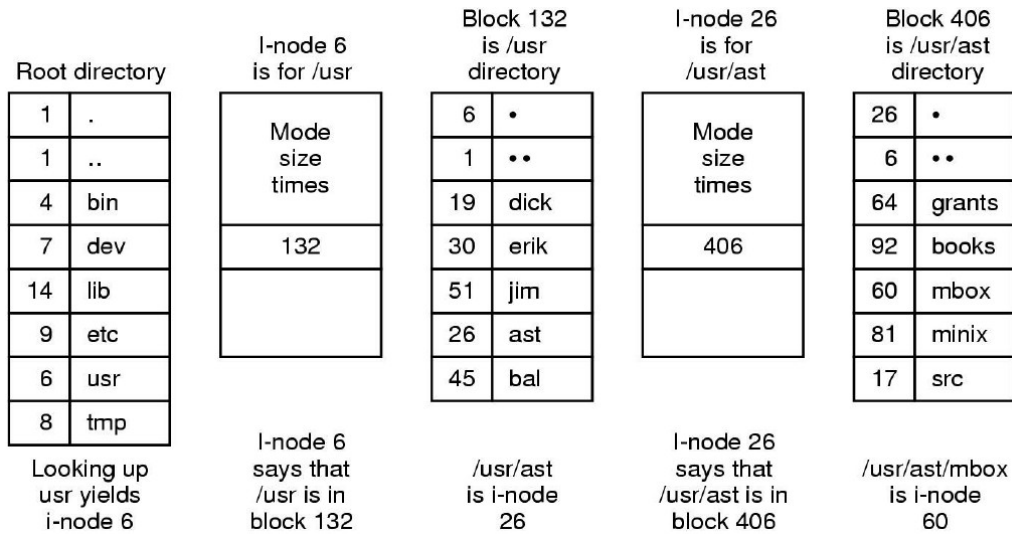
Se una operazione è stata scritta sul log, ma il log non è ancora stato confermato, viene perduta completamente, ovvero non viene svolta alcuna istruzione di questa operazione.

Se però abbiamo una sequenza di N operazioni, che portano il nostro FS da A coerente a B coerente, se non eseguiamo tutte le N operazioni allora non è garantito che lo stato risultato sia coerente.

Tuttavia, se nel log sono state salvate tutte le N operazioni da fare, anche se non abbiamo salvato B, e *se queste sono state confermate*, allora a prescindere da quante operazioni delle N che dovevamo fare abbiamo fatto, noi le facciamo tutte da capo partendo da A. In questo modo noi siamo certi di essere nello stato B.

Se abbiamo il log, anche se abbiamo un salto di corrente, possiamo tornare in breve tempo allo stato coerente, mentre se dovessimo fare il fsck ad ogni accensione, il tempo impiegato sarebbe notevole. In ogni caso, siccome il log comunque non risolve tutti i problemi (i problemi fisici di cui parlavamo prima, per esempio a causa di rottura di un settore del disco), in UNIX per esempio ogni 180 giorni viene chiesto di svolgere un fsck, in modo proprio di controllare che il file system che stiamo utilizzando sia coerente.

Esempio di file system in UNIX:



Introduzione alla sicurezza

[poco importante]

Per iniziare a parlare di sicurezza, dobbiamo prima dare alcune definizioni importanti:

- Con **Sicurezza** intendiamo il problema generale, che coinvolge non solo il sistema informatico, ma anche aspetti amministrativi, legali, politici e finanziari. Concetto “assoluto”: sicuro / non sicuro. Non possiamo mai essere troppo sicuri.
- Con **Trust** indichiamo la misura della fiducia sulla sicurezza di un sistema informatico. Questo è concetto “relativo”: abbiamo diversi gradi di fiducia. Trust e sicurezza spesso non coincidono.
- Con **Protezione** intendiamo l'insieme dei meccanismi utilizzati per proteggere il sistema informativo.

Spesso, se non c'è una cultura/consapevolezza della sicurezza, possiamo fare degli attacchi detti “di social engineering”: è il caso di phishing.

Obiettivi della sicurezza

Gli obiettivi della sicurezza, e quindi essenzialmente ciò che vogliamo proteggere, sono:

- **Data confidentiality**, ovvero come mantenere la segretezza/riservatezza dei dati.
- **Data integrity**, ovvero come evitare che i dati vengano alterati.
- **System availability**, ovvero come garantire che il sistema continuerà ad operare.

Alcuni di questi obiettivi sono in contrasto fra di loro: un sistema scollegato dal mondo garantisce la confidenzialità ma è poco disponibile.

Tutte queste 3 cose possono essere violate:

- Se violiamo la riservatezza, abbiamo un **disclosure**.
- Se violiamo l'integrità, avviene l'**alteration**.
- Se violiamo la disponibilità, abbiamo **denial of service**.

Lezione del 13/04/2021 – Sicurezza: Crittografia

Ripassino FS

Per creare un file vuoto da 100kB possiamo usare il comando `truncate -s 10k myfd`, e per vedere l'hexdump di questo file possiamo usare il comando `hd`. Con `mount` si indica mettere la gerarchia del file system di un device o partizione (o altro) nel nostro file system, in modo che sia facilmente raggiungibile.

Sicurezza, alcune considerazioni (o meglio, principi informali):

Non esistono sistemi che sono totalmente sicuri. Al contrario, invece, è molto facile creare sistemi che sono insicuri.

Un concetto chiave di software engineering è la separazione delle politiche (policy) dai meccanismi [l'abbiamo visto anche qui]. La componente che prende le decisioni per le politiche può essere totalmente diverse dalla componente che prende le decisioni per i meccanismi. In questo modo possiamo cambiare la politica senza cambiare i meccanismi e viceversa. Tratteremo soprattutto dei meccanismi.

Crittografia

Con crittografia si intende l'insieme dei meccanismi usati per trasformare messaggi in chiaro (plaintext) in messaggi cifrati (ciphertext). In questo modo i messaggi cifrati possono essere letti solo da chi ne ha l'autorizzazione. Vi è poi bisogno di un insieme di informazioni (chiave) che permettano di convertire un messaggio cifrato in un messaggio in chiaro.

Anche nei sistemi operativi ci sono sistemi di crittografia, che servono per garantire autenticazioni all'interno dei sistemi.

L'importante di questi sistemi è che non si trovi il modo di trovare la chiave dal messaggio cifrato, avendo già il messaggio in chiaro. Infatti, la funzione di crittografia dev'essere sempre iniettiva.

Occorrono delle funzioni che siano a senso unico (**one-way function, OWF**), in modo che avendo una funzione f , calcolare $f(x)$ sia relativamente facile, che calcolare $f^{-1}(f(x))$ sia difficile.

Le funzioni di crittografia generalmente “mescolano” i bit in modo complesso, e spesso e volentieri tramite diverse iterazioni sullo stesso insieme di bit, con operazioni di bit swapping, inversioni, etc.

La **crittografia a chiave privata** (o chiave simmetrica) è una tipologia particolare di crittografia in cui la chiave per crittografare i messaggi è la stessa usata per decrittargli. La chiave privata deve essere mantenuta segreta, e deve essere conosciuta ad entrambi gli estremi della comunicazione.

Il vantaggio principale di questo metodo è che la crittografia risulta molto veloce ed efficiente, mentre la distribuzione delle chiavi private è un problema di sicurezza.

La **crittografia a chiave pubblica**, invece, fa uso di due chiavi: una chiave pubblica, usata per criptare i messaggi in chiaro, e una chiave privata usata per decrittare i messaggi cifrati. In questo modo la chiave pubblica può essere appunto pubblicata, e conosciuta da tutti.

Le due chiavi sono comunque collegate in qualche modo, ma dev'essere complicato ottenere la chiave privata da quella pubblica.

Ci sono due approcci per la crittografia:

- **Security by obscurity:** ovvero ottenere sicurezza mantenendo gli algoritmi per la crittografia segreti. Questa è una politica fallimentare (es. Alan Turing con Enigma).
- **Sicurezza basata su chiavi:** la sicurezza viene ottenuta mantenendo le chiavi segrete, ed utilizzando spazi di chiavi enormi in modo da rendere difficili gli attacchi di forza bruta. Inoltre, se l'algoritmo è pubblico, un maggior numero di persone può analizzare il suo comportamento, ed individuare eventuali problemi (es, OpenSSL sul web, usato su internet, è una libreria pubblica).

Un esempio di algoritmo di cifratura è DES (Data Encryption Standard), che è un algoritmo a **chiave segreta/simmetrica** (oggi si usa anche 3DES che crea chiavi a 2×56 bit).

Ecco uno schema del funzionamento di DES:

- data una funzione a senso unico $OWF(k, x)$
 - dove k è una chiave a 56 bit, x è una chiave a 64 bit
- dato un messaggio **m** di 64 bit, lo si divide in
 - L_0 (32 bit più significativi)
 - R_0 (32 bit meno significativi)
- si calcola $L_i = R_{i-1}$; $R_i = \text{XOR}(L_{i-1}, OWF(k, R_{i-1}))$
- l'operazione si ripete per 16 volte, e L_{16} e R_{16} rappresentano la forma codificata

L'algoritmo DES come si può notare è molto meccanico, possiamo anche suare dei chip meccanici appositi che lo fanno estremamente velocemente.

Un altro algoritmo famoso di crittazione è RSA, che è un algoritmo a **chiave pubblica**.

- vengono scelti due numeri primi molto grandi **p** e **q** (almeno 100 cifre)
- si chiami **n=pq**.
- si sceglie un valore **d** in modo tale che sia primo con **(p-1)(q-1)**
 - cioè: **MCD(d, (p-1)(q-1))=1**.
- sia **e** l'inverso moltiplicativo di **d mod (p-1)(q-1)**
 - cioè: **de mod (p-1)(q-1)=1**
- allora
 - **E(m) = C = m^e mod n**
 - **D(C) = m = C^d mod n**

Attacchi

Gli attacchi possono essere attivi o passivi:

- gli **attacchi attivi** perturbano il sistema attaccato, e possono essere facili da individuare, dato che possiamo trovare degli indizi del sistema che portano a pensare che sia compromesso.
- gli **attacchi passivi** invece non perturbano il sistema, ma riescono a reperire informazioni utilizzando *eavesdropping* (spiando canali di comunicazione). Per questo motivo sono molto difficile da individuare.

Se pensiamo che in quartier generale ci sia una zona/perimetro considerato sicuro, fatto di persone o processi a cui viene data fiducia (zona demilitarizzata), possiamo dividere gli attacchi in interni o esterni:

- **interni**, sono attacchi che vengono all'interno del sistema potretto, a causa per esempio di un "tradimento" interno, oppure a causa di un attacco esterno avvenuto precedente (es. cavallo di troia, in cui prima hanno fatto entrare il cavallo dall'esterno).
- **esterno**, sono attacchi che avvengono tramite interfacce di comunicazione con l'esterno.

Dal punto di vista tecnico, gli obiettivi degli attaccanti sono acquisire una qualche forma di controllo della macchina, per poi acquire, se possibile, un controllo totale della macchina.

Nei sistemi tipici multiutente, abbiamo la possibilità di far accedere più persone allo stesso sistema informatico.

Possiamo fare una distinzione fra **utenti normali**, che hanno accesso ad un sottinsieme di risorse personali, e **superutenti o amministratori** (o root o admin) che hanno accesso all'intero insieme di risorse del sistema, e la controllano totalmente.

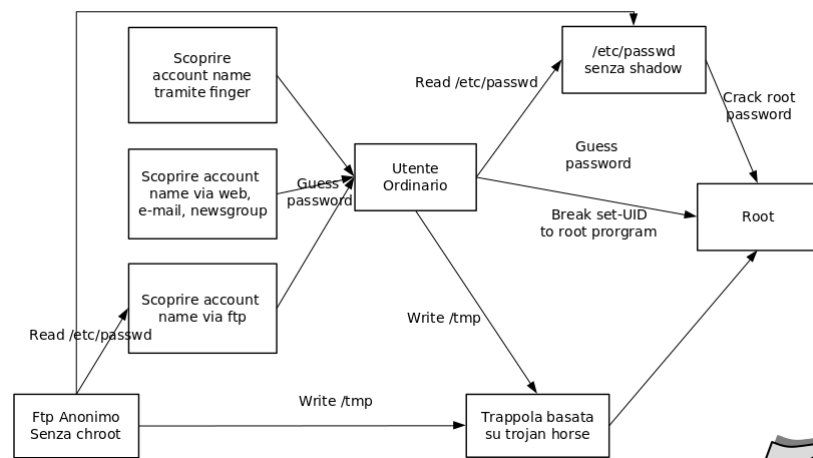
Tutti gli elementi di una macchina possono essere coinvolti in un attacco. Tra questi, anche l'hardware. Inoltre, se c'è accesso fisico a una macchina, la protezione diventa difficile da garantire: infatti, per creare un DoS basta staccare una spina, oppure per leggere dei dati un potrebbe rubare un hard disk, e leggere i file anche se erano protetti nella directory dell'utente. Volendo, uno potrebbe criptare i dati nel proprio hard disk, ma in questo modo otterrebbe un sistema molto meno efficiente.

Anche il kernel può essere soggetto ad attacchi, perciò dovrà fornire dei meccanismi di autenticazione e di autorizzazione.

Le librerie anche possono contenere delle grandi quantità di codice spesso e volentieri che viene eseguito spesso e volentieri in modalità superutente.

Le applicazioni anche possono un ponte per attaccare un sistema, e gli utenti sono l'anello più debole (perciò non devono essere inconsapevoli).

Ecco uno schema dei tipici attacchi:



Lezione del 16/04/2021 – Buffer overflow, TOCTOU, Trojan e virus

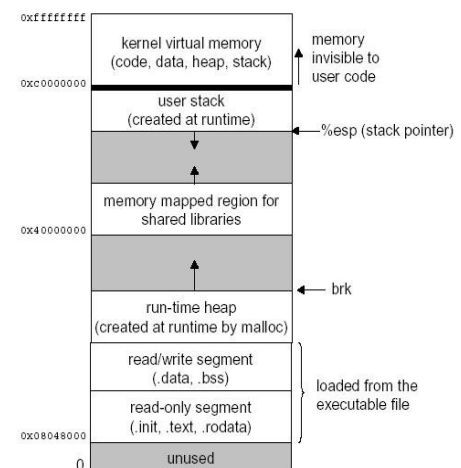
Buffer Overflow

Questa condizione si ha quando un programma legge dati provenienti dall'esterno, mettendoli in un buffer, e non si implementa un controllo della lunghezza del buffer. In poche parole, viene inserita una stringa di dati troppo grande per essere contenuta dal buffer, quindi l'input tracima l'ampiezza del buffer e la condizione non viene gestita.

Nel migliore dei casi, il programma va in crash e si ha un segmentation fault (è comunque però un problema di DoS).

Nel peggiore dei casi, invece, è possibile implementare dei trucchi per "iniettare" codice, e fare in modo che questa venga eseguita. Per esempio, quando il buffer è una variabile locale/automatica di una funzione ovvero memorizzata in uno stack, e sapendo che lo stack cresce a ritroso dagli indirizzi alti a quelli bassi, quindi se noi eccediamo il buffer, andiamo a sovrascrivere altre informazioni che stanno sullo stack, e per esempio potremmo andare a sovrascrivere l'indirizzo di ritorno della funzione.

Più del 50% degli incidenti riportati al CERT sono causati da buffer overflow.



Come fare un buffer overflow? Per farlo, dobbiamo conoscere alcune specifiche del sistema operativo. Inoltre, dobbiamo conoscere un po' di linguaggio macchina e dell'organizzazione della memoria (es. struttura segmento codice, segmento dati, segmento stack...)

Ecco un esempio tipico

```
int main()
{
    int buffer[10];
    buffer[20] = 5;
}
```

Questo è un esempio tipico di buffer overflow in C, infatti il compilatore C non crea eseguibili che fanno controlli in modo automatico (altrimenti otterremo eseguibili di dimensioni enormi).
Dunque, codice del genere, in cui il buffer viene creato nello stack e si modificano i dati di un indice maggiore dell'ampiezza del buffer, **può andare a sovrascrivere informazioni sul buffer.**

Vediamo ora perché avviene buffer overflow in modo più tecnico: ad ogni chiamata di funzione vengono allocati dei dati sullo stack, **tra cui indirizzo di ritorno, argomenti della funzione e dati/variabili locali.** Lo stack cresce verso il basso, e quindi in questo modo, come dicevamo prima, altri dati vengono facilmente sovrascritti, e possono iniettare codice maligno opportunatamente caricato modificando l'indirizzo di ritorno.

Esistono anche alcune funzioni di C deboli, che possono portare ad un buffer overflow siccome non controllano la dimensione dei loro argomenti, e sono:

- `gets()`, che è una funzione che legge lo standard input.
- `strcpy()`, copia una stringa in un'altra
- `sprintf()`, formatta una stringa in un buffer

Come fare quindi a difendersi dal buffer overflow? L'unica vera misura è evitare sloppy programming, quindi non programmare senza ragionare. Inoltre, sarebbe necessario evitare le funzioni C appena citate sopra.

È consigliabile aggiungere codice che controlli il valore di ritorno a run-time, valutando le dimensioni degli stack frame. In caso la dimensione degli stack frame presenti qualche errore, si termina il programma.

Possiamo anche aggiungere delle contromisure a tempo di compilazione, per esempio esistono patch per compilatori che aggiungono controlli sulla dimensione dei buffer. Tuttavia, questi rendono il codice molto meno performante. Esistono sono patch che controllano unicamente l'indirizzo di ritorno (es. StackGuard e StackShield).

Linux presenta anch'esso una soluzione per i buffer overflow, che consiste nel variare ogni volta di un po' l'indirizzo d'inizio dello stack.

Attacco TOC/TOU (Time Of Check/Time Of Usage)

In questo tipo di attacco, si fa un controllo di un permesso in un dato momento, e si esegue l'operazione che aveva tale permesso in un altro momento, e quindi dovuta al fatto che l'operazione di controllo e di utilizzo vengano fatte assieme in modo atomico (infatti è un tipico problema da race condition). È anche il motivo per cui la syscall `access()` è deprecata.

Victim	Attacker
<pre>if (access("file", W_OK) != 0) { exit(1); } fd = open("file", O_WRONLY); // Actually writing over /etc/passwd write(fd, buffer, sizeof(buffer));</pre>	<pre>// // // After the access check symlink("/etc/passwd", "file"); // Before the open, "file" points to the password database // //</pre>

Nel codice della vittima, appena si è finito di fare il permission check del file, l'attaccante inserisce un file con un link simbolico a `/etc/passwd`, in questo modo potrà scrivere nel file delle password.

I TOCTOU possono essere anche usati per privilege escalation.

[Disclaimer: non ho capito bene se è effettivamente così, ma il prof non l'ha spiegato troppo bene] Se si aggiunge un `setuid()` all'utente che ha fatto il check o prima di fare l'accesso effettivo al file.

Trojan Horse

I **cavalli di troia** sono programmi che replicano le funzionalità e l'aspetto di programmi di uso comune, oppure sono programmi dall'apparenza innocua, che però contengono codice maligno. Tipicamente, possono disturbare l'utente catturando le sue informazioni e inviandole al creatore del programma, informazioni che possono essere critiche per la sicurezza del sistema oppure "private" dell'utente (spyware).

Possono anche compromettere o distruggere le informazioni importanti per il funzionamento del sistemi.

Il successo di questi programmi dipende unicamente dall'accorgimento e dalla competenza dell'utente, e non dal programmatore del sistema. Per contenere il problema, si possono confinare gli ambienti in modo che solo l'utente del sistema che ha aperto il programma sia effettivamente attaccato.

Volendo, se uno non si fida di certi programmi, può eseguirli all'interno di una macchina virtuale, in modo che non ci siano troppi problemi.

Con questo tipo di programmi è importante tenere a mente che se un utente x lancia un certo programma, allora tutti i dati di tale utente saranno accessibili a quel programma.

Per esempio, se abbiamo un server web nel nostro sistema, questo non viene eseguito come root, ma come un utente www-data, che ha accesso solo alla parte del sistema/del file system dedicati al webserver.

Altri esempi di trojan horse sono:

- Login spoofing, in cui si crea sullo schermo vengono mostrati una interfaccia di login fasulli, e in questo modo l'attaccante può rubare facilmente i dati all'utente.
- Oppure un utente che inserisce nel \$PATH una versione di programmi maligni (es. un ls maligno).

Bombe logiche

Un esempio di bomba logica può essere un creatore di un software utilizzato internamente in una compagnia, e che inserisce un software che può attivarsi sotto particolari condizioni (es. se viene licenziato). Per difenderci da questo, possiamo fare spesso code reviews.

Backdoor / Trapdoor

Sono essenzialmente degli accessi secondari del sistema che aggirano i sistemi di protezione ordinari. Il creatore di un software, in questo modo, può lasciare porte di servizio, per entrare aggirando i sistemi di protezione. (Un esempio può essere il virus MyDoom. Oppure Microsoft con il Back Orifice, che si era tenuta una backdoor ai tempi di Windows NT nella porta 31337. Oggi questa porta viene usata per installare gli aggiornamenti, o _NSAKEY, altra cagata della Microsoft.)

Virus e Worm

Un **virus** è un frammento di DNA capace di creare delle proteine improprie causando effetti collaterali, quindi il virus da solo non vive, ed ha bisogno di un organismo per continuare il suo funzionamento. In informatica, è un frammento di programma che può infettare e ricopiarsi in altri programmi non maligni modificandoli. Un **worm**, invece, biologicamente è più simile ad un batterio siccome è un programma maligno completo, che si diffonde in una rete.

Oltre al modo di operare, altre differenze principali fra un Virus e un Worm sono:

- I worm operano SOLO sulle reti, i virus possono usare qualunque supporto
- I worm sono programmi autonomi, i virus infettano programmi esistenti

Tuttavia, oggi non c'è più una vera e propria distinzione. Un tempo semplicemente si pensava che se un computer fosse offline, quindi disconnesso dalla rete, non potesse essere soggetto a Worm e quindi

programmi maligni, ma i virus dimostrarono esattamente il contrario, siccome si diffondevano tramite file, chiavette, o CD.

L'attività di un virus si divide in due fasi:

- **Riproduzione (infezione):** in questa fase deve bilanciare infezione e possibilità di essere scoperto, usando tecniche per nascondersi e infettando i programmi.
- **Attivazione**, che può essere scatenata da uno specifico evento. In generale, un virus attivato compie azioni dannose (o semplicemente consuma risorse).

Un virus deve essere eseguito per potersi diffondere e poi attivarsi. E per forzare la propria esecuzione, si può:

- accodare in un programma esistente (.exe in Windows, ELF in Linux)
- oppure sfruttare un meccanismo di bootstrap (avviandosi all'esecuzione di un boot sector del disco, o del master boot sector)
- oppure accodandosi a file dati che permettono l'esecuzione di codice (es. posta elettronica, macro virus in Office).

Come ci si difende dai virus? L'errore più grande è che ci si difende da essi con un antivirus. Ciò che consente ai virus di poter esistere è pensare che i programmi possano essere modificabili, ovvero dobbiamo costruire programmi che non abbiano modi per modificare il programma da parte degli utenti. Oppure non dividere utenti normale dall'amministratore, per sola ragione di comodità.

Esistono anche **virus stealth**, che si nascondono dagli antivirus in quanto non presentano la “signature” (ovvero il frammento di codice) che caratterizza un virus, e tale firma normalmente viene caratterizzata dagli antivirus (es. il worm code red inizia con `/default.ida?NNNNN`). Essenzialmente infatti gli antivirus cercano semplicemente un pattern nel bytecode dei virus. I virus stealth semplicemente includono dei noop o degli incrementi al posto delle somme, in modo da non essere riconosciuti.

Lezione del 20/04/2021 – Worm di Morris, Autenticazione: Password, PAM, ACL e Capabilities con le loro versioni POSIX

Worm di Morris

Il Worm di Morris era uno dei primi worm conosciuti, ed era stato creato per dimostrare quanto fossero incompetenti gli amministratori di sistema all'epoca.

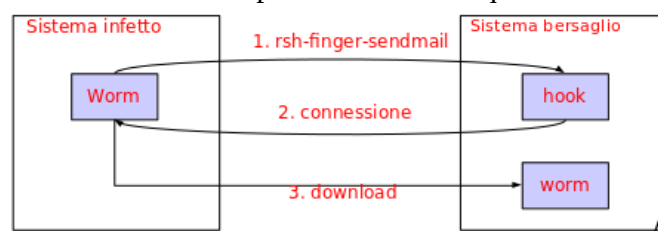
All'epoca, esistevano 3 servizi (o daemon, come ha detto il prof, ma non so se sia giusto) che erano delle “falle di sicurezza” del sistema. Questi erano i comandi:

- **finger**, che come abbiamo visto fa vedere quali utenti sono loggati in un dato momento.
- **sendmail**, che essenzialmente era un servizio grazie al quale si poteva inserire un file .forward nella propria home directory, con indicazioni a chi si vuole rimandare la posta. Potevo mettere però il carattere della pipe (“|”), in questo modo la posta veniva elaborata da un processo.
- **rsh** (progenitore dell'ssh, molto meno sicuro, siccome abilitava l'accesso senza password alla macchina, solo conoscendo l'utente).

Morris usò queste 3 falle per creare il Worm. Mandò il codice sorgente ad una macchina, poi lo compilò usando gcc nella macchina ospite, e poi si eseguiva. Il worm poi andava a vedere tutti gli host conosciuti in quel momento, e inviava a questi host il Worm.

Questo worm, tra l'altro, non faceva nulla di male, ma era solamente programmato per replicarsi.

Nonostante ciò, buttò giù l'intero internet mondiale all'epoca, siccome comunque il traffico in questo modo era esponenziale, e la rete non era fatta per sostenere tali quantità.



Il Worm di Morris permette di capire bene la differenza fra virus e worm: il virus è un segmento di programma, che va a modificare un programma esistente, il virus non vive da solo. Mentre il worm è un programma indipendente. Inoltre, il virus è fatto per i programmi di ogni genere, mentre i worm sono fatti per i programmi demoni nel server. **Mentre il virus si avvia a causa di un utente che lo esegue, il worm è completamente autonomo.**

Per difendersi dai virus, basta usare sistemi operativi che impediscono ai programmi di modificare sé stessi ed altri programmi, mentre per difendersi dai worm, aggiornare costantemente i programmi (e i software liberi permettono di avere aggiornamenti stabili e sicuri, che non compromettono altre parti).

Firewall

Un firewall è composto da:

- un bastione esterno, ovvero una macchina che collega la rete esterna.
- una rete, detta zona militarizzata, che collega il bastione interno.
- all'interno del bastione interno, c'è la rete interna.

Tutti i servizi che devono essere visti da fuori si inseriscono nella zona militarizzata, in modo da evitare che l'utente esterno si colleghi con l'interno.

Esiste poi il cosiddetto **teorema dei firewall**, che dice che il codice che viene eseguito sui bastioni deve essere semplicissimo, siccome quelli sono i punti in cui si valuta la sicurezza della rete interna.

Autenticazione

Autenticazione ed autorizzazione sono due facce della stessa medaglia: un sistema operativo deve dire chi può fare cosa e per garantire che chi stia chiedendo di fare cosa sia effettivamente lui.

L'autenticazione è alla base di tutti i meccanismi di protezione, e senza autenticazione tutti i meccanismi di sicurezza che vedremo saranno completamente inutili.

I meccanismi per ottenere autenticazioni sono basati su qualcosa che l'utente "è", qualcosa che l'utente "ha" oppure qualcosa che l'utente "conosce".

È bene se l'elenco degli del sistema non sia pubblico, siccome riduce le chance di violazione e di accesso.

Autenticazione basata su password

Il primo di autenticazione che vediamo è l'**autenticazione basata su password** (e quindi basato su qualcosa che uso sa). Una password è una parola chiave segreta, che consente di identificare la persona.

La password può essere un sistema debole se si ha una scarsa cultura delle password (es. uso di password banali, come qwerty12, oppure con post-it con la propria password).

In più, la password può essere facilmente "rubata", attraverso per esempio login spoofing.

Oggi dobbiamo usare molte password diverse per molti servizi, e quindi la sicurezza delle nostre password, si sposta alla sicurezza del luogo/metodo che usiamo per mantenere e salvare le password. Nei sistemi più obsoleti, le password venivano memorizzate in chiaro in un file, che era protetto da normali meccanismi di protezione.

Adesso si usano file crittati per la password, oppure usando una funzione hash per memorizzare la password degli utenti nel file delle password (es. `/etc/passwd` in Linux).

[digressione]

In Linux, c'è un file `/etc/shadow` che contiene le la password crittata ed è accessibile solo a root.

Ci sono anche degli utenti ai quali non si può accedere, che quindi non sono veri e propri utenti, ma che servono a dare dei permessi a dei certi files.

[fine digressione]

Per attaccare le password, possiamo avere dei metodi differenti.

Nel caso di password protette da hash, possiamo usare un dizionario di password banali, che codifichiamo con una data funzione di hash, e le confrontiamo con le password ottenute nel file delle password, in modo da capire se una di esse combacia ed arrivare alla password di partenza (**cracking**).

[digressione, il prof si diverte molto a fare in sta lezione]

Esiste il mito della password sicura, in deve essere almeno 8 caratteri, con simboli e 0 e 1, tuttavia una password del genere è relativamente facile da craccare e difficile da ricordare.

Le migliori password sono semplicemente 4 parole senza alcuna relazione semantica, che sono molto più difficili da craccare e sono più semplici da memorizzare (a patto che non si usino simboli o numeri troppo complessi ovviamente).

Si chiama uno script kiddie un “hacker” che usa script fatti da altri che non capisce per violare la sicurezza dei sistemi, anche se senza cattive intenzioni, ma per impressionare i suoi amici e conoscenti.
[fine digressione]

Meccanismo di salt per le password

Se uno riesce ad avere il file delle password crittate, può eseguire un attacco detto **attacco database** (o **rainbow table**), che è ancora più subdolo dell’attacco dizionario visto prima (ovvero il cracking), siccome prendiamo il vocabolario in chiaro (proprio il vocabolario delle parole italiane) e lo crittiamo, poi vediamo se una di queste parole corrisponde ad una delle database entry.

Uno dei meccanismi usati per difendersi da questo tipo di attacchi è l’uso del salt, ovvero del sale, in cui sostanzialmente noi non crittiamo solo la parola stessa, ma crittiamo questa + un numero casuale (che è appunto il salt) che viene memorizzato in chiaro nel file delle password.

In questo modo nel database dell’attaccante, devo aggiungere non solo la codifica della parola “mamma”, ma anche la codifica di “mamma” + ciascun valore del sale. Oggi la codifica delle password viene sempre fatta con sistemi di salt.

Packet sniffing

Il **packet sniffer** sono particolari software che analizzano il traffico di rete, e che cercano di individuare pacchetti contenenti informazioni importanti (tra cui password). Un modo per difendersi da questo è attraverso l’uso di crittazione dei dati (es. usando SSL). Protocolli come ftp e telnet per questo motivo sono deprecati.

Challenge e response

Se vogliamo fare password ancora più sicure, possiamo usare il meccanismo di **challenge e respond**. Basato su funzioni che sono **one-way function**, si basa su questo sistema:

- La password (chiave) k è nota sia all’utente che al sistema a cui si vuole accedere (eventualmente **codificata**)
- Il sistema propone una challenge (sfida), ovvero un valore numerico c che viene spedito all’utente.
- Sia il sistema che l’utente calcolano $f(c, k)$.
- L’utente comunica al sistema il valore di $f(c, k)$.
- Il sistema confronta questo valore con il valore calcolato localmente.

In questo modo i packet sniffer non possono scoprire i dati.

Un modo che un sysadmin può usare per amministrare la sicurezza è quello di craccare lui stesso le password (con programmi come JohnTheRipper) per verificarne la loro sicurezza. Oppure si può fare uso di password one-shot (ovvero usa e getta).

Password One-shot

Le **password one-shot** sono password fatte in modo che l’utente debba utilizzare una password nuova ad ogni accesso. L’utente potrà avere un elenco di password stampato, e scegliere elementi successivi dalla lista. Tuttavia questo metodo è molto pericoloso, infatti la lista può essere letta e copiata. Un tempo era usato anche in ambiti militari e nelle banche.

Autenticazione tramite oggetti fisici

Possiamo usare anche degli oggetti fisici per autenticarci. È il caso di:

- Bancomat, facili da copiare e basano la loro sicurezza su due meccanismi, composto da codice pin e bancomat.
- Smartcard, difficili da copiare siccome usano una piccola unità di calcolo, e permettono di usare dei meccanismi di challenge e response.
- Autenticazione biometrica, come impronte digitali, retina... (qualcosa che uno è).

PAM (Pluggable Authentication Model)

Tradizionalmente, in UNIX, l'autenticazione avviene in due diversi contesti, ed ogni servizio aveva la propria implementazione dell'autenticazione. I meccanismi erano fissi e non configurabili/modulari, e questo era un problema, siccome se c'era un fallimento da qualche parte, allora si fregava l'intero sistema.

Adesso, invece, i meccanismi sono altamente configurabili, attraverso moduli liberamente caricabili. In generale, ogni programma che ha bisogno di una qualsiasi forma di autenticazione, fa uso della **libreria PAM**, che si occupa esattamente di questo, ovvero di fornire un sistema di autenticazione. In questo modo, applicazioni come il login, passwd, su.. fanno tutte delle chiamate ai moduli di autenticazione usando l'API di PAM.

[la parte qua sotto è solo per completezza.]

Ecco un esempio:

%PAM-1.0

auth	sufficient	pam_rootok.so	
auth	required	pam_unix2.so	nullok
account	required	pam_unix2.so	
password	required	pam_pwcheck.so	nullok
password	required	pam_unix2.so	nullok use_first_pass use_authtok
session	required	pam_homecheck.so	
session	required	pam_unix2.so	debug # none or trace

Questo è il modulo per il comando su.

Il primo campo descrive la classe d'operazione, e può essere:
auth: procedure per l'autenticazione dell'utente
account: per modificare gli attributi di un account
password: per modificare la password
session: per debug e logging su syslog

I moduli vengono valutati in ordine, e ogni modulo può rispondere con *grant* oppure *deny*. Il secondo campo dice come deve essere valutato il valore di ritorno:

sufficient: se il modulo ritorna grant, la risposta è positiva e i moduli successivi non vengono considerati
required: se il modulo ritorna deny, la risposta è negativa e i moduli successivi non vengono considerati
required: questo modulo deve ritornare grant affinché l'operazione abbia successo
optional: questo modulo influisce sul risultato solo se è l'unico modulo presente

Autorizzazione: protezione e controllo dell'accesso

L'autorizzazione serve per stabilire essenzialmente chi può fare cosa. Questo è possibile anche grazie al fatto che il sistema può essere in **modo user** (in cui il kernel verifica se l'operazione può essere fatta o meno) o **modo kernel** (in cui si esegue codice del kernel e si può accedere a tutto).

Formalmente, l'autorizzazione è l'insieme di meccanismi e politiche con cui il S.O. "decide" se un soggetto ha il permesso di eseguire una determinata azione su un oggetto. Il modo più diffuso di realizzare l'autorizzazione tramite software è attraverso la **Matrice di Accesso (con Access Control List o capability)**.

I principi fondamentali della autorizzazione sono:

- Principio di **failsafe**: ovvero nessun soggetto ha diritti per default
- Principio di **privilegio minimo**: ogni soggetto ha, in ogni istante, i soli diritti necessari per quella fase dell'elaborazione. Se può fare altro che non è necessario, allora potrebbe essere un problema di sicurezza. Es: root/administrator e utenti.

Alcune definizioni importanti sulle autorizzazioni

Abbiamo:

- Insieme dei soggetti **S**: i soggetti sono delle entità attive che eseguono azioni. Nel nostro caso sono i processi.
- Insieme degli oggetti **O**: Gli oggetti sono delle entità (passive/attive) su cui vengono eseguite le azioni (inoltre l'insieme dei soggetti è un sott'insieme dell'insieme degli oggetti). Nel caso di UNIX, sono file, dispositivi e processi.
- Insieme dei diritti d'accesso **R**: rappresenta l'insieme delle azioni che possono essere eseguite. In UNIX questo insieme è rappresentato dall'insieme dei diritti d'accesso (ovvero rwx, read write execute).

Un **dominio di protezione** è un insieme di coppie $\langle o, rs \rangle$ dove $o \in O$ e $rs \subseteq R$. Detto in modo più informale, ogni coppia specifica un oggetto e un insieme di azioni che possono essere eseguite su tale oggetto.

Al posto di dire per ogni processo cosa può fare e cosa non può fare, stabiliamo che un processo abbia un proprietario, e uniformiamo i permessi e i livelli d'accesso a livello del proprietario.

Normalmente, ogni utente opera all'interno di un dominio di protezione, che determina cosa può fare e cosa non può fare. In UNIX in particolare, il dominio di protezione è determinato da user-id e group-id.

Matrice d'accesso e ACL

Si può pensare al problema dell'autorizzazione come la gestione della **matrice d'accesso** (anche se è una visione puramente teorica).

La matrice d'accesso è una matrice dominio/oggetto, in cui l'elemento $A(i,j)$ contiene i diritti che il dominio D_i prevede per l'oggetto O_j . Quando viene creato un nuovo oggetto, si aggiunge una colonna alla matrice d'accesso, e il contenuto di tale colonna è deciso al momento di creazione dell'oggetto.

oggetti domini	File1	File2	File3	Stampante
D1	read			
D2		read write	read	
D3			read write	write
D4				write

Questo approccio però è puramente teorico, infatti in un sistema reale abbiamo tantissimi processi diversi e la matrice sarebbe enorme.

Per compattare queste matrici allora si fa uso delle **ACL (ovvero Access Control Lists)**. Con questo metodo associamo le informazioni di protezione all'oggetto, tramite una lista di elementi $\langle \text{dominio}, \text{diritti di accesso} \rangle$. Metaforicamente, sarebbe come attaccare un'etichetta ad un oggetto fisico (es. un disinfettante) che ci dice "questo disinfettante può essere usato solo dagli studenti". Quindi non è il soggetto ha sapere che non ha un permesso su un certo oggetto, bensì l'oggetto stesso a comunicare al soggetto cosa può fare e cosa non può fare.

In UNIX, la stringa $rw-r-xr-x$ è una forma compatta della ACL. Se vogliamo cambiare i permessi, cambieremo "l'etichetta" e questo cambiamento diventa immediatamente efficace.

È ovvio che la verifica dell'etichetta e la sua gestione dev'essere fatta da un soggetto terzo (il kernel) in modo che anche l'autorizzazione d'accesso e la modifica dell'etichetta venga fatta in modo protetto.

Capabilities per l'autorizzazione d'accesso (DA NON CONFONDERE CON LE POSIX CAPABILITIES, che vedremo [qui])

Questo approccio è duale a quello della ACL: infatti non associamo più le informazioni di protezione all'oggetto, bensì le associamo al dominio, quindi essenzialmente spostiamo la memorizzazione.

Dunque, avremo una lista di capabilities, ovvero coppie $\langle \text{oggetti}, \text{diritti di accesso} \rangle$.

Possiamo vedere un capability come una sorta di "chiave" per l'accesso alla serratura dell'oggetto.

Perché il meccanismo delle capabilities funzioni, le capabilities non possono essere coniate (Non possiamo creare delle chiavi doppie, sennò la sicurezza sarebbe compromessa).

Le capabilities possono essere gestite in due modi:

- Possiamo mantenere le capabilities nello spazio dell'utente. Questo è anche l'approccio più comune.
 - In questo caso le capabilities sono protette da meccanismi crittografici, e queste sono memorizzate dai processi, ma non possono essere modificabili.
Il modo con cui viene crittato è con chiave singola, perché chi scrive le capabilities è lo stesso che lo legge.
- Altrimenti, possiamo mantenere le capability in uno spazio del kernel associato al processo. In questo modo però si vanifica il lato positivo delle capability (ovvero che i permessi sono in mano all'utente). Questo sistema è comunque parzialmente usato da Linux.

Nel file system si usano le ACL, ma molto spesso, quando noi ci autenticiamo via web ad un servizio, per permettere di poter accedere alla pagina senza loggarci di nuovo ci viene dato un cookie, che essenzialmente è una capability.

Perché però in questo caso non usiamo una ACL? Basta pensare che in un FS il degli utenti è ben definito, molto spesso i servizi accedibili via rete sono acceduti da un gran numero di persone, magari anche gente che si logga una volta e poi mai più... quindi non conviene usare una ACL, perché sarebbe un enorme spreco di memoria. Per questo motivo, le capabilities sono usate nei sistemi distribuiti.

Revoca dei diritti d'accesso

Come facciamo a revocare i diritti di accesso ad un utente che non ci sta più molto simpatico (es. la nostra ex fidanzat*)?

Innanzitutto, la revoca può essere:

- immediata o ritardata (subito o si può attendere)
- selettiva o generale (per alcuni i domini o per tutti)
- parziale o totale (tutti i diritti o solo alcuni)
- temporanea o permanente. (la capability stessa si potrebbe essere temporanea).

Nei sistemi basati su ACL, per revocare i dati, basta aggiornare in modo corrispondente le strutture dati d'accesso.

Nei sistemi basati su capability, l'informazione relativa ai permessi è memorizzata presso i processi, e quindi la rimozione diventa più difficile, e per rimuoverlo si possono adottare più approcci diversi:

[**DISCLAIMER**: questa parte il prof non l'ha spiegata ma l'ho aggiunta per completezza]

- **Capability a validità temporale limitata**: una capability "scade" dopo un prefissato periodo di tempo. In questo modo si ha una revoca ritardata.
- **Doppia memorizzazione**: ogni capability viene controllata prima di essere utilizzata, tuttavia in questo modo si perdono alcuni dei benefici delle capability.
- **Capability indirette**: vengono concessi diritti non agli oggetti ma a elementi di una tabella globale che puntano agli oggetti. È possibile revocare diritti cancellando elementi della tabella intermedia.
- **Cambiamento dell'identità dell'oggetto (contatore nel nome)**: i processi devono chiedere nuovamente l'autorizzazione di accesso, ma può essere pesante se ci sono frequenti variazioni.

Modello di UNIX per il controllo di accesso

In UNIX, ogni processo possiede 6 o più ID associati ad esso:

- **real user ID, real group ID**: identificano il vero utente e gruppo che esegue il processo. questi valori sono presi dalla entry nel /etc/passwd file e non cambiano durante la vita di un processo.
- **effective user ID, effective group ID, supplementary group IDs**: sono quelle effettivamente utilizzate per determinare i nostri diritti di accesso al file system.
- **saved set-user-ID e saved set-group-ID**: contengono copie della effective user id e della effective group id; inizializzate con la system call setuid.

Normalmente, l'**effective-user-id** e **real-user-id** coincidono (lo stesso vale per i **group-id**), tranne però negli eseguibili che eseguono una `setuid()` o `setgid()`.

In Linux, infatti, abbiamo degli eseguibili che hanno bisogno di svolgere alcune operazioni con diritti d'accesso superiori. Es: l'utente normale non può nemmeno leggere il file `/etc/shadow`, ma quando dobbiamo cambiare la password con il comando `passwd`, quel comando deve poter accedere al file `/etc/shadow`.

Ci sono operazioni che possono cambiare i permessi dei nostri file:

- nei bit dei permessi di un file eseguibile, il bit set-user-ID causa la seguente operazione:
effective user id := owner del file
- nei bit dei permessi di un file eseguibile, il bit set-group-ID causa la seguente operazione:
effective group id := owning group del file
- nei bit dei permessi di una directory, il bit set-group-ID causa la seguente operazione:
i nuovi file creati nella directory ereditano il group owner

Questa cosa succede anche quando spostiamo un file in una cartella, ad esempio quando facciamo la consegna del progetto, il file consegnato ha permessi particolari settati, che ne impediscono la lettura agli altri studenti.

Un altro esempio è sempre quello che avviene quando eseguiamo il comando `passwd` per cambiare la password:

1. l'owner del comando `passwd` è root
2. quando `passwd` viene eseguito, il suo effective user id è uguale a root
3. il comando può scrivere su `/etc/passwd`

Alcune precisazioni sull'accesso a file e cartelle in UNIX:

- Per aprire un file per nome, necessario il diritto di esecuzione (search) su tutte le directory che fanno parte del path. Attenzione ai nomi relativi: current directory è implicita.
- Per creare un nuovo file in una directory: necessari i diritti di esecuzione (x) e scrittura (w) sulla directory.
- Per cancellare un file in una directory: necessari i diritti di esecuzione (x) e scrittura sulla directory (w). Non abbiamo bisogno di diritti sul file, infatti posso cancellare un file su cui non ho permessi se ho permessi di lettura/scrittura della directory.

Sticky bit

Il 12° bit dei permessi è detto **sticky bit**, ma il nome "corretto" è saved text mode (da cui la **t** nei permessi mostrati da `ls`). Questo bit, nelle directory, impedisce ad un utente di cancellare file che non gli appartengono, **nonostante abbia i diritti di scrittura sulla directory** (Esempio di utilizzazione: `/tmp`). Negli eseguibili è obsoleto.

Accesso a un file in UNIX-like

Consideriamo di voler accedere ad un file in UNIX, allora avremo che gli attributi *effective uid*, *effective gid*, *supplementary gids* sono proprietà del processo in esecuzione, mentre *owner* e *group owner* sono proprietà del file a cui si vuole accedere.

L'algoritmo in UNIX che dà autorizzazione di accesso a un file è:

- IF *effective uid* == 0 (root) THEN **access allowed**
- ELSE IF *effective uid* == owner AND appropriate user permission THEN **access allowed**
- ELSE IF *group owner* \in *supplementary gids* \cup {*effective gid*} AND appropriate group permission THEN **access allowed**
- ELSE IF appropriate other permission THEN **access allowed**
- ELSE **access denied**

Tutte le volte che devo creare un file, uso una umask. Tutti i bit che sono accesi nella maschera, verranno spenti nell'access mode del file creato.

POSIX Working Group, POSIX ACL e POSIX Capabilities

L'intero sistema di autorizzazione (Access Control) in UNIX è creato solo mantenendo 12 per il codice di autorizzazione nel file (per l'ACL proprio), e poi un paio di interni per il group-id e lo user-id. In questo modo è molto efficiente.

Tuttavia, a volte questo non basta, e dobbiamo fare uso di un workaround per avere un sistema più complesso.

Per questo, il gruppo POSIX ha tentato di standardizzare numerose problematiche relative alla sicurezza, aggiungendo nuove funzionalità tra cui:

- POSIX Access Control List .
- POSIX Capabilities (!= dalle capabilities viste in precedenza).
- Mandatory Access Control (MAC)
- Audit
- Information Labeling

Linux per ora ha implementato solo i primi due.

In particolare, le ACL sono implementate in tutti i Linux moderni, e per permettere backwards compatibility con i sistemi più vecchi, le POSIX ACL non sostituiscono le vecchie ACL, infatti nell'inode troviamo sia le ACL ridotte (quelle standard), e le ACL estese vengono messe come attributi aggiuntivi del file (i file system moderni permettono infatti di avere per ogni file un blocco di estensione, e lì vengono indicate le ACL moderne). Questo è anche il motivo per cui non esistono syscall esclusive alle ACL moderne.

Le differenze fra le ACL minime (vecchie) e quelle estese (nuove) sono:

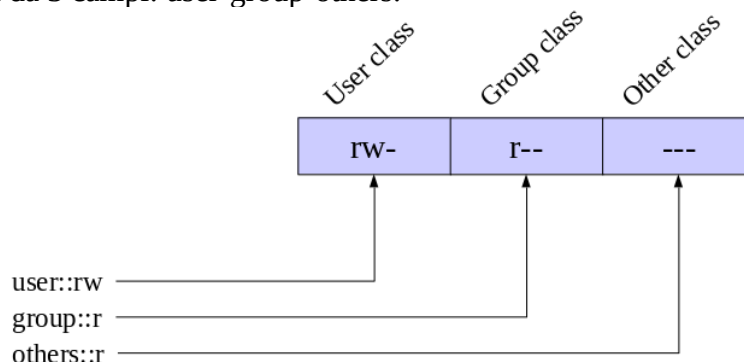
- **ACL minime**
 - Contengono solo tre entry.
 - Sono equivalenti ai bit di permesso di UNIX tradizionale.
- **ACL estese**
 - Hanno più di tre ACL entry.
 - Contengono una mask entry.
 - Possono contenere un qualunque (teoricamente, perché comunque l'area di memorizzazione è limitata) numero di named user e di named group entries.

Lezione del 23/04/2021 – Continuazione ACL estese (POSIX), DAC e MAC, namespace

Come funzionano le ACL minime e le estese?

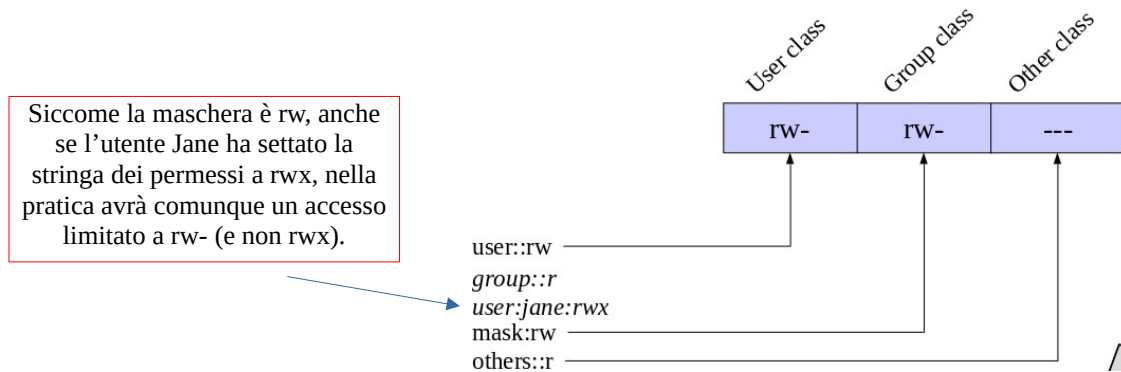
Le ACL minime in UNIX sono le classiche che abbiamo visto fino ad ora, ovvero la stringa dei permessi di cui parlavamo nel primo semestre.

La stringa è composta da 3 campi: user-group-others.



Nelle **ACL estese**, invece, aggiungono anche altri campi (quelli in corsivo nella foto qui sotto). Quindi, oltre alle informazioni presenti nelle ACL minime, possiamo anche trovare delle informazioni sui permessi di un utente o un gruppo specifico.

Quello che vediamo nella zona del gruppo, però, non rappresentano i permessi di un dato gruppo, bensì è la **maschera** (da non confondere con la umask). La maschera rappresenta il diritto di accesso massimo che si può dare con i permessi indicati nella stringa, quindi user specifici o gruppi non posso mai superare la maschera. In questo modo uno ha sotto occhio qual è il diritto massimo, ed è un ulteriore modo di salvaguardare l'autorizzazioe.



L'owning group, anche se la maschera è rw, avrà comunque solo permessi r (vale infatti **l'intersezione fra i permessi**). Anche quando si crea un file, si fa una intersezione fra i permessi della ACL e i permessi specificati nella creazione del file stesso.

Per implementare le ACL estese, si usano File Systems con Extended Attributes (ovvero, con una sezione estesa) e mettiamo le informazioni sulla ACL in questa sezione (come avevamo specificato prima).

[**DISCLAIMER**: questa parte non è molto chiara (come molte cose di sta lezione). Svolgo la procedura standard per quando non si capisce: scrivo le esatte parole dette da lui.]

Ma in realtà quello che succede è che, nei filesystem EA, c'è un puntatore un puntatore in più che punta ad un blocco, che se esiste, indica gli attributi estesi. E per gli attributi estesi che sono coppie <chiave, valore>, c'è una chiave che identifica le ACL estese, e sotto quella chiave vengono memorizzate in questo blocco le ACL estese.

A sua volta, anche all'interno del blocco delle ACL estese, vengono memorizzate come <nome, valore> dei singoli permessi dati.

[fine parte confusa]

In base ai file systems, si hanno più implementazioni diverse dello spazio dedicato alle EA (e quindi ACL). In ext2/ext3/ext4 per esempio si memorizzano le EA in un blocco separato (come stava cercando di spiegare il prof) puntato dall'inode, con possibilità di eseguire sharing del blocco. Il limite della grandezza di sto blocco è 1KB.

Questo fatto può dare complicazioni quando copiamo dei file da un file system di una certa architettura ad un file di un file system differente.

[Qui il prof ha anche parlato di quella volta che invitò Hans Reiser (creatore del ReiserFS) a cena, e che poche settimane dopo avevano scoperto che era un killer O_O]

Molte tool standard come ls, mv, cp etc... supportano tutti il POSIX ACL. scp però non le supporta ancora.

POSIX Capability

Le capability POSIX sono un modo per separare i privilegi che ha root. Nel modello classico di autorizzazione di UNIX c'erano utenti e root, e quest'ultimo era quello che poteva fare tutto nel sistema. Root inoltre ha diversi privilegi, per esempio fa boot, spegne la macchina...

In poche parole, root poteva accedere senza restrizioni a qualsiasi risorsa del sistema. Molti programmi, però, eseguono come root solo per ottenere alcuni di questi privilegi.

Le capabilities POSIX sono quindi un sistema per permettere di attribuire solo alcuni privilegi dell'utente root che possono essere utili a un processo in un dato momento.

Root può dare capability ai processi.

Ci sono quindi 3 insiemi di permessi:

- **Effective Set:** contiene le capability che il processo possiede ad un certo istante, ed è contenuto nel Permitted Set (detti anche permessi effettivi).
- **Permitted Set:** contiene il massimo insieme di capability che un processo possiede.
- **Inheritable Set:** contiene il sottoinsieme di capability che un processo può lasciare in eredità ai suoi sottoprocessi (detti anche permessi ereditabili).

[detto dal prof]

I primi (permessi effettivi) sono quelli che l'utente può usare, e che possono essere cambiati nei limiti dei permessi, i secondi (permessi ereditabili) invece sono quelli che uno può passare ai propri figli (cosa che avviene quando facciamo una exec(), ma non una fork() che invece mantiene le capabilities così come sono).

In poche parole, riusciamo grazie a questi ad avere una granulità più bassa dei permessi di root.

[end]

DAC & MAC

Il **Discretionary Access Control** (DAC) è il modello che di solito noi programmatori abbiamo in mente quando vogliamo creare un file, ovvero noi decidiamo quali permessi si possono dare a un certo oggetto. (I controlli di accesso sono basati sull'identità dei soggetti e sui permessi di accesso *assegnati* agli oggetti).

Questo però non è l'unico modello possibile: ci sono infatti organizzazioni in cui si hanno **modelli di accesso non discrezionali (Mandatory Access Control, MAC)**, in cui i permessi vengono attribuiti in base al ruolo dell'utente che ha creato tale file, e non dalla sua volontà. (Il controllo degli accessi è basato su regole e informazioni *associate ai soggetti* ed agli oggetti). Con questo sistema, il proprietario non ha privilegi speciali: le etichette vengono associate esternamente e non determinate dal proprietario.

Un esempio di Mandatory Access Control è il protocollo La Padula, che non è un protocollo informatico ma bensì un protocollo militare, secondo cui le informazioni si possono dividere in classificazioni di segretezza (Top Secret < Secret < Confidential < Unclassified). Secondo questa classificazione, inoltre, tutti possono scrivere documenti “verso il basso”, ovvero di classificazione inferiore a un certo valore, ma non possono leggerli (solo quelli da certo grado di segretezza in giù in base alla classificazione).

Namespace nei sistemi UNIX moderni

Le capabilities sono per le exec() quello che i namespace sono per le fork(), clone() o specifiche.

Nei sistemi UNIX, dopo che abbiamo fatto login, tutti i processi vedono l'intero FS, la stessa interfaccia di rete etc... invece con i **namespace si possono creare letteralmente degli spazi di nome, e quindi degli ambiti in cui per esempio il networking è diverso**; così, alcuni processi vedranno delle interfacce di rete virtuali (e diverse da quella attuale/reale) e possono connettersi a reti virtuali.

I namespace consentono di creare delle **sandbox**, ovvero degli spazi di confinamento in cui i processi possono essere confinati e possono svolgere solo alcuni tipi di accesso ai filesystem etc...

I browser fanno anche loro uso di namespace per creare delle aree protette, siccome eseguono codice javascript che potrebbe essere potenzialmente pericoloso.

Attraverso il comando **lsns**, possiamo vedere la lista dei namespace in esecuzione in un dato momento.

Lezione del 27/04/2021 – Ricevimento / esercizi

Possiamo trovare gli esercizi con la loro relativa risoluzione a [questo link](#).

Possiamo installare un modulo nel kernel Linux attraverso il comando `insmod [modulo]`, e per vedere la lista dei moduli installati usiamo il comando `lsmod`.

A questo punto tutte le lezioni che ci sono state dopo, sono state lezioni di correzione degli esercizi.

Grazie per aver letto questa guida, è il frutto di più di 500 ore di ascolto e riascolto delle lezioni.

A presto,

Angelo.