Sistemi operativi cap1

Un sistema operativo è un'interfaccia tra l'utente e l'hardware (livello intermedio). Il SO è quindi un software in grado di parlare con l'hardware e fa da tramite per gli utenti con il mondo sottostante. Per innescare il sistema operativo,gli utenti possono utilizzare servizi offerti sotto forma di **syscall**. Le syscall non le invochiamo direttamente ma ci appoggiamo ad una libreria,ad esempio libreria c,che ci dà una funzione già fatta e si farà carico scendendo di livelli di arrivare alla write. Per poter parlare con l'hardware,il SO ha dei privilegi ed esegue in una particolare modalità di esecuzione detta **kernel mode**,ovvero una modalità di esecuzione privilegiata in cui chi sta eseguendo può fare tutto (ad esempio dal livello utente non possiamo utilizzare direttamente istruzioni assembler di i/o). Al di sopra del livello delle **syscall** abbiamo le **interfacce utente**,cioè ciò che il SO ci mette a disposizione per poterne evocare i servizi e per poter "parlare" con l'hardware (prompt,GUI…). Al di sopra delle interfacce utente abbiamo poi i **programmi** che utilizziamo ogni giorno (web,email,music player);questi programmi eseguono in una modalità detta **user mode**,una modalità in cui i programmi non possono fare tutto e cioè non possono parlare direttamente con l'hardware (ad esempio possono scrivere nei registri general purpose del processore ma non possono effettuare direttamente la scrittura sul disco e per farla devono quindi invocare una syscall : si appoggia alla libreria c,fprintf/printf e così via).

Il **kernel (nucleo)** è un software mediatore tra applicazioni ed hardware,è la parte del SO che risiede in memoria principale,esegue principalmente "on-demand" (il codice del SO in memoria entra in funzione quando l'utente chiede qualcosa o quando ci sono interrupt da parte dell'hardware) e risponde a syscall,eccezioni ed interrupt.
Il **SO in senso ampio** è invece costituito dal kernel,dal **sistema di base** (consente al SO di avviarsi e di presentare un'interfaccia all'utente),da **programmi di utilità** (task manager,deframmentazione) e da **librerie di sistema**. I software di base (GUI,prompt,user interface…) fanno parte del SO mentre i **programmi applicativi** (compilatori,browser,client email ecc) non sono sistemi operativi.

La **user mode (privilegi ridotti)** è disponibile solo per un sottoinsieme di istruzioni macchina,per esempio non può eseguire istruzioni assembly per i/o le quali devono essere richieste al SO.
La **kernel (supervisor) mode** ha accesso a tutto l'hardware e può eseguire qualsiasi istruzione della macchina. I programmi in user mode richiedono l'esecuzione di operazioni privilegiate al **kernel** tramite syscall (se dobbiamo fare una stampa a video si passa il controllo al kernel il quale parlerà con l'hw ed eseguirà una serie di operazioni).

Un sistema operativo fornisce un **insieme di astrazioni**,cioè visioni estremamente semplificate di interfacce complesse. Le astrazioni si costruiscono tipicamente con funzioni di libreria. Un esempio è l'astrazione del file invece di byte memorizzati sul disco. Le astrazioni permettono quindi di usare disco,i/o,rete ecc. in maniera semplificata. Tra gli esempi di astrazione abbiamo:

- file/file system → memoria secondaria;
- spazio di indirizzamento → memoria centrale;
- processi/thread → esecuzione programmi;
- socket → rete.

Un sistema operativo è anche un **gestore delle risorse** : si occupa della gestione delle risorse hardware e dei criteri con cui assegnare una risorsa. Ad esempio se abbiamo un solo processore,se navigo su rete,ascolto musica,scrivo su word ecc. in un certo istante solo uno di quei processi sarà effettivamente in esecuzione e il SO sta operando una **condivisione delle risorse (multiplexing)** la quale può essere nel **tempo** (il SO dovrà decidere come alternare i processi sul singolo processore) e nello **spazio** (in memoria centrale vengono caricate più immagini appartenenti a processi diversi).
Se abbiamo un solo processore e su di esso sta girando un programma utente,il SO non può girare essendo esso stesso un programma,in particolare un programma di controllo che fornisce astrazioni e funziona on demand. Il SO stesso utilizza le stesse risorse destinate ai programmi applicativi e quindi le deve condividere con essi. Tra le principali funzioni di un SO abbiamo:

- fornire una user interface (UI) : prompt,GUI;
- program execution : caricamento in memoria ed esecuzione;
- operazioni i/o;
- file system e file;
- comunicazione;
- rilevamento errori;
- allocazione delle risorse;
- protezione e sicurezza.

Il primo sistema operativo compare nella metà degli anni 50/60 (seconda generazione) ed è chiamato **FMS,IBSYS** cioè un minimo di logica di controllo dei job che dovevano essere alternati in esecuzione sulla macchina.

Il SO **IBM OS/360** era costituito da milioni di righe di codice ed era pensato per funzionare su tutti i modelli IBM di quel periodo. Il progetto durò un certo numero di anni e dal 1963-1966 richiese 5000 anni uomo di sviluppo. Ciò portò ad un SO pieno di bug con un costo di 4 volte più grande rispetto a quello iniziale.

Monoprogrammazione vs multiprogrammazione

In un sistema **monoprogrammato** la CPU si pone in stato “inattivo” fino a che l’ i/o termina.
In un sistema **multiprogrammato** ci sono più job caricati in memoria centrale : mentre un job attende il completamento dell’i/o,un altro job può usufruire della CPU. In questo modo si massimizza l’utilizzazione della CPU,si crea un’alternanza per portare avanti più job contemporaneamente non essendo più vincolati al più lento di tutti.

Timesharing (multitasking)

Il **timesharing** è una variante della multiprogrammazione dove ad ogni job viene assegnato un **quanto** di tempo del processore e a turno ogni job sfrutta la CPU per il quanto di tempo ad esso assegnato. Le motivazioni sono : gestione di un gruppo di task interattivi,più utenti che accedono simultaneamente al sistema da più terminali e tempo del processore condiviso tra più utenti. Ogni volta che un programma va in esecuzione c’è il rischio che esso monopolizzi il processore e che quindi il SO non esegua mai. In un esempio con due programmi abbiamo un programma 1 in esecuzione che potrebbe monopolizzare il processore. Il timesharing prevede un “meccanismo di salvaguardia”,il quale qualsiasi cosa ci sia sul processore prevede un interrupt hardware periodico detto **timer interrupt**,il quale viene sollevato ogni tot di tempo per fare in modo che il SO riesca a mandare fuori chi sta sul processore (altrimenti si rischia di non poter buttare fuori chi sta occupando la CPU a meno che non si effettui una syscall). Una volta sollevato il timer interrupt,entra in gioco il SO che effettua una operazione di **scheduling**,cioè si attiva una porzione del kernel che vede quali sono i processi in esecuzione e in base a certi criteri decide chi mandare in esecuzione. In questo caso viene lanciato il programma 2. Anche il programma 2 potrebbe monopolizzare il processore ma c’è sempre il timer interrupt che evita questa situazione.
Ciò presuppone due cose : il processo in esecuzione non si accorge che gli è stato tolto il processore e che viene rimesso in esecuzione. Quindi ad un certo punto viene fatta una fotografia dello stato dei registri del processore che deve essere salvata in qualche parte per poter poi riprendere l’esecuzione del programma che era stato liberato dal processore.

MULTICS è un sistema timesharing che non ebbe un grandissimo successo. Ad avere successo fu **UNIX**,versione ridotta e monoutente di MULTIX. **Linux** è un clone di UNIX. MacOS deriva da una versione di UNIX e Android è uno strato software “sopra” Linux.

Struttura di un SO

Un SO è un programma di notevole complessità e dimensione. E' fondamentale applicare durante il suo progetto e realizzazione le tecniche proprie dell'ingegneria del software,al fine di garantire : correttezza,modularità,facilità di manutenzione ecc.

Una prima forma di SO è un **sistema monolitico o macrokernel** : l'intero SO è un singolo programma che gira in kernel mode (si tratta di un unico file binario in cui c'è tutto). Tra i vantaggi di questo sistema c'è la velocità di esecuzione mentre tra gli svantaggi abbiamo le dimensioni del file binario,la robustezza e non avere il completo controllo su ciò che sta accadendo realmente. I primi SO monolitici erano effettivamente costituiti da un solo programma senza particolari suddivisioni. Un SO monolitico può avere però anche un minimo di struttura tramite **progettazione modulare** : non abbiamo un unico file binario che contiene tutto ma il SO è organizzato in moduli,ciascuno destinato ad offrire una delle funzionalità del sistema (Es di questi SO sono kernel unix-like).
Un'altra visione di questi sistemi monolitici è quella a **livelli** dove quello più basso è quello vicino all'hardware e quello più alto è quello utente. Il SO è modulare con i moduli organizzati in una struttura gerarchica (livelli). I livelli non possono essere invocati arbitrariamente ma le interazioni possono avvenire solo tra livelli adiacenti (es. THE).
Un kernel monolitico richiede che tutto sia caricato in memoria e che tutto sia eseguito in kernel mode. Non necessariamente però tutto ciò che serve deve essere presente già fisicamente nel kernel e caricato in memoria. In Linux e altri SO c'è infatti il concetto di **modulo caricabile**,ovvero file oggetto contenente funzionalità del kernel non indispensabili per l'inizializzazione delle periferiche. Questi moduli sono attivabili/disattivabili on-demand a tempo di esecuzione (a mano o automaticamente) e rappresentano un modo per ridurre la dimensione del kernel ed aumentare la robustezza. In Linux,su prompt,con **lsmod** vengono elencati i moduli caricati e con **insmod/rmmod** possiamo caricare o rimuovere un modulo.

L'altra forma di SO è il **microkernel**. Un microkernel implementa solo i meccanismi assolutamente necessari : gestore interrupt,scheduling processi,comunicazione tra processi ; le restanti funzionalità vengono implementate all'esterno del kernel,da processi di sistema (in **user mode**). Quindi in un microkernel una parte viene eseguita in kernel mode (meccanismi assolutamente necessari) e una parte in user mode. Tra i vantaggi di questo sistema abbiamo che il SO occupa meno spazio in memoria,una facilità di estensione,robustezza ai guasti (se crasha un servizio non viene compromesso il kernel. Tra i servizi in user mode abbiamo il servizio Reincarnation server che riavvia i servizi che crashano e non compromettono quindi il kernel). Tra gli svantaggi abbiamo una minore efficienza : mentre nel macrokernel tutto funziona tramite invocazione di funzioni,in un microkernel le unità separate devono comunicare tramite scambio di messaggi,cosa molto più pesante rispetto all'invocazione di funzioni. Esempi di SO microkernel sono Minix,Mach.

I SO più comuni come windows,OS X adottano una soluzione intermedia detta **hybrid kernel**,una via di mezzo tra macro e microkernel. In questa soluzione i driver fanno parte del kernel mode (a differenza del microkernel) e gli altri servizi sono ancora in user mode.

Richiami hardware e alcuni meccanismi fondamentali

All'interno del **processore (CPU)** abbiamo registri general purpose e registri speciali (PC,SP,PSW…). I registri svolgono una funzione di memorizzazione e il tempo di accesso è pari ad una frazione di clock. Il grosso delle istruzioni sono fatte su registri: RISC vs CISC (Reduced instruction set e Complex instruction set; nel RISC bisogna effettuare la load del dato nei registri e poi c'è la add registro registro mentre nel CISC un operando è un operando memoria).
Il ciclo di esecuzione di un'istruzione è fetch,decode,execute. Abbiamo un **modello sequenziale** dove una fase non inizia se la precedente non è terminata : ad esempio con 3 istruzioni,dopo che la EX della I1 è terminata può iniziare l'IF della I2 e così via. Il modello utilizzato dai processori moderni è quello del **pipeline**,che prevede la sovrapposizione temporale dell'esecuzione delle istruzioni aumentando il throughput. I processori **superscalari**,invece,replicano unità funzionali per fare in modo che le istruzioni pronte ad essere eseguite vengano subito eseguite mentre quelle con dipendenze vengono messe in attesa fino a quando il dato che attendono non viene prodotto.
La **memoria principale (RAM)** è critica in un calcolatore. La RAM ha dei tempi di accesso di 10-50 nsec e una capacità tra 16/64 GB. Tra i registri (livello più basso) e la RAM abbiamo la memoria **cache** la quale ha un tempo di accesso di 1-8 nsec e serve a contenere un sottoinsieme degli elementi della memoria principale (quelli utilizzati più frequentemente). Abbiamo infine la memoria secondaria costituita dal **disco rigido** con un tempo di accesso di 10 msec e una capacità di memoria molto grande. Abbiamo poi processori **multicore** (chip con più processori) e cache **multilivello** con una cache di livello L1 integrata nel core,una cache di livello L2 condivisa e in altri schemi (intel core i7) abbiamo una cache di livello L1 e L2 nel core e una cache di livello L3 condivisa. L'**hyperthreading** consiste nel prendere il singolo core e andare a replicarne alcune parti (senza creare un core a sé stante). Questa tecnica viene utilizzata quando un processo deve essere tolto dalla CPU e in particolare viene replicata una parte dei registri in modo tale che anziché svuotare i registri e ricaricarli con valori nuovi,quelli del processo precedente si lasciano popolati e si spostano sull'altro thread. Le **GPU** sono processori specializzati su operazioni parallelizzabili (array e matrici).

In generale un programma in esecuzione è pensato rispetto ad uno **spazio di indirizzamento virtuale** che non coincide con quello **fisico**. La **MMU (memory management unit)** si occupa di tradurre gli indirizzi virtuali in indirizzi fisici.

Le periferiche sono organizzate con due componenti: un **controller** e la **periferica** vera e propria. Il **controller** è ciò che fisicamente parla con il **bus**. Il controller espone sul bus una serie di registri come quelli utilizzati per i dati (data_in e data_out),registro di stato e registro di controllo. Questo insieme di registri è detto **porta i/o** ed espone il controller sul bus (in base ai valori dei registri ci saranno operazioni diverse da compiere).
Supponiamo che un utente voglia scrivere sul disco. Viene invocata una funzione di libreria e scendendo di livello si esegue una syscall. Si passa quindi in kernel mode e in particolare gli oggetti/entità coinvolte nel comunicare con le periferiche sono detti **driver** i quali vengono invocati dal kernel. Il driver inizia l'i/o e cioè inizia a comunicare con i registri (invia dati e segnali di controllo). Il codice del driver in esecuzione sul processore accede ai registri di i/o in due modi:

- **port mapped** i/o : per comunicare con i registri i/o ci si rivolge a specifiche istruzioni assembler *in/out* specificando fisicamente il registro del controller. Queste istruzioni non possono essere eseguite in user mode;
- **memory mapped** i/o : i registri del controller vengono mappati nello spazio di indirizzamento del SO e trattarli quindi come se fossero locazioni di memoria vere e proprie. Si può quindi leggere o scrivere utilizzando le istruzioni classiche assembly come *mv*.

Una volta che il dato è arrivato nel registro i/o il driver deve sapere quando la scrittura nella periferica è terminata. Ci sono tre strategie:

- **busy waiting** : verifica periodica dello stato del controller (verifica se il controller ha terminato);
- **interrupt** : il controller notifica al processore quando ha terminato;
- **direct memory access (DMA)** : un chip diverso che sta sul bus e che viene configurato all'inizio dal driver e il trasferimento dati avviene direttamente tra il chip DMA e il controller. Una volta terminata l'operazione il chip DMA manda un interrupt al controller per avvisarlo.

Meccanismi fondamentali

Il SO può essere innescato tramite :

1. **System call** : richiesta "deliberata" di un servizio al SO,effettuata volutamente da un programma;
2. **Eccezione** : evento (tipicamente non deliberato) generato durante l'esecuzione di un'istruzione (ad es. divisione per zero,errore di indirizzamento);
3. **Interrupt hardware** : eventi hardware,non causati dal programma in esecuzione (per es. dispositivi I/O).

Esiste poi un'altra definizione,quella di **trap**,che può essere definita come una qualsiasi forma di trasferimento al controllo al SO,un evento causato dal programma in esecuzione,un'istruzione per passare da user a kernel mode (syscall) o per interruzioni software. Esistono quindi varie interpretazioni per la trap.
Inoltre,si definiscono **eventi asincroni** gli eventi non causati dall'istruzione in esecuzione ma causati da una sorgente esterna (ad es. dispositivi I/O).
Si definiscono **eventi sincroni** gli eventi causati dall'istruzione in esecuzione,ad esempio system call ed eccezioni rilevate dal processore.

Interrupt request

1) richiesta i/o : si richiede ad esempio di scrivere/leggere sul disco;
2) il controller segnala ad un particolare dispositivo detto **interrupt controller** di aver terminato l'operazione i/o per cui era stato chiamato in causa. L'interrupt controller è rappresentato come un oggetto che partecipa sul bus. Nei primi computer era un chip a parte mentre oggigiorno è integrato nel **southbridge chipset** della scheda madre. Il disco tramite il controller segnala al controller degli interrupt la necessità di "voler essere preso in considerazione".
3) L'interrupt controller,a sua volta,segnala la cosa al processore attivando un pin nella CPU. Il processore a questo punto sa che qualcuno vuole attenzione (il disco ha finito la scrittura e vuole farlo capire al processore). Il processore deve conoscere anche chi è che richiede attenzione e quindi c'è un ulteriore passaggio.
4) L'interrupt controller mette sul bus un numero (**identificativo del dispositivo**) detto **vettore di interrupt (interrupt vector)**.

Il ciclo del processore verifica la presenza di interruzioni dopo la EX. Se c'è un'interruzione parte una **ISR (Interrupt service routine)**.

Interrupt handling

1. La CPU salva il PC utente (ed eventuali registri "critici" come il PSW) sul kernel stack;
2. La CPU carica il PC con l'indirizzo dell'ISR (dalla **tabella dei vettori degli interrupt**). Ad esempio il kernel linux si poggia sull'**IDT** (interrupt descriptor table). Il vettore degli interrupt che era stato letto prima sul bus entra quindi ora in gioco e indicherà quale funzione far partire per gestire l'interrupt. L'IDT è costituito da 256 entries e il vettore di interrupt identifica una di queste locazioni. Non tutti i vettori di interrupt sono cause di hw interrupt : i vettori **0-31** sono dedicati alle eccezioni , il **128** è dedicato alle syscall e tutto il resto (**32-127** e **129-255**) sono vettori di interrupt dedicati all'hw. Ogni locazione contiene un descrittore (8 byte) e da esso si costituisce l'**indirizzo dell'ISR**.
3. Salvataggio dei registri (entry della tabella dei processi del processo attuale). Ciò viene fatto perché una volta che parte la ISR i registri verranno sporcati e

si dovrà poi riprendere l'esecuzione del processo nello stato in cui si trovava quando è stato interrotto.
4. Impostazione di un nuovo stack a livello del kernel su cui viene gestita l'interruzione. Questo è uno stack **opzionale**,dato che si potrebbe continuare ad utilizzare lo stack precedente.
5. Esecuzione dell'ISR.
6. Terminata la ISR,lo scheduler (una routine software) decide quale processo eseguire come successivo. Il processo successivo può coincidere con il processo precedente ma non necessariamente.
7. La procedura in C ritorna al codice assembly (non un "vero" step).
8. Caricamento dei registri (entry della tabella dei processi del nuovo processo attuale).
9. Ripristino esecuzione dopo l'interrupt. Ad esempio l'istruzione x86 **iret** (return from interrupt) restituisce il controllo all'user space.

I passi 1-2 sono hardware. C'è poi un frammento di codice scritto in assembler che salta nel process control block. La ISR è invece scritta in C. Quindi il passo 7 serve per poter saltare al process control block.

System call

Come sappiamo le syscall servono ad innescare il SO. Sono un'interfaccia che il SO offre ai programmi applicativi. Le syscall sono tipicamente offerte tramite librerie,ad esempio una funzione di libreria per ogni syscall. Gruppi di syscall formano **servizi** : file system service,process management service ecc.
Un esempio di invocazione tramite libreria può essere il seguente:

- opzione **1** : syscall (**1**,1,msg,sizeof(msg));
- opzione **2** : write (1,msg,sizeof(msg)).

Una syscall prevede **due sezioni** : **identificativo (syscall number)** e **altri parametri** (fino ad un max di 6) come il descrittore (numero che indica dove scrivere in caso di una scrittura) , il buffer (l'array di byte che contiene il dato effettivo) e il numero di byte. I parametri di una syscall sono passati **tramite registri**. La scelta dei registri non è arbitraria ma c'è una convenzione per le syscall : il primo parametro della syscall (identificativo) deve andare al registro **rax** in modo tale che quando il SO viene innestato andrà a leggere nel registro rax per trovare l'identificativo della syscall ; ci sono poi altri registri che servono a contenere gli altri parametri della syscall (questi registri sono ad esempio *rdi,rsi,rdx,r10,r8,r9* e sono come già detto per un max di 6). Se la syscall ha più di 6 parametri aggiuntivi,bisognerà utilizzare uno dei registri come puntatore ad una struttura dati che contiene gli altri parametri.

Come avviene una syscall

Una prima operazione preliminare che avviene in user mode,in seguito all'invocazione della libreria C,prevede il caricamento dei parametri nei registri con la

convenzione vista precedentemente (passi 1 a 5). Una volta popolati i registri si esegue un'istruzione ***trap*** per entrare in **kernel space** (passo 6). Questa trap può essere realizzata in due modi:

- fino alle macchine a 32 bit,veniva sollevata una vera e propria **interruzione software** (la 128). Quindi dopo aver scritto nei registri,come ultima istruzione assembler c'era *int $0x80* (l'interruzione viene generata volontariamente);
- oggigiorno sulle macchine a 64 bit,troviamo un'istruzione assembly ***syscall*** che permette di transitare in kernel space.

Una volta entrati in kernel mode,si individua e si esegue il corretto **gestore** della chiamata di sistema: si esamina in particolare il contenuto del registro **rax**; ci sarà poi un'altra tabella che per ogni identificativo di system call ha un puntatore all'implementazione della system call (**syscall handler** passi 7-8) e si inizia così ad eseguire il codice della system call. Una volta terminata l'esecuzione il controllo può (o non) essere restituito alla libreria C che era stata invocata (passo 9) e poi il controllo ritorna al programma utente (passo 10). I passi 9 e 10 non è detto che avvengano necessariamente al termine della system call → la syscall potrebbe bloccare il chiamante ad esempio per operazioni i/o. In questo caso se la syscall ha bloccato un processo il processore deve rimanere attivo e quindi si effettuerà un **context switch** mandando in esecuzione un processo che ha le carte in regola per poter essere eseguito al posto di quello bloccato.

Eccezioni

Ogni volta che viene sollevata un'eccezione il SO deve intervenire. Esso può fare due cose:

- tradurre l'eccezione in un **segnale** verso il processo. Ad ogni eccezione corrisponde quindi uno specifico segnale. Ad esempio SIGFPE (fatal arithmetic exception) , SIGILL (illegal instruction) … I segnali servono a dare "una seconda chance" ad un processo. Abbiamo due possibilità: la prima è scrivere una funzione che viene invocata nel momento in cui si verifica un'eccezione (per poter fare ciò il main program deve registrare l'handler : ad esempio se si potrebbe verificare un'eccezione aritmetica nel main dobbiamo invocare la funzione *signal(SIGFPE,handler)* dove SIGFPE è il segnale ed handler è la funzione scritta dal programmatore).
- il processo in esecuzione viene **ucciso** dal SO. Un'eccezione di questo caso è il **page fault**

Avvio del computer

Il **BIOS** è il Basic Input Output System ed è un firmware di sistema (mini programma che si trova in una memoria flash della scheda madre). Il BIOS è oggi sostituito da UEFI (Unified Extensible Firmware Interface). Appena accendiamo il computer avvengono due fasi:

prima fase:

- la CPU legge da un indirizzo mappato sulla flash dove si trova il BIOS;
- si esegue il codice del BIOS;
- il codice del BIOS rileva e inizializza le risorse (RAM,controller interrupt,bus PCI);
- set up firmware per alcuni servizi critici (low-level i/o);

seconda fase:

- il BIOS determina il dispositivo di avvio (da dove leggere per caricare il SO) e ne legge/esegue il primo settore,detto Master Boot Record (**MBR**). Il MBR contiene un programma che determina la "partizione attiva" (un'unità può essere suddivisa in partizioni e una di esse deve essere designata come attiva) e legge un **secondo bootloader**;
- viene letto/avviato il SO dalla partizione;
- si recupera la configurazione HW dal BIOS e se necessario si installano nuovi driver;
- i driver vengono caricati nel kernel,si creano strutture dati,si avviano processi background,compare la **presentazione di login** o della **GUI**.

—---

Processo

Un processo è un'astrazione di **programma in esecuzione**. Ad un processo sono associati (per poter realizzare questa astrazione) :

- spazio degli indirizzi e thread di controllo: ad ogni processo è associato un suo **spazio degli indirizzi**,ovvero un elenco di locazioni di memoria da 0 ad un massimo,che il processo può leggere e scrivere. Lo spazio degli indirizzi è costituito dal segmento stack (comprende variabili automatiche e parametri di funzioni),segmento heap (variabili dinamiche),segmento data (variabili globali)

e segmento text (codice eseguibile). Il segmento data cresce verso l'alto e lo stack verso il basso. In mezzo c'è uno spazio inutilizzato. Lo stack cresce in questo spazio in modo automatico secondo le necessità mentre l'area data aumenta in modo esplicito usando la chiamata di sistema *brk*,la quale specifica il nuovo indirizzo dove il segmento data deve finire (non definita in POSIX e quindi si usa *malloc*). Il PC punterà all'area text mentre lo SP punterà al primo elemento dello stack.;

- risorse : un esempio sono i **file**. Ogni volta che un processo viene avviato ha sempre 3 "file" di default,il file 0 (stdin),il file 1 (stdout) e il file 2 (stderr). Ogni volta che vengono creati nuovi file avremo poi descrittori diversi. Sono visti come file anche le sockets,dispositivi ecc.
- strutture dati usate dal kernel per rappresentarlo e gestirlo (il **PCB**).

Supponiamo ci siano tre processi attivi durante l'attività di un utente. Il SO,periodicamente,decide di fermare un processo e avviare l'altro,per esempio perché il primo ha terminato il tempo sulla CPU. Quando un processo viene temporaneamente sospeso in questo modo,deve essere riavviato nello stesso stato di quando è stato fermato e quindi tutte le informazioni relative al processo devono essere salvate in modo esplicito durante la sospensione. In molti SO tutta l'informazione riguardante ciascun processo (a parte i contenuti dello spazio degli indirizzi) è salvata all'interno della **tabella dei processi**,costituita dai diversi **PCB**.
Il **Process Control Block (PCB)** è una struttura dati e ne esiste uno per ogni processo. Il PCB contiene tutta l'informazione riguardante il processo (tranne i contenuti dello spazio degli indirizzi). Un processo (sospeso) consiste quindi del suo spazio degli indirizzi e della sua voce nella tabella dei processi,insieme al contenuto dei suoi registri. Ad esempio nella tabella abbiamo il Process ID (identificativo del processo),Processo Padre (ogni processo per essere creato necessita che un altro processo lo crei),Stato del processo (indica se il processo è in esecuzione,in attesa o è blocked),per la gestione della memoria ci indica dove inizia e finisce l'aria text,l'area data e lo stack ecc.

Il meccanismo alla base della **multiprogrammazione**,come già detto,è quello di avere in memoria centrale più processi caricati per permettere ad un altro processo di essere mandato in esecuzione (tramite context switch) se un altro processo è bloccato. Il program counter è unico e la CPU esegue 1 processo alla volta e "passa" avanti/indietro tra i processi : per i sistemi **monoprocessore** c'è l'illusione di parallelismo,ossia **pseudo-parallelismo (concorrenza)** dove alterno due thread di esecuzione (significa che si sovrappongono due attività nel tempo: riesco a far partire un processo prima che sia concluso un altro processo) ; per i sistemi **multiprocessore** abbiamo invece un vero e proprio **parallelismo** dove ogni processore esegue un solo thread ma questi vengono portati avanti contemporaneamente sugli n processori (quindi anche con n processori ogni CPU eseguirà comunque un processo alla volta).

I processi sono **indipendenti** : ogni processo ha un proprio spazio degli indirizzi,per ogni processo avanza il suo thread di esecuzione e i processi sono indipendenti tra di essi e ciò vuol dire che l'esecuzione di un processo non impatta su un altro. In alcuni casi potrebbe essere necessario far **interagire** più processi : ad esempio un processo ha necessità di un output di un altro processo. In questo caso sono necessari dei meccanismi espliciti come la memoria condivisa per scambiare dati. In generale,il SO non offre garanzie temporali o di ordinamento (non garantisce che gli n processi siano sempre eseguiti nello stesso ordine).

Da un punto di vista "comportamentale" i processi o utilizzano CPU o sono in attesa di un evento come ad esempio il sollevamento di un interrupt dal disco al completamento di una scrittura. I rettangoli indicano l'utilizzo della CPU e la linea l'attesa dell'evento. I processi potrebbero appartenere a due categorie principali:

- Processi **CPU bound** : utilizzano molta CPU e fanno poco i/o. Sono caratterizzati da rettangoli molto lunghi e si pongono in attesa di eventi solo per poco tempo.
- Processi **I/O bound** : sfruttano fasi di utilizzo del processore molto brevi (dette anche **burst**) e per il grosso del tempo sono in attesa di eventi (come i/o).

Creazione di un processo

Un processo,per essere creato,deve essere creato da un altro processo. Avremo quindi un processo in esecuzione che esegue una specifica **chiamata di sistema** per la creazione di un processo (in UNIX **fork**). La fork crea un duplicato esatto del processo originale,includendo tutti i descrittori dei file,registri ecc. In questo modo si crea una relazione tra **processo padre** (che ha effettuato la fork) e **processo figlio** (creato dal padre) che prenderanno vie separate. Tutte le variabili hanno valori identici al momento della chiamata a *fork* , ma poiché i dati del genitore sono copiati per creare un figlio,i cambiamenti successivi in uno dei due non influiscono sull'altro. Infatti,la memoria del figlio può essere condivisa **copy on write** con il genitore e cioè il genitore e il figlio condividono una singola copia fisica della memoria fino a quando uno dei due non modifica un valore in una posizione di memoria. A quel punto il SO crea una copia della piccola parte di memoria contenente la posizione e riduce così al minimo la quantità di memoria da copiare a priori,visto che gran parte di essa può restare condivisa. In più parte del programma non cambia (es area text) e quindi può essere sempre condiviso tra genitore e figlio. La *fork* restituisce un valore che nel figlio è zero e nel genitore è uguale al **PID** del figlio (Process IDentifier). Tramite il PID restituito,i due processi possono vedere quale sia il genitore e quale il figlio. Nella maggior parte dei casi,dopo una *fork*,il figlio dovrà eseguire un codice diverso da quello del padre. All'avvio del SO una specifica fase consiste nel creare un particolare processo,padre di tutti i processi,che viene creato dal SO ed è chiamato **init** (**systemd** in Linux). I processi vengono creati durante l'avvio del SO,dopo l'avvio

del SO (dai processi in esecuzione),su richiesta/azione dell'utente (doppio click). Ogni volta che si avvia il SO vengono infatti generati diversi processi. Alcuni di essi sono processi attivi,che interagiscono con gli utenti mentre altri sono in background,non associati ad utenti ma che svolgono funzioni specifiche. I processi che sono in background per gestire attività (come email) sono detti **demoni**. In UNIX per elencare i processi in esecuzione può essere utilizzato il comando **pstree**.

Chiusura di un processo

Una volta creato,un processo inizia a girare ed esegue quello che è il suo compito. In ogni caso,esso non dura per sempre e potrà terminare per una delle seguenti cause:

- **uscita normale** : la maggior parte dei processi termina perché ha completato il proprio lavoro. Quando un compilatore ha compilato il programma che gli è stato dato,esegue una chiamata di sistema che indica al SO che ha finito. Questa chiamata in UNIX è detta ***exit***
- **errore critico** : dovuto o ad un input mancante o errato (ad esempio quando si prova a compilare un .c ma il file non esiste);
- **uscita su errore** : si tratta di un errore causato dal processo,spesso dovuto ad un difetto del programma. Alcuni esempi possono essere un'istruzione non valida,il riferimento ad una cella di memoria non esistente o la divisione per 0;
- **terminazione da un altro processo** : il processo esegue una chiamata indicando al SO di chiuderne altri. In UNIX questa chiamata è la ***kill***. Il "killer" deve avere l'autorizzazione necessaria per terminare il processo.

Stati di un processo

Ogni processo è un'entità indipendente,con il proprio PC e il proprio stato interno,ma i processi hanno spesso bisogno di interagire con altri processi. Un processo può generare output che un altro processo utilizza come input. Un processo può trovarsi in tre stati :

1. **Running (in esecuzione)** : il processo sta effettivamente utilizzando la CPU;
2. **Ready (pronto)** : il processo può essere eseguito,ma è temporaneamente sospeso per consentire ad un altro processo di essere eseguito (CPU non disponibile);
3. **Blocked (bloccato)** : incapace di proseguire finché non avviene un evento esterno.

Da un punto di vista logico i primi due stati sono simili. In entrambi casi il processo è determinato a continuare e solo nel secondo caso la CPU è temporaneamente non disponibile. Il terzo stato è diverso dai primi due perché il processo non può essere

eseguito,anche nel caso in cui la CPU è libera. Fra questi tre stati ci sono quattro transizioni.
La **prima** transizione è quella **running→blocked**. Questa si ha quando il SO scopre che un processo non può continuare subito. In alcuni sistemi il processo può eseguire una chiamata di sistema,come *pause*,per entrare in blocked. In altri sistemi,incluso UNIX,quando un processo legge da una pipe o da un file speciale (ad esempio un terminale) e non vi è input disponibile,il processo va in blocked in modo automatico.
La **seconda** transizione è quella **running→ready** ed è seguita dalla **terza ready→running**. Queste due transizioni sono causate dallo **scheduler** dei processi,una parte del SO,senza che il processo nemmeno lo sappia. La seconda transizione avviene quando un processo rilascia volontariamente la CPU (**scheduling non preemptive**) invocando una specifica system call (**sched_yield**) oppure quando scade il quanto di tempo ad esso assegnato (**scheduling preemptive**). Nel caso di scheduling preemptive è necessario un **timer interrupt** che consenta al SO e,quindi allo scheduler,di intervenire. Gli scheduler moderni sono preemptive e,inoltre,nel caso non preemptive durante un timer interrupt non sono prese decisioni di scheduling. La terza transizione si verifica invece quando tutti gli altri processi hanno avuto il loro equo intervallo di tempo ed è ora per il primo processo di riprendersi la CPU per continuare l'esecuzione.
La **quarta** transizione è quella **blocked→ready** e avviene quando si verifica l'evento esterno di cui un processo era in attesa (come l'arrivo di un input). Se nessun altro processo è in esecuzione in quel momento,sarà innescata la transizione 3 e il processo inizierà ad essere eseguito. Altrimenti potrebbe attendere in stato ready per un attimo fino a quando la CPU sarà disponibile e arrivi il suo turno.

SO cap2

Threads

Un **thread** è il flusso di controllo sequenziale di un processo. Nei sistemi operativi tradizionali,ogni processo dispone di uno spazio degli indirizzi e di un singolo thread di controllo. Ci sono poi frequenti situazioni in cui è utile avere più thread di controllo in esecuzione nello stesso spazio degli indirizzi,quasi in parallelo,come se fossero processi (quasi) separati. I thread vengono utilizzati per vari motivi. Un primo motivo è che in molte applicazioni ci sono più attività contemporanee e a volte alcune possono bloccarsi. Suddividendo una di esse in diversi thread sequenziali eseguiti quasi in parallelo,il modello di programmazione diventa più semplice. Un altro motivo è quello che i thread,essendo più leggeri dei processi,sono più facili da creare e da distruggere. Un ultimo motivo è dato dalle prestazioni. I thread non producono un guadagno di prestazioni quando sono tutti diretti alla CPU,ma quando c'è un'attività di elaborazione considerevole unita a un i/o considerevole,i thread permettono alle due attività di sovrapporsi e l'applicazione risulta più veloce.

Modello del thread classico

Il modello a processi si basa su due concetti indipendenti : raggruppamento di risorse ed esecuzione. A volte è utile separarli,ed entrano così in gioco i thread. Mentre ad un processo sono associati lo spazio degli indirizzi contenente area text,data e altre risorse,ad un thread è associato un PC che tiene traccia di quale istruzione eseguire come successiva,dei registri,uno stack per contenere la storia della sua esecuzione e un proprio stato. Nonostante un thread debba essere eseguito in un processo,il thread e il relativo processo sono concetti diversi e possono essere trattati separatamente. I processi sono usati per raggruppare risorse,i thread sono entità schedulate per l'esecuzione della CPU.
Il valore aggiunto dei thread è quello di consentire molteplici esecuzioni che hanno luogo nello stesso ambiente (e spazio indirizzi) del processo,con un grado di indipendenza l'una dall'altra. Avere più thread eseguiti in parallelo in un processo è come avere più processi in parallelo su un computer. Nel primo caso essi condividono spazio indirizzi e altre risorse mentre nel secondo memoria fisica,dischi,stampanti e risorse. Dato che i thread condividono alcune proprietà dei processi,sono anche chiamati **processi leggeri (lightweight process)**. Quando un processo multithread è eseguito su una sola CPU,i thread sono eseguiti a turno. La CPU passa rapidamente avanti e indietro tra i thread,dando l'illusione che siano eseguiti in parallelo,anche se la CPU è più lenta. I diversi thread nello stesso processo non sono indipendenti quanto processi diversi. Tutti i thread hanno esattamente lo stesso spazio degli indirizzi e ciò vuol dire che condividono anche le stesse variabili globali. Dato che ogni thread può accedere a tutto l'indirizzo di memoria all'interno dello spazio degli indirizzi del processo,un thread può leggere,scrivere o cancellare lo stack di un altro thread. Non c'è quindi **protezione** tra i thread perché è impossibile e non dovrebbe essere necessario. Processi diversi potrebbero essere di utenti differenti e potrebbero contrapporsi,ma un singolo processo è di proprietà di un singolo utente,il quale dovrebbe aver creato molteplici thread,che dovrebbero cooperare e non entrare in conflitto. Come un processo tradizionale anche un thread potrà trovarsi in uno degli stati running,ready,blocked.
E' importante capire che ogni thread ha uno stack personale che contiene una struttura per ogni procedura chiamata e non ancora terminata. Questa struttura contiene variabili locali di procedura e l'indirizzo di ritorno da usare al termine della procedura per tornare il controllo al chiamante.
Pur essendo spesso utili,i thread causano anche complicazioni gravi nel modello di multiprogrammazione. Consideriamo gli effetti della chiamata *fork*. Nel caso in cui il processo padre ha più thread e il figlio non li avesse,potrebbe non funzionare correttamente,dato che ogni thread potrebbe essere essenziale. Se invece ad esempio il figlio ha gli stessi thread del padre e uno di quelli del padre diventa blocked su una chiamata di *read* ci sarebbero due thread bloccati dalla tastiera,uno per il padre e uno per il figlio.
I progettisti dei SO devono fare delle scelte chiare e definire in modo attento la semantica,in modo tale che gli utenti comprendano il comportamento dei thread. Ad

esempio,in Linux,una *fork* di un processo multithread creerà un unico thread nel figlio. Un altro caso è quello dei segnali (che partono dopo un'eccezione). I segnali verranno recapitati a tutti i thread o al singolo thread. In Linux,ad esempio,un segnale può essere gestito da un qualsiasi thread scelto dal SO. Se uno o più thread si registrano per lo stesso segnale,il SO sceglie un thread a caso e gli consente di gestire il segnale.
Esistono principalmente due luoghi nei quali implementare i thread : spazio utente e kernel.

Kernel-level thread

Il kernel è a conoscenza dei thread del processo : ogni volta che si crea un thread nel processo,il kernel lo vede come un thread diverso che dovrà essere gestito e schedulato. Il kernel dispone di una tabella dei thread che tiene traccia di tutti i thread del sistema. Quando un thread vuole crearne uno nuovo o distruggerne uno esistente,fa una chiamata al kernel,che esegue la creazione o la distruzione aggiornando la tabella dei thread del kernel. La chiamata di sistema fornita dal SO che serve a creare un thread (solo un nuovo flusso di esecuzione in un processo,non un nuovo processo) in Linux è la ***clone***. Il SO fornisce supporto al multithreading. Il fatto che il kernel deve supportare i thread,implica che quando si invoca una *clone*,il kernel dovrà prevedere una thread table per salvare tutte le informazioni relative al thread (in realtà si estende la tabella dei processi per ospitare le informazioni private relative ai thread e quindi i registri del thread,il suo stato,il suo stack). Quando un thread si blocca,il kernel può scegliere di eseguire un altro thread dello stesso processo (se è pronto) o un thread di un altro processo. Quando invece un thread viene distrutto,viene segnato come non eseguibile,ma le sue strutture dati nel kernel non vengono intaccate.

User-level thread

Il pacchetto di thread si trova interamente nello spazio utente. Il kernel non sa nulla dei thread del processo e "vede" il consueto processo ordinario,a singolo thread. Al processo single thread corrispondono n-thread del livello utente. I thread sono implementati tramite una **thread library** : ogni volta che il SO ritorna il controllo,cioè manda in esecuzione il thread al processo,deve decidere quale thread mandare in esecuzione (di quelli utente) dalla libreria ; le informazioni sui thread sono gestite dalla libreria a livello utente. Lo **scheduling** è gestito quindi dalla libreria che decide quale thread mandare in esecuzione quando il processo diventa *running* e questo scheduling è disaccoppiato da quello del SO (la libreria potrebbe prevedere un algoritmo di scheduling completamente personalizzato).
Un esempio è quello di sistemi che non supportano il multithreading.
I vantaggi di questo sistema sono:

- minore overhead per il context switch : non bisogna scendere in kernel mode per fare un context switch tra thread (il kernel non conosce i thread);
- lo scheduling dei thread è indipendente da quello dei processi;
- le applicazioni sono **portabili** : si può scrivere programma multithread generico,avendo come riferimento le funzioni di libreria e questo programma si potrà eseguire su un'altra piattaforma a patto che esista l'implementazione di quella specifica su quella piattaforma.

Gli svantaggi di questo sistema sono:

- un thread può monopolizzare la CPU,a meno che non la rilasci volontariamente. Lo scheduler della libreria non ha a disposizione un meccanismo di salvaguardia come il timer interrupt e quindi si fa affidamento al fatto che il thread rilasci volontariamente la CPU tramite la ***yield***;
- se un thread in esecuzione invoca una system call bloccante,tutti i thread di quel processo si bloccano (questo problema non c'è nei kernel-level thread). Per risolvere questo problema ci sono due soluzioni. La prima,impraticabile,consiste nel rendere tutte le system call non bloccanti (se vogliamo fare ad esempio una *read*,la system call ritorna immediatamente se non ci sono dati invece di mettersi in attesa e aspettare l'interrupt quando clicchiamo un tasto). La seconda,più realistica,consiste nel verificare se una system call genererà un blocco. Ciò richiede un **wrapper** (codice aggiuntivo) attorno all'invocazione della system call che viene eseguito quando viene invocata la system call. Nel wrapper si invoca un'altra system call che servirà a testare se la system call effettiva si bloccherà o meno. Se il wrapper verifica che la system call non si bloccherà,il thread la invocherà altrimenti si salveranno le informazioni per riavviarlo nella thread table e si manderà in esecuzione un altro thread. In futuro,quando sarà passato un tot di tempo,si rimanderà in esecuzione il thread che era stato bloccato e la system call verrà di nuovo testata ed eventualmente eseguita.
- c'è poco vantaggio nell'utilizzo del multithreading in sistemi multicore. Se avessi n processori non avrò modo di mandare in esecuzione più thread dello stesso processo in parallelo.

Implementazione ibrida (o combinata)

A più thread al livello d'utente corrispondono un numero di thread al livello del kernel inferiore o uguale : ogni thread del kernel ha un insieme di thread a livello utente che lo "usa" a turno. Il grosso dello scheduling e sincronizzazione è fatto nello spazio utente.

Thread POSIX

Per permettere la scrittura di programmi **portabili** che usino i thread,IEEE ha definito i thread nello standard IEEE 1002.1c. Il package dei thread così definito è chiamato **phtread**,supportato dalla maggior parte dei sistemi UNIX. Lo standard definisce più

di 60 chiamate di funzione. Tutti i thread phtread hanno precise proprietà : un identificatore,un insieme di registri (incluso PC) e un insieme di attributi,memorizzati in una struttura di attributi tra i quali abbiamo la dimensione dello stack,parametri di scheduling e altri elementi per usare il thread. Un nuovo thread si crea usando la chiamata ***phtread_create***. L'identificatore del thread appena creato è restituito come valore della funzione. La chiamata è simile alla ***fork***,con l'identificatore del thread che gioca il ruolo del PID,per identificare i thread cui si fa riferimento in altre chiamate. Quando un thread ha completato il lavoro ad esso assegnato,può terminare chiamando ***phtread_exit***,chiamata che ferma il thread e libera il suo stack. Talvolta accade che un thread non sia logicamente bloccato,ma senta di essere stato in esecuzione abbastanza a lungo e voglia dare ad un altro thread la possibilità di essere eseguito. Per questo viene effettuata la chiamata ***phtread_yield***.

ESERCITAZIONE N1 → System Call per la gestione dei processi

Linux è un sistema multiprogrammato,quindi è possibile eseguire più processi indipendenti contemporaneamente. Inoltre,ciascun utente può avere diversi processi attivi contemporaneamente,così su un grosso sistema possono esserci centinaia o migliaia di processi in esecuzione. Abbiamo infatti anche processi in background,chiamati **daemon**,avviati da uno script di shell all'avvio del sistema. Un daemon tipico è **cron**. Esso si attiva una volta al minuto per controllare se c'è qualche lavoro da fare,nel caso lo fa,poi torna a riposo fino al controllo successivo. Questo daemon è necessario dato che in Linux è possibile schedulare attività per minuti,ore,giorni o mesi. Il cron si usa anche per avviare attività periodiche,come i backup giornalieri. Altri daemon gestiscono la posta elettronica in entrata e in uscita,la coda di stampa,controllano se ci sono pagine libere sufficienti nella memoria e così via. In Linux è estremamente semplice creare i processi : tramite la chiamata di sistema ***fork*** si crea una copia esatta del processo originale,che al momento della copia diventa il **processo padre**. Il nuovo processo è chiamato **processo figlio**. Padre e figlio hanno ciascuno le proprie immagini private della memoria (aree dati globali,stack,heap e user area che contiene informazioni specifiche per il processo come file aperti,directory corrente ecc). Se il genitore modifica in seguito una qualsiasi delle sue variabili,le sue modifiche non sono visibili al figlio e viceversa. I file aperti sono condivisi tra genitore e figlio : se un determinato file era aperto nel genitore prima della *fork*,rimarrà aperto in seguito sia nel genitore sia nel figlio e le modifiche apportate da uno dei due saranno visibili all'altro. Il fatto che immagini della memoria,variabili,registri e tutto il resto siano identici in padre e figlio genera una difficoltà nel sapere come i processi capiscano se eseguire il codice padre o il codice figlio. Ciò si intuisce tramite la *fork* che restituisce al figlio uno 0 e al genitore un valore diverso da 0,che è il **PID (Process IDentifier)** del figlio. Entrambi i processi normalmente controllano il valore restituito e agiscono di conseguenza. Dato che si copia anche il PC,entrambi i processi ripartono dall'istruzione successiva alla *fork*. La copia è un processo costoso. L'**area testo**

(non modificabile) è tipicamente condivisa tra padre e figlio in molte implementazioni UNIX. Una modalità di condivisione è **copy-on-write** : inizialmente il figlio condivide la memoria del padre,configurata come read-only ; se uno dei due decide di modificarne una parte,il kernel provvede esplicitamente ad effettuare una copia di tale parte della memoria.
I processi sono identificati dai loro PID. Quando si crea un processo,al padre viene dato il PID del figlio. Se il figlio vuole conoscere il suo PID si usa la chiamata di sistema ***getpid***. I PID sono usati in modi diversi. Ad esempio,quando un figlio termina,il suo PID viene assegnato al padre. Questo può essere importante perché un padre può avere molti figli. Dato che i figli possono avere figli,un processo originale può costruire un intero albero di figli,nipoti e altri discendenti. I processi in Linux possono comunicare tra loro usando una forma di scambio di messaggi. E' possibile creare tra due processi un canale,sul quale uno dei due può scrivere un flusso di byte da far leggere all'altro. Questi canali sono chiamati **pipe**.

Implementazione fork :

```
PID = fork ( ) ;

if (PID < 0) {

        handle error () ; // fork fallita (per es. memoria o qualche tabella piena)
}

else if (PID > 0) {

        CODICE PADRE

}

else {

        CODICE FIGLIO

}
```

Il PID è un numero intero tra 0 e un massimo assegnato dal kernel all'atto della sua creazione. Per avere il PID del padre si userà ***getppid***.
Dopo la chiamata della *fork*,il figlio dovrà eseguire un codice diverso dal padre. Consideriamo il caso della shell : legge un comando da terminale,genera attraverso la *fork* un processo figlio,aspetta che il figlio esegua il comando e poi legge il comando successivo quando il figlio termina. Per aspettare che il figlio finisca,il genitore esegue una chiamata di sistema ***waitpid***,che semplicemente aspetta che il figlio (o i figli) termini. La *waitpid* ha tre parametri. Il primo consente al chiamante di

attendere un figlio specifico;se è -1,andrà bene qualsiasi figlio (cioè il primo che termina). Il secondo parametro è l'indirizzo di una variabile che verrà impostata allo stato di uscita del figlio (chiusura normale o anomala e valore di uscita). Il terzo determina se il chiamante si blocca o ritorna nel caso in cui nessun figlio sia già terminato. La ***wait***,invece,consente al padre di raccogliere lo stato di terminazione dei figli. Lo **stato** contiene il valore passato dal figlio alla system call exit.

*int wait (int *stato);*
*int waitpid (pid_t pid , int *stato , int options)*

Se i figli non sono ancora terminati,il kernel **sospende** il processo padre finché uno dei figli (wait) o uno specifico figlio (waitpid) non è terminato.
Tra i casi tipici abbiamo :

- Un processo (figlio) termina **prima** che il proprio padre abbia avuto modo di attenderne la fine. Il processo terminato è definito **zombie**. Anche se terminato,ha ancora un PID e il PCB,necessari affinché il padre possa raccoglierne il valore di uscita;
- Il processo padre termina **prima** dei suoi processi figli. I processi figli diventano **orfani**. I processi orfani sono adottati da *init (systemd)* e,quindi,il PID del padre diventa 1.

La chiamata di sistema *kill* consente ad un processo di inviare un segnale ad un altro processo correlato uccidendolo.

La chiamata di sistema ***exec*** effettua una sostituzione di codice : consente ad un processo di eseguire un programma differente e il **nuovo programma** viene eseguito nel contesto del processo chiamante (il pid non cambia). L'invocazione di *exec* fa ritorno al chiamante in caso di errore,altrimenti il controllo passa al nuovo programma. Dopo l'*exec* un processo avrà nuovo codice,dati globali,stack e heap e manterrà la user area (tranne PC e informazioni relative al codice). Nel caso più generale,la *exec* ha tre parametri : nome del file che deve essere eseguito,un puntatore all'array dei parametri e un puntatore all'array dell'ambiente. Esistono varie procedure di libreria,come *execl,execv,execle ed execve*,per consentire ai parametri di essere omessi o specificati in vari modi. Tutte queste procedure invocano la stessa chiamata di sistema sottostante.
Con la *execl* il processo corrente viene sovrascritto con il nuovo programma specificato, e il controllo viene trasferito al nuovo programma.
La funzione *execle()* è simile a *execl()*, ma consente di specificare anche l'ambiente del nuovo programma. L'ambiente si riferisce alle variabili d'ambiente che il nuovo programma dovrebbe utilizzare.

Modello multiprogrammazione (approssimato)

L'utilizzo della CPU può essere migliorato per mezzo della **multiprogrammazione**. In parole povere,se il processo medio esegue calcoli solo per il 20% del tempo in cui risiede in memoria,con cinque processi in memoria in contemporanea la CPU dovrebbe essere occupata tutto il tempo. Questo modello è irrealisticamente ottimista,dato che presuppone tacitamente che nessuno dei cinque processi sia mai in attesa di I/O nello stesso momento. Considerando l'uso della CPU da un punto di vista statistico si ottiene un modello migliore. Supponiamo che un processo impieghi una frazione *p* del suo tempo in attesa che l'I/O sia completato. Con *n* processi in memoria contemporaneamente,la probabilità che tutti gli *n* processi stiano aspettando l'I/O (nel caso in cui la CPU sarebbe inattiva) è p^n. L'utilizzo della CPU è quindi dato dalla formula :

Utilizzo CPU = 1 - p^n.

Il **grado di multiprogrammazione** è indicato come una funzione di *n*. Se i processi impiegano l'80% del loro tempo in attesa dell'I/O,devono esserci contemporaneamente almeno 10 processi in memoria per fare in modo che lo spreco di CPU sia inferiore al 10%. Dato che i tempi di attesa di I/O dell'80% e più non sono rari il modello probabilistico appena visto è un'approssimazione. Esso presuppone in modo implicito che tutti gli *n* processi siano indipendenti,e ciò significa che è abbastanza accettabile che un sistema con cinque processi in memoria abbia tre in esecuzione e due in attesa. Supponendo di avere un processo che sfrutta 50% di CPU non è sufficiente prendere un altro processo che sfrutta 50% di CPU per arrivare al 100% : se il processo B attende eventi nel momento in cui quando A utilizza processore si arriva al 100% ; se A e B fanno I/O allo stesso momento l'utilizzo della CPU rimane al 50%. Con una singola CPU,tuttavia,non possiamo avere tre processi in esecuzione contemporaneamente,così un processo che diventa pronto mentre la CPU è occupata dovrà aspettare e quindi i processi non sono indipendenti.

Scheduling

Quando un computer è multiprogrammato,spesso ha molteplici processi o thread che competono per ottenere la CPU nel medesimo istante. Questa situazione si verifica quando due o più di essi sono in stato pronto. Se è disponibile una sola CPU,è necessario scegliere quale processo eseguire come successivo. La parte del SO che effettua la scelta e assegna le risorse al processo è chiamata **scheduler** mentre il processo è scelto tramite un **algoritmo di scheduling**. Vedremo lo **scheduler della CPU** (o a breve termine) che seleziona il processo tra 2 o più processi in stato pronto. Vale la pena di notare che se le CPU diventano più veloci,i processi tendono a essere più i/o bound : le CPU migliorano molto più velocemente dei dischi e quindi è probabile che in futuro lo scheduling dei processi i/o bound

diventi più importante. L'idea base è che un processo di questo tipo che vuole essere eseguito dovrebbe poterlo fare in fretta,per poter inviare la propria richiesta al disco e tenerlo impegnato. Per mantenere la CPU completamente impegnata sono necessari parecchi processi I/O bound. Molti problemi che si applicano allo scheduling dei processi si applicano anche a quello dei thread. Quando il kernel gestisce i thread,lo scheduling è fatto per thread e che un thread appartenga ad un processo o ad un altro importa poco o nulla.

Quando effettuare lo scheduling

- scheduling **non preemptive** : quando un processo termina (naturale o errore) bisogna scegliere un altro processo da mandare in esecuzione. Quando un processo si blocca in attesa di un evento → un processo che sta sul processore ed effettua ad esempio una system call che lo blocca,deve lasciare la CPU ed interviene quindi lo scheduler. Al massimo si può sperare che il processo rilasci volontariamente la CPU tramite chiamata di sistema (yield).
- scheduling **preemptive** : quando viene creato un processo o quando si verifica un interrupt. Lo scheduling preemptive comprende anche i primi due casi. Il SO si prende quindi il diritto di liberare la CPU dal processo in esecuzione e dare preferenza ad un altro processo. Nasce questa esigenza perché per i processi CPU bound che possono monopolizzare la CPU,conviene dotarsi di un meccanismo di salvaguardia (timer interrupt). Quando viene creato un nuovo processo,infatti,si deve decidere se eseguire il processo genitore o il processo figlio. Essendo entrambi in stato pronto si tratta di una normale decisione di scheduling e può andare in entrambe le direzioni,ossia lo scheduler può legittimamente scegliere di eseguire per primo il genitore o il figlio.I SO moderni usano scheduling preemptive.

Ricapitolando : lo scheduling **non preemptive** sceglie un processo per eseguirlo e poi lo lascia semplicemente girare fino a quando si blocca (sia su un I/O o in attesa di un altro processo) o rilascia volontariamente la CPU. In caso di interrupt,non vengono prese decisioni di scheduling e al termine dell'elaborazione dell'interrupt viene ripristinato il processo precedentemente in esecuzione,a meno che abbiamo un processo con priorità più alta in attesa. Lo scheduling **preemptive** sceglie invece un processo e lo lascia girare per un tempo massimo prefissato. Se alla fine dell'intervallo di tempo è ancora in esecuzione,il processo è sospeso e lo scheduler ne sceglie un altro da eseguire (se è disponibile). Questo tipo di scheduling richiede che vi sia un timer interrupt alla fine dell'intervallo per restituire il controllo della CPU allo scheduler.

Obiettivo di un algoritmo di scheduling

- equità : dare ad ogni processo un'equa condivisione della CPU;

- imposizione della policy : assicurarsi che la policy dichiarata sia attuata;
- bilanciamento : tenere impegnate tutte le parti del sistema;
- throughput : massimizzare il numero di job per unità di tempo;
- tempo di turnaround : ridurre al minimo il tempo tra il momento in cui un job viene sottoposto e quello in cui termina;
- utilizzo della CPU : mantenere la CPU sempre impegnata;
- tempo di risposta : è il tempo che trascorre dall'istante della sottomissione di una richiesta fino all'istante in cui la risposta inizia ad essere ricevuta;
- rispetto delle scadenze : evitare la perdita di dati;
- adeguatezza : far fronte alle aspettative dell'utente;
- tempo di attesa : tempo che il processo ha speso nello stato ready; ← Importante;

Non esiste un algoritmo di scheduling perfetto. Gli obiettivi sono tanti,interdipendenti,e,talvolta,anche in conflitto tra di essi : un algoritmo di scheduling implica un compromesso che tenga conto anche della tipologia di sistema/dominio applicativo. Distinguiamo così tre tipologie :

- sistemi **batch** : grandi server a cui sottopongo il lavoro ma non mi aspetto una risposta istantanea. Tra i parametri di questi sistemi abbiamo throughput,tempo di turnaround,utilizzo della CPU;
- sistemi **interattivi** : sistemi dove digitiamo qualcosa e ci si aspetta una risposta veloce. Tra i parametri da tener conto abbiamo tempo di risposta,tempo di attesa e adeguatezza;
- sistemi **real-time** : rispetto delle scadenze e prevedibilità.

Un algoritmo di scheduling che massimizza il throughput non necessariamente minimizza il tempo di turnaround. Per esempio,dato un mix di job brevi e lunghi,uno scheduler che esegua sempre job brevi e mai job lunghi potrebbe ottenere un throughput eccellente,ma a costo di un tempo di turnaround inaccettabile per i job lunghi. Se i job brevi continuano ad arrivare ad un tasso abbastanza costante,i processi lunghi potrebbero non essere mai eseguiti,ottenendo un tempo medio di turnaround infinito pur mantenendo un throughput elevato.

Algoritmi di scheduling

First-come,first-served (FCFS)

I processi sono assegnati alla CPU nell'ordine di arrivo. Fondamentalmente c'è una singola coda di processi in stato pronto. Quando il primo entra nel sistema,viene avviato immediatamente e gli è consentito di essere eseguito finché vuole. Se la durata di esecuzione è troppo lunga non viene interrotto. Gli altri processi vengono messi in fondo alla coda. Quando il processo in esecuzione si blocca,viene eseguito il primo processo della coda. Quando un processo bloccato ritorna pronto,è posto in fondo alla coda come un processo appena arrivato.

Supponiamo di avere tre processi P1,P2,P3. P1 richiede 24 unità di tempo,P2 3 unità e P3 3 unità. I processi verranno eseguiti nell'ordine di arrivo P1-P2-P3 con un tempo di attesa P1 = 0 , P2 = 24 , P3 = 27 e un tempo di attesa medio pari a (0+24+27)/3 = 17. Per ridurre questo tempo di attesa si potrebbe invertire l'ordine di esecuzione,eseguendo prima P2 e P3 dato che hanno un minore tempo di servizio e poi P1. In questo modo avremo un tempo di attesa P2 = 0 , P3 = 3 , P1 = 6 e tempo di attesa medio pari a 3 ma l'equità non sarà più rispettata dato che avremo una preferenza sui processi da eseguire prima.
La grande forza di questo algoritmo è che è facile da capire,facile da programmare ed equo. Con esso,una sola lista concatenata tiene traccia di tutti i processi pronti. Per scegliere il processo da eseguire basta prendere quello all'inizio della coda e per aggiungerne uno basta aggiungerlo alla fine della coda. Sfortunatamente ha anche un forte svantaggio. Supponiamo che ci sia un processo CPU bound eseguito per 1 secondo alla volta e molti processi I/O bound che usano poco tempo la CPU,ma che ognuno debba eseguire 1000 letture del disco per completare il lavoro. Il processo CPU bound è eseguito per 1s e poi legge un blocco di disco. Vengono poi eseguiti i processi I/O e iniziano le letture. Il risultato finale è che ogni processo I/O bound riuscirà a leggere 1 blocco al secondo e impiegherà 1000 secondi per terminare. Il fenomeno per il quale un processo "corto" può aspettare molto è detto **convoy effect** : i processi I/O-bound devono attendere il completamento di quelli CPU-bound. E' un algoritmo senza prelazione.

Shortest job first

Questo algoritmo parte dal presupposto che i tempi di esecuzione dei processi siano noti in anticipo. Quando nella coda di input si trovano parecchi job di pari importanza in attesa di essere avviati,lo scheduler preleva per primo il job più breve. Supponendo di avere i job A,B,C,D con tempi di esecuzione pari a 8,4,4 e 4 minuti,eseguendoli in ordine il tempo di turnaround sarebbe di 8 minuti per A,12 minuti per B,16 per C e 20 per D con una media di 14 minuti e un tempo di attesa pari a (0+8+12+16)/4 = 9. Se invece usassimo l'algoritmo SJF,il turnaround sarebbe di 4 per B,8 per C,12 per D e 20 per A con una media di 11 minuti e un tempo di attesa pari a (0+4+8+12)/4 = 6. Questo algoritmo è indiscutibilmente ottimale solo nel caso in cui tutti i job siano disponibili contemporaneamente. E' un algoritmo senza prelazione.

Shortest remaining time next

E' la versione con prelazione dell'algoritmo SJF. Con questo algoritmo lo scheduler sceglie sempre il processo che impiegherà meno tempo per terminare l'esecuzione e anche in questo caso il tempo di esecuzione deve essere noto in anticipo. All'arrivo

di un nuovo job,il suo tempo totale è confrontato al tempo restante dei processi attuali. Se il nuovo job richiede meno tempo del processo attuale per terminare,il processo attuale viene sospeso ed è avviato il nuovo job.
Supponiamo di avere questo scenario :

tempo di arrivo	processo	service time
0	A	8
2	B	2
4	C	4

Quando arriva B al tempo 2,B ha tempo di servizio 2 inferiore agli 8 di A. A viene tolto,si segna quanto tempo rimane da eseguire per A (8 unità - 2 eseguite = 6) e scelgo chi richiede lo shortest remaining time ovvero il tempo di esecuzione rimanente più piccolo,in questo caso B. Al tempo 4 arriva C,A dovrà aspettare ancora e B sarà terminato e C sarà eseguito. Alla fine A potrà consumare le 6 unità di tempo rimaste.

Shortest process next (SPN)

L'algoritmo SJF produce sempre il minor tempo medio di risposta sui sistemi batch,quindi sarebbe bello poterlo usare anche nei sistemi interattivi che generalmente seguono lo schema di attesa/esecuzione di un comando. Se consideriamo l'esecuzione di ogni comando come singolo "job",potremmo minimizzare complessivamente il tempo di risposta eseguendo per primo il job più breve. Il problema è identificare quale degli attuali processi eseguibili sia il più breve. Questo algoritmo infatti non prevede che si conoscano a monte i tempi di esecuzione. Un approccio è quello di fare delle stime basate sul comportamento passato ed eseguire il processo con il tempo di esecuzione stimato più breve. Viene costruita per ogni processo una previsione della durata del prossimo CPU burst (utilizzo intensivo) tramite una **media esponenziale (exponential averaging)** :

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Indica la predizione e cioè ciò che al tempo attuale si ipotizza essere quella che sarà la durata del next burst CPU (per quanto tempo al prossimo turno utilizzerà la CPU). T verrà inserita in una tabella e sarà l'informazione con cui lo scheduler deciderà chi eseguire,in particolare si prenderà il processo con T più piccolo. La predizione è costituita da due contributi :

- una parte della predizione viene calcolata tenendo conto per quanto tempo il processo ha appena usato processore (ad esempio nell'ultimo burst il processo ha occupato processore per 10s). Questo contributo **tn** ed indica l'attuale lunghezza del n-esimo CPU burst.
- l'altra parte viene calcolata tenendo conto della storia passata dell'utilizzo del processore da parte dei processi. E' il parametro **Tn**.

I due parametri sono regolati da **alfa** che varia tra **0<=alfa<=1**. Un valore tipico di alfa è **alfa = 1⁄2 e** cioè per costruire la stima ci si basa per metà sull'ultima osservazione e per metà su tutta la storia passata. Il caso limite è **alfa = 0**,dove l'ultima osservazione non viene considerata e ci si basa solo sulla storia passata. Se invece **alfa = 1** si costruisce la previsione tenendo solo conto dell'ultimo effettivo utilizzo del processore (tn).

In generale : vedi slide per calcolo media esponenziale.

Es :

Immaginiamo che al tempo 0,ho la stima al burst precedente pari a 10 e come durata dell'ultimo burst pari a 6. Se alfa è pari ad ½ avrò una predizione pari ad 8 (½ * 6 + (1-½)10 = 8). L'algoritmo si baserà quindi su questo parametro pari ad 8 è cioè lo scheduler selezionerà sulla carta il processo che al prossimo burst prenderà quel tempo. Se ho stimato 8,il processo viene schedulato e si passa all'iterazione successiva. La durata del burst effettivo è stata 4 quindi la predizione è risultata sbagliata. Rieffettuo il calcolo : ½*4 +(1-½)*8 = 6. Se 6 tra tutti i processi che hanno lo SPN è effettivamente il minore selezionerò questo. La predizione difficilmente coinciderà con la durata effettiva del burst al passo successivo ma più o meno cerca di seguirla. Ci sarà quindi un transitorio dopo cui la predizione si adatterà agli utilizzi effettivi e la sua rapidità ad esaurirsi dipende dal parametro alfa.

Round robin

E' uno degli algoritmi più vecchi,più semplici,più equi e maggiormente utilizzati. E' un algoritmo con prelazione dove ad ogni processo viene assegnato un intervallo di tempo,detto **quanto**,durante il quale gli è consentito di essere eseguito. Se alla fine del quanto il processo è ancora in esecuzione,la CPU viene **prelazionata** in favore di un altro processo. Se il processo si blocca o termina prima che sia trascorso il quanto,il passaggio della CPU avviene naturalmente quando il processo si blocca. Il round-robin è facile da implementare. Lo scheduler deve semplicemente mantenere una lista dei processi eseguibili. Quando il processo esaurisce il suo quanto,viene posto alla fine della lista.
Supponiamo di avere il seguente scenario :

processo	arrival time	service time
A	0	3
B	2	6
C	4	4

Al tempo 0 è in esecuzione A,al tempo 2 si fa un altro quanto e arriva B. A avrà terminato il suo quanto e quindi inizia il quanto di tempo di B. Al tempo 4 arriva C,e viene eseguito il quanto di tempo per C. Man mano che i processi completano l'esecuzione,usciranno dal round-robin.
La durata del quanto è **critica**. La gestione del passaggio da un processo ad un altro (salvare e caricare i registri e le mappe di memoria,aggiornare le tabelle e i vari elenchi,svuotare e ricaricare la cache e così via) richiede una certa quantità di tempo.
Se il quanto è troppo **breve** → troppi context switch.
Se il quanto è troppo **lungo** → si degenera in un FCFS e causa una risposta scadente alle richieste interattive brevi..
Se il quanto è più lungo del burst di CPU medio,la prelazione non avviene molto spesso. Molti processi effettueranno invece un'operazione di blocco prima che finisca il quanto,causando uno scambio di processo. Eliminare la prelazione migliora le prestazioni poiché gli scambi di processo avvengono solo quando sono logicamente necessari,ossia quando un processo si blocca e non può continuare.
Generalmente un **valore tipico** del quanto è compreso tra i 20 e i 50 ms.

Scheduling a priorità

I processi sono raggruppati in **classi di priorità**. La **priorità** è rappresentata tipicamente da un intero che può variare in un certo range e che è associato al processo. La priorità indica quanto è importante e quanta importanza dare ad un determinato processo.
Nella **versione base** i processi sono raggruppati per priorità e il SO alterna i processi a pari priorità con un approccio round-robin. Fintantoché ci sono processi in una coda di priorità,le priorità meno importanti (decrescenti) non verranno considerate (fino a quando la coda a priorità maggiore non si esaurisce). La probabilità che i processi che si trovano nelle code più scarse non arrivino al processore è concreta. Il rischio che si corre è chiamato **starvation** dei processi a minore priorità : se continuano ad arrivare processi a priorità elevata che avranno precedenza,quelli a priorità minore rischiano di non vedere mai processore.

Gestire le priorità non è semplice. Per evitare che i processi a priorità elevata monopolizzino la CPU e processi a priorità più bassa o che facciano molto I/O non vedano mai processore abbiamo diverse soluzioni :

- riduzione “volontaria” della priorità dei propri processi da parte dell’utente (in realtà viene gestito in modo automatico). Ad esempio se si ha un processo a priorità elevata e ci si accorge che esso non termina ogni volta esaurito il suo quanto,abbassiamo la sua priorità e lo facciamo scendere nelle code. La riduzione della priorità può essere realizzata dall’utente stesso in UNIX tramite comando ***nice***. Prendiamo i processi che,una volta che si vedono assegnato il quanto di tempo,appena entrano sul processore non consumano tutto il quanto. Questa informazione è utilizzata per priorizzare i processi,dando priorità maggiore al round successivo ai processi che non riescono ad utilizzare tutto il proprio quanto di tempo. Si tratta di una buona soluzione per i processi I/O-bound. La priorità è data da **1/f** dove **f** è la frazione dell’ultimo quanto usata dal processo. Supponiamo che q = 50 e un processo usato per 1 unità di tempo → priorità = 1/(1/50) = 50. Se invece q = 50 e un processo usato per 25 unità di tempo → priorità = 1/(25/50) = 2. Se avrò tanti processi I/O-bound,però,avranno tutti priorità elevata e quindi i processi CPU bound vengono penalizzati.
- riduzione della priorità del processo in esecuzione ad ogni clock interrupt. Se questa azione fa scendere la sua priorità al di sotto di quella del processo successivo avviene un context switch. In alternativa,a ciascun processo può essere assegnato un quanto di tempo massimo per l’esecuzione;nel momento in cui il quanto termina,è data l'opportunità di esecuzione al processo con priorità immediatamente inferiore. Dopo aver inibito un processo per un periodo abbastanza lungo,è necessario che un algoritmo ne aumenti la priorità per fare in modo che possa riprendere l’esecuzione,altrimenti tutti i processi avrebbero priorità pari a 0.

Multilevel feedback queue

E’ un algoritmo che prevede diverse code con diverse priorità. Un parametro di questo algoritmo è quale approccio di scheduling viene eseguito nella singola coda,tipicamente il round-robin. Quando viene creato il nuovo processo,gli viene assegnata la massima priorità. Questo algoritmo prevede che i processi che ad esempio sono a priorità massima e non hanno esaurito il loro quanto di tempo vengano spostati in una coda a priorità più bassa. Nel caso in cui nemmeno con il quanto della coda a priorità minore si riesce ad eseguire il processo esso viene spostato ancora in un’altra coda a priorità minore e così via fino a quando viene eseguito. Le code,non sono vincolate infatti ad avere lo stesso quanto di tempo.

Scheduling dei thread

Consideriamo i thread a **livello utente**. Dato che il kernel non è a conoscenza dell'esistenza dei thread,opera come al solito scegliendo un processo,diciamo A,e passandogli il controllo per il suo quanto. Lo scheduler dei thread interno ad A decide quale thread eseguire,diciamo A1. Poiché per questi thread non vi sono interrupt del clock,A1 può continuare l'esecuzione quanto vuole. Se utilizza l'intero quanto del processo,il kernel sceglie un altro processo da eseguire. Quando il processo A finalmente viene eseguito,il thread A1 riprenderà l'esecuzione e continuerà a consumare tutto il tempo di A fino a quando avrà finito. Questo comportamento però non influirà su altri processi,ai quali lo scheduler continuerà ad allocare ciò che ritiene opportuno a prescindere da ciò che avviene all'interno del processo. Consideriamo il caso che i thread di A abbiano relativamente poco lavoro da svolgere per burst di CPU,ad esempio 5 ms di lavoro su 50 ms di quanto;ognuno lavora per un po',dopodiché cede la CPU allo scheduler dei thread. Ciò potrebbe portare alla sequenza A1,A2,A3,A1,A2,A3,A1 prima che il kernel passi al processo B. L'algoritmo di scheduling usato può essere uno di quelli descritti in precedenza. Lo scheduler interno al processo (implementato dalla thread library) decide quale thread eseguire. La sola restrizione è l'assenza di interrupt del clock che interrompano un thread eseguito troppo a lungo,ma di solito non è un problema perché i thread cooperano.
Consideriamo ora i thread a **livello kernel**. In questo caso il kernel sceglie un particolare thread da eseguire. Non deve tener conto del processo cui appartiene il thread,ma può farlo se vuole. Al thread è attribuito un quanto,e se lo eccede viene sospeso forzatamente. Una differenza rilevante fra i thread a livello utente e quelli a livello kernel sta nelle prestazioni : uno scambio di thread a livello utente richiede una manciata di istruzioni macchina,con i thread del kernel sono necessari un context switch,modifica della mappa della memoria e invalidazione della cache,il che è più lento di parecchi ordini di grandezza. Per contro,un thread del kernel bloccato in attesa di I/O non sospende l'intero processo,come avviene invece a livello utente. Dato che il kernel sa che passare da un thread nel processo A ad un thread nel processo B è più costoso che eseguire un secondo thread in A,può tener conto di questa informazione quando deve prendere una decisione. Ad esempio,dati due thread ugualmente importanti,uno dei quali appartenente allo stesso processo di un thread che si è bloccato e l'altro appartenente a un diverso processo,la preferenza potrebbe essere assegnata al primo. Un altro fattore importante è che i thread a livello utente possono utilizzare uno scheduler dei thread specifico dell'applicazione.

SO cap3

Ci riferiremo alla **main memory**,la memoria RAM,ovvero la memoria dove sono caricati i processi e da dove il processore va a leggere. Parleremo dell'allocazione della memoria ai processi,deallocazione,protezione,condivisione controllata,gestione indirizzi…

Nessuna astrazione di memoria

La più semplice astrazione della memoria è l'assenza di astrazione. I primi computer non avevano astrazione della memoria. Ogni programma vedeva semplicemente la memoria fisica. Quando un programma eseguiva un'istruzione come **MV R1,1000**,il computer spostava semplicemente il contenuto della locazione di memoria fisica 1000 a R1. In questo modo il modello di memoria presentato ai programmatori era semplicemente memoria fisica,un insieme di indirizzi da 0 ad un certo massimo,ogni indirizzo corrispondente ad una cella contenente un certo numero di bit,solitamente 8. L'associazione di istruzioni/dati ad indirizzi di memoria è fatta a tempo di **compilazione** : in fase di compilazione è noto dove il processo risiederà in memoria;se,in un momento successivo,la locazione cambiasse,sarebbe necessario ricompilare il codice. Questo scenario è detto **codice assoluto (absolute code)**. Questo schema a soli indirizzi fisici funziona solo nel caso monoprogrammato. Se la memoria è però sufficientemente grande,posso avere più processi in diverse parti della memoria. Quindi se ho due programmi diversi,essi vengono collocati in parti diverse della memoria fisica. In questo caso,però,se le istruzioni vengono caricate nella loro forma originale,utilizzeranno sempre gli stessi indirizzi e potrebbero puntare a locazioni diverse entrando in locazioni di altri programmi. Una possibile soluzione a ciò è la **rilocazione statica** : a tempo di compilazione si utilizzano indirizzi "fisici" arbitrari e a tempo di caricamento (il SO conosce l'indirizzo fisico di partenza del programma) gli indirizzi all'interno del codice vengono sostituiti sommandoli all'indirizzo di partenza. Il compilatore deve essere predisposto per generare codice **rilocabile**. Questa soluzione funziona bene se eseguita in modo corretto,ma non è una soluzione molto generale e rallenta il caricamento. Inoltre richiede un'informazione aggiuntiva in tutti i programmi eseguibili per indicare quali parole contengono indirizzi riposizionabili e quali no. Ad esempio **JMP 28** può essere riposizionata,ma un'istruzione come **MOV R1,28** che sposta 28 in R1 non deve essere riposizionata. Il loader necessita quindi di un sistema per distinguere una costante da un indirizzo. Un'altra soluzione è quella della **protezione della memoria** : la memoria è suddivisa in blocchi e a ciascuno è assegnata una **chiave di protezione** ; nel PSW (Program Status Work Register) viene caricato il valore della chiave del blocco di memoria in cui il processo in esecuzione è autorizzato a scrivere/leggere. Ogni volta che il processo fa riferimento ad un indirizzo si va a verificare che l'indirizzo di interesse corrispondesse ad un blocco etichettato con la chiave nel PSW in quell'istante. Se non è così il SO intercetta questi processi e si evita che il processo acceda ad aree di memoria non indirizzate ad esso.

Spazio degli indirizzi

Per **spazio di indirizzi** si intende l'insieme degli indirizzi che un processo può usare per indirizzare la memoria. Ogni processo ha il suo spazio degli indirizzi "personale",**indipendente** dagli altri processi e i suoi indirizzi vanno da 0 ad un certo massimo. Tutte le istruzioni sono impostate per puntare ad indirizzi logici : PC,SP contengono indirizzi logici,ptr a variabili punteranno all'indirizzo logico nell'immagine del processo,jmp e branch si riferiscono ad indirizzi logici e così via. Il problema che

si pone è come dare l'illusione a tutti i processi di avere un set di indirizzi a cui accedere. Ad esempio l'indirizzo 4096 in un programma corrisponde ad una locazione fisica diversa dall'indirizzo 4096 di un altro processo. L'associazione di istruzioni/dati ad indirizzi di memoria è fatta a tempo di **esecuzione** con **rilocazione dinamica**: gli indirizzi logici non coincidono con gli indirizzi fisici (gli indirizzi logici e fisici sono identici,come nel caso di associazione a tempo di compilazione;nell'associazione in fase di caricamento gli indirizzi logici (rilocabili) sono rimpiazzati da quelli fisici);richiede supporto specifico dell'architettura (ad esempio specifici registri hardware);durante il suo ciclo di vita un processo può essere spostato da un blocco di memoria di un altro.
Ad occuparsi della traduzione degli indirizzi logici in indirizzi fisici è il dispositivo hardware **MMU (Memory management unit)**. Quindi l'indirizzo **logico** è l'indirizzo generato dalla CPU,detto anche **indirizzo virtuale** mentre l'indirizzo **fisico** è l'indirizzo visto dall'unità di memoria. Il programma tratta indirizzi logici e non fisici. E' infatti la MMU ad associare gli indirizzi logici agli indirizzi fisici in fase di esecuzione.

Registri base e registri limite

Si tratta di una semplice versione della **rilocazione dinamica** : mappa lo spazio degli indirizzi di ogni processo su una parte diversa della memoria fisica. Ogni CPU viene equipaggiata con due registri hardware speciali,chiamati **registro base (o di rilocazione)** e **registro limite**. Quando vengono utilizzati questi registri,i programmi sono caricati in posizioni di memoria consecutive dovunque vi sia spazio e senza rilocazione durante il caricamento. Al momento dell'esecuzione di un processo,nel registro base viene caricato l'indirizzo fisico dove comincia il suo programma in memoria e nel registro limite viene caricata la lunghezza del programma. Ogni volta che un processo fa riferimento alla memoria,per prelevare un'istruzione o per leggere o scrivere una parola di dati,prima di inviare l'indirizzo sul bus di memoria l'hardware della CPU aggiunge automaticamente il valore di base all'indirizzo generato dalla CPU e contemporaneamente controlla se l'indirizzo offerto sia uguale o maggiore al valore del registro limite,nel qual caso è generato un errore di eccezione (di indirizzamento) e l'accesso viene terminato. Questa implementazione hardware con due registri è una forma base di **MMU**. Uno svantaggio della rilocazione per mezzo dei registri limite e base è la necessità di eseguire una somma e un confronto ad ogni riferimento alla memoria. I confronti possono avvenire velocemente,ma le somme sono lente a causa del tempo di propagazione,a meno che non vengano usati circuiti speciali.

Swapping

Se la memoria fisica del computer è abbastanza ampia da contenere tutti i processi,gli schemi visti finora più o meno funzioneranno. Nella realtà tuttavia la quantità totale di RAM necessaria per tutti i processi è spesso molto più grande della memoria fisica. Una parte della memoria fisica è occupata dal kernel. Supponiamo di

avere uno scenario con una macchina che supporti uno spazio logico di indirizzi pari a 2^(16) = 64 kB. Non necessariamente il processo sfrutta tutta la dimensione dello spazio logico,ad esempio potrebbe avere bisogno solo di 48 kB. Supponiamo di portare in memoria fisica i 48 kB del processo. Arriva il processo B con il suo spazio e lo carico in memoria. Arriva C e se ho sufficiente spazio lo carico in memoria. Arriva D e in questo caso la quota di memoria rimanente è più piccola dell'immagine dell'eseguibile di D. In questo caso o si aspetta che qualcuno termini oppure si effettua un'operazione di **swapping** : si prende l'immagine in memoria di un processo e si copia sul disco,liberando così spazio in memoria. Quindi viene attuato lo swapping sul disco di A e D viene caricato in una parte dello spazio di memoria liberato da A. In seguito esce B e ritorna A. Dato che ora è in una posizione diversa,i suoi indirizzi devono essere rilocati dal software al momento dello swapping o (più probabilmente) dall'hardware durante l'esecuzione del programma. Quando lo swapping crea più spazi vuoti della memoria,è possibile combinarli tutti in un unico spazio vuoto spostando tutti i processi il più in basso possibile. Questa tecnica è chiamata **memory compaction (compattazione della memoria)**. Bisogna capire quanta memoria dovrebbe essere allocata per un processo quando viene creato o viene riportato in memoria dal disco tramite swapping. Se i processi sono creati con una dimensione fissa che non cambia mai,l'allocazione è semplice e il SO alloca esattamente il necessario. Se invece i segmenti dei dati dei dati del processi possono crescere,ad esempio **allocando dinamicamente memoria dallo heap**,appena il processo prova a crescere sorge un problema. Se c'è spazio vuoto adiacente al processo,può essere allocato e il processo può crescere in tale spazio. Se invece il processo è adiacente ad un altro processo,quello che sta crescendo dovrà essere spostato in uno spazio vuoto della memoria abbastanza grande da ospitarlo,oppure si dovrà effettuare lo swapping sul disco di uno o più processi,in modo da creare uno spazio vuoto abbastanza grande. Se un processo non può crescere nella memoria e l'area di swap del disco è piena,il processo deve essere sospeso finché non sia liberato dello spazio.

Nel caso in cui ci si aspetta che la maggior parte dei processi cresca durante l'esecuzione,probabilmente è opportuno allocare un po' di **memoria extra** ogni volta che un processo viene scambiato o spostato,per ridurre l'overhead associato allo swapping o allo spostamento dei processi che non stanno più dentro la memoria allocata inizialmente. Tuttavia,quando si fa lo swapping di un processo sul disco,si dovrebbe scambiare solo la memoria effettivamente in uso : è uno spreco fare lo swapping anche della memoria extra.

L'altra possibilità è quella di allocare area dati e testo in modo contiguo e di allocare lo stack nella parte superiore dello spazio degli indirizzi logici,facendo in modo che lo stack cresca verso il basso e l'heap verso l'alto sperando che non sforino l'uno nell'altro (buffer overflow → faccio sforare l'heap nello stack,sovrascrivo lo stack e il processo punterà da tutt'altra parte). Se si esaurisce la memoria tra stack e dati,il processo dovrà essere spostato in uno spazio vuoto con spazio sufficiente,o scambiato dalla memoria al disco fino a quando sia possibile creare uno spazio abbastanza grande,oppure dovrà essere terminato.

Memoria virtuale

Per quanto visto fino ad ora,un processo deve essere sempre collocato in una porzione di memoria contigua. Si pone poi il problema di come eseguire programmi più grandi della memoria fisica disponibile. Il problema dei programmi più grandi della memoria è presente fin dalle origini dell'informatica. Negli anni '60 fu adottata una soluzione che divideva i programmi in piccole parti dette **overlay**. All'avvio di un programma veniva caricato dalla memoria solo il gestore degli overlay,che caricava subito l'overlay 0. Al termine,veniva indicato al gestore degli overlay di caricare l'overlay 1 sopra l'overlay 0 in memoria oppure sovrascrivendo l'overlay 0 (in mancanza di spazio). Il lavoro di scambio degli overlay veniva eseguito dal SO,ma la suddivisione dal programmatore. Si arrivò poi alla **memoria virtuale** : un programma ha il suo spazio degli indirizzi (virtuali) suddiviso in **pagine** posizionabili in maniera indipendente e di dimensione fissa pari a 4 kB. L'altro approccio è quello della **segmentazione**,dove lo spazio degli indirizzi logici viene diviso in segmenti con dimensione variabile. Ogni pagina è un intervallo di indirizzi contigui. Queste pagine sono mappate su memoria fisica,ma per eseguire il programma non è indispensabile che tutte le pagine siano contemporaneamente in memoria fisica. Quando il programma fa riferimento a una parte del suo spazio di indirizzi che è in memoria fisica,l'HW esegue direttamente la mappatura necessaria. Quando il programma fa riferimento ad una parte del suo spazio degli indirizzi che non è in memoria fisica,il SO viene allertato,va a prelevare la parte mancante ed esegue nuovamente fallita.
Con la memoria virtuale,invece di avere una rilocazione separata solo per i segmenti dati e testo,l'intero spazio degli indirizzi può essere rimappato sulla memoria fisica in unità abbastanza piccole.

Paginazione

Prendiamo 64 kB come insieme degli indirizzi logici o virtuali che un processo ha a disposizione. Con indirizzi virtuali su 16 bit ho un intervallo da 0-65535. Lo spazio virtuale a disposizione del processo non conviene più trattarlo come blocco unico in maniera contigua,dato che difficilmente si troverà un blocco che conterrà tutti i 64 kB e si genererà frammentazione. Lo spazio virtuale viene quindi diviso in un certo numero di pezzi detti **pagine** di dimensione pari a 4 kB. Se lo spazio logico è di 2^(16) e lo divido in pagine di dimensione pari a 4 kB avrò 16 pagine : 4 kB = 2^(2) * 2^(10) = 2^(12) → 2^(16)/2^(12) = 2^4 = 16 pagine. Avrò così 16 range diversi.
Anche la memoria fisica non viene più trattata come unico blocco contiguo e viene divisa in pezzi di dimensione pari alla dimensione della pagina. I singoli pezzi della memoria fisica sono detti **frame** o **frame page**. Con una memoria fisica a 32 kB,ad esempio,con pagine a 4 kB avrò a disposizione 8 frame : 32 kB = 2^(5) * 2^(10) = 2^(15) → 2^(15)/2^(12) = 2^(3) = 8 frame.

Quando è utilizzata la memoria virtuale,gli indirizzi non vanno direttamente al bus di memoria,ma ad una MMU che mappa gli indirizzi virtuali sugli indirizzi della memoria fisica.
I pezzi dello spazio di indirizzamento logico possono essere caricati in un frame. Non è più necessaria la contiguità : se ad esempio prendo il range 8k-20k posso allocare i singoli 3 range in maniera indipendente su tre frame non contigui. Si riduce così la **frammentazione esterna**. Inoltre,non è più necessario che tutta l'immagine sia allocata in memoria centrale tutta nello stesso tempo. Si potranno quindi avere delle pagine che non sono caricate in memoria e che rimangono sul disco. Se ad esempio il programma prova ad accedere all'indirizzo 0,l'indirizzo virtuale 0 viene mandato alla MMU. La MMU vede che questo indirizzo virtuale cade nella pagina 0,ovvero nel range 0-4095,che secondo la sua mappatura in figura,è il frame 2 (8192-12287). Trasforma così l'indirizzo in 8192 ed emette sul bus l'indirizzo 8192. La memoria non sa dell'esistenza dell'MMU e vede semplicemente una richiesta di lettura o scrittura all'indirizzo 8192.
Per sapere la pagina a quale frame corrisponde viene utilizzata una struttura dati del kernel detta **page table**.

Page table

Supponiamo di avere uno scenario con indirizzi virtuali a 16 bit (0-65535) e dimensione delle pagine pari a 4 kB. Avremo 16 pagine (0-15) e con una memoria fisica di 32 kB il numero di frame è pari a 8 (0-7). I 4 bit più significativi di un indirizzo a 16 bit rappresentano il numero di pagine : in generale,se la dimensione della pagine è 2^n e la dimensione della memoria è 2^m,il numero di pagine totale è 2^(m-n). Quindi,l'indirizzo virtuale si divide in una parte di **numero di pagine** e il resto in **offset** (12 bit per indirizzare la posizione relativa dell'indirizzo virtuale nel frame). I 4 bit più significativi o gli (m-n) bit più significativi vengono utilizzati come **indice** nella page table : ad esempio 0010 = 2 cioè l'identificativo del frame nella memoria fisica. Nell'immagine contiene 110 (sono 3 bit perché non ci deve essere per forza una relazione tra dimensione dello spazio logico e quella dello spazio fisico → con 8 frame sono necessari 3 bit (0-7)). Ogni riga della page table prende il nome di **voce** della page table o **page table entry** : essa indica il frame in cui si trova e fornisce un **bit presente/assente** che indica se la pagina si trova in memoria centrale e può essere acceduta o se deve essere generato un page fault. L'offset viene copiato poi dopo i 3 bit per costruire l'indirizzo fisico.
Nel caso di indirizzi a 64 bit e pagine di dimensione pari a 4 kB avrò uno spazio degli indirizzi di 2^(64). Dividendo questo spazio in pagine di 4 kB avrò 2^(64)/2^(12) = 2^(52) pagine. Fisicamente non è possibile creare una page table con 2^(52) voci. In pratica,nelle CPU a 64 bit sono usati solo 48 bit meno significativi dell'indirizzo virtuale : i 16 bit più significativi sono tutti 0 per lo spazio degli indirizzi virtuale utente e 1 per quello kernel. Il numero di pagine sarà quindi 2^(48)/2^(12) = 2^(36) pagine.
Ricapitolando,lo scopo della page table è quello di mappare le pagine virtuali sui frame delle pagine.

Struttura di una voce della page table

Ogni riga di una page table,in generale,hanno una dimensione pari a 64 bit :

- i **52 bit** più significativi sono utilizzati per memorizzare il frame number. Si utilizza 52 perché si ha un numero di pagine massimo pari a 2^(52);
- i **12 bit** restanti vengono utilizzati per fornire informazioni sulla pagina.

I 12 bit restanti si dividono in :

- bit **presente/assente** : indica se la voce è valida e può essere usata. Se il bit è 0,la pagina virtuale cui appartiene la voce non è effettivamente in memoria. L'accesso alla voce della tabella delle pagine con questo bit impostato a 0 genera un page fault;
- bit **protezione** : specificano quali tipi di accesso sono consentiti. Nella forma elementare questo campo contiene 1 bit,con 0 che significa lettura/scrittura e 1 per la sola lettura. Un'impostazione più sofisticata ha 3 bit,ogni bit per consentire lettura,scrittura ed esecuzione della pagina;
- bit **supervisor** : correlato al bit protezione ed indica se la pagina sia accessibile soltanto al codice con privilegi,ovvero al kernel oppure ai programmi utente. Qualsiasi tentativo di accesso ad una pagina supervisore da parte di un programma utente causa un page fault;
- bit **modificato** : viene impostato automaticamente dall'hardware quando viene scritta una pagina. Questo bit è valorizzato quando il SO decide di riutilizzare un frame. Se la sua pagina è stata modificata (è sporca) deve essere riscritta sulla memoria non volatile. Se non è stata modificata (pulita) può essere abbandonata,poiché la copia sul disco è ancora valida;
- bit **riferimento** : è impostato ogni qualvolta si faccia riferimento alla pagina,sia in lettura sia in scrittura. Serve ad aiutare il SO a scegliere una pagina da "sfrattare" quando si verifica un page fault;
- bit **caching disabilitato** : permette di disabilitare la cache per la pagina. E' importante per le pagine che mappano sui registri dei dispositivi invece che in memoria. Se il SO si trova in un ciclo veloce in attesa che qualche dispositivo I/O risponda ad un comando appena inviato,è fondamentale che l'hw continui a prelevare la parola dal dispositivo e non usi una vecchia copia presente nella cache. Questo bit non serve alle macchine che hanno uno spazio di I/O separato e che non usano l'I/O mappato in memoria.

Implementazione del paging

In ogni sistema di paginazione devono essere affrontate due questioni principali:

1. La mappatura dall'indirizzo virtuale a quello fisico deve essere veloce;

2. Anche se lo spazio virtuale degli indirizzi è enorme,la page table non deve essere troppo grande.

Il primo punto deriva dal fatto che la mappatura virtuale-fisica deve avvenire ad ogni riferimento alla memoria. Tutte le istruzioni vengono infatti dalla memoria e molte di esse fanno riferimento ad operandi in memoria e quindi per ogni istruzione è necessario fare uno,due o più riferimenti alla page table.
Il secondo punto deriva dal fatto che tutti i computer moderni usano indirizzi virtuali di almeno 32/64 bit. Anche se un processore moderno usa solo 48 dei 64 bit per l'indirizzamento,con una dimensione della pagina di 4 kB,uno spazio di indirizzi di 48 bit ha 64 miliardi di pagine con una page table per ogni programma da 64 miliardi di voci da 64 bit ciascuna.
La page table può essere implementata tramite **registri hardware della MMU**. E' una soluzione efficiente ma necessiterebbe di troppi registri hardware (come visto prima negli esempi con 64 bit). All'avvio di un processo,il SO carica i registri con la page table del processo,presa da una copia che tiene in memoria. Durante l'esecuzione del processo non sono necessari altri riferimenti alla memoria per la page table, Un altro problema si verifica con i context switch : si devono prendere i registri,svuotarli e caricarli con i mapping del nuovo processo che punterà ad un range fisico diverso. Non è una soluzione praticabile,a meno che non abbiamo spazi logici molto piccoli.
La seconda soluzione è avere una page table realizzata **in memoria** (in una struttura dati del kernel) ed è necessario almeno un registro hardware denominato **page-table register (PTBR)** che punti all'inizio della page table per indicare dove si trova. Il problema,in questo caso,ogni volta che si deve effettuare un accesso in memoria si devono in realtà fare due accessi,uno per tradurre l'indirizzo e l'altro per accedere all'istruzione/dato. Un vantaggio è che,ad ogni context switch,la mappatura virtuale-fisica viene cambiata ricaricando un solo registro.Non è in realtà praticabile.
La soluzione praticabile prevede una tabella delle pagine in memoria con l'aggiunta di un **TLB (translation lookaside buffer)**.

Translation lookaside buffer (TLB)

E' un dispositivo hardware (cache associativa) della MMU per mappare gli indirizzi virtuali senza passare dalla page table. Il numero di entry è ridotto (massimo 256 entry) e ognuna di essa ci dice ciascuna pagina virtuale a quale frame corrisponde.
Quando un indirizzo virtuale viene presentato alla MMU per la traduzione,l'hardware prima guarda se il suo numero di pagina virtuale è presente nel TLB confrontandolo in parallelo con tutte le voci. Questa operazione richiede un hardware specializzato,presente in tutte le MMU dotate di TLB. Se trova un riscontro valido e l'accesso non viola i bit di protezione,si ha un TLB hit e il frame è prelevato direttamente dal TLB,senza andare alla tabella delle pagine in memoria. Se il numero di pagina virtuale è presente nel TLB,ma l'istruzione prova a scrivere su una pagina di sola lettura,si genera un errore di protezione. Nel caso in cui invece il

numero di pagina virtuale non è nel TLB si verifica un miss del TLB. La MMU rileva il **TLB miss** ed esegue una normale ricerca nella page table. Quindi sfratta una delle voci dal TLB e la rimpiazza con la voce della page table appena trovata,così se quella pagina viene riutilizzata a breve,la seconda volta si avrà un **TLB hit** invece di un miss. Ogni volta che una voce è eliminata dal TLB,il bit *modificato* viene copiato di nuovo nella voce della page table nella memoria. Gli altri valori,eccetto il bit *riferimento* sono già lì. Quando il TLB viene caricato dalla page table,tutti i campi vengono presi dalla memoria. Se il SO vuole modificare i bit nella voce della page table (ad esempio per rendere scrivibile una pagina in sola lettura),lo farà modificandolo in memoria. Per accertarsi però che l'operazione successiva di scrittura sulla pagina fisica riesca,deve anche eliminare però dal TLB la voce corrispondente con i vecchi bit di autorizzazione. Il TLB quindi contiene alcune voci della page table (quelle più utilizzate). Questo perché solo una piccola parte delle voci della page table viene letta frequentemente. In caso di context switch il TLB viene invalidato-svuotato a meno che le voci del TLB non siano contrassegnate con il PID del processo a cui appartengono (in tal caso si parla di **tagged TLB**). I miss del TLB si dividono inoltre in :

- **soft miss** : il numero di pagina non è nel TLB ma la pagina si trova in memoria (non serve I/O da disco o SSD);
- **hard miss** : il numero di pagina non è nel TLB e la pagina non è in memoria (per prelevare la pagina bisognerà accedere al disco o SSD).

Tabella delle pagine multilivello

Per risolvere il problema della dimensione della page table,invece di avere un'unica tabella con 2^(52) voci,si procede in maniera gerarchica. In prima istanza si prevede una prima tabella con 1024 voci (1024 = 2^(10) = 1K). Queste voci non corrispondono ad entry ma sono dei puntatori ad altre tabelle con 1024 entry. Solo nella tabella di secondo livello si troverà il link alla entry della tabella delle pagine. Si ricorre a questo schema perché la tabella delle pagine potrebbe diventare enorme. Un altro motivo è quello che l'immagine di un processo difficilmente occuperà tutto lo spazio logico a disposizione. L'obiettivo di questo schema è quindi quello di allocare soltanto alcune delle sottotabelle per alcune delle entry che saranno effettivamente utilizzate. L'indirizzo,ad esempio a 32 bit,viene partizionato in questo modo :

- 10 bit per **PT1** : viene utilizzato per accedere alla **top-level** page table. Se si trova una voce che punta ad una tabella di secondo livello è una situazione pacifica,altrimenti l'indirizzo potrebbe essere illegale;
- 10 bit per **PT2** : viene utilizzato se la voce di primo livello è popolata e consente di accedere alla **second-level** page table puntata dalla voce 1 della top-level page table. Ogni tabella di secondo livello permette di indicizzare 4 MB di memoria (2^(10) voci con pagine di 4 KB = 2^(12) → 2^(22) =

2^(2)*2^(20) = 4 MB) e quindi posso indirizzare 1024 pagine. Qui si troverà la entry;
- 12 bit per l'offset.

Su macchine a 64 bit,abbiamo un sistema multilivello con 4 livelli con pagine da 4K e pagine con 2M. Gli ultimi 12 bit sono offset. I 48 bit effettivamente utilizzati vengono raggruppati in 4 gruppi da 9 per indicizzare 4 tabelle da 512 entry (2^(9) = 512). Si possono assorbire 9 bit nell'offset di 12 bit per avere un offset di 21 bit in modo tale da indicizzare la memoria a pagine di 2MB e quindi verranno utilizzate solo 3 tabelle.

TLB miss e page fault

Dopo un TLB miss si ricerca la voce nella page table. Si controlla poi il bit presente/assente (anche chiamato valid/invalid) per capire se la pagina si trova o meno in memoria. Se il bit presente/assente è pari ad **1** significa che la pagina è presente in memoria centrale e che semplicemente la voce non era nel TLB,il quale verrà aggiornato caricando la voce. Se il bit presente/assente è pari a **0** si verifica invece un **page fault**. In questo caso o il processore ha generato un indirizzo illecito e quindi viene fatto un **abort** del processo che ha generato il page fault oppure la pagina cercata esiste ma in quel momento si trova sul disco.

Gestione di un page fault

1. L'hardware esegue l'eccezione nel kernel,salvando il PC nello stack. Nella maggior parte delle macchine alcune informazioni sullo stato dell'istruzione corrente vengono salvate in registri speciali della CPU;
2. Viene avviata una routine di servizio interrupt in codice assembly che salva i registri e altre informazioni volatili per evitare che il SO li elimini,poi chiama il gestore dei page fault;
3. Il SO prova a scoprire quale pagina virtuale sia necessaria. Spesso questa informazione è contenuta in uno dei registri hardware. Se non lo è,il SO deve recuperare il PC,prelevare l'istruzione e analizzarla nel software per capire cosa stava facendo quando si è verificato l'errore;
4. Una volta noto l'indirizzo virtuale che ha causato il page fault,il sistema controlla che l'indirizzo sia valido e che la protezione sia coerente con l'accesso. Qualora non lo sia,viene mandato un segnale al processo o viene terminato. Se l'indirizzo è valido e non è avvenuto alcun errore di protezione,il sistema verifica se c'è un frame libero. Se non ci sono frame liberi,viene eseguito un **algoritmo di sostituzione** delle pagine per liberarne uno;
5. Se la pagina selezionata è sporca,viene schedulata per il trasferimento in memoria non volatile e ha luogo un context switch,sospendendo il processo in page fault e consentendo l'esecuzione di un altro processo fino al completamento del trasferimento su disco o SSD. In ogni caso il frame è

contrassegnato come impegnato per prevenirne l'uso per qualunque altro scopo;

6. Appena il frame è pulito (o immediatamente o dopo essere stato scritto in memoria non volatile),il SO ricerca l'indirizzo su disco dove si trova la pagina necessaria e schedula un'operazione del disco per portarla in memoria. Durante il caricamento della pagina,il processo in page fault è ancora sospeso e viene eseguito,se disponibile,un altro processo utente;
7. Quando l'interrupt del disco o dell'SSD indica che è arrivata la pagina,le page table vengono aggiornate in modo da riflettere la sua posizione e il frame è contrassegnato in stato normale;
8. L'istruzione in errore è riportata allo stato che aveva all'inizio e il PC è ripristinato in modo da puntare a quell'istruzione;
9. Il processo in errore è schedulato e il SO torna alla routine che lo aveva caricato;
10. La routine ricarica i registri e le altre informazioni di stato e ritorna allo spazio utente per riprendere l'esecuzione da dove si era interrotta.

Algoritmi di sostituzione delle pagine

Quando si verifica un page fault,per far spazio alla pagina entrante il SO deve scegliere una pagina da sfrattare. Se la pagina da rimuovere è stata modificata mentre era in memoria,deve essere riscritta sulla memoria non volatile per aggiornare la copia sul disco. Se invece la pagina non è stata modificata (ad esempio perché contiene il codice eseguibile del programma),la copia sul disco è già aggiornata e non c'è bisogno di riscrivere. La pagina da leggere sovrascrive semplicemente la pagina sfrattata. Negli algoritmi di sostituzione,inoltre,sorge il problema del fatto che quando una pagina deve essere rimossa dalla memoria,si deve capire se deve essere una pagina del processo in errore o se deve essere una pagina che appartiene ad un altro processo.
I bit che vengono utilizzati che negli algoritmi di sostituzione e che fanno parte della PTE (page table entry) sono :

- **bit modified (M)** : settato quando la pagina viene modificata (o sporcata) ed è anche chiamato **dirty bit**;
- **referenced (R)** : impostato ad 1 se ho acceduto alla pagina o in lettura o in scrittura. E' anche detto **accessed bit**.

Un algoritmo di sostituzione ottimale è un algoritmo dove ogni pagina viene **etichettata** con il numero di istruzioni da eseguire prima che essa riceva un riferimento. L'algoritmo rimuove la pagina con l'etichetta massima. E' un algoritmo però irrealizzabile dato che il numero di istruzioni dopo il quale la pagina verrà referenziata non è prevedibile.

Algoritmo First-in,First-out (FIFO)

Le pagine presenti in memoria sono organizzate in una coda. Per ogni pagina viene salvato il tempo di caricamento. La pagina più vecchia è posta in testa e la più recente in coda. Ad un page fault,la pagina in testa viene rimossa,mentre la nuova è inserita in coda. Il problema che potrebbe sorgere è che in realtà la pagina più vecchia potrebbe essere ancora utile. Per questo motivo il FIFO è raramente usato nella sua forma più rigida.

Algoritmo Second chance

E' un algoritmo che migliora l'algoritmo FIFO. Esso evita il problema di gettare una pagina usata di frequente controllando il bit R della pagina più vecchia. Il bit R,a differenza del bit modified,viene periodicamente resettato dal SO. Quando si accede ad una page,R viene impostato ad 1. Ad ogni clock interrupt il SO lo resetta : supponiamo che la pagina è stata referenziata e che R = 1;al clock interrupt il bit viene riportato a 0 → se la pagina non viene più acceduta in lettura/scrittura R rimarrà 0 e in futuro il SO si accorgerà che quella pagina non è stata più acceduta. Se R=0,la pagina è vecchia e inutilizzata e viene sostituita immediatamente. Se R=1,il bit viene azzerato,la pagina è posta in fondo all'elenco e il momento in cui è stata caricata in memoria viene aggiornato per farla sembrare appena arrivata. Poi la ricerca continua. Supponiamo di avere le pagine da A ad H mantenute in una lista collegata e ordinate in base al momento in cui sono arrivate nella memoria. Supponiamo si verifichi un page fault all'istante 20. La pagina più vecchia è A,arrivata al momento 0 all'inizio del processo. Se A ha il bit R azzerato,allora è rimossa dalla memoria,sia scrivendola su memoria non volatile se è sporca sia semplicemente scaricandola se è pulita. Se invece il bit R è impostato,A viene portata in fondo alla lista e il suo "momento di caricamento" è reimpostato all'attuale (20). Anche il bit R viene azzerato. La ricerca della pagina adatta prosegue con B. Quello che la seconda chance sta cercando è una vecchia pagina che non sia stata oggetto di riferimenti durante l'ultimo intervallo del clock. Se tutte le pagine hanno avuto riferimenti,la seconda chance degenera in un FIFO puro.

Algoritmo Clock

L'algoritmo second chance è ragionevole ma è inutilmente inefficiente perché fa scorrere di continuo le pagine lungo la lista. Un approccio migliore è quello di tenere tutti i frame su una lista circolare a forma di orologio dove la lancetta indica la pagina più vecchia. Quando si verifica un page fault,la pagina indicata dalla lancetta viene controllata. Se R=0 viene sfrattata,la nuova pagina viene inserita al suo posto nell'orologio e la lancetta viene spostata avanti di una posizione. Se invece R=1,viene azzerato e la lancetta passa alla pagina successiva. Questo processo è ripetuto finché non si trova una pagina con R=0.

Algoritmo Not Recently Used (NRU)

All’avvio di un processo i bit R ed M sono impostati a 0 dal SO. Periodicamente (ad ogni clock interrupt),il bit R è ripulito,per contraddistinguere le pagine che non hanno avuto riferimenti recentemente da quelle che ne hanno avuti. Quando avviene un page fault,il SO ispeziona tutte le pagine e le divide in **4 classi** basate sui valori attuali dei loro bit R ed M :

- **Class 0** : R=0 , M=0;
- **Class 1** : R=0 , M=1;
- **Class 2** : R=1 , M=0;
- **Class 3** : R=1 , M=1.

Le pagine di classe 1 sembrano a prima vista impossibili,ma appaiono quando un clock interrupt azzera il bit R di una pagina di classe 3. I clock interrupt non azzerano il bit M perché questa informazione è necessaria per sapere se la pagina deve essere riscritta sul disco o meno. Azzerare R ma non M produce una page di classe 1. In altre parole una page di classe 1 è stata modificata molto tempo fa e da allora non è stata più toccata.
L’algoritmo **NRU** rimuove una pagina a caso dalla classe non vuota con il numero più basso. E’ chiara l’idea che è meglio rimuovere una pagina modificata che non è stata oggetto di riferimento nemmeno una volta nell’ultimo intervallo del clock che una pagina pulita usata frequentemente. L’algoritmo è facilmente comprensibile,discretamente efficiente da implementare e prestazioni adeguate.

Algoritmo Least recently used (LRU)

Una buona approssimazione dell’algoritmo si basa sull’osservazione che le pagine usate più frequentemente nelle ultime istruzioni lo saranno anche nelle successive. Al contrario,quelle inutilizzate da secoli lo resteranno ancora per molto,quindi in caso di page fault si può sfrattare la pagina rimasta inutilizzata per più tempo. Questa strategia si chiama paginazione **LRU**. E’ un algoritmo fattibile ma non è assolutamente economico. Per implementarlo a pieno è necessario mantenere una lista concatenata di tutte le pagine in memoria,con quelle più usate in testa e quelle meno usate in coda. La difficoltà sta nel fatto che l’elenco deve essere aggiornato ad ogni riferimento alla memoria e ciò può essere supportato tramite contatori hardware incrementati ad ogni riferimento. Trovare una pagina in memoria,cancellarla e poi portarla in testa è un’operazione che costa del tempo,anche se eseguita nell’hardware. Per questo motivo,questo algoritmo non viene utilizzato nella pratica,ma viene utilizzato l’algoritmo **Not Frequently Used (NFU)**. Ad ogni clock interrupt si prende il valore del bit R e si somma ad un **contatore software** associato ad ogni pagina. Se R=0 al clock interrupt il contatore non verrà incrementato,altrimenti il contatore della pagina viene incrementato. In caso di page fault si sostituisce la pagina con il valore del contatore più basso.

Working set (set di lavoro)

La **località di riferimento** ci dice che,durante il suo ciclo di vita,un processo ha accesso ad un numero relativamente piccolo delle sue pagine. C'è quindi una frazione di pagine che viene acceduta molto più frequentemente delle altre. Questa definizione ci porta ad un'altra definizione,ovvero il **working set**,definito come l'insieme delle pagine che un processo sta usando "attualmente". Una volta che il processo è andato a regime,dato che il processo sfrutta solo una porzione piccola delle pagine,esse faranno parte del suo working set che rimarrà stabile nel tempo. Su un sistema di riferimento il working set può essere rappresentato mettendo sull'asse delle ordinate il numero di pagine accedute nelle k istruzioni precedenti e sull'asse delle ascisse il numero di riferimenti k pari al numero di riferimenti/istruzioni. Si ottiene una funzione monotona crescente con un incremento rapido all'inizio (page fault elevato nelle prime istruzioni). Dopo un tot di istruzioni il working set sarà in memoria ed il processo avrà poca necessità di portare nuove pagine in memoria. Quando k diventa grande,il limite di w(k,t) è finito.dato che un programma non può fare riferimento a più pagine di quante ne contenga il suo spazio degli indirizzi e pochi programmi utilizzeranno ogni singola pagina. Si arriva così a due modelli di paginazione diversi :

- **demand paging** : ogni volta che si carica un processo in memoria si carica soltanto la pagina iniziale del processo e man mano che il processo inizia a riferire il suo spazio logico si generano molti page fault che necessiteranno il caricamento delle pagine necessarie fino ad arrivare a regime;
- **pre-paging** : ogni volta che si carica un processo in memoria si carica la pagina iniziale del processo e tutto il suo working set.

Inoltre,un programma che causa page fault ogni poche istruzioni è definito in **thrashing**. L'algoritmo basato sul working set si basa sul fatto che,in caso di page fault,si rimuove una pagina fuori dal working set ovvero una pagina del working set che ha iniziato ad essere utilizzata di meno. In pratica il working set è definito su **base temporale** come l'insieme delle pagine riferite dal processo nell'ultimo lasso di tempo (current virtual time = tempo di CPU effettivamente usata dal processo dal momento del suo avvio. Si utilizza perché è molto più semplice lavorare su tempi più piccoli) **tau**. Nella PTE bisognerà salvare un'informazione in più relativa al **tempo di ultimo utilizzo della pagina**. Ogni PTE avrà quindi questo campo e il bit R che verrà azzerato ad ogni clock interrupt dal SO. Ad ogni page fault si scansiona la page table. Se R=1 il current virtual time viene scritto nell'istante di ultimo utilizzo della PTE della pagina,per indicare che la pagina era in uso al momento del page fault. Dato che la pagina ha avuto un riferimento durante l'ultimo ciclo del clock,è chiaramente nel working set e non è candidata alla rimozione (si suppone che tau sia un multiplo del clock). Se invece R=0 significa che la pagina non ha avuto riferimenti durante il ciclo di clock attuale e può essere candidata alla rimozione. Si calcola quindi l'**età della pagina** pari a : **età = current virtual time - istante di**

ultimo utilizzo. Questo valore viene confrontato con *tau*. Se **età <= tau** significa che la pagina è stata utilizzata nell'ultimo *tau* di tempo e quindi viene risparmiata,ma la pagina con età maggiore viene contrassegnata; se invece **età > tau** significa che la pagina è uscita dal working set e viene quindi sostituita. Nel peggiore dei casi,tutte le pagine hanno avuto un riferimento nell'ultimo ciclo del clock (hanno tutte R=1),perciò ne viene scelta una a caso per la rimozione,preferibilmente,se esiste,una pagina pulita.

Area di scambio

Abbiamo visto come avviene la scelta della pagina da rimuovere. Non è stato però detto dove sia posta nella memoria non volatile dopo essere stata paginata fuori dalla memoria. L'algoritmo più semplice per allocare in memoria non volatile lo spazio delle pagine è avere una partizione speciale per lo scambio sul disco,o ancora meglio su un dispositivo di memorizzazione separato dal file system in modo tale da bilanciare il carico di I/O. I sistemi UNIX funzionano tradizionalmente così. Questa partizione ha un file system particolare,che elimina tutto il sovraccarico della conversione degli offset dei file in indirizzi di blocchi usando invece i numeri dei blocchi relativi all'inizio della partizione.
All'avvio del sistema questa partizione di scambio è vuota ed è rappresentata in memoria come una singola voce che ne indica l'inizio e la dimensione. All'avvio del processo (nello schema più semplice),nella partizione viene riservata una parte di dimensione pari a quella del processo,e l'area restante viene ridotta di questa quantità. All'avvio di nuovi processi,ad ognuno è assegnata una parte della partizione di scambio di dimensione uguale alla loro immagine. Quando terminano,lo spazio da essi occupato in memoria di massa viene liberato.
A ciascun processo è associato l'indirizzo in memoria non volatile della sua area di scambio,ossia dove si trova la sua immagine nella partizione di scambio. Questa informazione è tenuta nella tabella dei processi. L'indirizzo in cui scrivere una pagina si ottiene sommando l'offset della pagina nel suo spazio virtuale degli indirizzi all'inizio dell'area di scambio. Prima che un processo possa partire,però,l'area di scambio deve essere inizializzata. Ciò può essere fatto copiando l'intera immagine del processo nell'area di scambio,così da poterla utilizzare quando è necessario oppure si carica l'intero processo in memoria per poi paginarlo quando è necessario. Questo modello presenta un problema : i processi possono aumentare di dimensione durante l'esecuzione e quindi può essere meglio riservare aree di scambio separate per il testo,dati e stack e consentire a ciascuna di loro di essere composta da più parti della memoria non volatile.
L'estremo opposto è di non allocare niente in anticipo e allocare lo spazio in memoria non volatile per ciascuna pagina quando viene scambiata sul disco,e deallocarlo quando viene riportata in memoria. In questo modo i processi in memoria non sono vincolati a un dato spazio di scambio. Lo svantaggio è che per tener traccia di ogni pagina in memoria non volatile è necessario avere un indirizzo del disco in memoria. Ci deve essere quindi una tabella per ogni processo che indichi

dove si trova ogni pagina nella memoria non volatile (**disk map**). Sul disco si hanno solo le pagine che non stanno in memoria.
Quindi,nella paginazione con area di scambio **statica**,l'area di scambio sul disco è grande quanto lo spazio logico del processo e ogni pagina ha una posizione assegnata fissa nella quale viene scritta quando è sfrattata dalla memoria principale. Una pagina in memoria ha sempre una copia "ombra" sul disco,ma questa copia potrebbe non essere aggiornata nel caso in cui la pagina sia stata modificata dopo il suo caricamento. Le pagine in grigio nella memoria indicano pagine non presenti in essa mentre quelle grigie sul disco sono sostituite in teoria dalle copie nella memoria;nel caso una pagina in memoria debba essere scambiata verso il disco e non sia stata modificata dal momento del suo caricamento,sarà usata la copia ombra. Nel caso in cui si verifica un page fault,sarà solo necessario sapere dove inizia nel disco l'area dedicata al processo per recuperare la pagina necessaria.
Nella paginazione con area di scambio **dinamica** le pagine non hanno un indirizzo su disco fisso. Quando una pagina viene scambiata verso il disco,viene scelta "al volo" una pagina libera del disco e la mappa del disco (che ha spazio per un indirizzo del disco per ogni pagina virtuale) viene aggiornata di conseguenza. Una pagina in memoria non ha alcuna copia sul disco e le voci delle pagine nella disk map contengono un indirizzo del disco non valido o un bit che le contrassegna come non in uso.

Cenni sulla segmentazione

E' un approccio alternativo alla paginazione per suddividere lo spazio degli indirizzi. Invece che dividere lo spazio degli indirizzi in pagine da 4K,un programma è visto come un **insiemi di segmenti**. Un **segmento** è un'unità logica come:

- un programma principale;
- funzioni;
- variabili locali e globali;
- pile;
- tabella dei simboli;
- vettori;
- …

Invece di avere uno spazio logico monodimensionale,esso è frammentato in segmenti che corrispondono ad entità logiche che hanno significato all'interno del programma. Ogni segmento consiste di una sequenza lineare di indirizzi,da 0 ad un certo massimo. La lunghezza dei segmenti può essere qualsiasi,da 0 al massimo consentito. Segmenti diversi hanno dimensioni diverse e la loro lunghezza può variare durante l'esecuzione. Si ricorre a questo metodo per non obbligare il programmatore a gestire l'espansione e la contrazione delle tabelle (testo,simboli,costanti,stack). La lunghezza del segmento dello stack può essere aumentata ogni volta che viene inserito qualcosa al suo interno e diminuita ogni volta

che qualcosa viene estratta. Dato che ogni segmento costituisce uno spazio degli indirizzi separato,segmenti diversi possono crescere o decrescere indipendentemente senza influenzarsi l'un l'altro. Se uno stack in un segmento ha bisogno di altro spazio degli indirizzi per crescere,può averlo perché nel suo spazio non c'è altro contro cui andare a sbattere. Raramente capita che i segmenti si riempiano,essendo molto grandi. L'**indirizzo logico** diventa bidimensionale ed è costituito dall'**identificativo del segmento** e un **offset** che indica la posizione all'interno del segmento. Non abbiamo contiguità e c'è un equivalente della page table chiamata **tabella dei segmenti**. Il SO tramite la tabella dei segmenti conosce ogni segmento a quale indirizzo fisico corrisponde (colonna base). La traduzione di un indirizzo logico in indirizzo fisico avviene in questo modo : dato un indirizzo logico **<s,d>** con **s = numero segmento** e **d = offset**,si usa **s** come indice della tabella per ricavare l'indirizzo **base**;si verifica poi che **d < limite** → se è vero allora si effettua la somma **indirizzo fisico = base + d** altrimenti si solleva un'eccezione.
La segmentazione facilita inoltre la condivisione di procedure o di dati fra molti processi. Un esempio comune è quello delle librerie condivise. In un sistema segmentato,ad esempio,la libreria grafica può essere posta in un segmento e condivisa da più processi eliminando la necessità di averla nello spazio degli indirizzi di ogni processo.

Linux address space

Gli eseguibili,in generale,sono parzialmente linkati : ci sono riferimenti ad una serie di librerie che si chiamano librerie condivise (in linux so → shared object e in windows le dll),le quali non sono linkate staticamente ma viene in realtà creato un eseguibile parzialmente linkato e il collegamento tra l'invocazione ad una funzione della libreria dinamica con la libreria vera e propria avviene soltanto a tempo di esecuzione. Nella memoria fisica,il codice della libreria dinamica,viene posto una volta e in seguito gli spazi di indirizzamento virtuali punteranno alla stessa parte della memoria fisica. Le librerie statiche fanno già invece parte dell'eseguibile sul disco. Tipicamente un modulo oggetto è costituito da una sezione testo (codice) e da una sezione dati. La sezione dati,che si riferisce a dati non locali (definiti fuori da ogni blocco come stack e heap),prevede una distinzione tra dati inizializzati,dati non inizializzati. Il linker prende tutti i moduli oggetto e li linearizza in un unico spazio creando qualcosa che ha solo riferimenti a sé stesso e crea così un eseguibile. Il contenuto dell'eseguibile dovrà essere inserito in uno spazio di indirizzamento che verrà creato dal kernel nel momento in cui si deve eseguire il programma. Gli indirizzi più bassi sono occupati dall'area testo e dai dati dell'eseguibile,presi proprio dall'eseguibile al momento del caricamento. Lo stack e heap non fanno parte dell'eseguibile e vengono allocati dal kernel man mano che si crea il processo e si crea lo spazio di indirizzamento. Tra heap e stack c'è un gap,con l'heap che cresce verso l'alto e lo stack verso il basso. In questo gap vengono create regioni chiamate **memory-mapped**,dove verranno mappate le librerie condivise. Ciò è utile per fare in

modo che se due processi utilizzino le stesse funzioni di libreria,queste vengano condivise caricandole una volta sola invece che ad ogni chiamata.
In Linux,è possibile consultare tante informazioni riguardo i processi in esecuzione,le quali sono contenute in un file-system speciale chiamato **ls/proc**. In esso abbiamo una cartella per ogni PID ed in particolare,per quanto riguarda lo spazio di indirizzamento,ci interessa visualizzare il file ***maps*** tramite comando ***cat proc/PID/maps***. Esso ci permetterà di vedere come sono organizzati gli indirizzi logici del processo.
Se lanciassimo il programma con *strace*,vediamo subito una ***execve***,usata per caricare l'eseguibile e ciò crea nello spazio di indirizzamento una prima sezione. Il kernel deve poi creare diverse aree di memoria come heap e stack,con chiamate di sistema come *mmap*,dove prenderà le librerie e le mapperà nello spazio di indirizzamento.
Lo spazio degli indirizzi è diviso in due "tronconi" :

- **kernel (mai sostituito);**
- **processo (sostituito ad ogni context switch)**.

Di tutto lo spazio logico a 32/64 bit l'utente può accedere soltanto ad una parte di esso mentre l'altra parte è riservata al kernel. Ciò perché la page table,a partire da un certo punto in poi,è la stessa per tutti i processi dato che il codice del kernel rimane sempre uguale. La parte inferiore è diversa per ogni processo e contiene descrittori del processo,le page table,stack kernel ecc. Per evitare che un utente tenti di accedere ad un indirizzo riservato al kernel viene utilizzato nella PTE il **bit supervisor**,che se posto ad 1,indica che la page è riservata al kernel e quindi si verifica un abort.

Nei sistemi a 32 bit l'ultimo GB è riservato per il kernel. Il "grosso" (immagine kernel e frame fisici 896MB) degli indirizzi dell'ultimo GB sono mappati linearmente con la memoria fisica. Per la traduzione basta quindi sommare o sottrarre una costante. Il fatto che nello spazio degli indirizzi logici ci siano anche quelli del kernel potrebbe non essere un problema grazie al bit supervisor. Si è poi scoperto che in realtà l'utente,pur emettendo indirizzi che non gli appartengono,riusciva a leggere comunque dati del kernel dato che la verifica del bit supervisor avveniva qualche colpo di clock dopo (si riusciva a leggere una parte della cache del kernel). Questa vulnerabilità è chiamata **melt-down**. In alcuni sistemi,per risolvere questo problema,viene implementato un meccanismo chiamato **Kernel Page Table Isolation (KPTI)** che prevede due tabelle delle pagine separate,una del kernel con tutto lo spazio degli indirizzi e quella dei processi con lo spazio utente con uno **stub** per entrare e uscire in maniera efficiente dal kernel,senza vedere nello spazio utente gli indirizzi riservati al kernel. → NON LO CHIEDE.

File system

Quando si parla di file system si parla del salvataggio delle informazioni a **lungo termine**. La RAM è volatile e ciò significa che tutto ciò che contiene sopravvive finché il processo non termina. Molto di ciò che viene prodotto in un computer generalmente ci servirà in futuro e quindi deve essere salvato in memoria secondaria. I requisiti di questo salvataggio sono :

- possibilità di salvare enormi quantità di informazioni;
- le informazioni devono permanere oltre la fine del processo che le usa;
- le informazioni devono poter essere acquisite,eventualmente in contemporanea,da più processi.

La tecnologia per memorizzare i dati in modo permanente è quella della memoria secondaria. Abbiamo in particolare **dischi magnetici (HDD)** e **unità a stato solido (SSD)**. Assumiamo al momento una memoria secondaria come un'area di **storage** in cui salvare i dati,composta da una serie di **blocchi** ed organizzata come un array lineare. Ciascun blocco ha un identificativo e l'interfaccia dello storage ci dà due primitive : **scrivi** un blocco k o **leggi** un blocco k.
Ad alto livello a queste primitive corrispondono system call come *opne,read,write,readdir…* . L'obiettivo è quello di vedere come raggiungere all'interno del disco i blocchi che contengono l'informazione desiderata.

Un **file** è una unità logica di informazioni,correlate e registrate nella memoria secondaria (in modo persistente finché l'utente non decide di cancellarlo),cui è assegnato un identificativo (nome che servirà al file system per rintracciarlo) e creato da un processo. Sono un **meccanismo di astrazione** per salvare informazioni sul disco e leggerle in seguito.

Il **file system** è la parte del SO responsabile per la gestione e l'organizzazione dei file. Alcuni compiti di un file system sono :

- implementazione e persistenza dei file;
- recupero-accesso dei file;
- un meccanismo di naming;
- condivisione e protezione dei file;
- …

File e directory (vista "utente")

Vedremo come sono usati i file e quali proprietà hanno.

File naming

Quando un processo crea un file,gli attribuisce un nome. Quando il processo termina,il file continua ad esistere ed altri processi possono accedervi usando il suo nome. Le regole esatte per la denominazione dei file possono variare a seconda del sistema,ma tutti i sistemi operativi attuali considerano validi i nomi composti da stringhe di lettere. Alcuni file system,come quello usato in MS-DOS limitavano la lunghezza dei nomi dei file ad un massimo di 8 caratteri (**FAT12**) mentre la maggior parte dei sistemi moderni supporta nomi fino a 255 caratteri (**Ext4**). Alcuni file system distinguono tra maiuscole e minuscole,altri no. Unix è uno di questi mentre MS-DOS,ad esempio,non fa distinzione.
La parte che segue il punto,ad esempio in *prog.c*,è chiamata **estensione** del file e generalmente indica una particolarità del file. In MS-DOS,ad esempio,i nomi dei file vanno da 1 a 8 caratteri,più un'estensione opzionale da 1 a 3 caratteri. In UNIX,l'eventuale dimensione dell'estensione dipende dall'utente. In alcuni sistemi (ad esempio in tutti i tipi di UNIX) le estensioni sono solo convenzioni e non sono forzate dal SO. Un file chiamato *file.txt* potrebbe essere un file di testo,ma l'estensione vale più da promemoria per l'utente che per trasmettere al computer una qualsiasi informazione reale. Un compilatore C,invece,potrebbe effettivamente richiedere l'estensione *.c* per i file da compilare,rifiutandosi di farlo se non la presentano,ma ciò non interessa al SO. Windows,invece,considera le estensioni e dà loro un significato specifico.

Struttura dei file

I file possono essere strutturati in tanti modi diversi. Le possibilità più comuni sono 3:

1. file visto come una **sequenza non strutturata di byte**. Il SO non sa cosa c'è nel file e non gli interessa. Tutto ciò che vede sono byte. Qualsiasi significato deve essere imposto dai programmi a livello utente (assumono significato diverso a seconda che siano eseguibili,testo ecc.). Sia UNIX sia Windows utilizzano questo approccio. Il fatto che il SO consideri i file come sequenze di byte fornisce la massima flessibilità. I programmi utente possono mettere ciò che vogliono nei loro file e chiamarli nel modo che preferiscono. Il SO non offre aiuti e non crea ostacoli.
2. file visto come una **sequenza di record** di lunghezza fissa,ciascuno con una certa struttura interna. L'idea di un file composto da una sequenza di record è incentrata sul fatto che l'operazione di lettura restituisca un record e l'operazione di scrittura sovrascriva o aggiunga un record;
3. file visto come un **albero di record**,non necessariamente della stessa lunghezza,ognuno contenente un **campo chiave** in una posizione fissa del record. L'albero è ordinato sul campo chiave per permettere una ricerca rapida per una chiave particolare. Non si ricerca quindi il record successivo ma quello con la chiave desiderata.

Tipi di file

Molti SO supportano diversi tipi di file. I file si dividono in :

- **file normali** : sequenze di byte che contengono informazioni dell'utente (dati,programmi sorgente,programmi oggetto). I file normali si dividono in **file ASCII** e file **binari**. I file ASCII sono composti da righe di testo. In alcuni sistemi ciascuna riga termina con un carattere di "a capo",in altri è usato il carattere di "nuova riga". Le righe non sono necessariamente della stessa lunghezza. Il vantaggio dei file ASCII è che possono essere visualizzati e stampati così come sono e possono essere corretti in qualsiasi editor di testo. I file binari,invece,non sono file ASCII. Stampandoli si ottiene solo un elenco di caratteri a caso (es. eseguibile). Generalmente hanno una struttura intera conosciuta dai programmi preposti ad usarli. Se si prova ad eseguire un file di testo avremo un problema : non si hanno i permessi (facendo ls -la possiamo vedere tutte le informazioni del file → non ha permessi di esecuzione e per darglieli si usa *chmod* dandogli la terna 777). Se rendessi l' eseguibile *a.out* come file di testo tramite *cp a.out a.txt* esso rimarrà comunque con i permessi di esecuzione,e posso comunque eseguirlo come ./a.txt. C'è poi un comando chiamato *file* che dando un file ci fornisce informazioni su cosa esso rappresenta. Un file può essere "zippato" tramite ad esempio *zip a.zip a.out*. Lo zip di un eseguibile,per essere eseguito deve avere i permessi (glieli diamo come prima 777),ma non può essere eseguito perché è un file binario che è pensato per essere processato dal programma che effettua la decompressione del file. Non può essere neanche stampato a video perché abbiamo i byte che rappresentano il risultato della compressione che l'utility ha effettuato sul file (non sono ASCII);
- **directory** : file di sistema usati per mantenere la struttura del file system. Essi permettono cioè di ricostruire l'associazione tra il nome del file e fisicamente dove è memorizzato sul disco;
- **file speciali** : si dividono in **file speciali a caratteri**,relativi all'I/O e usati per modellare i dispositivi seriali di I/O,come terminali,stampanti e reti e in **file speciali a blocchi** usati per modellare i dischi.

I file binari eseguibili sono costituiti da cinque paragrafi : intestazione (**header**),testo,dati,bit di rilocazione (**relocation bit**) e tabella dei simboli. L'header parte con un **numero magico**,che identifica il file come eseguibile (per prevenire l'esecuzione accidentale di un file non in questo formato). Abbiamo poi le dimensioni delle varie parti del file,l'indirizzo da cui parte l'esecuzione e alcuni indicatori (flag). Dopo l'header abbiamo il testo e i dati del programma vero e proprio,caricati in memoria e rilocati tramite i bit di rilocazione. La tabella dei simboli è usata per il debug.

Accesso ai file

Abbiamo due tipi di accesso ai file : **accesso sequenziale** e **accesso casuale**. Nei file con accesso sequenziale,un processo legge i byte o i record in ordine,a partire dal principio;non può né saltare,né leggere in ordine sparso. I file con accesso sequenziale possono essere riavvolti,in modo da poterli leggere tutte le volte che si vuole. Erano comodi quando i supporti di memorizzazione erano i nastri magnetici,prima dei dischi. Nei file con accesso casuale,invece,possiamo leggere i byte o i record senza un ordine,o accedere ai record secondo una chiave anziché in base alla posizione. Per specificare dove cominciare a leggere possono essere utilizzati due metodi. Nel primo ogni operazione ***read*** fornisce la posizione del file dalla quale iniziare a leggere. Nel secondo è fornita un'operazione speciale,***seek***,per impostare la posizione corrente. Dopo una *seek* il file può essere letto sequenzialmente dalla posizione definita come corrente.

Attributi dei file

Ogni file ha un nome e i propri dati. Tutti i SO associano ulteriori informazioni a ciascun file,ad esempio la data e l'ora in cui è stato modificato l'ultima volta e la dimensione. Queste voci del file sono dette **attributi** (o **metadati**). Abbiamo attributi che si riferiscono alla protezione e indicano chi può accedervi. In alcuni sistemi l'utente deve fornire una password per accedere ad un file,nel qual caso la password deve essere uno degli attributi. I flag sono bit o campi corti che controllano o abilitano alcune proprietà specifiche. I file nascosti,ad esempio,non compaiono nell'elenco di tutti i file. Il flag archivio è un bit che tiene traccia del fatto che sia stata eseguita recentemente una copia di sicurezza (backup) del file. Il programma di backup lo azzera e il SO lo imposta ad 1 se il file viene modificato in modo tale che il programma di backup capisca quali file necessitano di un backup. Il flag temporaneo permette ad un file di essere contrassegnato per l'eliminazione automatica quando il programma che l'ha creato termina.

Operazione sui file

1. **create** : il file è creato senza dati. Lo scopo della chiamata è annunciare la presenza del file e impostarne alcuni attributi;
2. **delete** : usata per eliminare un file quando non serve più;
3. **open** : prima di usare un file,un processo deve aprirlo. Lo scopo di questa chiamata è permettere al sistema di portare nella memoria principale gli attributi e la lista degli indirizzi del disco per un accesso rapido nelle chiamate a seguire;
4. **close** : quando tutti gli accessi sono terminati,gli attributi e gli indirizzi del disco non servono più,quindi il file viene chiuso per liberare lo spazio delle tabelle interne;
5. **read** : i dati sono letti dal file. Il chiamante deve specificare quanti dati sono necessari e deve fornire un buffer in cui metterli;

6. **write** : i dati sono nuovamente scritti sul file,generalmente dalla posizione attuale. Se la posizione attuale è la fine del file,la dimensione del file aumenta;se è nel mezzo i dati esistenti sono sovrascritti e persi per sempre;
7. **append** : aggiunge dati alla fine del file;
8. **seek** : riposiziona il puntatore del file in una sua posizione specifica;
9. **get attributes** : per lavorare,i processi hanno spesso bisogno di leggere gli attributi dei file. Ad esempio il programma UNIX *make*,quando chiamato,esamina la data e l'ora di modifica di tutti i file sorgente e dei file oggetto;
10. **set attributes** : alcuni attributi possono essere impostati dall'utente e possono essere modificati dopo la creazione del file con questa chiamata di sistema;
11. **rename**.

Directory

Una **directory** è un file speciale creato con l'obiettivo di risolvere le corrispondenze tra il nome del file in formato testuale e il suo identificativo.
La forma più semplice di sistema di directory è di avere una sola directory contenente tutti i file,chiamata **directory principale (root directory)**. Sui primi computer questo sistema era comune. Il vantaggio di questo schema è la semplicità e la capacità di localizzare i file rapidamente,dato che abbiamo un solo posto in cui cercare. Il singolo livello è adeguato a semplici applicazioni dedicate (come RFID),ma per gli utenti moderni con migliaia di file sarebbe impossibile rintracciare qualcosa se tutti i file si trovassero in un'unica directory. Serve quindi un modo per poter raggruppare i file correlati. Viene così introdotta una **gerarchia** e cioè **directory ramificate ad albero**. Inoltre,se più utenti condividono un file server comune,ogni utente può avere una directory principale privata per la propria gerarchia. Ad esempio abbiamo una root con 3 directory A,B,C di tre utenti differenti ed ognuna di esse presenta delle sotto-directory (subdirectory).
Quando il file system è organizzato in un albero di directory,i nomi dei file vanno specificati in qualche modo. Vengono usati due metodi. Il primo è quello di assegnare ad ogni file un **nome di percorso assoluto** composto dal percorso che inizia dalla directory principale e arriva al file. In UNIX i componenti del percorso sono divisi tramite **/**,in Windows tramite **** e in MULTICS **>**. L'altro metodo è quello del **nome di percorso relativo**. E' usato congiuntamente al concetto di **directory di lavoro**,anche detta **directory corrente**. Un utente può designare una directory come directory corrente,e in quel caso tutti i nomi di percorso che non cominciano con la directory principale sono considerati relativi alla directory corrente. Molti SO che supportano un sistema di directory gerarchico hanno due voci speciali in ogni directory, "**.**" e "**..**" che servono a riferirsi rispettivamente alla directory corrente e alla directory genitore,con l'esclusione della directory radice che anche nel caso di .. fa riferimento a se stessa.

Operazioni su directory

Le chiamate di sistema consentite per la gestione delle directory differiscono maggiormente a seconda del sistema rispetto alle chiamate per i file. Alcuni esempi sono :

1. **create** : viene creata una directory vuota,a parte punto e puntopunto che vengono inserite automaticamente dal programma *mkdir*;
2. **delete** : viene eliminata una directory. Una directory può essere eliminata solo se è vuota. Una directory contenente punto e puntopunto è considerata vuota dal momento che non sono eliminabili;
3. **opendir** : le directory possono essere lette. Ad esempio,per elencare tutti i file in una directory,un programma la apre per leggere i nomi;
4. **closedir** : quando una directory è stata letta,deve essere chiusa per liberare lo spazio delle tabelle interne;
5. **readdir** : restituisce la voce successiva all'interno di una directory aperta. In precedenza si poteva leggere le directory tramite la chiamata *read*,ma questo approccio ha lo svantaggio di obbligare il programmatore a conoscere e ad avere a che fare con la struttura interna delle directory. *readdir*,invece,restituisce sempre una voce in formato standard,non importa quale struttura di directory sia usata;
6. **rename**;
7. **link** : permette ad un file di apparire in più di una directory. Specifica un file esistente e un nome di percorso e crea un collegamento tra il file esistente e il nome specificato nel percorso.
8. **unlink** : viene rimossa una voce dalla directory. Se il file che viene scollegato è presente in una sola directory viene eliminato dal file system.

Implementazione del file system

Agli implementatori,a differenza degli utenti,interessa il modo in cui sono memorizzati file e directory,com'è gestito lo spazio su disco e come far funzionare tutto in modo efficace ed affidabile.

Implementazione dei file

L'aspetto più importante dell'implementazione della memorizzazione dei file è tener traccia di quali blocchi del disco siano associati ad un determinato file. I metodi di implementazione che vedremo sono :

- **allocazione contigua**;
- **allocazione a liste concatenate**;
- **allocazione a liste concatenate con tabella in memoria**;
- **i-node**.

E’ importante dire che un **blocco** è l’unità minima con cui leggiamo/scriviamo.

Allocazione contigua

E’ lo schema di allocazione più semplice. Consiste nel memorizzare ciascun file come una sequenza contigua di blocchi del disco. In questo modo,su un disco con blocchi da 1K,ad un file di 50KB sarebbero allocati 50 blocchi consecutivi. Con blocchi di 2KB ne sarebbero allocati 25 consecutivi. Quando il file viene salvato,il SO alla base dello stack scriverà i vari blocchi in modo consecutivo. Ogni file parte dall’inizio di un nuovo blocco,quindi se un file fosse di 3 blocchi e 1/2 ,parte dello spazio alla fine dell’ultimo blocco sarebbe sprecato. L’allocazione contigua presenta due vantaggi importanti. Innanzitutto è semplice da implementare,poiché per tenere traccia della posizione dei blocchi di un file bastano due numeri : indirizzo sul disco del primo blocco e numero di blocchi del file. Dato il numero del primo blocco,il numero di qualunque altro blocco si ricava con una somma. Il secondo vantaggio è quello delle prestazioni,le quali in lettura sono eccellenti anche sul disco perché l’intero file può essere letto dal disco con una sola operazione. Serve una sola ricerca (del primo blocco),dopo la quale non servono altre ricerche.
Lo svantaggio è quello della **frammentazione esterna**. Quando un file viene eliminato,vengono liberati i suoi blocchi,e ciò lascia un intervallo di blocchi liberi sul disco. Il disco non viene immediatamente compattato per chiudere il vuoto,visto che ciò potrebbe comportare la copia di tutti i blocchi che seguono il vuoto,potenzialmente milioni. Di conseguenza,alla fine il disco risulta composto da file e vuoti. Inizialmente questa frammentazione non è un problema,dato che ogni nuovo file può essere scritto alla fine del disco,di seguito al precedente. Tuttavia lo è nell’eventualità che il disco si riempia e diventi necessario compattare il disco,operazione dal costo proibitivo,o riutilizzare lo spazio libero nei vuoti. Quest’ultima operazione richiede il mantenimento di una lista di vuoti,il che è fattibile. Tuttavia,quando viene creato un nuovo file sarebbe necessario conoscere la sua dimensione finale per poter scegliere un vuoto abbastanza grande. In questo scenario,quando l’utente avvia un’applicazione,la prima cosa che il programma richiede è di quanti byte sarà il file. Se alla fine il numero dato si rivela troppo piccolo,il programma potrebbe terminare perché il vuoto sul disco sarebbe pieno e non ci sarebbe posto per il resto del file. I file quindi non possono crescere di dimensione : se rimuovo dei pezzi e voglio crescere di dimensione si rischia di sforare in altri file.
La deframmentazione (shift) potrebbe risolvere il problema.

Allocazione a liste concatenate

Il secondo metodo per memorizzare i file è configurare ciascuno come una lista concatenata di blocchi del disco. La prima parte di ciascun blocco è usata come puntatore al successivo,il resto del blocco è per i dati. Diversamente dall'allocazione contigua,questo metodo permette di usare ogni blocco del disco. Non si perde spazio per la frammentazione del disco,eccetto per la **frammentazione interna** dell'ultimo blocco che contiene un valore per indicare che non punta a nulla. La voce delle directory deve memorizzare semplicemente l'indirizzo su disco del primo blocco ed eventualmente quello dell'ultimo blocco. Quindi questo approccio ha come vantaggi l'assenza di frammentazione esterna e un minor costo di allocazione.
Tra gli svantaggi si ha :

- possibilità di errore se un link viene danneggiato,dato che si perde il puntatore al blocco successivo;
- costo della ricerca di un blocco : leggere un file sequenzialmente è semplice,però l'accesso casuale è estremamente lento. Per arrivare al blocco *n*,il SO deve partire dal principio e leggere prima *n-1* blocchi,uno alla volta. E' chiaro che tante operazioni di lettura comportano tempi lunghissimi dato che il file può essere spezzettato in più parti del disco;
- overhead dello spazio : la quantità di spazio per i dati di un blocco non è più una potenza di due,poiché il puntatore occupa alcuni byte. Dato che molti programmi leggono e scrivono in blocchi la cui dimensione è una potenza di due,avere una dimensione peculiare risulta meno efficiente. Se i primi byte di ciascun blocco sono occupati da un puntatore al blocco successivo,leggere l'intera dimensione del blocco richiede l'acquisizione e la concatenazione di informazioni da due blocchi del disco,il che genera ulteriore overhead dovuto alla copia.

Allocazione a liste concatenate con una tabella in memoria

Gli svantaggi dell'allocazione a liste concatenate possono essere eliminati prendendo la parola puntatore di ogni blocco del disco e ponendola in una tabella in memoria,chiamata **file allocation table (FAT)**. La tabella è vista come un array : gli indici contengono il puntatore al blocco successivo. Ad esempio se un file A ha una sequenza 4,7,2,10,12 avremo all'indice 4 il puntatore a 7,all'indice 7 il puntatore a 2 e così via. La sequenza termina con un indicatore speciale,come -1,non valido come numero di blocco. Usando questa organizzazione,l'intero blocco è disponibile per i dati e l'accesso casuale è molto più semplice. Anche se si deve seguire la catena per trovare un dato offset all'interno del file,essa è interamente in memoria,così può essere seguita senza fare riferimento al disco. Come nel metodo precedente,è sufficiente che la voce della directory contenga un singolo intero,quello del blocco iniziale,per localizzare tutti i blocchi. Lo svantaggio principale di questo metodo è che l'intera tabella deve restare sempre in memoria. Con un disco da 1 TB e blocchi da 1 KB,la tabella deve avere miliardi di voci,una per ciascun blocco del disco (2^40/2^10

posizioni). Le dimensioni della tabella,quindi,è uno svantaggio per dischi di grosse dimensioni : crescerà in maniera proporzionale alla dimensione del disco,cioè,più è grande il disco più la tabella sarà grande. Un altro svantaggio è quello che,a sistema fermo la FAT si trova sul disco e quando inizia l'esecuzione dovrà essere portata in memoria,toglierà spazio utile (eventualmente dovrà essere paginata).

i-node

L'**index-node** è una struttura dati che elenca gli attributi e gli indirizzi dei blocchi dei file. Ad ogni file viene associato un i-node,che permette di trovare tutti i blocchi di quel file. Il grande vantaggio di questo schema rispetto alle liste concatenate con FAT è che l'i-node deve essere in memoria solo quando il file corrispondente è aperto. Se inoltre ciascun i-node occupa *n* byte e può essere aperto un numero massimo *k* di file in contemporanea,la memoria occupata dall'array che contiene gli i-node è di soli *k*n* byte per tutti i file aperti. Occorre riservare in anticipo solo questa quantità di spazio. Questo array solitamente è molto più piccolo dello spazio occupato dalla FAT. La FAT è infatti proporzionale alla dimensione del disco e quindi se il disco ha *n* blocchi,la tabella avrà bisogno di *n* voci e all'aumentare della capacità del disco,la FAT cresce in proporzione lineare. Lo schema degli i-node invece richiede in memoria un array proporzionale al numero massimo di file che potrebbero essere aperti contemporaneamente e quindi la dimensione del disco è ininfluente. Uno dei problemi degli i-node è quello che riguarda la dimensione del file. Ogni i-node ha infatti spazio per un numero fisso di indirizzi del disco e questa potrebbe essere superata dalla dimensione di un file. La soluzione è quella di riservare l'ultimo indirizzo del disco non per un blocco di dati,ma per l'indirizzo di un blocco contenente ulteriori indirizzi di blocchi del disco. (1)

Implementazione delle directory

La funzione principale del sistema delle directory è mappare il nome ASCII del file sulle informazioni necessarie per localizzare i dati. Una questione strettamente correlata è dove debbano essere memorizzati gli attributi. Una possibilità è memorizzarli direttamente nella voce della directory. In questo modello una directory è composta da una lista di voci a dimensione fissa,una per file,contenente un nome di file (di lunghezza fissa),una struttura degli attributi del file e uno o più indirizzi del disco (sino ad un certo massimo) che indicano dove sono i blocchi del disco. Questo approccio è il caso di **FAT 12/16** :

- 8 caratteri per il nome;
- 3 per l'estensione;
- attributi : numero del primo blocco,tempo,data,dimensione…

Un'altra possibilità è memorizzare gli attributi negli i-node,per i sistemi che li usano,anziché inserire tutto nelle voci delle directory. In questo caso la voce della

directory può essere più breve. E' il caso del mondo **UNIX** : abbiamo il nome del file su 14 caratteri e tutto il resto è contenuto nell'i-node associato al file,identificato dal numero indicato nella voce stessa della directory. (2)

File system V7 UNIX (1) (2)

Il file system ha la forma di un albero che nasce dalla directory principale,con l'aggiunta di link,a formare un grafo aciclico orientato. I nomi dei file hanno fino a 14 caratteri e possono contenere qualunque carattere,escluso "/" e NUL (vale 0 ed è utilizzato per riempire i nomi con meno di 14 caratteri. Una voce di directory ha in questo caso solo due campi (come detto in (2)) : il nome del file su 14 byte e il numero di i-node corrispondente a quel file su 2 byte. Questi parametri limitano il numero di file per file system a 64K. Gli i-node UNIX contengono alcuni attributi. Gli attributi contengono la dimensione del file,orari (creazione,ultimo accesso e ultima modifica),proprietario,gruppo,protezione e numero di voci di directory che puntano a quel file. L'ultimo campo è necessario a causa dei link. Quando viene creato un nuovo link ad un i-node,il suo contatore è aumentato di uno. Quando viene rimosso,è decrementato. Quando raggiunge 0,l'i-node viene riciclato e i suoi blocchi del disco tornano nella lista dei blocchi liberi. Per gestire file grandissimi,i primi 10 indirizzi del disco sono memorizzati nel'i-node stesso e prima di essi abbiamo gli attributi. In questo modo per i file piccoli tutte le informazioni necessarie sono nell'i-node che viene prelevato dal disco e posto in memoria principale quando è aperto il file. Per i file più grandi,uno degli indirizzi dell'i-node è l'indirizzo di un blocco del disco chiamato **blocco indiretto singolo**,che contiene indirizzi aggiuntivi del disco. Se anche questo non basta,nell'i-node c'è un altro indirizzo contenente una lista di blocchi indiretti singoli,chiamato **blocco indiretto doppio**. Ognuno di questi blocchi indiretti singoli punta a qualche centinaio di blocchi di dati. Se ancora non bastasse può essere usato un **blocco indiretto triplo**.

Gestione file name lunghi e a dimensione variabile

Finora abbiamo supposto che i file abbiano nomi brevi e di lunghezza fissa. In MS-DOS i file hanno nomi da 1 ad 8 caratteri ed un'estensione opzionale di 3 caratteri. Nella versione 7 di UNIX i nomi dei file hanno da 1 a 14 caratteri,inclusa l'eventuale estensione. Quasi tutti i SO moderni supportano però nomi di file di lunghezza variabile.
L'approccio più semplice per implementare file name a dimensione variabile è impostare un limite sul nome del file,generalmente 255 caratteri e poi usare un modello di implementazione (FAT o i-node) riservando 255 caratteri per ciascun nome di file. Questo sistema è semplice,ma spreca una buona parte dello spazio delle directory,dato che pochi file hanno nomi così lunghi e quindi una parte rimarrà vuota. Abbiamo quindi due alternative.
La **prima** alternativa è rinunciare all'idea che le voci di directory abbiano la stessa lunghezza. Con questo metodo,ciascuna directory contiene una parte fissa,che inizia

con la lunghezza della voce,seguita poi da un formato fisso che include gli attributi e il nome del file. Ogni nome di file termina con un carattere speciale,solitamente 0. Per consentire a ciascuna voce di directory di iniziare alla fine della parola,ogni nome di file è riempito fino ad arrivare ad un numero intero di parole. Questo approccio è utile perché ci permette di utilizzare solo il numero di caratteri necessari per il nome del file. Lo svantaggio è che quando il file viene cancellato,nella directory è introdotto un vuoto di ampiezza variabile,che potrebbe non bastare a contenere il file da inserire successivamente. Un altro problema è quello che una singola voce di directory può estendersi su più pagine,cosicché può accadere un page fault durante la lettura del nome di un file.
Un altro metodo per gestire nomi di lunghezza variabile è creare tutte le voci di directory di lunghezza fissa e tenere i nomi dei file in uno *heap* alla fine della directory. La parte iniziale del file è costituita da puntatori all'*heap* e dagli attributi dei singoli file. Il vantaggio è che quando viene rimossa una voce,il successivo file da inserire ci starà sempre. Naturalmente l'heap va gestito e riordinato. A differenza del primo metodo,non è più necessario che i nomi dei file inizino ai confini delle parole e quindi non abbiamo bisogno di caratteri di riempimento alla fine dei nomi dei file.

Layout del disco (Master Boot Record)

Dobbiamo capire come recuperare la directory a cui appartiene il file interessato all'interno di tutta la struttura di directory del file system.
L'approccio ormai superato è quello di avere un **MBR**. Fintantoché il sistema non si è avviato,egli si aspetta di trovare alcune informazioni in punti prestabiliti. In particolare,il **settore 0** di un disco prende il nome di **Master Boot Record**. Il MBR ha un'informazione sulle partizioni di un disco. Esso contiene una tabella delle partizioni che contiene l'indirizzo-fine di ciascuna partizione e indica qual è la **partizione attiva**,cioè la partizione che contiene il SO. All'avvio,una volta individuata la partizione attiva,viene letto il primo blocco della partizione chiamato **boot block** e successivamente si avvia tutto il SO. A parte il fatto di iniziare con un blocco di boot,il layout di una partizione del disco cambia molto a seconda del file system,che spesso contiene diversi elementi. Il primo è il **superblocco** : contiene i parametri chiave riguardanti il file system e viene letto all'avvio del computer,o quando il file system viene usato per la prima volta.

Layout del disco (UEFI)

L'approccio più moderno è quello Unified Extensible Firmware Interface. In questo modello,anziché avere un MBR nel settore 0 del dispositivo di avvio,il SO cerca la posizione della tabella delle partizioni nel secondo blocco del dispositivo,riservando il primo blocco come marcatore speciale per il software che si aspetta di trovarvi un MBR. Il marcatore dice che non c'è il MBR. La **GPT (GUID partition table)**,intanto,contiene informazioni sulla posizione delle varie partizioni sul disco.

UEFI conserva nell'ultimo blocco un backup della GPT. Una GPT contiene l'inizio e la fine di ogni partizione;una volta trovata la GPT,il firmware ha funzionalità sufficiente a leggere file system di tipi specifici (tipo FAT).
A differenza dell'approccio precedente,dove il SO doveva eseguire il boot record per iniziare "a lavorare",in questo approccio il processo di avvio può utilizzare un vero file system contenente programmi,file di configurazione e tutto ciò che può servire durante l'avvio.

NOTA

Un disco può essere visto come un array di blocchi. Una parte di essi è riservata a memorizzare dati. Un'altra parte di blocchi descrive invece il file system e per contenere **metadati**,cioè dati che servono a descrivere altri dati.

The Second Extended File System (ext2)

E' uno dei file system più noti nella community Linux. Supporta tutte le caratteristiche del file system UNIX ed è basato sull'assunzione che i dati nei file sono organizzati in blocchi. Questo file system è suddiviso logicamente in più parti (**cylinder groups** o **gruppi di blocchi**) al fine di memorizzare i file per ridurne i tempi di accesso. Abbiamo sempre il blocco di boot. Il primo blocco dopo quello di boot è chiamato **superblocco**,il quale contiene una serie di informazioni sul file system : tipo di file system,spazio disponibile,blocchi disponibili ecc. Ha inoltre l'informazione relativa alla dimensione del blocco e ciò permette all'utente,in fase di formattazione,di specificare la dimensione di un blocco (generalmente 4K). In ogni gruppo c'è una copia del superblock (solo la prima copia è usata dal file system). Dopo il superblock abbiamo il **descrittore del gruppo** : esso contiene le informazioni sul gruppo (ad esempio il numero di blocco nella bitmap dei blocchi e degli i-node per il gruppo). Dopo il descrittore del gruppo abbiamo la **bitmap dei blocchi** : dato che in alcuni blocchi del disco potremmo avere file e in altri no,la bitmap dei blocchi ci dice quali blocchi sono in uso e quali no. Quando si crea un file e si deve salvare,si consulta la bitmap dei blocchi per vedere quali sono i blocchi liberi da usare per memorizzare il file. Dopo la bitmap dei blocchi c'è la **bitmap degli i-node** : in un file system esiste un numero finito di i-node,i quali possono essere utilizzati per tenere traccia dei file;la bitmap degli i-node ci dice quali di questi sono liberi per poter essere utilizzati (1 utilizzato - 0 libero). Abbiamo poi gli **i-node** veri e propri ed infine i **blocchi di dati** (dati veri e propri).
Vediamo ora come questi elementi ci aiutano a recuperare un file dal disco.

Lookup di una directory

Immaginiamo di dover progettare il percorso ***/usr/ast/mbox***. Il superblocco contiene un numero,ovvero l'**i-node** della directory radice. L'i-node della directory radice ha una posizione nota sul disco. Tramite le informazioni contenute nell'i-node della directory radice si risale ai blocchi di dati in cui effettivamente è memorizzata la directory. Il SO,una volta recuperati i blocchi che descrivono la directory,effettua una ricerca e trova ***usr***. Da questa voce della directory si scopre che il file *usr* è descritto dall'i-node numero 6. Il SO deve accedere quindi alla sezione degli i-node,prelevare l'i-node numero 6 e caricarlo in memoria. L'i-node,oltre agli attributi del file,contiene l'informazione del blocco in cui il file si trova (ad esempio si scopre che *usr* corrisponde al blocco 132). Si carica poi il blocco dati puntato dall'i-node e si ottiene una nuova directory. Si cerca */ast* in questa directory e si scopre che è descritta dall'i-node numero 26. Si accede all'area degli i-node e troviamo il blocco 406. Si carica il blocco dati puntato dall'i-node e si ottiene una directory che conterrà l'ultima parte del percorso.
Quindi se i-node punta ad una directory avremo una "tabella" altrimenti blocchi dati che rappresentano il dato.

File condivisi

Quando parecchi utenti lavorano insieme ad un progetto,spesso hanno bisogno di condividere i file,quindi risulta comodo che un file condiviso appaia contemporaneamente in directory diverse,appartenenti ad utenti diversi (un esempio è *crea collegamento* → file condiviso con la directory desktop e la directory in cui si trova). Supponiamo di avere due directory B e C. Un file contenuto nella directory C viene condiviso con la directory B. La connessione tra la directory B e il file condiviso è chiamata **link**. Il file system è ora un **grafo aciclico orientato** o **DAG (Directed Acyclic Graph)**. La condivisione dei file è comoda,ma introduce alcuni problemi. Se le directory contengono realmente indirizzi su disco,quando il file viene collegato deve essere eseguita una copia degli indirizzi nella directory B. Se B o C successivamente accodano qualcosa al file,i nuovi blocchi saranno elencati solo nella directory dell'utente che ha eseguito l'accodamento;la modifica non sarà visibile all'altro utente,e ciò vanifica la condivisione. Questo problema è risolvibile in due modi.
Nella **prima soluzione** i blocchi del disco non sono elencati nelle directory,ma in una piccola struttura dati associata al file stesso. Le directory hanno solo dei puntatori a questa piccola struttura dati (i-node) o al blocco. E' l'approccio usato da UNIX.
Nella **seconda soluzione**,B si collega ad uno dei file di C tramite la creazione da parte del sistema di un nuovo file,di tipo LINK,e inserendo quel file nella directory di B. Il nuovo file contiene solo il nome di percorso del file a cui si collega. Quando B legge dal file collegato,il SO vede che il file in lettura è di tipo LINK,cerca il nome del file e legge quel file. Questo approccio è chiamato **link simbolico**,per distinguerlo dal collegamento tradizionale detto **hard link**.
Questi metodi hanno dei pro e dei contro. Nel primo metodo,nel momento in cui B si collega al file condiviso,l'i-node registra C come proprietario del file. Creare un link

non cambia la proprietà,ma incrementa il conteggio dei link nell'i-node,così il sistema sa a quante directory stanno puntando al file. Se successivamente C prova ad eliminare il file,il sistema deve affrontare un problema. Se rimuove il file e pulisce l'i-node,B avrà una voce di directory che punta ad un i-node non valido. Se l'i-node in seguito è riassegnato ad un altro file,il collegamento di B punterà al file errato. Il sistema può vedere dal conteggio dell'i-node che il file è ancora in uso,ma non ha un modo semplice per trovare tutte le voci di directory per il file,al fine di cancellarle. Non è possibile salvare i puntatori alle directory nell'i-node,dato che potrebbe esservi un numero illimitato di directory. L'unica cosa da fare è quella di cancellare la voce della directory di C,ma lasciando l'i-node intatto,con il conteggio impostato ad 1. A questo punto B è l'unico utente ad avere una voce di directory per un file il cui proprietario è C. Se il sistema ha un sistema di contabilità o di quote,C continuerà ad essere addebitato per il file finché B deciderà di eliminarlo,se mai lo farà;a quel punto il conteggio raggiungerà 0 e il file sarà eliminato.
Con i link simbolici questo problema non si pone poiché solo il vero proprietario ha un puntatore all'i-node. Gli utenti che hanno un collegamento al file hanno solo nomi di percorso,non puntatori a i-node. quando il proprietario elimina il file,lo distrugge. I successivi tentativi di utilizzarlo tramite link simbolici falliranno,in quanto il sistema non sarà in grado di localizzare il file. L'eliminazione di un link simbolico non ha alcun effetto sul file. Il problema dei link simbolici è l'appesantimento della gestione. Il file contenente il percorso deve essere letto,quindi il percorso va analizzato e seguito,componente per componente,sino a raggiungere l'i-node. Tutta questa attività può richiedere un numero considerevole di accessi extra al disco. Inoltre serve un i-node extra per ciascun link simbolico,e anche un ulteriore blocco del disco in cui memorizzare il percorso,per quanto se il nome del percorso è breve può essere memorizzato nell'i-node stesso. I link simbolici hanno il vantaggio di poter essere utilizzati per collegare i file su macchine ovunque nel mondo,semplicemente aggiungendo al percorso del file sulla macchina nella quale risiede l'indirizzo di rete della macchina in questione.
Un ultimo problema,che riguarda i link in generale,è che quando sono consentiti i collegamenti,i file possono avere due o più percorsi. I programmi che partono da una directory definita e trovano tutti i file in quella directory e nelle sue sottodirectory localizzeranno più volte un file collegato.

Gestione del disco e file system

Dischi magnetici

I dischi magnetici sono caratterizzati dal fatto che le operazioni di lettura e scrittura sono ugualmente veloci,rendendoli l'ideale per la memoria secondaria. Array di questi dischi sono utilizzati per fornire memoria altamente affidabile. I dischi rigidi sono composti da una pila di piatti di alluminio. Su ciascun piatto è depositato uno strato sottile di materiale magnetizzabile.Sono organizzati in **cilindri**,ciascuno contenente tante tracce quante sono le testine impilate verticalmente. Le tracce sono

divise in settori e il numero di settori lungo la circonferenza raggiunge generalmente varie centinaia. Il numero di testine varia da 1 a 16. Le **tracce** sono anelli concentrici del piatto ; i **settori** sono una quota parte della traccia e sono la granuralità minima di memorizzazione dati sul disco. Il cilindro è invece l'insieme delle tracce di tutti i piatti che hanno la stessa distanza dall'asse centrale. Un disco è **indirizzabile** in due modi :

1. **CHS :** cylinder-head-sector. Ogni settore nel disco è identificato da una tripla di numeri (x,y,z);
2. **LBA** : logical block addressing. I settori dei dischi sono numerati consecutivamente partendo da 0,senza tener conto in alcun modo della geometria del disco. L'elettronica interna del disco,dato un certo numero riesce a risalire a quale settore ci si riferisce.

Man mano che ci si allontana dal centro,le tracce diventeranno più lunghe. Per questo motivo,uno schema in cui i settori occupano via via sempre più spazio porterebbe a sprecare spazio. Nella pratica,infatti,i dischi sono divisi in zone (partendo dal centro) e quando ci si è allontanati a sufficienza dal centro e la traccia è diventata più lunga,si potranno ospitare più settori.

Formato di un settore

Prima dell'uso,ciascun piatto deve sottostare ad una **formattazione a basso livello** eseguita via software. La formattazione consiste di una serie di tracce concentriche,ognuna contenente un certo numero di settori,con dei piccoli spazi tra di loro. La maggior parte dei dischi usa settori a 512 byte. Il settore è così costituito:

- **preambolo** : insieme di bit che permette all'hardware di riconoscere l'inizio del settore. Contiene inoltre il cilindro e i numeri dei settori e qualche altra informazione;
- **dati** : la dimensione della parte dei dati è determinata dal programma di formattazione a basso livello;
- **ECC** : codice di correzione errori. Contiene informazioni ridondanti che possono essere usate per ripristinare eventuali errori di lettura. La dimensione e il contenuto di questo campo variano a seconda del produttore,di quanto spazio del disco il progettista è disposto a lasciare al fine di garantire un'alta affidabilità e di quanto sia complesso il codice ECC.

In sostanza,dato il settore,la quota parte che si riserva al dato vero e proprio è più piccola del settore vero e proprio.
Tra un settore e l'altro abbiamo un piccolo **gap**. Quindi se abbiamo un disco di 1 TB,la capacità reale sarà sicuramente inferiore.
Accesso al disco

Per effettuare una lettura o una scrittura di un blocco del disco,il tempo richiesto è determinato da tre fattori :

1. **tempo di ricerca o tempo di seek** : tempo di ricerca medio per posizionare la testina sulla traccia desiderata;
2. **ritardo di rotazione** : tempo necessario affinché il disco effettui mezza rotazione. Se ad esempio ci siamo posizionati sulla traccia,ma non siamo sul settore desiderato,il disco deve ruotare per selezionare il settore e nel caso peggiore farà una mezza rotazione;
3. **tempo di trasferimento** : tempo necessario alla rotazione completa di una traccia moltiplicato per la frazione della traccia da dover percorrere. Una volta posizionata la testina sul settore di interesse,si legge il settore e quindi il piatto deve girare.

Ad esempio per leggere un blocco di **k** byte in un disco con 1 MB per traccia,tempo di rotazione pari a 8,33 ms,e tempo di ricerca medio pari a 5 ms il tempo di accesso sarà :

5 + 4,165 + (k/1milione) * 8,33

4,165 corrisponde al ritardo di rotazione e 1kk a 1MB. Il tempo di accesso è influenzato maggiormente dal tempo di seek e dalla rotazione. Il rapporto k/1MB influisce solo quando inizia ad essere elevato.

Dimensione dei blocchi

Il SO usa una **dimensione del blocco** in generale maggiore rispetto a quella del **settore del disco**. Ogni blocco consiste in un certo numero di settori consecutivi. La dimensione tipica di un blocco è di 4 KB. Si arriva a questo numero per un compromesso ragionevole per l'utente medio. Su un disco magnetico,un blocco di grandi dimensioni significa che qualunque file,anche da un solo byte,impegna un intero blocco,quindi i file piccoli sprecano una grande quantità di spazio del disco. All'opposto,un blocco piccolo significa distribuire la maggior parte dei file su più blocchi e incorrere in più ricerche e ritardi di rotazione per leggerli,a discapito delle informazioni. (VEDI FRAMMENTAZIONE INTERNA LEZ.16 1.14.00 1.23.00).
Analizzando uno schema p.267,la curva tratteggiata mostra la velocità di trasferimento dei dati per un disco in funzione della dimensione dei blocchi. Per il calcolo dell'efficienza dello spazio dobbiamo ipotizzare una dimensione media dei file,ad esempio 4 KB. La curva continua mostra l'efficienza dello spazio in funzione della dimensione dei blocchi. Le due curve possono essere spiegate come segue. Il tempo di accesso per un blocco consta per la stragrande maggioranza di tempo di ricerca e ritardo di rotazione,quindi,dato che per accedere ad un blocco servono 9ms (dall es. precedente),più dati sono prelevati meglio è. A partire da questa considerazione la velocità di trasferimento dei dati sale quasi linearmente con la

dimensione del blocco (fino a quando i trasferimenti iniziano ad impiegare così tanto da rendere rilevante il tempo di trasferimento). Consideriamo l'efficienza dello spazio. Con file di 4 KB e blocchi di 1 KB,2 KB e 4 KB i file usano rispettivamente 4,2 e 1 blocchi,senza sprechi. Con blocchi di 8 KB e file di 4 KB l'efficienza scende al 50% e con blocchi di 16 KB al 25%. In realtà pochi i file sono multipli esatti dei blocchi del disco,e così nell'ultimo blocco si spreca sempre un po' di spazio (frammentazione interna). Le due curve ci dicono quindi che blocchi piccoli sono negativi per le prestazioni,ma ottimi per l'utilizzo dello spazio.

Tenere traccia dei blocchi liberi

Per tenere traccia dei blocchi liberi possono essere utilizzati due metodi. Il primo consiste nell'utilizzare una lista concatenata di blocchi del disco,con ciascun blocco contenente tanti numeri di blocchi del disco liberi quanto possibile. Con un blocco da 1 KB e un numero a 32 bit di blocchi del disco,ciascun blocco sulla **lista dei blocchi liberi** contiene i numeri di 255 blocchi liberi (uno slot è richiesto dal puntatore al blocco successivo). Generalmente si usano blocchi liberi per memorizzare la lista dei blocchi liberi,cosicché non si occupa ulteriore spazio. Il secondo metodo di gestione dello spazio libero è la **bitmap**. Un disco con *n* blocchi richiede una bitmap con *n* bit. I blocchi liberi sono indicati dal valore 1 nella mappa,quelli allocati dallo 0 (o viceversa). Se ad esempio abbiamo un disco da 1 TB serve una mappa da 1 miliardo di bit,il che richiede circa 130 000 blocchi da 1 KB per eseguire la memorizzazioni. La bitmap richiede meno spazio,dato che usa 1 bit per blocco contro i 32 bit del modello a lista concatenata. Lo schema della lista concatenata richiede meno spazio della bitmap solo se il disco è quasi pieno (cioè se ha meno blocchi liberi). Se i blocchi liberi tendono ad essere in lunghi tratti di blocchi consecutivi,il sistema della lista dei blocchi liberi può essere modificato per tener traccia di serie di blocchi anziché di blocchi singoli. A ciascun blocco può essere associato un conteggio a 8,16 o 32 bit,che fornisce il numero di blocchi liberi consecutivi. Nell'ipotesi migliore,un disco fondamentalmente vuoto potrebbe essere rappresentato da due numeri : indirizzo del primo blocco libero seguito dal conteggio dei blocchi liberi. Se invece il disco diventa molto frammentato,tener traccia delle serie è meno efficiente che tener traccia dei singoli blocchi,dato che non deve essere memorizzato solo l'indirizzo,ma anche il conteggio.
Per il metodo della lista dei blocchi liberi,è necessario tenere in memoria solo un blocco di puntatori. Al momento della creazione di un file,i blocchi necessari vengono presi dal blocco dei puntatori;quando i puntatori finiscono viene letto un nuovo blocco di puntatori dal disco. In modo analogo,quando un file viene cancellato,i suoi blocchi vengono liberati e aggiunti al blocco dei puntatori nella memoria principale;quando questo blocco è pieno viene scritto sul disco.

Quote del disco

Sistemi UNIX/Linux sono pensati per servire più utenti,i quali sono liberi di creare e cancellare file per i loro account ma abbiamo comunque un solo disco. Per impedire che gli utenti occupino troppo spazio sul disco,i SO multiutente forniscono spesso un meccanismo per imporre le quote del disco. L'idea è che l'amministratore di sistema assegni a ciascun utente un numero massimo di file e blocchi e che il SO si accerti che gli utenti non superino la loro quota. Un meccanismo tipico è quello che segue.
Quando un utente apre un file,gli attributi e gli indirizzi del disco vengono localizzati e posti in una tabella dei file aperti nella memoria principale. Fra gli attributi c'è una voce che indica il proprietario. Ogni aumento di dimensione del file è contabilizzato nella quota del proprietario. Una seconda tabella contiene il record delle quote di ogni utente che ha un file attualmente aperto,anche se il file è stato aperto da qualcun altro. La **tabella delle quote** contiene un estratto dal file delle quote su disco degli utenti i cui file sono attualmente aperti. Quando tutti i file sono chiusi il record viene riscritto sul file delle quote. Quando si crea una nuova voce nella tabella dei file aperti,viene inserito un puntatore al record delle quote del proprietario per trovare i vari limiti. Ogni volta che un blocco viene aggiunto al file,viene incrementato il numero totale dei blocchi in carico al proprietario e viene eseguito un controllo sul limite "**soft**" e "**hard**". Il limite "soft" può essere oltrepassato (vengono emessi dei warning e quindi la tabella contiene anche una quota dei warning rimanenti che viene decrementata di uno. Se arriva a 0,l'utente ha ignorato gli avvisi una volta di troppo e non gli è più consentito di accedere al sistema. Il permesso deve essere discusso con l'amministratore del sistema),quello "hard" no. L'aumento di dimensione di un file quando è stato già raggiunto il limite "hard" sui blocchi provoca un errore. Durante una sessione quindi gli utenti possono superare i loro limiti "soft" a condizione che rimuovano gli eccessi prima di scollegarsi. Controlli analoghi esistono anche sul numero di file per evitare che un utente si accaparri tutti gli i-node.

Esercitazione Shell Scripting

Fino ad ora abbiamo visto comandi eseguiti tramite shell,ed utilizzo di meccanismi come pipe e redirezione. Nella pratica,può servire eseguire **più comandi** (al verificarsi di certe condizioni) o **iterarli**. Molte shell,tra cui **bash**,supportano un linguaggio di scripting (variabili,strutture di controllo,sintassi,etc.). Lo shell scripting è un linguaggio ottimizzato per eseguire task "**shell-related**" :

- creazione di pipeline di comandi;
- salvare risultati su un file;
- cercare file;
- effettuare ricerche testuali;
- creare,cancellare,spostare directory e files;
- gestire processi;
- …

Riguarda quindi tutto ciò che si può costruire sulla base dei tantissimi comandi e programmi di utilità.

Comandi base

- **pwd** : print current directory;
- **ls** : list directory content;
- **cd** : change directory;
- **mkdir** : make directory;
- **rm** : remove files or directories (**rm -r**);
- **mv** : move - rename files;
- **cp** : copy files or directories;
- **ps aux** : snapshot of current process.

Per questi comandi e gli altri comandi in seguito,si ricorda l'utilizzo dei flag **-h** oppure **–help**,o anche del comando **man** (manuale).

Ricerca di file e directory : find

La ricerca di file e directory è un task molto comune. Tutti i sistemi UNIX-like forniscono il comando ***find***.

-name : serve a specificare un pattern dei file che stiamo cercando. Se usiamo "*" significa 0 o un numero arbitrario di caratteri,e quindi un nome arbitrario. L'asterisco è chiamato **wildcard**.

Se vogliamo cercare tutti i file all'interno di una cartella "resources" useremo : *find resources -name "*".* Se volessi cercare solo i file testuali,userei "*.txt" che cerca tutto ciò che ha nome arbitrario e con estensione .txt.

Un'altra opzione è quella di utilizzare *-type* per specificare quale file vogliamo cercare : *-type d* cerca le directory , *-type f* cerca i file. Ad esempio :

find resources -name "" -type d // cerca tutte le directory con nome arbitrario in resources*

find resources -name "" -type f // cerca tutti i file con nome arbitrario in resources*

find non solo ricerca dei file con un certo pattern,ma permette anche di applicare un comando arbitrario. Ad esempio :

find resources -name "" -type f -exec ls -la {} \;*

cerca tutti i file nella cartella e ci fornisce le loro informazioni tramite comando aggiuntivo *ls -la*. **{}** ****; rappresenta il nome del file su cui è applicato il comando (viene cioè sostituito col nome del file).

Se sto cercando già file con "*.txt" , -type f non è obbligatorio,dato che dovrei avere già i file ma in UNIX nessuno vieta di creare una directory .txt.

Se volessi copiare tutti i file *.txt* contenuti in *resources* in un'altra directory,userò :
find resources -name ".txt" -type f -exec cp {} namedir \;*

Visualizzazione righe

- **cat** : concatena file e mostra sullo standard output;
- **head** : mostra la prima parte di un file;
- **tail** : mostra la parte finale di un file;

head -n visualizza le prime *n* righe di un file.
tail -n visualizza le ultime *n* righe di un file.

Conteggio e ordinamento

- **wc** : conta. Ad esempio conta le linee di un file tramite : *wc -l words.txt*. In base all'opzione può contare i byte,i caratteri ecc.
- **uniq** : elimina i duplicati dal file. In particolare scorre i file ma elimina (nella versione senza flag) solo i duplicati consecutivi.
- **sort** : ordina un file. Senza opzioni fa un ordinamento di tipo lessico-grafico e quindi anche i numeri sono considerati caratteri.

sort -n ordina numericamente (anziché in maniera lessicografica);
sort -r ordina in maniera inversa;
sort -nr,ordina numericamente in ordine decrescente.

uniq,oltre ad eliminare i duplicati consecutivi,ci può dire quanti elementi erano duplicati tramite ***uniq -c***.

Tramite ***sort words.txt | uniq -c*** ordiniamo il file e vediamo quali erano le occorrenze del termine/riga nell'input,eliminando i duplicati consecutivi.

Come isolo il termine più frequente o meno frequente?

più frequente → uso *sort words.txt* | *uniq -c* per ordinare il file e contare le occorrenze. Collego poi *sort words.txt* | *uniq -c* | *sort -n* per ordinare numericamente e infine ci aggiungo *sort words.txt* | *uniq -c* | *sort -n* | *tail -n 1* per isolare il più frequente.

meno frequente → uso *sort words.txt* | *uniq -c* per ordinare il file e contare le occorrenze. Collego poi *sort words.txt* | *uniq -c* | *sort -nr* per ordinare numericamente in ordine decrescente e infine ci aggiungo *sort words.txt* | *uniq -c* | *sort -n* | *tail -n 1* per isolare il meno frequente.

Come trovo il file nella directory *resources* con il numero maggiore di righe?
find resources -name "" type -f -exec wc -l {} \;* | *sort -nr* | *head -n 1*

Data handling

awk è un linguaggio orientato alla manipolazione dati di tipo testuale (da file o standard input). Ha molte funzionalità ma lo useremo per isolare l'n-esimo token dalle righe di input. Questo comando,dato un input testuale,il quale si assume essere organizzato in maniera tabellare,permette di trattare l'input come una tabella e permette di operare sulle colonne.

awk '{print $IDcolonna , $n}'

Ad esempio dato il comando :

find resources -name "" type -f -exec wc -l {} \;* | *sort -nr* | *head -n 1*

collegando *awk '{print $2}'* alla fine,stamperemo solo il nome del file.

Quali sono i 5 processi che stanno occupando più memoria?

Devo usare *ps aux* e vedere la colonna RSS (Resident Set Size - memoria occupata dal processo in RAM). Per avere un output pulito posso usare questo comando :

ps aux | *awk '{print $6,$2}* | *sort -nr* | *awk '{print "RSS: " , $1 , "PID: ", $2}'* | *head -n 5*

grep permette di specificare una stringa (in generale pattern) e di conservare solo le righe di input con quel pattern.
Con ***grep -v*** mostra le righe che non hanno quella stringa.

Comando di prima con grep :

```
ps aux | grep -v "PID" | awk '{print $6,$2}' | sort -nr | head -n 3
```

awk può essere utilizzato anche per mostrare colonne se altre colonne soddisfano un certo criterio. Un esempio può essere il seguente :

ps aux | awk '($2 == 15438) {print $1,$5,$6}'

Questo comando stampa le colonne 1,5,6 se la colonna 2 assume il valore 15438.

Creare uno script

Bisogna innanzitutto inserire **#!**,detti **shebang** per indicare al sistema quale interprete utilizzare (qualsiasi altra riga con # è un commento),il quale sarà **/bin/bash** e quindi sarà **#! /bin/bash**.
Supponiamo di dover creare un programma di backup,passando da riga di comando la directory in cui salvare i file ad esempio lanciando ***./backup.sh mydir***.
Dobbiamo innanzitutto prendere il parametro da riga di comando e leggerlo nello script : **BACKUPDIR = $1**. $1 sta ad identificare il primo argomento,assegnato alla variabile BACKUPDIR ($2 secondo argomento,$3 terzo e così via). Bisognerà poi eliminare questa directory se esiste già e crearla se non esiste,usando rispettivamente i comandi ***rm -rf*** e ***mkdir***.
Troviamo poi i file con *resources/ -name ".txt" -type f -exec cp {} $BACKUPDIR \;*

In bash :

- **single quote (')** : inserire caratteri tra single quote,ne preserva il valore letterale;
- **double quote ('')** : inserire caratteri tra double quote,ne preserva il valore letterale,fatta eccezione per $ '\!;
- **back quote (`)** : esegue il comando tra i back quote,come **$()**.

Per assegnare il risultato di un comando abbiamo due alternative :

var=$(command)
var=`**command**`

Se si prova ad eseguire un file ***.sh*** non avremo i permessi e quindi dobbiamo darglieli tramite ***chmod 777*** (dà i permessi a tutto).

SO cap5

Oltre a fornire astrazioni come processi e thread,spazi degli indirizzi e file,un SO controlla anche tutti i dispositivi di I/O del computer,inviando comandi ai dispositivi,intercettando interrupt e gestendo gli errori. Dovrebbe anche fornire

un’interfaccia semplice e facile da usare fra i dispositivi e il resto del sistema,la quale dovrebbe essere la stessa per tutti i dispositivi (indipendenza dai dispositivi). Il codice per l’I/O rappresenta una parte significativa della totalità del SO.

Dispositivi di I/O

I dispositivi di I/O possono essere divisi in due categorie principali : **dispositivi a blocchi** e **dispositivi a caratteri**. Un dispositivo a blocchi archivia informazioni in blocchi di dimensioni fisse,ciascuno con il proprio indirizzo. Le dimensioni dei blocchi vanno solitamente da 512 a 32768 byte. Tutti i trasferimenti sono in unità di uno o più blocchi (consecutivi) interi. La caratteristica essenziale di questi dispositivi è che ciascun blocco può essere letto o scritto indipendentemente dagli altri. Dischi fissi magnetici e unità SSD sono classici dispositivi a blocchi,così come unità a nastro che oggi si usano nei data center. L’altro tipo di dispositivi di I/O è quello a caratteri. Un’unità a caratteri rilascia o accetta un flusso di caratteri,senza alcuna struttura a blocchi. Non è indirizzabile e non ha alcuna operazione di ricerca. Alcuni esempi sono stampanti,interfacce di rete,mouse ecc. Altri dispositivi di I/O sono i **clock**,non indirizzabili a blocchi e non generano né accettano flussi di caratteri : il loro compito è quello di produrre interrupt a intervalli ben definiti. I due modelli visti sono abbastanza generici da poter essere usati come base per rendere indipendenti dal dispositivo alcuni dei software del SO che si occupano di I/O. Ad esempio il file system,si occupa solo di dispositivi a blocchi astratti e lascia la parte dipendente dai dispositivi a software di livello più basso. I dispositivi di I/O possono avere velocità molto diverse e ciò mette pressione al software che deve gestire al meglio le velocità di trasferimento variabili.

Controller dei dispositivi

I dispositivi di I/O consistono tipicamente di una componente meccanica e di una elettronica;queste due parti possono essere separate per fornire una progettazione più modulare e generale. La componente elettronica è detta **controller del dispositivo (device controller)** o **adattatore (adapter)**. Sui computer è spesso presente nella forma di chip sulla scheda madre o di una scheda a circuiti stampati inseribile in uno slot di espansione (PCIe). La parte meccanica è invece il dispositivo stesso. La scheda del controller presenta spesso un connettore in cui inserire un cavo che si collega al dispositivo stesso. Molti controller possono gestire due,quattro,otto o più dispositivi identici. L’interfaccia tra il controller e il dispositivo è generalmente di livello molto basso. Ad esempio,un disco potrebbe avere 3kk tracce,ciascuna formattata in un numero da 200 a 500 settori di 4096 byte. Ciò che esce dall’unità,però,è un flusso seriale di bit che parte con un **preambolo**,seguito dagli 8x4096 = 32768 bit in un settore e alla fine un **codice di correzione degli errori (ECC)**. Il preambolo viene scritto al momento della formattazione del disco e contiene il numero dei cilindri e dei settori,la dimensione e dati simili.

Il compito del controller è convertire il flusso seriale di bit in un blocco di byte ed eseguire le correzioni degli errori necessarie. Tipicamente il blocco dei byte è prima assemblato bit per bit,in un buffer interno al controller e dopo aver verificato l'assenza di errori viene mandato alla memoria principale.

I/O mappato in memoria

Ogni controller dispone di alcuni registri usati per le comunicazioni con la CPU. Scrivendo in questi registri,il SO può ordinare al dispositivo di inviare dati,accettarli,accendersi e spegnersi o eseguire altre azioni. Leggendo da questi registri,il SO apprende quale sia lo stato del dispositivo,se sia pronto ad accettare un nuovo comando e così via. La questione che sorge è come la CPU comunichi con i registri di controllo e i buffer dei dati del dispositivo (il SO può scrivere o leggere in essi). Esistono due alternative : **port-mapped** e **memory-mapped**.
Nella prima,a ciascun registro di controllo è assegnato un numero di **porta di I/O**,un intero di 8 o 16 bit. L'insieme di tutte le porte di I/O forma lo **spazio delle porte di I/O**,protetto in modo che i normali programmi utente non possano accedervi (solo il SO può farlo). Se ad esempio si usa un'istruzione speciale come IN REG,PORT , la CPU può leggere il registro di controllo PORT e salvare il risultato nel registro di CPU REG. Allo stesso modo,usando OUT PORT,REG la CPU può scrivere il contenuto di REG in PORT. In questo schema,gli spazi degli indirizzi della memoria e dell'I/O sono diversi. Inoltre,le istruzioni IN R0,4 e MOV R0,4 sono completamente diverse in questo caso. La prima legge i contenuti della porta di I/O 4 e li salva in R0,la seconda legge invece i contenuti della parola di memoria 4 e li salva in R0. Il 4 si riferisce a spazi degli indirizzi diversi e non correlati.
Il secondo approccio,quello **memory-mapped**,consiste nel mappare tutti i registri di controllo nello spazio della memoria. Abbiamo quindi un unico spazio di indirizzamento che comprende anche le porte di I/O. A ciascun registro di controllo è assegnato un indirizzo di memoria univoco a cui non è assegnata memoria. Generalmente gli indirizzi assegnati sono quelli nella parte superiore dello spazio degli indirizzi. I registri di I/O sono quindi acceduti come se fossero in memoria.
Abbiamo poi uno **schema ibrido**,con buffer dei dati dei dispositivi di I/O mappati in memoria e porte di I/O separate per i registri di controllo.
Vediamo come funzionano.
In tutti i casi,quando la CPU vuole leggere una parola,dalla memoria o da una porta di I/O,mette l'indirizzo di cui ha bisogno nelle linee degli indirizzi del bus e dichiara un segnale di READ sulla linea di controllo del bus. Una seconda linea di segnale è usata per indicare se si debba impiegare lo spazio dell'I/O o lo spazio della memoria. La risposta arriva dalla memoria nel caso la richiesta riguardi il suo spazio,e dal dispositivo di I/O nell'altro caso. Se siamo nel caso memory-mapped,tutti i moduli di memoria e tutti i dispositivi di I/O confrontano le linee degli indirizzi con l'intervallo degli indirizzi di loro competenza,e il modulo o il dispositivo con l'intervallo giusto risponde alla richiesta.

I due schemi per indirizzare i controller presentano punti di forza e punti deboli diversi.
Il primo vantaggio dell'I/O **memory-mapped** è quello che se sono necessarie istruzioni speciali di I/O per leggere e scrivere i registri di controllo dei dispositivi,per accedervi è necessario codice assembly,dato che in C o C++ non è possibile eseguire un'istruzione IN o OUT (invocare una procedura di questo genere aggiunge overhead al controllo dell'I/O). Con I/O memory mapped,invece,i registri di controllo dei dispositivi sono solo variabili in memoria e possono essere trattati in C come variabili. Quindi,con I/O memory-mapped,un driver di un dispositivo di I/O può essere scritto completamente in C. Il secondo vantaggio è che non serve alcun meccanismo di protezione speciale per evitare che i processi utente eseguano l'I/O : basta che il SO eviti di mettere la parte dello spazio degli indirizzi contenente i registri di controllo nello spazio degli indirizzi virtuali di un qualunque utente. Il terzo vantaggio è che ogni istruzione che può fare riferimenti alla memoria può fare anche riferimenti ai registri di controllo. Per esempio,se c'è un'istruzione TEST che verifica se il valore in una parola di memoria è 0,può essere usata anche per verificare se il valore di un registro di controllo è 0,il che potrebbe significare che il dispositivo è inattivo e può accettare un nuovo comando. Se non viene utilizzato il memory-mapped,il registro di controllo deve essere prima letto dalla CPU e poi testato,richiedendo due istruzioni invece di una. Non sono necessarie quindi istruzioni aggiuntive. Vediamo ora gli **svantaggi**. Il primo è che la maggior parte dei computer ha una cache delle parole della memoria. Mettere nella cache un registro di controllo potrebbe avere effetti disastrosi. Se un registro di controllo viene messo in cache,infatti,tutti i riferimenti successivi prenderebbero il valore dalla cache senza interrogare il dispositivo e se diventasse disponibile,il software non se ne accorgerebbe. Per evitare questa situazione,l'hardware deve avere la capacità di disabilitare in maniera selettiva la cache. Il secondo svantaggio è che se c'è un solo spazio degli indirizzi,tutti i moduli di memoria e tutti i dispositivi di I/O devono esaminare tutti i riferimenti alla memoria per vedere a quali rispondere. Se il computer ha un **singolo bus**,è semplice fare in modo che ciascuno controlli tutti gli indirizzi. La tendenza dei computer moderni è però quella di avere un **bus della memoria dedicato ad alta velocità**. Questo bus è appositamente costruito per ottimizzare le prestazioni della memoria,senza compromessi dovuti a dispositivi di I/O lenti. Il problema di avere questo bus separato su una macchina che usa l'I/O memory-mapped,è che i dispositivi di I/O non hanno modo di vedere gli indirizzi di memoria,che vanno sul bus della memoria,e quindi non hanno modo di rispondergli. Inoltre,per far funzionare l'I/O mappato in memoria su un sistema con più bus sono necessarie misure speciali. Una possibilità è quella di inviare prima alla memoria tutti i riferimenti alla memoria. Se la memoria non risponde,la CPU prova gli altri bus. Può funzionare ma richiede complessità hw. Un secondo design è mettere sul bus della memoria un dispositivo "spia" che intercetti tutti gli indirizzi che potenzialmente interessano dei dispositivi di I/O. Il problema è che i dispositivi di I/O potrebbero non essere in grado di elaborare le richieste alla velocità cui invece è in grado di rispondere la memoria. Terza possibilità è quella di filtrare gli indirizzi nel controller

della memoria. In questo caso il chip del controller della memoria contiene registri di indirizzi precaricati al momento dell'avvio. Lo svantaggio è la necessità di capire al momento dell'avvio quali indirizzi di memoria non sono realmente indirizzi di memoria.

Eseguire operazione di I/O

Per eseguire l'I/O esistono tre metodi principali :

- I/O programmato;
- I/O guidato dagli interrupt;
- I/O con DMA (Direct Memory Access).

I/O programmato

L'I/O programmato consiste nel delegare tutto il lavoro alla CPU. Possiamo spiegarlo tramite un esempio. Consideriamo un processo utente che voglia scrivere la stringa di 8 caratteri "ABCDEFGH" sulla stampante tramite un'interfaccia seriale. Per prima cosa,il software assembla la stringa in un buffer dello spazio utente. Il processo utente acquisisce poi la stampante per la scrittura,facendo una chiamata di sistema per aprirla (*printer_open*). Se la stampante al momento è usata da un altro processo utente,la chiamata fallirà restituendo un codice di errore,o si bloccherà finché la stampante non torna disponibile,a seconda del SO e dei parametri della chiamata. Una volta ottenuta la stampante il processo utente esegue una chiamata di sistema richiedendo al SO di stampare la stringa (*printer_print (buffer,...)*). Il SO,di solito,copia poi il buffer con la stringa in un array nello spazio del kernel,dove è più facilmente accessibile (dato che il kernel potrebbe dover cambiare la mappa della memoria per accedere allo spazio utente) ed è anche al riparo da eventuali modifiche da parte del processo utente. Controlla poi se la stampante è disponibile,e se non lo è aspetta che lo diventi. A questo punto il SO copia il primo carattere nel registro dei dati della stampante,in questo caso usando I/O mappato in memoria. Questa azione attiva la stampante. Il carattere potrebbe non apparire ancora perché alcune stampanti mettono nel loro buffer una riga o una pagina prima di avviare la stampa. Non appena ha copiato il primo carattere nella stampante,il SO controlla se la stampante è pronta per riceverne un altro. Di solito la stampante ha un secondo registro che fornisce il suo stato. L'azione di scrittura nel registro dei dati fa sì che lo stato diventi "not ready". Quando il controller della stampante ha elaborato il carattere corrente,indica la sua disponibilità impostando qualche bit o mettendo qualche valore nel registro di stato. Il SO aspetta che la stampante sia nuovamente pronta;quando lo è stampa il carattere successivo e il ciclo continua fino a quando non viene stampata l'intera stringa per poi ritornare il controllo al processo utente.
Ciò può essere riassunto tramite il seguente codice :

```
copy_from_user(buffer,p,count) // p è il buffer del kernel
for (i = 0 ; i < count ; i++) { // si ripete per tutti i caratteri della stringa

        while (*printer_status_reg != READY) ; // si ripete fino a quando lo stato READY
                *printer_data_register = p[i]; // output di un carattere
}

return_to_user();
```

I dati vengono copiati nel kernel,il So entra in un ciclo emettendo un carattere per volta. L'aspetto essenziale dell'I/O programmato è che dopo aver emesso un carattere,la CPU interroga continuamente il dispositivo per vedere se è pronto ad accettarne un altro. Questo comportamento è detto **polling** o **busy waiting** (il processo attende il completamento dell'operazione di I/O). L'I/O programmato è semplice,ma ha lo svantaggio di occupare la CPU a tempo pieno fino a che l'I/O non è terminato. Se il tempo per stampare un carattere è molto breve,allora il busy waiting va bene così come in un sistema embedded dedicato. In sistemi più complessi è invece inefficiente.

I/O guidato dagli interrupt

Supponiamo di avere una stampante senza buffer ma che stampa un carattere dopo l'altro appena arriva. Ad esempio se stampa 100 caratteri/secondo,ogni carattere impiega 10ms ad essere stampato. Ciò significa che dopo l'operazione di scrittura nel registro dei dati della stampante di ciascun carattere,la CPU si fermerà per un ciclo di inattività di 10 ms in attesa che le sia consentito l'output del carattere successivo. In questo tempo può avvenire un context switch e l'esecuzione di un altro processo per evitare che i 10 ms vengano sprecati. Per permettere alla CPU di fare altro nell'attesa che la stampante diventi pronta è possibile usare degli interrupt. Dopo la chiamata di sistema per stampare la stringa,il buffer viene copiato nello spazio del kernel e il primo carattere è copiato nella stampante appena è in grado di accettarlo. A questo punto la CPU chiama lo scheduler e viene eseguito un altro processo. Il processo che ha richiesto la stampa della stringa è bloccato finché non è stampata l'intera stringa. Il lavoro eseguito al momento della chiamata di sistema è :

```
copy_from_user(buffer,p,count) // copio nel kernel i dati
enable_interrupts(); // abilito gli interrupt
while(*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler();
```

Quando la stampante ha stampato il carattere ed è pronta ad accettare il successivo,genera un interrupt che ferma il processo attuale e ne salva lo stato. Poi è eseguita la procedura di interrupt della stampante :

```
if (count == 0) {
        unblock_user();
} else {
        *printer_data_register = p[i];
        count = count-1;
        i = i+1;
}
        acknowledge_interrupt();
        return_from_interrupt();
```

Se non ci sono più caratteri da stampare,il gestore degli interrupt esegue qualche azione per sbloccare l'utente (unblock). Altrimenti,esegue l'output del carattere successivo,riconosce l'interrupt e ritorna al processo che stava eseguendo prima dell'interrupt,che riprende da dove era stato lasciato.

Direct Memory Access (DMA)

A prescindere che una CPU abbia o meno I/O mappato in memoria,deve poter accedere ai controller dei dispositivi per scambiare i dati. La CPU può richiedere dati da un controller di I/O un byte alla volta,ma ciò spreca il tempo della CPU,quindi si usa un sistema chiamato **DMA**. Supponiamo che la CPU acceda a tutti i dispositivi e alla memoria con un singolo bus (principi identici per sistemi più complessi). Il SO può usare il DMA solo se l'hardware ha un controller DMA,presente nella maggior parte dei sistemi. A volte è integrato nei controller dei dischi e all'interno di altri controller,ma sarebbe così necessario un controller DMA distinto per ogni dispositivo. Più frequentemente è disponibile un controller DMA singolo (ad esempio sulla scheda madre) per regolare i trasferimenti tra più dispositivi,spesso in concomitanza. Ovunque si trovi,il controller DMA ha accesso al bus di sistema indipendentemente dalla CPU. Esso contiene alcuni registri che possono essere scritti e letti dalla CPU,tra cui un registro degli indirizzi di memoria,un registro dei conteggi dei byte e uno o più registri di controllo. Questi ultimi specificano le porte di I/O da usare,la direzione del trasferimento (lettura da un dispositivo o scrittura verso un dispositivo),l'unità di trasferimento (byte o parola) e il numero di byte da trasferire alla volta. Vediamo come funziona il DMA tramite un esempio di lettura di dati da un disco:

1. La CPU "**programma**" il controller DMA impostandone i registri in modo che conosca che cosa deve trasferire e dove;

2. Il DMA invia una richiesta di lettura al controller del disco. Questa richiesta si presenta come qualunque altra richiesta e il controller del disco non sa,se gli arriva dalla CPU o dal controller DMA;
3. Avviene l'operazione di scrittura in memoria. Generalmente l'indirizzo di memoria in cui scrivere è sulle linee degli indirizzi del bus,così quando il controller del disco preleva la parola successiva dal buffer interno,sa dove scriverla;
4. Una volta completata l'operazione di scrittura,il controller del disco manda un segnale di conferma al controller DMA,sempre tramite il bus. Il controller DMA incrementa poi l'indirizzo di memoria da usare e decrementa il conteggio dei byte. Se questo conteggio è maggiore di 0,i passi da 2 a 4 vengono eseguiti finché il conteggio arriva a 0;
5. Una volta che il conteggio è arrivato a 0,il DMA invia un interrupt alla CPU per avvisarla che il trasferimento è completato. Quando l'esecuzione passa al SO,non deve copiare il blocco del disco in memoria perché è già lì.

Il DMA appena visto gestisce un trasferimento alla volta ma esistono DMA più sofisticati che possono gestire più trasferimenti insieme. Questi ultimi hanno internamente più set di registri,uno per ciascun canale. La CPU inizia caricando ciascun set di registri con i parametri pertinenti al suo trasferimento. Ogni trasferimento deve usare un diverso controller di dispositivo. Dopo che ogni parola è trasferita (passi 2 e 4),il DMA decide qual è il prossimo dispositivo da servire. Può essere impostato per usare un algoritmo round-robin o può avere uno schema di priorità al fine di favorire alcuni dispositivi rispetto ad altri. Possono essere contemporaneamente in attesa più richieste a controller di dispositivi diversi,purché ci sia un modo inequivocabile per distinguere le conferme. Per questa ragione sul bus è spesso usata una linea di conferma separata per ciascun canale DMA. Molti bus possono operare in due modalità : **word-at-a-time (una parola alla volta)** e **a blocco**. Spesso i controller DMA possono lavorare in entrambi i modi. Nel primo,l'operazione avviene come descritto in precedenza : il controller DMA richiede il trasferimento di una parola e la ottiene;se anche la CPU richiede il bus,deve attendere. Il meccanismo è chiamato **cycle stealing (furto di un ciclo)**,dato che il controller del dispositivo ogni tanto ruba un ciclo del bus alla CPU,rallentandola leggermente. Nella modalità **blocco**,il controller DMA comunica al dispositivo di acquisire il bus,avviare una serie di trasferimenti e poi rilasciare il bus. Questa forma di operazione è chiamata **modalità burst**;è più efficiente del cycle stealing perché acquisire il bus richiede tempo e possono essere trasferite più parole al costo di una sola acquisizione del bus. Il lato negativo di questa modalità è che può bloccare la CPU e altri dispositivi per un tempo considerevole nel caso di un burst di grossa entità. Nel modello **fly-by mode**,il DMA comunica al controller del dispositivo di trasferire i dati direttamente alla memoria principale. Un metodo alternativo usato da alcuni DMA è quello di inviare la parola dal controller del dispositivo al controller del DMA,che poi invia una seconda richiesta del bus per scrivere la parola ovunque debba andare. Questo schema,chiamato **flow-through**,richiede un ciclo di bus extra

per ogni parola trasferita,ma è più flessibile perché può anche eseguire copie da dispositivo a dispositivo e copie da memoria a memoria (inviando alla memoria prima un comando di lettura e poi un comando di scrittura ad un diverso indirizzo).
La maggior parte dei controller DMA usa per i trasferimenti **indirizzi di memoria fisici**;ciò richiede che il SO converta l'indirizzo virtuale del buffer di memoria necessario in un indirizzo fisico e poi scriva questo indirizzo fisico nel registro degli indirizzi del controller DMA. Uno schema alternativo usato in pochi controller DMA prevede la scrittura di indirizzi virtuali nel controller DMA,che quindi deve usare l'MMU per eseguire la traduzione da virtuale a fisico. Gli indirizzi virtuali possono essere messi sul bus solo quando l'MMU è parte della memoria (raro) invece che parte della CPU. Il disco ha bisogno di un buffer interno per due motivi. Il primo è che usando un buffer interno il controller del disco può verificare la checksum prima di avviare il trasferimento e se è sbagliata si segnale un errore e il trasferimento viene annullato. Il secondo è che,una volta partito il trasferimento,i bit iniziano ad arrivare dal disco a velocità costante,che il controller sia pronto a riceverli o no. Se il controller provasse a scrivere i dati direttamente in memoria dovrebbe andare sul bus di sistema per ciascuna parola trasferita. Se il bus fosse occupato da qualche altro dispositivo di I/O (ad esempio in burst),il controller dovrebbe attendere. Se la parola successiva del disco arrivasse prima che la precedente fosse memorizzata,il controller dovrebbe memorizzarla da qualche parte. Se il bus fosse molto impegnato,il controller potrebbe finire per contenere svariate parole e a dover svolgere parecchie attività amministrative. Quando il blocco è nel buffer interno,non c'è bisogno del bus finché non parte il DMA,quindi il design del controller è molto più semplice perché il trasferimento del DMA alla memoria non ha problemi di temporizzazione.

I/O con DMA

Ritornando all'I/O guidato dagli interrupt è che avviene un interrupt ad ogni carattere. Gli interrupt richiedono tempo e quindi si spreca una certa quantità di tempo della CPU. Una soluzione è l'uso del DMA. In questo caso l'idea è di lasciare che il controller DMA invii i caratteri alla stampante uno alla volta,senza disturbare la CPU. In sostanza,il DMA è I/O programmato,solo che il lavoro è delegato al controller DMA anziché alla CPU principale. Ciò richiede hardware speciale (controller DMA),ma lascia che la CPU faccia altre cose durante l'I/O. Una bozza di codice corrispondente è :

```
copy_from_user(buffer,p,count);
set_up_DMA_controller();
scheduler();

acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

Quando si esegue la procedura di servizio interrupt,i dati richiesti sono stati già trasferiti. Il vantaggio del DMA consiste nella riduzione del numero degli interrupt da uno per carattere ad uno per buffer stampato. Però il controller DMA è molto più lento della CPU principale;se il controller DMA non è in grado di condurre il dispositivo a velocità massima o se la CPU non ha comunque nulla da fare mentre è in attesa dell'interrupt del DMA,potrebbe andar meglio l'I/O gestito dagli interrupt o anche quello programmato. La maggior parte delle volte conviene comunque usare un DMA.

Livelli del software di I/O

Il software di I/O è generalmente organizzato in quattro livelli :

1. Software per l'I/O a livello utente;
2. Software del SO indipendente dal dispositivo;
3. Driver dei dispositivi;
4. Gestori degli interrupt

Ciascuno di essi ha una funzione ben definita da eseguire e un'interfaccia ben definita verso i livelli adiacenti. La funzionalità e le interfacce sono diverse a seconda del sistema. Vediamo le funzioni dei vari livelli :

1. Software per l'I/O a livello utente : eseguire la chiamata I/O,formattare l'I/O,spooling;
2. Software del SO indipendente dal dispositivo : denominazione,protezione,blocco,buffering,allocazione;
3. Driver dei dispositivi : impostazione dei registri dei dispositivi,controllo dello stato;
4. Gestori degli interrupt : attivare il driver quando è completato l'I/O;
5. Hardware : eseguire l'operazione di I/O.

Driver dei dispositivi (device driver)

Il controllo di ogni dispositivo di I/O connesso al computer necessita di un certo codice specifico. Questo codice è chiamato **driver di dispositivo (device driver)** ed è solitamente scritto dal produttore del dispositivo stesso e rilasciato in dotazione. Dato che ogni SO necessita dei propri driver,i produttori generalmente forniscono driver per i SO più conosciuti. Ogni driver del dispositivo gestisce normalmente un tipo di dispositivo o al massimo una classe di dispositivi strettamente correlati. Nella maggior parte dei casi non è una buona idea avere un driver che controlli più dispositivi molto diversi tra loro (ad esempio mouse e joystick). A volte però driver profondamente diversi sono basati sulla stessa tecnologia,come l'**USB (Universal Serial Bus)**. I dispositivi che lo utilizzano svolgono compiti totalmente diversi. I driver

USB sono posti su uno stack. Alla base,solitamente nell'hardware,si trova il livello di collegamento USB che gestisce questioni hardware come la segnalazione e la decodifica dei flussi di segnali in pacchetti USB;viene utilizzato dai livelli superiori che hanno a che fare con i pacchetti dati e con le funzionalità comuni condivise dalla maggior parte dei dispositivi USB. Al di sopra di tutto ci sono poi le API di alto livello,come interfacce per memorie di massa,fotocamere ecc.

Per poter accedere all'hardware del dispositivo,cioè ai registri del controller,il device driver deve normalmente essere **parte del kernel** del SO. E' possibile costruire driver eseguiti nello spazio utente,con chiamate di sistema per eseguire la lettura e la scrittura dei registri dei dispositivi. Questo design isola il kernel dai driver e i driver l'uno dall'altro eliminando una causa importante di crash del sistema : driver difettosi che in un modo o nell'altro interferiscono con il kernel. Fanno eccezione alcuni kernel,come MINIX,dove i driver dei dispositivi sono eseguiti come procedure utente.
Dato che i progettisti dei SO sanno che vi saranno installati driver scritti da terzi,l'architettura del SO deve consentirlo. I driver di dispositivo si posizionano normalmente sotto il resto del SO (nel kernel space). I SO solitamente classificano i driver in categorie. Quelle più comuni sono **dispositivi a blocchi**,come i dischi,contenenti molteplici blocchi di dati indirizzabili indipendentemente e **dispositivi a caratteri**,come stampanti e tastiere,che generano o accettano un flusso di caratteri. La maggior parte dei SO definisce un'interfaccia standard che deve essere supportata da tutti i dispositivi a blocchi e una seconda interfaccia standard che deve essere supportata da tutti i dispositivi a caratteri. Esse sono composte da alcune procedure che il resto del SO può chiamare affinché il driver faccia il suo lavoro come la lettura di un blocco o la scrittura di una stringa di caratteri. Come sappiamo,in alcuni sistemi il SO è un singolo programma binario contenente al suo interno i driver compilati che gli servono,ideale se i dispositivi di I/O cambiavano raramente. Se veniva aggiunto un nuovo dispositivo,l'amministratore di sistema ricompilava il kernel col nuovo driver. Pochi sono in grado di farlo e perciò si è arrivati a caricare dinamicamente i driver nel sistema **durante l'esecuzione**. Sistemi diversi gestiscono questo caricamento in modi diversi. Un driver di dispositivo ha **funzioni varie**. La più ovvia è accettare richieste di scrittura e lettura astratte provenienti dal soprastante software indipendente dal dispositivo e fare in modo da portarle a termine. Un'altra funzione è inizializzare,se necessario,il dispositivo e gestire i requisiti di alimentazione e il registro degli eventi.
Molti dispositivi hanno una struttura generale simile. Un **driver classico** inizia verificando se i parametri di input sono validi o meno. Se non lo sono,viene restituito un errore. Se sono validi potrebbe rendersi necessaria una traduzione dall'astratto al concreto. Per i driver di un disco questo potrebbe significare la conversione del numero di un blocco lineare in numeri di testina,traccia,settore e cilindro,mentre per gli SSD il numero di blocco dovrebbe essere mappato su blocco e pagina flash corretti. In seguito il driver potrebbe verificare se il dispositivo sia attualmente in uso. Se lo è,la richiesta andrà in coda per un'elaborazione successiva. Se il dispositivo è inattivo verrà esaminato lo stato dell'hardware per vedere se la richiesta può essere

gestita immediatamente. Controllare il dispositivo significa inviare una serie di comandi. Il driver è il luogo dove si determina la sequenza di comandi,a seconda di ciò che deve essere fatto. Una volta che il driver sa quali comandi sta per inviare,inizia a scriverli nei registri del dispositivo del controller. Dopo la scrittura di ciascun comando nel controller,potrebbe rendersi necessario verificare se il controller abbia accettato il comando e sia pronto ad accettare il successivo;questa sequenza continua finché non sono stati inviati tutti i comandi. Ad alcuni controller può essere passata una lista concatenata di comandi (in memoria),con l'ordine di leggerli ed elaborarli tutti autonomamente,senza ulteriore aiuto dal SO. Dopo l'invio di ciascun comando si possono verificare **due scenari**. Nel **primo** il driver del dispositivo deve aspettare che il controller abbia eseguito un po' di lavoro,quindi si blocca finché arriva un interrupt a sbloccarlo. Nel **secondo**,invece,l'operazione finisce senza ritardo e quindi il driver non ha bisogno di bloccarsi. In entrambi i casi,dopo il completamento dell'operazione,il driver deve **controllare eventuali errori**. Se è tutto a posto,il driver potrebbe avere dei dati (ad esempio un blocco appena letto) da passare al software indipendente dal dispositivo. Alla fine restituirà alcune informazioni di stato per comunicare gli errori al **chiamante**. Se ci sono in coda altre richieste,ora può esserne selezionata ed avviata una.
Il driver è in realtà più complicato. Un dispositivo I/O potrebbe terminare mentre il suo driver è in esecuzione,inviando un interrupt che potrebbe causare l'esecuzione di un driver di dispositivo. Per esempio,mentre il driver della rete sta elaborando un pacchetto di ingresso,potrebbe giungere un altro pacchetto. Perciò i driver dovrebbero essere **rientranti**,ossia un driver in esecuzione deve aspettarsi di essere chiamato una seconda volta prima di aver completato la prima chiamata. In un sistema "a caldo" i driver possono essere aggiunti ed eliminati a computer acceso. Può accadere quindi che un driver occupato nella lettura di un dispositivo venga informato dal sistema che l'utente l'ha rimosso. L'attuale trasferimento di I/O deve essere annullato senza danneggiare strutture dati del kernel ed è necessario eliminare ogni richiesta diretta al dispositivo svanito e comunicarlo ai chiamanti. Al driver non sono consentite chiamate di sistema,però devono spesso interagire col kernel. Di solito sono permesse chiamate a determinate procedure del kernel come allocare e deallocare pagine fisse di memoria da usare come buffer,gestione MMU,controller DMA ecc.

Software per l'I/O indipendente dal dispositivo

Sebbene parte del software per l'I/O sia specifico di un determinato dispositivo,altre parti sono indipendenti dal dispositivo stesso (device indipendent). Il limite esatto fra i driver e il software indipendente dal dispositivo dipende dal sistema e dal dispositivo,poiché alcune funzioni che potrebbero essere svolte in modalità indipendente dal dispositivo sono effettivamente svolte dai driver,sia per efficienza sia per altre ragioni. La funzione base del software per l'I/O indipendente dal dispositivo è quella di eseguire tutte quelle funzioni di I/O trasversali a tutti i dispositivi e fornire un'interfaccia uniforme al software a livello utente.

Le funzioni svolte da questo livello sono :

1. Interfacciamento uniforme dei driver dei dispositivi;
2. Buffering;
3. Segnalazione errori;
4. Allocazione e rilascio dei dispositivi dedicati;
5. Dimensione dei blocchi indipendente dal dispositivo.

Interfacciamento uniforme dei driver dei dispositivi

Una questione fondamentale di un SO è come rendere tutti i dispositivi e i driver più o meno simili tra loro. Se dischi,stampanti,tastiere e così via sono tutti interfacciati ogni volta in modo diverso,ogni volta che arriva un nuovo dispositivo il SO va modificato. Intaccare il SO per ogni nuovo dispositivo non è una buona idea. Un aspetto della questione è l'interfaccia tra i driver di dispositivo e il resto del SO. Abbiamo due scenari. Il **primo** è quello dove ogni driver ha una diversa interfaccia verso il SO e ciò significa che le funzioni dei driver chiamabili dal sistema sono diverse a seconda del driver. Potrebbe significare anche che le funzioni del kernel che servono al driver sono diverse a seconda del driver. Il **secondo** è quello dove tutti i driver hanno la stessa interfaccia. Risulta così molto più semplice inserire un nuovo driver,a patto che sia conforme all'interfaccia comune. Significa,inoltre,che chi scrive i driver sa che cosa ci si aspetta da loro. In termini pratici,non tutti i dispositivi sono assolutamente identici,ma di solito i tipi di dispositivi sono pochi e in genere sono quasi uguali.
Per ogni classe di dispositivi,come dischi o stampanti,il SO definisce un insieme di funzioni che il driver deve supportare. Per un disco queste includeranno lettura e scrittura,accensione e spegnimento,formattazione e altre azioni specifiche. Spesso il driver contiene una tabella con puntatori a sé stesso per queste funzioni. Una volta caricato il driver,il SO registra l'indirizzo di questa tabella di puntatori a funzioni,così quando ha bisogno di chiamarne una può fare una chiamata indiretta tramite questa tabella. Questa tabella di puntatori a funzioni definisce l'interfaccia tra il driver e il resto del SO. Un altro aspetto dell'uniformità di un'interfaccia riguarda la **denominazione dei dispositivi di I/O**. Il software indipendente dal dispositivo si occupa di mappare i nomi simbolici dei dispositivi nel driver adatto. In UNIX,ogni device di I/O viene visto,a tutti gli effetti come un **file (file speciale)**. I file speciali si trovano nella directory **dev**. Sono caratterizzati da tipo (blocchi/caratteri),classe dispositivo (major device number) e istanza dispositivo (minor device number) → ls -la /dev/sda* (VEDI COSA FA). L'apertura di un file speciale restituisce un **descrittore**,che punta ad una **struct** per effettuare la lettura-scrittura di blocchi o flussi di caratteri dal/sul dispositivo. L'interfaccia di accesso è **uniforme** e il driver implementa le funzionalità. Ad esempio,un nome di dispositivo come */dev/disk0* specifica in modo univoco l'i-node di un file speciale e questo i-node contiene il **major device number (numero di dispositivo primario)**,usato per localizzare il driver adeguato. L'i-node contiene anche il **minor device number (numero di dispositivo secondario)**,passato come parametro al driver al fine di specificare

l'unità da leggere o scrivere. Tutti i dispositivi hanno un numero di dispositivo primario e secondario,e il numero di dispositivo primario è quello che indica il driver e consente di accedervi. Per quanto riguarda la protezione,sia in UNIX sia in Windows i dispositivi appaiono nel file system come oggetti denominati e quindi sono visti come file a tutti gli effetti a cui poter applicare le regole per la protezione dei file.

Buffering

Per una varietà di ragioni,anche il buffering costituisce un problema,sia per i dispositivi a blocchi sia per quelli a caratteri. Consideriamo un processo che voglia leggere dei dati da un modem. Una possibile strategia per gestire i caratteri in ingresso consiste nell'avere un processo utente che esegue una chiamata di sistema **read()** e si blocca in attesa di un carattere. Ogni carattere in arrivo causa un interrupt. La procedura di servizio degli interrupt passa il carattere al processo utente e si sblocca. Dopo aver messo il carattere da qualche parte,il processo legge un altro carattere e si blocca di nuovo. Il problema che sorge in questo caso è che il processo utente deve essere riavviato ad ogni carattere in ingresso. Consentire che un processo sia eseguito più volte per brevi periodi non è efficiente e quindi non è uno schema valido.
Un miglioramento è quello dove il processo fornisce un buffer di *n* caratteri nello spazio utente e fa quindi una read di *n* caratteri. La procedura di servizio degli interrupt mette i caratteri in ingresso nel buffer finché non è pieno. Solo a quel punto il processo viene risvegliato. Nel caso in cui però il buffer viene paginato in uscita all'arrivo di un carattere,il buffer potrebbe essere bloccato in memoria,ma se molti processi iniziano a bloccare pagine in memoria senza un criterio,l'insieme di pagine disponibili si riduce con un calo delle prestazioni.
Un ulteriore approccio è quello di creare un buffer all'interno del kernel e fare in modo che il gestore degli interrupt vi inserisca i caratteri. Quando questo buffer è pieno,la pagina con il buffer viene paginata in ingresso,se necessario,e il buffer vi viene copiato in una sola operazione. Questo schema che copia dal buffer kernel al buffer utente è più efficace. Se però arrivano caratteri mentre la pagina con il buffer viene letta dal disco,dato che il buffer è pieno non c'è posto dove metterli. Si può risolvere questo aspetto inserendo un ulteriore buffer nel kernel : dopo aver riempito il primo,ma prima che sia svuotato,si utilizza il secondo. Quando il secondo è pieno lo si copia nel buffer utente (supponendo che l'utente lo abbia richiesto);mentre il secondo buffer viene copiato,il primo può ricevere nuovi caratteri. In questo modo i due buffer fanno a turno : mentre uno viene usato per copiare nello spazio utente,l'altro accumula i nuovi input. Questo tipo di buffering è detto **buffering doppio**. L'uso di un buffer è importante anche in fase di **output**. Consideriamo ad esempio come viene eseguito l'output su un modem senza buffer nel kernel. Il processo utente esegue una chiamata di sistema *write* per eseguire l'output di *n* caratteri. A questo punto il sistema ha due possibilità. Può bloccare l'utente finché siano stati scritti tutti i caratteri,ma ciò potrebbe richiedere molto tempo su una linea

lenta. Potrebbe rilasciare immediatamente l'utente e gestire l'I/O mentre l'utente calcola qualcos'altro,ma ciò causa il problema che il processo non potrà capire quando l'output è stato completato e che quindi può riutilizzare il buffer. Il sistema potrebbe generare un segnale o un interrupt software,ma questo stile di programmazione è difficile e incline a race condition.
Una soluzione decisamente migliore per il kernel è quella di copiare i dati in un buffer del kernel,e cioè dallo spazio utente al kernel space e sbloccare immediatamente il chiamante. A questo punto non importa quando l'I/O effettivo è stato completato;l'utente è libero di riutilizzare il buffer nel momento in cui è sbloccato.
Il buffering è una tecnica largamente usata ma ha una controindicazione : se i dati sono messi troppe volte nel buffer,le prestazioni ne risentono. Consideriamo ad esempio una rete. Quando un utente effettua una chiamata di sistema per scrivere sulla rete,il kernel copia il pacchetto in un buffer del kernel per permettere all'utente di procedere immediatamente. A questo punto il programma utente può riutilizzare il buffer. Quando viene chiamato,il driver copia i dati sul controller di rete per l'output. Il motivo per cui non esegue l'output direttamente dalla memoria del kernel è che,una volta avviata,una trasmissione a pacchetti deve continuare a velocità uniforme. Il driver non può garantire una velocità uniforme di accesso alla memoria,dato che i canali DMA e altri dispositivi di I/O possono rubare molti cicli. Non riuscire a trasmettere una parola in tempo rovinerebbe il pacchetto e perciò,mettendolo nel buffer del controller si evita il problema. Dopo che il pacchetto è stato copiato nel controller di rete,viene copiato sulla rete. I bit arrivano al destinatario rapidamente dopo il loro invio,così l'ultimo bit inviato raggiunge immediatamente il ricevente,dove il pacchetto è stato bufferizzato nel controller. Il pacchetto è poi copiato nel buffer del kernel del ricevente e infine copiato nel buffer del processo ricevente. Di solito a questo punto il ricevente restituisce una conferma;quando il mittente la riceve è pronto a inviare il pacchetto successivo. E' evidente che tutte queste operazioni di copia rallentano considerevolmente la velocità di trasmissione,dato che tutti i passaggi devono avvenire sequenzialmente. Un contro è quindi il possibile elevato numero di copie di un dato. Le copie buffer utente kernel e kernel utente si fanno ma il vero collo di bottiglia sono le copie memorie dispositivo,che passano tramite DMA e impiegano diverso tempo.

Segnalazione degli errori

Gli errori sono molto frequenti nel contesto di I/O. Quando accadono il SO deve gestirli meglio che può. Molti errori sono specifici del dispositivo e devono essere gestiti tramite un driver appropriato,ma la struttura per la loro gestione è indipendente dal dispositivo. Una classe di errori di I/O è quella degli errori di programmazione;si verificano quando un processo richiede qualcosa di impossibile,come la scrittura su un dispositivo di input (tastiera,mouse …) o lettura da un dispositivo di output (stampante,plotter …). Altri errori sono fornite un indirizzo di buffer o un altro parametro non valido e specificare un dispositivo errato e così via. Si deve in questi casi inviare un codice d'errore al chiamante. Un altra classe di

errori è quella di errori di I/O,ad esempio un tentativo di scrittura su un blocco danneggiato o leggere da una videocamera spenta. In questi casi il driver decide cosa fare;se non sa che strada intraprendere,il problema passa al software indipendente dal dispositivo. L'azione che esso intraprende dipende dall'errore e dall'ambiente. Se è un errore di lettura e c'è un utente interattivo disponibile,può visualizzare una finestra di dialogo con l'utente. Le opzioni possono includere il riprovare un certo numero di volte,ignorare l'errore o terminare il processo chiamante. Se non c'è l'utente disponibile,viene riportato un codice d'errore (cosa che accade in altri casi).

Allocazione e rilascio dei dispositivi dedicati

Alcuni dispositivi,come le stampanti,possono essere usati solo da un singolo processo in un determinato momento. Sta al SO esaminare le richieste di uso del dispositivo e accettarle o rifiutarle,a seconda che il dispositivo sia disponibile o meno. Un modo semplice per gestire le richieste è quello di chiedere ai processi di fare la *open* direttamente su file speciali per i dispositivi. La chiusura di questi dispositivi rilascia i file. Un approccio alternativo è quello di avere meccanismi speciali per richiedere e rilasciare dispositivi dedicati : anziché fallire,un tentativo di acquisizione di un dispositivo non disponibile blocca il chiamante. I processi bloccati sono messi in coda e quando il dispositivo torna disponibile il primo processo in coda lo acquisisce e continua l'esecuzione.

Dimensione dei blocchi indipendente dal dispositivo

SSD diversi hanno pagine flash di diverse dimensioni,e dischi diversi possono avere settori di dimensioni diverse. Il software indipendente dal dispositivo ha il compito di occultare questo aspetto e fornire una dimensione di blocco uniforme ai livelli posti più in alto,ad esempio trattando alcuni settori o pagine flash come blocco logico singolo. In questo modo i livelli superiori hanno a che fare solo con dispositivi astratti,che useranno la stessa dimensione di blocco logico indipendentemente dalla dimensione del settore fisico.

Livello utente

Anche se la maggior parte del software di I/O risieda nel SO,una piccola parte è composta da **librerie collegate** con i programmi utente e anche da interi programmi eseguiti al di fuori del kernel. Le chiamate di sistema,incluse quelle per l'I/O,sono normalmente eseguite da procedure di libreria.

Memoria di massa (Dischi e SSD)

Dischi magnetici

I dischi rigidi magnetici sono caratterizzati dal fatto che le operazioni di lettura e scrittura sono ugualmente veloci,rendendoli l'ideale come memoria secondaria. Array di questi dischi sono utilizzati anche per fornire memoria altamente affidabile. Sono organizzati in cilindri,ciascuno contenente tante tracce (cerchi concentrici) quante sono le testine impilate verticalmente. Le tracce sono divise in settori e il numero di settori lungo la circonferenza raggiunge generalmente varie centinaia. Le testine variano da 1 a 16. Nei dischi **SATA (serial ATA)**,l'unità disco contiene un microcontroller che svolge una parte considerevole del lavoro e permette al controller vero e proprio di inviare un insieme di comandi ad alto livello. Il controller si occupa del caching delle tracce,il rimappaggio dei blocchi difettosi ed altro. Una caratteristica che ha importanti implicazioni per il driver del disco è la possibilità che un controller possa eseguire ricerche su due o più unità allo stesso tempo,dette **ricerche sovrapposte (overlapped seek)**. Mentre il controller e il software sono in attesa che si completi una ricerca su un'unità,il controller può iniziare una ricerca su un'altra unità. Molti controller possono anche leggere o scrivere su un'unità mentre stanno eseguendo una ricerca su una o più altre unità,e un sistema con più hard disk con controller integrati può utilizzarli contemporaneamente,almeno per i trasferimenti tra disco e memoria buffer del controller. In ogni caso è possibile un solo trasferimento alla volta tra il controller e la memoria principale.
nota : vedi cap4

Formattazione dei dischi

Formattazione a basso livello → vedi cap4.
La posizione del settore 0 su ciascuna traccia viene spostata di un certo offset rispetto al settore 0 della traccia precedente in fase di formattazione a basso livello. Questo offset,chiamato **cylinder skew (sfasamento del cilindro)**,serve a migliorare le prestazioni. L'idea è quella di consentire al disco di leggere più tracce in un'unica operazione continua senza perdere dati. Supponiamo che una richiesta abbia bisogno di 18 settori a partire dal settore 0 sulla traccia più interna. La lettura dei primi 16 settori richiede una rotazione del disco,ma serve una ricerca per posizionarsi sulla nuova traccia per prendere il settore 17. Durante lo spostamento di una traccia della testina,la rotazione ha fatto finire il settore 0 oltre la testina,quindi serve un'altra rotazione prima che vi ripassi. Questo problema viene risolto con l'offset dei settori. Si noti che anche il passaggio tra una testina e l'altra comporta un tempo finito,quindi abbiamo anche un **head skew** oltre a quello dei cilindri,ma non è molto grande.
Se consideriamo un controller con un buffer di un settore a cui sia stato dato un comando di lettura di due settori consecutivi,dopo la lettura del primo settore e il calcolo dell'ECC,i dati devono essere trasferiti alla memoria principale. Durante questo trasferimento il settore seguente supererà la testina,e dopo aver completata la copia il controller dovrà aspettare quasi un intero periodo di rotazione prima che il secondo settore torni sotto la testina. Con **interleaving singolo**,si dà un po' di

respiro al controller tra due settori consecutivi mentre copia il buffer in memoria principale. Se il processo di copia è molto lento,allora può rendersi necessario l'**interleaving doppio**. Se il controller ha un buffer di un solo settore,non importa se la copia dal buffer alla memoria principale è eseguita dal controller,dalla CPU principale o anche da un chip DMA,si impiegherà sempre del tempo. Per evitare l'interleaving il controller dovrebbe bufferizzare un'intera traccia.
Il passaggio finale nella preparazione di un disco è la **formattazione ad alto livello** di ciascuna partizione (separatamente). Questa operazione predispone un blocco di avvio,l'amministratore dello spazio di memorizzazione libero (elenco o bitmap dei blocchi liberi),directory radice e un file system (vuoto). Inserisce nella tabella delle partizioni un codice che indica quale file system sia utilizzato nella partizione,dato che molti SO supportano più file system compatibili e il sistema può essere poi avviato.

Algoritmi di scheduling del braccio del disco

Per la maggior parte dei dischi,il tempo di ricerca è decisamente maggiore degli altri due,cosicché la riduzione del tempo medio di ricerca migliora sostanzialmente le prestazioni del sistema. Vedremo tre algoritmi :

1. **First-Come,First-Served (FCFS);**
2. **Shortest Seek First (SSF);**
3. **Algoritmo dell'ascensore.**

First-come,First-served

Il driver accetta una richiesta per volta e le esegue nello stesso ordine,in modo tale da servire la prima che arriva. Molti driver dei dischi mantengono una tabella,indicizzata per numero di cilindro,con tutte le richieste in attesa di ciascun cilindro unite in una lista concatenata alla testa delle voci della tabella. Ciò è necessario perché potrebbe verificarsi,se il disco è utilizzato intensamente,che mentre il braccio sta compiendo una ricerca sulla base di una richiesta,altri processi generino altre richieste per il disco.
Immaginiamo uno scenario con un disco di 40 cilindri. Arriva una richiesta di lettura di un blocco sul cilindro 11. Mentre è in corso la ricerca del cilindro 11,arrivano richieste dei cilindri 1,36,16,34,9 e 12. Vengono inserite nella tabella delle richieste in attesa,con una lista distinta per ciascun cilindro. Al termine della richiesta del cilindro 11,il driver sceglie quale richiesta gestire come successiva. Con questo algoritmo seguirebbe l'ordine di arrivo delle richieste e quindi : 1,36 e così via. Questo algoritmo richiederebbe rispettivamente movimenti del braccio di 10,35,20,18,25 e 3 cilindri,per un totale di **111**.

Shortest Seek First

Per migliorare le prestazioni dell'algoritmo FCFS,si può apportare una modifica per ridurre al minimo il tempo di ricerca,gestendo sempre come richiesta seguente la più vicina. Prendiamo sempre lo scenario con richiesta iniziale 11 e richieste successive 1,36,16,34,9,12. In questo caso la sequenza è 12,9,16,1,34,36 e i movimenti del braccio sono 1,3,7,15,33 e 2,per un totale di **61 cilindri**. Si riduce così lo spostamento totale del braccio quasi della metà in confronto all'FCFS. Abbiamo però un problema. Supponiamo che continuino ad arrivare altre richieste durante l'elaborazione delle altre. Per esempio,se dopo essere andato al cilindro 16 giunge una nuova richiesta per il cilindro 8,quella richiesta avrà priorità alta rispetto a quella del cilindro 1. Se arriva poi una richiesta del cilindro 13,il braccio vi si porterà escludendo di nuovo il cilindro 1. Con un disco che subisce un forte carico di lavoro,il braccio avrà la tendenza a restare per la maggior parte del tempo nel mezzo del disco,pertanto le richieste ad entrambi gli estremi dovranno aspettare finché una fluttuazione statistica del carico di lavoro porterà a non avere richieste nel mezzo. Le richieste lontane dal centro saranno servite in modo scadente. In questo caso gli obiettivi di un tempo di risposta minimo e di imparzialità sono in contrasto.

Algoritmo dell'ascensore

Prendiamo come esempio gli edifici alti. Il problema dello scheduling di un ascensore in un grattacielo è simile a quello del braccio di un disco. Arrivano continuamente richieste che chiamano l'ascensore ai piani (cilindri) a caso. Il computer che gestisce l'ascensore potrebbe facilmente tenere traccia della sequenza delle richieste in cui gli utenti hanno premuto il pulsante di chiamata e servirli tramite FCFS o SSF. Per trovare un punto di incontro tra efficienza e imparzialità,la maggior parte degli ascensori usa un algoritmo diverso : continuano a muoversi in una direzione finché non vi sono più richieste inevase in quella direzione,quindi cambiano direzione. L'**algoritmo dell'ascensore** richiede al software di mantenere un bit : quello della direzione attuale,**UP** o **DOWN**. Al termine di una richiesta,il driver del disco controlla il bit. Se è UP,il braccio è spostato alla richiesta successiva in attesa verso l'alto. Se non vi sono richieste in attesa più in alto,il bit della direzione viene invertito. Quando il bit è impostato a DOWN,lo spostamento è nella posizione successiva richiesta verso il basso,se c'è;in assenza di altra richieste,il braccio si ferma e aspetta. Il software dei dischi,di solito,non cerca di pre-posizionare speculativamente la testina (a differenza degli ascensori che dopo un po' tornano a piano terra perché è più probabile che una nuova richiesta sia vicina ad esso).
Tornando allo scenario di richieste d'esempio consideriamo il bit inizialmente UP. L'ordine in cui sono serviti i cilindri è 12,16,34,9 e 1, il che comporta spostamenti del braccio di 1,4,18,2,27 e 9 per un totale di **60 cilindri**. In questo caso l'algoritmo dell'ascensore è leggermente migliore dell'SSF,sebbene in genere sia peggiore. Una caratteristica dell'algoritmo dell'ascensore è che,data una qualunque raccolta di richieste,il limite massimo dello spostamento totale è fisso : due volte il numero dei cilindri.

Solid State Drive (SSD)

Le unità SSD impiegano la memoria flash e funzionano in modo molto diverso dai dischi magnetici. I dischi magnetici sono costituiti da una sequenza di piccoli magneti: due poli diversi vicini corrispondono ad uno 0 e due uguali ad 1. Di solito negli SSD è utilizzata memoria flash con tecnologia NAND (anziché NOR);gran parte della differenza è legata agli aspetti fisici sui quali si basa la memorizzazione. A prescindere dalla tecnologia,abbiamo differenze tra dischi ed SSD : nella memoria flash non abbiamo parti in movimento,quindi non esistono nemmeno i problemi di tempi di ricerca e di ritardi dovuti alla rotazione. Ciò significa che il tempo di accesso è molto migliore,nell'ordine delle decine di microsecondi anziché di millesimi di secondo. Significa che negli SSD non c'è molta differenza in termini di prestazioni tra ricerche casuali e sequenziali. A differenza dei dischi magnetici,la tecnologia flash ha prestazioni asimmetriche in lettura e scrittura : le letture sono molto più veloci delle scritture. Ad esempio,se una lettura richiede qualche decina di microsecondi,le scritture possono richiederne centinaia. Un primo motivo è la programmazione delle celle flash che implementano i bit;è una questione di fisica. Un secondo motivo,di maggiore impatto,è che è possibile scrivere un'unità di dati solo dopo aver cancellato un'area idonea del dispositivo. Infatti la memoria flash fa una distinzione tra **unità di I/O** (pagine,tipicamente da 4KB) e **unità di cancellazione (**blocchi,spesso da 64-256 unità di I/O,quindi diversi MB). Il concetto di **pagina** è diverso da quello di **pagina di memoria** e quindi parleremo di **pagina flash** e **blocco flash**. Per scrivere una pagina flash,l'SSD deve prima cancellare un blocco flash,operazione costosa che impiega centinaia di microsecondi;fortunatamente,dopo la cancellazione è disponibile spazio libero per molte pagine flash e l'SSD può scrivere le pagine flash nel blocco flash in ordine. In altre parole,scrive nel blocco prima la pagina 0,1,2 e così via. Non può scrivere la pagina 0 seguita dalla 2 e poi la 1. Inoltre,l'SSD non può sovrascrivere una pagina flash scritta in precedenza : deve prima cancellare di nuovo tutto il blocco flash. La modifica di dati su un SSD,invece,rende semplicemente non valida la vecchia pagina flash e riscrive il nuovo contenuto in un altro blocco. Se non ci sono blocchi flash con pagine flash libere,sarà necessario cancellare prima un altro blocco. Non è desiderabile continuare a scrivere sempre le stesse pagine flash,comunque,perché la memoria flash soffre di usura. Scritture e cancellazioni ripetute sono onerose,e a un certo punto le celle flash che contengono i bit non sono più utilizzabili. Un ciclo **program/erase (P/E,programmazione cancellazione)** consiste nel cancellare una cella e scrivere nuovi contenuti al suo interno. La tipica cella di memoria flash ha una durata massima di qualche migliaio o centinaio di migliaia di cicli P/E. E' quindi importante spalmare l'usura il più possibile sulle celle di memoria.

Il componente del dispositivo responsabile della gestione uniforme dell'usura è chiamato **FTL (Flash Translation Layer)**. E' responsabile anche di parecchi altri aspetti e gira su un semplice processore con accesso ad una memoria veloce. I dati sono memorizzati nei pacchetti flash (FB);ciascun pacchetto flash è composto da più **die** (wafer di silicio),ciascuno dei quali contiene a sua volta un certo numero di

piani,ovvero collezioni di blocchi flash contenenti pagine flash. Per accedere ad una specifica pagina flash sull'SSD è necessario indirizzare correttamente,in ordine gerarchico : il pacchetto flash,il die,il piano,il blocco e la pagina. E' però un indirizzamento complesso e lontano dal funzionamento del SO,che richiede semplicemente la lettura di un blocco del disco a un indirizzo lineare e logico. L'FTL usa delle tabelle di traduzione per indicare che il blocco logico 54321 è in realtà nel die 0 del pacchetto flash 1,nel piano 2,nel blocco 5. Queste tabelle di traduzione tornano utili anche per il livellamento dell'usura (wear leveling),poiché il dispositivo è libero di spostare una pagina in un blocco diverso (ad es. perché deve essere aggiornata),a patto che aggiorni la mappatura nella tabella di traduzione. L'FTL si occupa anche di gestire blocchi e pagine non più necessari. Supponiamo che dopo aver eliminato o spostato dati per qualche volta un blocco flash contenga alcune pagine flash non più valide. Poiché ora solo alcune pagine sono valide,il dispositivo può liberare spazio copiando le restanti pagine in un altro blocco con delle pagine libere e poi cancellando il blocco originale. Questa operazione è chiamata **garbage collection**.

Raid

Una tecnica che attualmente aiuta a migliorare l'affidabilità dei sistemi di memorizzazione in generale è nata come misura per migliorare le prestazioni dei sistemi di memorizzazione su dischi magnetici. Per velocizzare le prestazioni della CPU è sempre più utilizzata l'elaborazione parallela. Molti hanno pensato che il parallelismo dell'I/O potesse essere un'idea altrettanto buona. Nella loro pubblicazione del 1988,Patterson et al. suggerivano sei specifiche organizzazioni dei dischi che avrebbero potuto migliorarne le prestazioni e l'affidabilità. Queste idee furono rapidamente adottate dal settore e diedero vita ad una classe di dispositivi di I/O chiamati **RAID**,**Redundat Array Of Inexpensive Disks**. L'idea base del RAID è installare un contenitore pieno di dischi accanto al computer,solitamente un grosso server,sostituire la scheda controller dei dischi con un controller RAID,copiare i dati nel RAID e quindi continuare come al solito. Oltre ad apparire al software come un disco singolo,un'altra proprietà dei RAID è che i dati sono distribuiti su tutti i dischi. Per raggiungere questo scopo abbiamo diversi schemi,e oggi la maggior parte dei produttori indica da RAID 0 a RAID 6 le sette configurazioni standard. Il termine "**livello**" è improprio,visto che tra questi sei livelli non vi è alcuna gerarchia;si tratta solo di sette diverse modalità di organizzazione.
Il RAID di **livello 0** consiste nel vedere il singolo disco virtuale simulato dal RAID suddiviso in **strip (strisce)** di **k** settori ciascuna,con la strip 0 costituita dai settori da 0 a k-1,la strip 1 quelli da k a 2k-1 e così via. Per k = 1,ogni strip è un settore; per k = 2 ogni strip è costituita da due settori e così via. L'organizzazione del RAID di livello 0 scrive strip consecutive sulle unità in modalità round-robin. Questa distribuzione dei dati su più unità è chiamata **striping**. Se per esempio il software invia un comando per leggere un blocco di dati che consiste di quattro strip consecutive iniziando al limite di una strip,il controller RAID genera quattro comandi separati,uno

per ciascuno dei quattro dischi,facendoli operare in parallelo. In questo modo abbiamo un I/O parallelo senza che il software ne sia a conoscenza. Il RAID di livello 0 funziona meglio con richieste grandi e più lo sono meglio è. Se una richiesta è superiore al numero di unità moltiplicato per la dimensione della strip,alcune unità riceveranno più richieste,cosicché quando termineranno la prima richiesta faranno partire la seconda. Le prestazioni sono eccellenti e l'implementazione è semplice. Il RAID di livello 0 funziona peggio con i SO che abitualmente richiedono dati un settore per volta : i risultati saranno corretti,ma senza parallelismo e senza guadagno di prestazioni. Il RAID di livello 0 ha poi lo svantaggio dell'affidabilità : ogni volta che un'unità va in errore tutti i dati saranno persi completamente.

Il RAID di **livello 1** duplica tutti i dischi,con quattro dischi (in questo caso) primari e quattro dischi di backup. In scrittura ogni strip è scritta due volte. In lettura può essere usata qualunque copia,distribuendo il carico su più unità. Di conseguenza le prestazioni in scrittura non migliorano rispetto a un'unica unità,ma quelle in lettura possono migliorare fino al doppio. La tolleranza agli errori (fault tolerance) è eccellente : in caso di guasto di un disco viene semplicemente usata la copia sull'altro. Il ripristino consiste nell'installare una nuova unità e nel copiarvi l'intero disco di backup.

Il RAID di **livello 4** è come il RAID di livello 0,con parità **strip-per-strip** scritta su un'unità extra. Per esempio,se ogni strip è lunga k byte,viene eseguito l'OR esclusivo di tutte le strip,risultando in una strip di parità lunga k byte. Se un'unità va in crash,i byte persi possono essere ricalcolati a partire dall'unità di parità,leggendo l'intero insieme di unità. Questo schema protegge dalla perdita di un'unità,ma ha prestazioni scadenti per piccoli aggiornamenti. Se viene modificato un settore è necessario leggere tutte le unità al fine di ricalcolare la parità,che deve poi essere riscritta. In alternativa può leggere i vecchi dati dell'utente e vecchi dati della parità e da essi ricostruire la nuova parità. Anche con questo metodo,un piccolo aggiornamento richiede due letture e due scritture. Il forte carico sull'unità di parità può trasformarla in un collo di bottiglia.

Il RAID di **livello 5** elimina il collo di bottiglia del RAID di livello 4 distribuendo i bit di parità in modo uniforme su tutte le unità,in stile round-robin. Tuttavia,la ricostruzione di un'unità nell'evenienza di un suo crash è un procedimento complesso.

Il RAID di **livello 6** è come quello di livello 5,a parte il fatto che si utilizza un blocco di parità aggiuntivo;in altre parole,si fa uno striping dei dati su tutti i dischi con due blocchi di parità invece di uno solo. Come risultato,le operazioni di scrittura sono un po' più costose a causa dei calcoli della parità,ma le letture non hanno svantaggi prestazionali. Inoltre offre una maggiore affidabilità.

Nonostante gli SSD offrono prestazioni migliori e maggiore affidabilità rispetto ai dischi rigidi,i RAID vengono usati anche in questo caso dato che possono offrire

prestazioni e affidabilità migliori rispetto ad un solo SSD. Con gli SSD,per una maggiore affidabilità si può optare per i livelli RAID più alti,come RAID 1;può migliorare la prestazione in lettura (se un SSD è occupato,un altro è libero),ma non in scrittura perché tutti i dati vanno memorizzati due volte e devono essere verificati gli errori. Inoltre,visto che è possibile utilizzare solo la metà dello spazio di memorizzazione,RAID 1 è costoso,soprattutto perché gli SSD costano di più degli HDD. Anche i RAID livello 5 e 6 sono utilizzati con gli SSD;i vantaggi sono migliori prestazioni e affidabilità. Lo svantaggio è che la componente di scrittura è molto intesa,e richiedono un numero importante di scritture aggiuntive a causa dei blocchi di parità. Le scritture,inoltre,aumentano l'usura degli SSD.

Deadlock

Esempio motivazionale

Quando ci troviamo davanti ad uno scenario concorrente,vengono eseguite una serie di operazioni : possiamo avere una serie di risorse IPC coinvolte,ci potrebbero essere problemi di mutua esclusione oppure cooperazione di processi che producono e consumano da una determinata risorsa. Se le cose non vengono effettuate in ordine opportuno si potrebbe bloccare tutto.
Supponiamo di avere un processo A e un processo B che provano a chiedere due risorse. Abbiamo due semafori,risorsa 1 e risorsa 2 impostati ad 1 i quali servono a controllare l'accesso a due aree di memoria su cui serve la mutua esclusione. I processi vengono lanciati. Il codice del processo A fa una *wait* sul primo semaforo,il valore del primo semaforo viene decrementato e diventa 0 (essendo 0 il processo non viene sospeso). Supponiamo che dopo aver effettuato la *wait* il quanto di tempo di A sia esaurito e lo scheduler selezioni il processo B,il quale farà una *wait* sulla risorsa 2 (il valore del secondo semaforo viene decrementato di 1 e diventa 0). B finisce il suo quanto di tempo e si ritorna ad A,il quale effettua una *wait* su risorsa 2,decrementa il semaforo a -1 e a questo punto il processo viene messo nella coda dei processi in attesa. Lo scheduler seleziona dalla coda dei processi pronti B che dovrà fare una *wait* sulla prima risorsa,decrementa il semaforo a -1 e B va a finire nella coda dei processi in attesa del primo semaforo. Abbiamo così due processi bloccati in attesa di un evento che può essere generato solo da un altro processo appartenente al set dei processi in attesa. Se forzassi una politica per cui chi chiede delle risorse,le deve chiedere tutte nello stesso ordine si potrebbe risolvere.

Problema dei cinque filosofi

Esso rappresenta il caso in cui un certo numero di processi si alternano in due attività. I filosofi pensano e di tanto in tanto vanno a mangiare. Abbiamo una tavola con 5 piatti e 5 forchette. Il problema è che chi mangia ha bisogno di due forchette. Una soluzione che porterà ad un deadlock è : il filosofo pensa e prende la forchetta.

A questo punto ci sono 5 persone sedute con una forchetta,non potrà prenderne un'altra e non potrà mangiare.

Deadlock

Un deadlock può essere definito in questo modo : un set di processi è in **stato di deadlock** se ciascun processo del set è in attesa di un evento che può essere provocato solo da un altro processo del set.
Dato che tutti i processi sono in attesa e i processi in attesa rimangono tali fino a che non si verifica l'evento per cui si erano bloccati,l'elaborazione non continuerà dato che nessuno di essi provocherà mai uno degli eventi che potrebbe risvegliare uno degli altri membri del set e quindi rimarranno per sempre in attesa. Nella maggior parte dei casi l'evento che ciascun processo aspetta è il rilascio di qualche risorsa attualmente posseduta da un altro membro del set. In altre parole,ciascun membro del gruppo dei processi in deadlock è in attesa di una risorsa posseduta da un altro processo in deadlock. Nessun processo può essere eseguito,nessuno può rilasciare una risorsa e nessuno può essere risvegliato. Il numero di processi e il numero e la tipologia delle risorse possedute non ha importanza mentre il risultato vale per ogni tipo di risorsa sia hardware sia software. Questo tipo di deadlock è chiamato **deadlock delle risorse**. Non è l'unico tipo di deadlock,abbiamo ad esempio deadlock delle comunicazioni (due macchine connesse su una rete che si devono parlare. La comunicazione avviene tramite scambi di pacchetti e ad esempio se il pacchetto in andata si perde,chi deve ricevere non riceve nulla si blocca e si bloccherà anche il primo che ha mandato il pacchetto perché non riceverà nulla). In un deadlock delle risorse abbiamo bisogno in genere di un'area di memoria (hw/sw) bloccata.

Condizioni per i deadlock delle risorse

Un deadlock delle risorse può avvenire se sussistono quattro condizioni :

1. **condizione di mutua esclusione** : ciascuna risorsa è assegnata ad esattamente ad un processo oppure è libera;
2. **condizione di possesso e attesa** : i processi che allo stato attuale detengono risorse assegnate in precedenza possono richiederne di nuove;
3. **condizione di impossibilità di prelazione** : le risorse assegnate in precedenza non possono essere espropriate a forza ad un processo. Esse devono essere esplicitamente rilasciate dal processo che le detiene. (nota : una risorsa può essere in genere prelazionabile o non prelazionabile);
4. **condizione di attesa circolare** : ci deve essere una lista circolare di due o più processi,ciascuno dei quali è in attesa di una risorsa trattenuta dal membro successivo della catena.

Affinché avvenga un deadlock delle risorse devono sussistere tutte queste quattro condizioni. Se si riesce ad invalidare una di esse si può “risolvere” il problema del deadlock. Inoltre,ogni condizione si riferisce ad una politica che un sistema può avere o meno.

Gestione-risoluzione dei deadlock

Generalmente,per affrontare i deadlock si utilizzano quattro strategie :

1. Ignorare semplicemente il problema. Facendo finta che non esista,magari non esisterà. Quindi si lascia eseguire il tutto e se si blocca riavvieremo il computer. Si tratta dell’**algoritmo dello struzzo**. Si stima quante volte accade il problema,quante volte il sistema si blocca per altri motivi e quanto sia serio il deadlock. Se i deadlock ad esempio avvengono in media una volta ogni cinque anni,ma il sistema va in crash una volta alla settimana per problemi hardware e bug del SO,la maggior parte degli ingegneri non sarà interessata a pagare un pegno elevato in termini di prestazioni o convenienza per eliminare i deadlock.
2. **rilevamento e risoluzione** dei deadlock. Si tratta di **detection-recovery**. Si lascia che i deadlock avvengano,si rilevano e si agisce di conseguenza;
3. **evitare** i deadlock. Si tratta di **deadlock avoidance**;
4. **prevenzione** dei deadlock. Si tratta di **deadlock prevention**. Si impedisce strutturalmente una delle quattro condizioni richieste ad esempio richiedendo che le risorse vengano richieste in un certo ordine oppure prelazionando delle risorse ecc..

Modellazione dei deadlock

Il supporto nello studio dei deadlock è dato da **grafi** e **matrici**. I grafi hanno due tipi di nodi : i **processi**,mostrati come cerchi,e le **risorse**,mostrate come quadrati. Un arco orientato da un nodo risorsa ad un nodo processo significa che la risorsa è stata in precedenza richiesta,assegnata e attualmente detenuta da quel processo. Un arco da un processo ad una risorsa significa che il processo è attualmente bloccato in attesa di quella risorsa. Utilizzare i grafi per trovare un deadlock è molto semplice : basta verificare se c’è un ciclo all’interno di un grafo di allocazione delle risorse. Ad esempio abbiamo un processo C in attesa della risorsa T,posseduta dal processo D. Il processo D non sta per rilasciare la risorsa T perché è in attesa della risorsa U,posseduta da C e quindi entrambi i processi attenderanno per sempre.
Immaginiamo ora di avere tre processi A,B,C e tre risorse R,S,T. Il SO è libero di eseguire qualunque processo non bloccato in qualsiasi momento,quindi potrebbe decidere di eseguire A fino a quando conclude tutto il suo lavoro,poi B e alla fine C. Questo ordine non causa deadlock dato che non c’è competizione per le risorse ma allo stesso tempo non ha alcun parallelismo. Oltre alla richiesta e al rilascio delle

risorse,i processi svolgeranno elaborazioni e I/O. Quando i processi sono eseguiti sequenzialmente non è possibile che,mentre un processo è in attesa dell'I/O,un altro possa usare CPU. Così l'esecuzione sequenziale non può essere ottimale. Se inoltre nessuno dei processi esegue I/O,l'algoritmo *shortest job first* è migliore del *round-robin* e quindi in condizioni specifiche l'esecuzione sequenziale dei processi può essere la soluzione migliore.
Supponiamo che i processi eseguano sia I/O che elaborazioni e usiamo quindi il round-robin. Dopo che A richiede S,A si blocca in attesa di S. Nei due passaggi successivi anche B e C si bloccano,generando un ciclo e un deadlock.

Per gli algoritmi che vedremo,avremo due scenari :

- **una risorsa** per tipo. Immaginiamo che la risorsa sia una stampante (tipo = stampante). In questo scenario avremo una sola stampante e quindi una sola istanza;
- **più risorse** per tipo. Se la risorsa è una telecamera (tipo = telecamera),su un sistema possiamo avere n telecamere installate. Si fa quindi la differenza tra la tipologia di risorsa e il numero effettivo di istanze di quella risorsa. Ciò vuol dire che se ad esempio due processi richiedono una telecamera posso accontentarli entrambi.

In questo caso,non necessariamente,la presenza di un ciclo ci indica un deadlock. Nel caso di più risorse per ciclo,potremmo avere un ciclo ma non un deadlock : avendo più tipologie di una risorsa,prima o poi un processo ne libererà una e questa risorsa liberata può essere utilizzata da un altro processo senza creare un deadlock.

Quindi se un grafo non ha cicli non avremo deadlock. Se un grafo ha dei cicli,invece:

- se ho 1 risorsa per tipo → deadlock;
- se ho più risorse per tipo → possibilità di deadlock.

min 35.00 lez. 14

Deadlock detection e recovery

Il sistema non prova ad evitare che accadano i deadlock,ma lascia che si verifichino,cerca di rivelare quando si verificano e intraprendere qualche azione per risolvere la situazione.

caso 1 : una risorsa per tipo

Esiste una sola risorsa per tipo e ogni dispositivo può essere acquisito da un unico processo. L'approccio per rilevare il deadlock prevede di costruire il grafo ed eseguire uno degli algoritmi per scoprire se c'è almeno un ciclo nel grafo. Qualunque processo sia parte di un ciclo è in deadlock. Se non esistono cicli,il sistema non è in deadlock e può proseguire la sua normale esecuzione.

caso 2 : più risorse per ciascun tipo

Quando esistono più copie di alcune risorse,per rilevare i deadlock serve un approccio diverso. In questo caso il ciclo nel grafo non necessariamente rileva un deadlock,semplicemente perché con n risorse posso accontentare più processi. L'algoritmo che vedremo è basato su una matrice per rilevare i deadlock fra *n* processi,da P1 a Pn. Diciamo che il numero di classi di risorse sia *m*,con E1 che indica le risorse della classe 1,E2 le risorse della classe 2 e in generale Ei le risorse di classe *i* (1<=i<=m). **E** è il **vettore delle risorse esistenti**. Fornisce il numero totale di istanze di ciascuna risorsa. Per esempio,se la classe 1 è costituita dalle unità a nastro,E1 = 2 significa che ci sono 2 unità a nastro. In qualsiasi istante,alcune delle risorse sono assegnate e non sono disponibili. Diciamo che **A** è il **vettore delle risorse disponibili**,con **Ai** che fornisce il numero di istanze delle risorse *i* attualmente disponibili (cioè non assegnate). Se entrambe le unità a nastro sono assegnate,A1 sarà 0. Ci servono poi due array : **C** è la **matrice di allocazione corrente** ed **R** è la **matrice delle richieste**. La i-esima riga di C indica quante istanze di ciascuna classe di risorse detiene attualmente Pi (riga),perciò Cij è il numero di istanze della risorsa j detenuto dal processo i. Se sommo una colonna della matrice C,ottengo il numero di istanze utilizzate dai processi di quella determinata risorsa. Deve risultare quindi in ogni istante che :

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Quindi la somma di una colonna più le istanze disponibili di quella risorsa è pari alle risorse esistenti (non si può allocare più di quanto il sistema ci mette a disposizione). Analogamente,Rij è il numero di istanze della risorsa j richiesto da Pi.
L'algoritmo di **deadlock detection** si basa sul confronto tra vettori. Inizialmente ogni processo è impostato come non contrassegnato. Eseguendo l'algoritmo,i processi saranno contrassegnati per indicare che sono in grado di essere completati e pertanto non presentano deadlock. Quando l'algoritmo termina,qualsiasi processo non contrassegnato è in deadlock. Questo algoritmo considera lo scenario peggiore : tutti i processi trattengono tutte le risorse acquisite sino al loro termine. L'algoritmo può essere così formulato :

1. Cerca un processo non contrassegnato,Pi,per cui l'i-esima riga di R sia minore o uguale ad A. In altre parole,si cerca un processo per il quale si possono soddisfare le richieste. Se esso non esiste siamo in deadlock;
2. Se si trova questo processo,aggiungi l'i-esima riga di C ad A,contrassegna il processo e ritorna al passo 1. Il processo in questione alla fine libererà le sue risorse (oltre quelle già disponibili) ed avremo quindi un nuovo vettore A dato da A precedente sommato alle risorse liberate.
3. Se il processo non esiste,l'algoritmo termina. In caso contrario l'algoritmo riprende cercando dalla matrice delle richieste una riga che può essere soddisfatta (<= A).

Al termine dell'algoritmo,tutti gli eventuali processi (righe) non contrassegnati sono in deadlock. Nel passo 1 l'algoritmo cerca un processo eseguibile fino al suo completamento;un processo del genere è caratterizzato dalla corrispondenza delle sue richieste di risorse con le risorse attualmente disponibili. Il processo selezionato è poi eseguito fino alla fine,e a quel punto restituisce al set delle risorse disponibili le risorse che ha detenuto,quindi viene contrassegnato come completato. Se alla fine tutti i processi sono in grado di essere completati,nessuno è in deadlock.

Risoluzione di un deadlock

- **prelazione della risorsa**;
- **eliminazione di processi**;
- **checkpoint & rollback**;

Prelazione della risorsa

In alcuni casi si può togliere temporaneamente una risorsa al suo attuale proprietario e assegnarla ad un altro processo. In molti casi può essere richiesto un intervento manuale. Per esempio,per togliere una stampante al suo proprietario,l'operatore può raccogliere tutti i fogli già stampati e metterli in una pila,poi il processo può essere sospeso. A questo punto la stampante può essere assegnata ad un altro processo. Quando questo termina,la pila dei fogli precedenti può essere rimessa nel cassetto di uscita della stampante e si può riavviare il processo originale. La possibilità di togliere una risorsa ad un processo,farla usare ad un altro e poi restituirla senza che il primo processo lo noti dipende moltissimo dalla natura della risorsa. Risolvere un deadlock in questo modo è spesso difficile o impossibile.

Uso del rollback

Se i progettisti dei sistemi sanno che i deadlock sono probabili,possono fare in modo che periodicamente vengano generati dei **checkpoint** dei processi. Generare un checkpoint per un processo significa scrivere il suo stato in un file in modo da poterlo

riavviare più tardi. Il checkpoint contiene l'immagine della memoria e lo stato delle risorse,cioè le risorse attualmente assegnate al processo. Per avere molta efficacia,i nuovi checkpoint non dovrebbero sovrascrivere i precedenti ma dovrebbero essere scritti in nuovi file per accumulare una sequenza durante l'esecuzione. Quando si rileva un deadlock,è semplice vedere quali risorse sono necessarie. Per effettuare il ripristino,un processo che possiede una risorsa viene riportato (**rollback**) a un istante precedente all'acquisizione di quella risorsa,facendolo ripartire da uno dei suoi checkpoint precedenti. Tutto il lavoro eseguito dopo il checkpoint va perduto. In effetti,il processo è reimpostato ad un momento precedente,in cui non aveva la risorsa,ora assegnata ad uno dei processi in deadlock. Se il processo riavviato prova ad acquisire ancora la risorsa,dovrà attendere che diventi disponibile.

Eliminazione dei processi

E' il metodo più semplice per interrompere un deadlock. Una possibilità è eliminare un processo nel ciclo. Con un po' di fortuna gli altri processi potranno proseguire. Se non basta,si può ripetere l'operazione fino a che il ciclo non è interrotto. In alternativa si può scegliere di uccidere un processo fuori dal ciclo per liberare le risorse. Questo processo va scelto attentamente,dato che sta trattenendo risorse necessarie ad uno dei processi nel ciclo. Quando possibile,la cosa migliore è sopprimere un processo che può essere rieseguito senza alcun effetto negativo (ad esempio una compilazione).

Deadlock avoidance

Nell'analisi dell'individuazione dei deadlock abbiamo assunto che quando un processo richiede risorse,le richieda tutte insieme. Nella maggior parte dei sistemi,tuttavia,le risorse vengono richieste una alla volta. Il sistema deve essere in grado di decidere se l'assegnazione di una risorsa sia sicura o no ed eseguire l'allocazione solo nel primo caso. Il sistema non alloca se scopre se si rischia di andare in contro ad un deadlock. Nella pratica ciò porta ad uno svantaggio : i processi devono dichiarare,prima di partire,il numero massimo di risorse di ciascun tipo di cui ha bisogno.

Stati sicuri e non sicuri

Uno stato si dice **sicuro** se esiste un ordine di scheduling nel quale ogni processo può essere eseguito sino alla conclusione anche se tutti i processi richiedono all'improvviso il loro massimo numero di risorse. Avremo nella matrice di allocazione il numero di risorse possedute dal processo e una colonna aggiuntiva col numero massimo di risorse che possono essere richieste dal processo. In questo modo si può ottenere tramite la differenza max-has il numero di risorse che possono essere ancora richieste di quel tipo. Si ha uno stato **non sicuro** se accontentiamo tutti i processi nel momento in cui fanno la richiesta si arriva ad un punto in cui le risorse

terminano. Il **deadlock avoidance** consiste nell'accettare una richiesta se ci si trova in uno stato sicuro e nel rifiutarla se la richiesta ci porta in uno stato non sicuro (si mette in attesa). Gli stati sicuri non portano a deadlock mentre uno stato non sicuro potrebbe portare ad un deadlock.

Algoritmo del banchiere (un tipo di risorsa)

L'algoritmo controlla se,acconsentire ad una richiesta,conduca ancora ad uno stato sicuro. Se si,la richiesta è concessa altrimenti è rinviata. Per valutare se uno stato è sicuro,il banchiere controlla per vedere se ha abbastanza risorse per soddisfare alcuni clienti. Se è così,suppone che quei prestiti saranno ripagati e controlla il cliente che è al momento più vicino al limite,e così via. Se alla fine tutti i prestiti possono essere ripagati,lo stato è sicuro e la richiesta iniziale può essere accordata.

Algoritmo del banchiere (più tipi di risorsa)

Abbiamo una matrice che mostra quante istanze di ogni risorsa siano al momento assegnate a ciascun processo. Abbiamo poi un'altra matrice che mostra di quante risorse ogni processo ha ancora bisogno per concluderle. Come nel caso della risorsa singola,i processi devono dichiarare le loro necessità di risorse totali prima dell'esecuzione,in modo tale che il sistema possa calcolare in ogni momento la matrice delle **risorse ancora necessarie** (max-has). Avremo poi il vettore delle risorse esistenti **E**,il vettore delle risorse possedute **P** e le risorse disponibili **A**. L'algoritmo opera in questo modo per verificare se uno stato è sicuro :

1. Individua una riga,R,le cui necessità insoddisfatte relative alle risorse siano tutte minori o uguali ad A. Se non esiste questa riga,il sistema andrà in deadlock,dato che nessun processo può arrivare a completamento (assumendo che tutti i processi mantengano tutte le risorse fino all'uscita);
2. Suppone che il processo della riga richieda tutte le risorse di cui necessita e termini. Contrassegna questo processo come terminato e aggiunge tutte le sue risorse ad A;
3. Ripete 1-2 fino a quando tutti i processi siano contrassegnati come conclusi (si ha così uno stato iniziale sicuro) oppure non rimanga alcun processo le cui richieste di risorse possano essere soddisfatte (in questo caso il sistema non era sicuro).

Nella pratica questo algoritmo può essere applicato ma richiede che il processo,prima di partire,dichiari il massimo di cui ha bisogno. In pratica quindi l'algoritmo è sostanzialmente inutile,in quanto i processi non sanno quasi mai in anticipo quali saranno le loro necessità massime di risorse. Inoltre,il numero dei processi non è fisso,ma variabile in maniera dinamica al connettersi o disconnettersi

di nuovi utenti. Oltretutto,risorse che si pensava fossero disponibili possono scomparire.

Livelock

In alcune situazioni,un processo cerca di essere gentile rilasciando i lock che ha già acquisito quando nota che non gli è possibile ottenere il successivo lock che gli servirebbe. Poi aspetta un millesimo di secondo,per esempio,e riprova. In teoria questa è una buona cosa,e dovrebbe aiutare a rilevare ed evitare i deadlock. Se però l'altro processo fa la stessa cosa esattamente allo stesso momento la cosa non andrà bene.
Consideriamo una primitiva atomica ***try_lock*** in cui il processo chiamante esegue un test su un mutex,accorpandoselo o restituendo un errore;in altre parole non si blocca mai (se c'è un permesso si acquisisce altrimenti ritorna con errore). I programmatori possono utilizzarlo solo tramite ***acquire_lock***,che a sua volta cerca di accaparrarsi il lock,ma nel caso non sia disponibile si blocca. Immaginiamo di avere due processi che funzionino in parallelo e utilizzino due risorse. Ciascuno ha bisogno di due risorse ed entrambi usano la primitiva *try_lock* per provare ad acquisire i lock necessari. Se il tentativo fallisce,il processo rilascia il lock che possiede e riprova.
Nel codice il processo 1 viene eseguito e acquisisce la risorsa 1,mentre il processo 2 viene eseguito e acquisisce la risorsa 2. A questo punto entrambi cercano di acquisire l'altro lock,fallendo. In pratica,rilasciano il lock che stanno mantenendo e riprovano. Il tutto prosegue fino a quando un utente (o un'altra entità),stanco di attendere,fa terminare uno dei due processi. In questo modo non abbiamo un deadlock (dato che nessun processo è bloccato),ma visto che non è possibile alcun progresso si ha un **livelock**.

Starvation

Si riferisce al fatto che un processo è in attesa indefinita di una risorsa (vedi scheduling basato su priorità) ma esso non si trova in uno stato bloccato ma nello stato pronto. Il processo che rischia la starvation potrebbe provare a fare altro se non arriva la risorsa dopo un tot di tempo.

SO cap7

Virtualizzazione

Uno scenario tipico è quello in cui abbiamo un software **S** che gira su un hardware target **T**. Abbiamo poi un software **V (virtual)** che gira su un hardware **H (host)**. Si vuole sostituire T con V,ed assicurare che S funzioni come prima. Si ricorre a ciò se non si vuole modificare S e se sono presenti una o più delle seguenti condizioni :

- l’hardware T non è disponibile;
- V è meno costoso di T;
- V è più flessibile di T;
- V offre un buon modello di protezione per S.

Un esempio è un’azienda che possiede server email,server Web,server e-commerce e tanti altri. Tutti questi server sono in esecuzione su computer diversi nello stesso rack,interconnessi tramite rete ad alta velocità,costituendo un multicomputer. Il motivo per cui girano su macchine separate è che una sola macchina potrebbe non essere in grado di gestire tutto il carico. L’altro motivo è l’affidabilità : il management non si fida del fatto che un SO funzioni 365 giorni all’anno senza errori. Per questo motivo,eseguendo i servizi su computer separati,anche in caso di blocco di un server gli altri non saranno coinvolti. Lo stesso vale anche per la sicurezza : in caso di attacco al server Web da parte di un intruso,questi non avrebbe accesso immediato anche alle email (sandboxing). Spesso le organizzazioni utilizzano più sistemi operativi per l’operatività quotidiana. Si ricorre così alla tecnologie delle **macchine virtuali**. L’idea fondamentale è che ci sia un **VMM (virtual machine monitor/hypervisor)** che crea l’illusione di più macchine virtuali in esecuzione sullo stesso hardware fisico. La **virtualizzazione** consente ad un singolo computer di ospitare più macchine virtuali,ciascuna delle quali può ospitare un SO diverso. In particolare abbiamo :

- **host** : piattaforma hardware sottostante;
- **VMM** : crea ed esegue macchine virtuali fornendo un’interfaccia identica all’host (eccetto il caso della paravirtualizzazione);
- **guest** : processo a cui è fornita una copia virtuale dell’host. E’ in generale un SO.

Il vantaggio di questo metodo è che un errore in una delle macchine virtuali non blocca le altre. In un sistema virtualizzato possono esserci diversi server in esecuzione su diverse macchine virtuali,mantenendo il modello a guasto parziale di un multicomputer a costo inferiore e con una maggiore facilità di manutenzione. E’ oggi possibile eseguire sullo stesso hardware più SO diversi,mantenendo i benefici dell’**isolamento** delle macchine virtuali in caso di attacco e molti altri vantaggi.
Il motivo per cui le virtualizzazioni funzionano è che la maggior parte dei problemi di interruzione dei servizi non è causato da errori hardware,ma da software progettato male,inaffidabile,pieno di bug e mal configurati e lo stesso vale per i SO. Con la tecnologia delle macchine virtuali,il solo software in esecuzione con alti privilegi è l’hypervisor,che ha un numero di righe di codice di due ordini di grandezza inferiore rispetto ad un SO completo,e quindi un numero di bug di due ordini di grandezza minore. Un hypervisor è più semplice di un SO perché esegue un solito compito : emulare più copie dell’architettura hardware. L’esecuzione di software nelle macchine virtuali,oltre a garantire un forte isolamento,ha anche altri vantaggi,uno dei quali è che la presenza di meno macchine fisiche fa risparmiare denaro in termini di

acquisto di hardware e consumo energetico. La virtualizzazione consente inoltre di provare idee. Nelle grandi aziende spesso accade che reparti o gruppi abbiano un'idea interessante e che per implementarla vadano ad acquistare un nuovo server. Se poi l'idea funziona e diventano necessari centinaia di migliaia di server,il data center si allarga. Spesso è difficile portare il software sulle macchine esistenti,perché ogni applicazione potrebbe avere bisogno di una versione diversa del SO,di librerie e di file di configurazioni propri e altro ancora. Con le macchine virtuali,ogni applicazione può avere il proprio ambiente operativo (**workload aggregation**). Anche le operazioni effettuate per bilanciare meglio il carico tra più server sono più efficienti con macchine virtuali. Quando si svolge la migrazione di una macchina virtuale è sufficiente spostare le immagini di memoria e disco,dato che con esse vengono spostate anche le tabelle de SO (**workload migration**).

Requisiti della virtualizzazione

E' importante che le macchine virtuali funzionino esattamente come i sistemi emulati; in particolare,deve essere possibile avviarle esattamente come macchine reali e installarvi i SO desiderati,proprio come avviene su un hardware reale. Il compito di garantire questa illusione,e per di più in maniera efficiente,è affidato all'hypervisor,che deve essere in grado di garantire tre elementi fondamentali :

1. **sicurezza** : l'hypervisor deve avere il controllo completo delle risorse virtualizzate;
2. **equivalenza** : il comportamento di un programma in esecuzione all'interno di una macchina virtuale deve essere identico a quello che lo stesso programma avrebbe in esecuzione sul nudo hardware;
3. **efficienza** : la maggior parte del codice in esecuzione sulla macchina virtuale dovrebbe funzionare senza interventi da parte dell'hypervisor.

Un sistema **sicuro** per eseguire le istruzioni è elaborarle tutte,una dopo l'altra,tramite un **interprete** ed eseguire esattamente ciò che occorre a quella istruzione. Alcune istruzioni possono essere eseguite direttamente,ma non molte. Per esempio,non si può chiedere al SO guest di disabilitare gli interrupt della macchina host né di modificare le mappature della tabella delle pagine. Il trucco consiste nel far solo credere al SO che gira nell'hypervisor di aver disabilitato gli interrupt o di aver cambiato la mappatura della page table.
Per quanto riguarda l'**equivalenza**,la virtualizzazione è stata a lungo un problema sull'architettura x86,a causa dei difetti di architettura dell'Intel 386,trascinatisi per vent'anni su ogni modello di CPU in ragione della retrocompatibilità. In parole povere,ogni CPU con una modalità kernel e una modalità utente ha un set di istruzioni che si comporta in modo diverso a seconda che venga eseguito in modalità kernel o in modalità utente; fra queste ci sono le istruzioni di I/O,le modifiche alle impostazioni della MMU e così via. Popek e Goldberg le chiamano **istruzioni sensibili**. Esiste poi un set di istruzioni che causano una trap quando vengono

eseguite in modalità utente; Popek e Goldberg le chiamano **istruzioni privilegiate**. I due affermano per la prima volta che una macchina è virtualizzabile solo se le istruzioni sensibile sono un sottoinsieme di quelle privilegiate ovvero se si cerca di eseguire in modalità utente un compito che non dovrebbe essere eseguito in questa modalità,l'hardware dovrebbe causare una trap. Nell'Intel 386,ad esempio,l'istruzione *popf* sostituisce il registro dei flag,che modifica il bit che abilita e disabilita gli interrupt. In modalità utente,il bit semplicemente non viene cambiato;la conseguenza è che il 386 e i suoi successori non potevano essere virtualizzati e quindi non potevano supportare direttamente un hypervisor. Oltre ai problemi con le istruzioni che non causano trap in modalità utente,vi sono istruzioni che,in modalità utente possono leggere stati sensibili senza causare trap (17 istruzioni sensibili non privilegiate). Ad esempio sui processori x86 prima del 2005,un programma può determinare se è in esecuzione in modalità utente o kernel leggendo il proprio selettore del segmento di codice. Un SO che si comporti in questo modo e scopra di trovarsi in modalità utente potrebbe prendere decisioni sbagliate basandosi su questa informazione.
Questo problema fu risolto quando Intel e AMD,a partire dal 2005,introdussero la virtualizzazione nelle proprie CPU. Sulle CPU Intel si chiama **VT (Virtualization Technology)**;sulle CPU AMD prende il nome di **SVM (Secure Virtual Machine)**. L'idea fondamentale è creare ambienti all'interno dei quali eseguire le macchine virtuali. Quando si avvia un SO guest in un ambiente,continua a funzionare fino a che non solleva un'eccezione e passa una trap all'hypervisor,per esempio eseguendo un'istruzione di I/O. L'insieme di operazioni che causano una trap è controllato da una mappa di bit hardware impostata dall'hypervisor;con queste estensioni diventa possibile il classico approccio alla macchina virtuale chiamato **trap-and emulate**.

Caratteristiche di un VMM

Un **VMM**,presenta una interfaccia (virtuale) della piattaforma ai guest. Un VMM deve essere protetto dal software guest e deve isolare gli stack quest (SO+applicazioni). Un VMM controlla gli accessi a CPU,memoria e dispositivi di I/O. Le modalità con cui un VMM crea l'illusione di possesso di risorse ai guest sono :

- multiplexing temporale : alla VM è consentito accesso diretto alla risorsa per un certo periodo di tempo (ad esempio,nel caso della CPU);
- partizionamento delle risorse;
- mediazione.

Hypervisor di tipo 1 e 2

In generale abbiamo due metodi di virtualizzazione. Un tipo di hypervisor è chiamato **hypervisor di tipo 1** e un altro **hypervisor di tipo 2**. L'hypervisor di tipo 1,anche detto nativo o bare-metal,è tecnicamente simile ad un SO,dato che è il solo programma in esecuzione nella modalità più privilegiata. Il suo compito è supportare più copie dell'hardware reale,chiamate **macchine virtuali**,in modo simile a quello in cui viene eseguito un normale SO. Si tratta di un SO "special purpose",le cui applicazioni sono i guest e,piuttosto che fornire l'interfaccia delle system call,crea,esegue e gestisce SO guest. La macchina virtuale viene eseguita come processo in modalità utente.
Un hypervisor di tipo 2,anche detto "hosted",è un programma che si basa,per esempio,su Windows o Linux per allocare e schedulare le risorse,in modo molto simile ad un processo. Si tratta quindi di un programma eseguito e gestito da un SO host. Ovviamente,anche un hypervisor di tipo 2 finge di essere un computer completo di CPU e diverse periferiche. Entrambi i tipi di hypervisor devono eseguire le istruzioni macchina in modo sicuro. Ad esempio,un SO in esecuzione sopra l'hypervisor può modificare e addirittura scompigliare le proprie tabelle delle pagine,ma non quelle degli altri. Il SO in esecuzione al di sopra dell'hypervisor è chiamato,in entrambi i casi **SO guest**. Nel caso di hypervisor di tipo 2,il SO che gestisce l'hardware prende il nome di **SO host**.
Un hypervisor di tipo 2 dipende,per gran parte della propria funzionalità,da un SO host (Windows,Linux …). Quando viene avviato per la prima volta,funziona come un computer appena acceso e ha bisogno quindi di un DVD,una chiavetta USB o di un CD-ROM contenenti il SO. L'unità contenente il supporto,però,può essere virtuale : per esempio,è possibile memorizzare l'immagine come file ISO sul disco dell'host e fare in modo che l'hypervisor finga di leggere da una vera unità DVD. Il SO viene quindi installato su un **disco virtuale** (che è in realtà semplicemente un file sotto Windows,Linux o macOS) eseguendo il programma di installazione che si trova sul DVD. Quando il SO è stato installato sul disco virtuale,può essere avviato ed eseguito senza rendersi minimamente conto che c'è il trucco.
Un hypervisor di tipo 1 che supporta una macchina virtuale,come sappiamo,viene eseguito direttamente sull'hardware. La macchina virtuale viene eseguita come processo utente in modalità utente,pertanto non le è consentita l'esecuzione di istruzioni sensibili. La macchina virtuale,tuttavia,esegue un SO che ritiene di essere in modalità kernel (anche se non lo è realmente);si parla in questo caso di **modalità kernel virtuale**. La macchina virtuale esegue anche processi utente che credono di essere in modalità utente (perché in effetti lo sono). Quando il SO guest,che pensa di trovarsi in modalità kernel esegue un'istruzione consentita solo quando la CPU si trova veramente in modalità kernel sorge un problema. Sulle CPU prive di VT l'istruzione normalmente fallisce e il SO va in crash. Se invece la CPU è provvista di VT,quando il SO guest esegue un'istruzione sensibile viene generata una trap sull'hypervisor. L'hypervisor può quindi controllare l'istruzione per verificare se è stata richiesta dal SO guest all'interno della macchina virtuale oppure da un programma utente,sempre nella macchina virtuale. Nel primo caso,fa in modo che l'istruzione possa essere eseguita,mentre nel secondo emula il comportamento che

avrebbe l'hardware reale qualora dovesse affrontare l'esecuzione di un'istruzione sensibile in modalità utente.

Tecniche di virtualizzazione

Nelle CPU intel sono previsti quattro livelli di privilegio,anche detti **ring**. Il ring 0 corrisponde al kernel,il ring 1 e il ring 2 ai driver e agli hypervisor e il ring 3 alle applicazioni. Le principali tecniche di virtualizzazione sono:

- **emulazione** : realizza ciò che fa il processore,ma in software. Un emulatore è un programma basato su un loop che imita le azioni del processore target (fetch,decode,execute). Le execute prevedono uno switch con un case per ogni istruzione target. Permette di eseguire programmi compilati per una particolare architettura,e quindi un particolare ISA,su un sistema di elaborazione dotato di un diverso ISA;
- **trap-and-emulate** (con supporto hardware) : parte del codice eseguito direttamente sulla CPU. Quando si solleva un'eccezione si passa una trap all'hypervisor,per esempio eseguendo un'istruzione di I/O (istruzioni sensibili). L'insieme di operazioni che causano una trap è controllato da una mappa di bit hardware impostata dall'hypervisor;
- **binary translation**;
- **paravirtualization** : il guest OS viene modificato al fine di semplificare alcuni aspetti. Ad esempio,i driver parlano direttamente con il VMM,anziché usare trap sulle istruzioni PIO/MMIO. La paravirtualizzazione non ha come scopo la creazione di una macchina virtuale che appaia assolutamente identica all'hardware sottostante. Presenta un'interfaccia software che espone esplicitamente il fatto che si tratti di un ambiente virtualizzato,offrendo per esempio un insieme di **hypercall** (chiamate all'hypervisor) che consentono al guest di inviare all'hypervisor delle richieste esplicite (analogamente ad una chiamata di sistema che offre alle applicazioni i servizi del kernel). I guest usano le hypercall per svolgere operazioni privilegiate sensibili come l'aggiornamento delle page table;dato che questi compiti sono svolti esplicitamente in cooperazione con l'hypervisor,il sistema può essere complessivamente più semplice e veloce.

Binary translation

Il principio di base è quello di tradurre codice guest in codice host equivalente e saltare al codice tradotto. Il codice tradotto viene memorizzato in una cache interna e viene riutilizzato nel caso deve essere nuovamente eseguito. Per molti anni i processori x86 hanno supportato quattro modalità di protezione,anche dette **anelli/ring**. Quello con privilegi più bassi è il ring 3,ed è in questo che vengono eseguiti i processi utente. In questo anello non possono essere eseguite istruzioni privilegiate. L'anello con privilegi più elevati è lo 0,che consente l'esecuzione di

qualsiasi istruzione;nella normale operatività,il kernel viene eseguito nell'anello 0. Gli altri due anelli (1 e 2) non vengono attualmente utilizzati da alcun SO e ciò significa che possono essere tranquillamente utilizzati dagli hypervisor. Molte soluzioni di virtualizzazione mantenevano l'hypervisor in modalità kernel (ring 0) e le applicazioni in modalità utente (ring 3),facendo girare il SO guest in un livello con privilegi intermedi (ring 1). Come risultato,il kernel ha più privilegi rispetto ai processi utente e qualsiasi tentativo di accedere alla memoria del kernel da un programma utente genera una violazione di accesso;al contempo,le istruzioni privilegiate eseguite sul SO guest generano una trap sull'hypervisor,che effettua alcuni controlli di integrità prima di eseguire le istruzioni su richiesta del SO guest. Per quanto riguarda le istruzioni sensibili nel codice kernel del guest,l'hypervisor fa in modo che non ne esistano più,riscrivendo il codice un blocco di base alla volta. Un **blocco di base** è una breve sequenza di istruzioni che termina con un'istruzione di diramazione e che,per definizione,non contiene salti,trap,return né altre istruzioni che possano modificare il flusso di controllo,a parte l'ultima,che è un'istruzione di diramazione. Appena prima di eseguire un blocco di base,l'hypervisor lo esamina per verificare se contiene istruzioni sensibili;in caso affermativo le sostituisce con una chiamata a una procedura dell'hypervisor che sia in grado di gestirle. La diramazione dell'ultima istruzione viene anch'essa sostituita da una chiamata all'interno dell'hypervisor,così da garantire che tutta la procedura possa essere ripetuta anche con il blocco di base successivo. I blocchi tradotti vengono inseriti in una cache e quindi non serve ripetere la traduzione in seguito. Dato che il blocco di base è stato eseguito,il controllo torna all'hypervisor,che trova il blocco seguente;se questo è stato già tradotto può essere eseguito immediatamente,altrimenti viene prima tradotto,inserito in una cache ed eseguito. Alla fine,la maggior parte del programma si troverà nella cache e verrà eseguito quasi a velocità massima. Si esegue di frequente una traduzione binaria di tutto il codice del SO guest in esecuzione nel ring 1,sostituendo anche le istruzioni privilegiate sensibili,che,in linea di principio,potrebbero generare delle trap. Il motivo è che le trap sono molto costose da gestire,e la traduzione binaria garantisce prestazioni migliori.

Container

I container sono un modo per **isolare** processi e risorse : un container incapsula un'applicazione e tutte le sue dipendenze (librerie,binari,file di configurazione,ecc.). Un container non virtualizza alcun hardware e condivide il kernel con l'host (ciascun container condivide il SO con gli altri container). Un container è molto più piccolo di una VM. Per garantire che un processo possa eseguire in maniera isolata da tutto il resto,abbiamo vari strumenti che UNIX mette a disposizione :

- **chroot (change root)** : system call UNIX per cambiare la root directory di un processo e dei suoi figli a una nuova locazione del file system. Sappiamo che il SO parte da / e poi abbiamo una serie di directory e sottodirectory. *chroot* permette di prendere un path arbitrario e dire che per un determinato

processo la root sarà la directory specificata. Il processo vedrà quindi un file system limitato solo a quella directory root. Avere una propria directory radice però non basta perché per avere un isolamento vero e proprio servono anche spazi dei nomi separati per identificatori di processi e di utenti,interfacce di rete (e indirizzi IP associati) e così via. Limitare l'accesso ad un particolare spazio dei nomi del file system con *chroot* è semplice. Il SO ricorda che per quel gruppo di processi tutte le operazioni sui file sono relative alla nuova radice. Quando ad esempio un processo apre tutte le interfacce di rete del sistema,il SO si accerta di aprire solo l'interfaccia (o le interfacce) assegnate al gruppo/container;

- **cgroup (control group)** : permette di organizzare processi in gruppi,monitorare e limitare l'uso di vari tipi di risorse come tempo di CPU,memoria,disco,banda di rete e così via. I cgroup sono flessibili in quanto non prescrivono in anticipo quali siano le risorse di cui tener traccia,quindi è possibile aggiungere qualsiasi risorsa possa essere limitata e tracciata. Associando a un cgroup un controller (detto anche sottosistema) di una specifica risorsa è possibile monitorare e/o limitare l'accesso alla risorsa per tutti i processi membri del cgroup.;
- **cpuset** : consente agli amministratori di associare specifiche CPU o core e sottoinsiemi di memoria a un gruppo di processi. I cpuset consentono di assegnare ad un cgroup un set di CPU e nodi di memoria. In questo modo gli amministratori possono specificare che quel cgroup può utilizzare quei core della CPU e che tutta la sua memoria sarà allocata solo dalla memoria di quei nodi. E' poi possibile subpartizionare le risorse di un cpuset genitore in cpuset figli;
- **namespaces** : ogni container ha un'istanza di ogni tipo di namespace (pid,rete…) per limitare gli oggetti host visibili. Si tratta di un sottoalbero che parte da una radice creata dall'amministratore tramite *chroot*.

Tramite questi concetti è possibile creare container isolati senza ricorrere ad hypervisor o alla virtualizzazione hardware. I container hanno un'amministrazione di sistema semplice : va mantenuto un solo SO,non un sistema distinto per ogni macchina virtuale. Tra gli svantaggi abbiamo invece il fatto che non è possibile eseguire più SO sulla stessa macchina (se si vuole eseguire Windows e UNIX contemporaneamente,i container non saranno d'aiuto). Un altro svantaggio è quello che l'isolamento è buono ma non assoluto : i container condividono lo stesso SO e possono interferire tra loro a quel livello. Se il SO ha limiti statici su determinate risorse,come il numero di file aperti,e un contenitore li sta usando (quasi) tutti,gli altri contenitori avranno problemi. Allo stesso modo una singola vulnerabilità mette a rischio tutti i container. L'isolamento degli hypervisor è quindi più solido.

Docker

Piattaforma open source per la creazione,gestione,e orchestrazione di container per Linux. L'obiettivo è quello di portare le applicazioni in modo semplice da una macchina all'altra (che supportano Docker),ottimizzando lo sviluppo,testing,rilascio e distribuzione di applicazioni.
Un'**immagine** è la descrizione statica dell'applicazione e di tutte le sue dipendenze. Si tratta di un "template",read-only,che contiene tutte le istruzioni necessarie a creare un container docker : può essere definita tramite una lista di istruzioni specificate in un file di testo (**Dockerfile**),oppure ottenuta da un **registry**.
Con ***docker image ls*** possiamo vedere quali sono le immagini create/scaricate.
Se lanciamo il comando ***docker run -it ubuntu bash*** avremo un'immagine di un mini sistemi operativo,con kernel host,che mette a disposizione comandi tipici in una distribuzione ubuntu e le librerie di base.
docker run è un comando che, presa un'immagine nel repository o che può essere specificata,fa partire il container. In particolare legge il contenuto dell'immagine e crea un container con le caratteristiche descritte dall'immagine.
L'opzione ***-it*** si apre un collegamento diretto con il container e quindi nel caso precedente si apre una shell nel container. Inoltre,siccome chi gira nel container è un'applicazione unica,questa gira in modalità root. Il container vede comunque un file system e una serie di comandi base ubuntu che sono nell'immagine scaricate : se lancio *gcc* nella shell del container non viene trovato. La visibilità del container è inoltre limitata : se faccio *ps aux* sull'host vedremo tutti i processi in esecuzione mentre se uso lo stesso comando nel container vedrò solo due processi. Quindi il container pensa di essere in esecuzione da solo e in particolare sarà il processo con PID 1,che sull'host è *init* e quindi i nomi all'interno del container sono isolati con tutto il resto. Il secondo processo è il comando lanciato. Per uscire da questo container si usa *exit*.

I componenti **chiave** sono :

- **processo demone** : processo che gira sul SO host. Ogni volta che scriviamo *docker run,docker image* stiamo parlando con questo processo tramite l'interfaccia a riga di comando;
- **registry (repository)** : database di immagini. Se non specifichiamo nulla si scarica dal database pubblico di docker.
- **immagine** : descritta prima;
- **container** : istanza eseguibile di un'immagine. Il container può essere creato,avviato,stoppato,migrato ecc. Le modifiche non si riflettono sull'immagine. E' possibile definire "quanto" il container è isolato dall'host;

Oltre ad immagini e container docker permette anche di istanziare **network** e **volumi**. I container hanno infatti una natura "effimera" : una volta creato ed eseguito il container,terminata l'esecuzione il ciclo di vita del container è finito. Il file system che vede il container è quindi limitato all'esistenza del container. Se ci serve salvare dati e file in maniera permanente si deve esplicitare la richiesta a Docker creando

un'entità chiamata **volume**,che verrà mappata sul file system dell'host : in questo modo si dà la possibilità al container di salvare qualcosa sull'host,altrimenti andrebbe perso. L'altra possibilità è quella di creare delle **reti (network)**. Lanciando il comando ***ifconfig*** sull'host possiamo vedere le interfacce di rete : vedremo così che Docker ha creato una rete sulla rete. In virtù dell'isolamento Docker vede a sua volta delle "reti ancor più virtuali" rispetto a quella della VM. Le reti di Docker hanno un bridge sulla scheda rete della macchina virtuale. I container docker tipicamente girano infatti sugli indirizzi IP ed espongono dei servizi su delle porte. In particolare abbiamo l'interfaccia della macchina host e poi le interfacce virtuali docker che hanno un indirizzo IP in bridge con la scheda di rete della macchina host (tutto ciò che viene scritto su quelle interfacce di rete viene propagata ai container). E' possibile creare tramite Docker interfacce virtuali e sottoreti container,ovvero container su LAN virtuali ed è quindi possibile agganciare un container da una rete e spostarlo su un'altra rete. Una volta partizionata un'applicazione complessa in pezzi semplici che si mantengono in piedi da soli (**microservizi**),questi dovranno parlarsi e lo fanno tipicamente tramite protocollo *http*.

Immaginiamo di voler scaricare un web server. Con ***docker search*** possiamo consultare il repository pubblico dando una stringa che sarà contenuta nell'immagine. Un web server è un'applicazione che si può installare su una macchina e sa parlare il protocollo *http*. Uno famoso open source è *ngix*. Usando ***docker search ngix*** ci restituirà tutte le immagini del repository pubblico di docker con *ngix* nel nome : le immagini che hanno nel nome la singola stringa senza / sono immagini ufficiali,quelle con lo / sono immagini prodotte da utenti e pubblicate sul repository docker. Per scaricare un'immagine possiamo farlo da un repository o possiamo crearla noi. Per scaricarla da un repository si usa ***docker image pull imagename*** (si può specificare anche una determinata versione dell'immagine ma in generale scaricherà la più recente).
docker run crea il container e lo mette in esecuzione.
Le opzioni di questo comando possono essere :

- **-i** : comando interattivo;
- **-t** : allocazione di un terminale;
- **- -name** : nome del container;
- **-d** : il container esegue in background → ogni volta che lanciamo un comando da prompt,il prompt si blocca fintantoché il comando non è ultimato. Dato che questo oggetto rimarrà in attesa fino a quando non lo uccidiamo,con questa opzione non si rimane bloccati col prompt;
- **-p h:e** : mappa la porta e (esposta) sulla porta h (host);
- **-P** : pubblica le porte esposte su porte random dell'host.

Un esempio è *docker run -P -d –name MyWebServer nginx*. Per vedere quali container sono in esecuzione si usa ***docker ps -a*** : in questo modo possiamo vedere che *MyWebServer* ascolta su una porta di default n.80 (**porta http**). Abbiamo

chiesto che le porte venissero esposte su porte random dell'host. La porta può essere vista anche tramite ***docker port namecontainer***. Il web server containerizzato può essere contattato dalla macchina host aprendo il browser e cercando indirizzo IP della macchina virtuale e numero di porta su cui è stato esposto il container : come se stessimo navigando sul web abbiamo in realtà contattato un web server che gira nel container della macchina virtuale (**indirizzoIP:port**). In questo modo non interessa il SO,basta scaricare l'immagine,creare il container e abbiamo un web server funzionante indipendente dal SO.

Creare un'immagine

Il file **application.py** è un programma python che utilizza un package chiamato **Flask** che consente di scrivere web service : ad un web service possono essere sottoposte richieste *http* le quali sono anche in grado di innescare codice arbitrario (programmi) lato server. In particolare si definisce il corpo dei metodi e andando a dire quando devono essere eseguiti in base all'URL : mettendo solo ID e porta si esegue il primo metodo mentre mettendo URL / invocherò il secondo metodo. Provando ad eseguire questa applicazione sulla macchina host,quest'ultima non la può eseguire perché mancano le dipendenze : bisogna quindi incapsulare il tutto in un'**immagine** in modo tale da rendere l'applicazione subito funzionante.
La base di partenza per creare un'immagine sono dei file testuali che devono essere chiamati esattamente **Dockerfile**. Il Dockerfile descrive come è fatta l'immagine,definisce un componente applicativo e contiene solo il software ad esso necessario. La prima istruzione è **FROM …** : le immagini vanno tipicamente costruite andando ad aggiungere cose nuove su immagini esistenti → FROM specifica l'immagine di base da cui partire (es. FROM ubuntu:latest).
Successivamente bisognerà installare i pacchetti necessari a far girare l'applicazione tramite direttiva **RUN** : nel nostro caso si richiede di fare un update dei repository,installare python e il package flask. I RUN servono quindi a costruire le dipendenze. Tramite la direttiva **ADD**,invece,si specifica un file che sta sull'host e una destinazione nel container (copia il file dall'host alla directory temporanea del container). Successivamente troviamo **EXPOSE**,che serve a dire che se l'applicazione è in attesa su una porta indica su quale porta mettere in attesa il container (es. EXPOSE 5000 → il container va in attesa sulla porta 5000). Infine,abbiamo la direttiva **CMD**,ovvero il comando da eseguire nel container quando esso è avviato (es. CMD [" " , " "]).

Una volta scritto il Dockerfile,per creare un'immagine useremo il comando ***docker build -t name .*** (-t serve ad indicare il nome dell'immagine — "**.**" serve invece ad indicare in quale directory il build deve cercare il Dockerfile).
Una volta creata l'immagine possiamo lanciarla tramite ***docker run -P -d –name namecontainer nameimg***.

Per uccidere un container si usa ***docker stop namecontainer*** *,* nel caso in cui è in esecuzione e poi ***docker rm namecontainer***.

SO cap9

Introduzione alla sicurezza

Molte aziende possiedono informazioni di valore da custodire con attenzione. Queste informazioni possono essere tecniche,commerciali,finanziarie o legali e sono memorizzate su computer. Anche i computer domestici contengono dati importanti come questi. Più informazioni sono custodite nei computer,più diventa importante proteggerle. Sorvegliare queste informazioni contro l'eventualità di un loro utilizzo non autorizzato è perciò uno degli obiettivi principali di tutti i SO ed è molto difficile da gestire. Molti dispositivi hanno un solo utente,quindi il pericolo che un utente dello stesso dispositivo possa spiare dati non suoi è virtualmente sparito. Questo non vale per i server condivisi,magari sul cloud;in questi casi l'interesse a mantenere isolati gli utenti è elevatissimo. Possono verificarsi inoltre casi di intercettazioni,soprattutto in rete. Se ad esempio un utente si trova sulla stessa rete Wi-Fi di un altro utente,può intercettare tutti i dati del suo traffico di rete. Anche se però si riuscisse ad intercettare questi dati,questi ultimi potrebbero non essere facilmente interpretati. Si parla così di **crittografia** : colui che intercetta i dati deve saperli decifrare. Purtroppo però questo sistema non funziona sempre alla perfezione,dato che se un utente riesce a penetrare nel computer di un altro utente,può intercettare tutti i messaggi in uscita prima che siano cifrati,e tutti i messaggi in ingresso dopo che sono stati decifrati. Penetrare in un computer altrui non è facile,ma è più facile di quanto dovrebbe. Il problema è causato dai bug nel software del computer. Quando un bug è nella sicurezza si parla di **vulnerabilità**. Quando qualcuno vuole prendere il controllo totale di un computer altrui,è sufficiente passare il numero di byte che occorrono per attivare il bug (**exploit**). Un attacker può lanciare un exploit manualmente o in automatico per eseguire software **malware**,che può assumere molte forme. Un esempio è il malware che infetta i computer iniettandosi come **virus** in altri file (spesso eseguibili). In altre parole,un virus ha bisogno di un altro programma e di una qualche forma di interazione con l'utente per potersi propagare. Un **worm**,invece,si diffonde in modo automatico e si propaga indipendentemente da ciò che fa l'utente.

Fondamenti della sicurezza dei SO

Con il termine **sicurezza** ci riferiamo al problema nella sua globalità e con il termine **dominio di protezione** ci riferiamo al preciso insieme di operazioni (come la lettura o la scrittura di un file o di una pagina di memoria) che un utente o un processo sono autorizzati ad eseguire sugli oggetti del sistema. Con **meccanismi di sicurezza**,invece,ci riferiamo a tecniche specifiche utilizzate dal SO per proteggere

le informazioni del computer;un esempio è l'impostazione del bit supervisore nella voce della tabella delle pagine riferita ad una pagina che deve essere inaccessibile alle applicazioni dell'utente. Infine,con **dominio di sicurezza** ci riferiamo in modo informale al software che da una parte deve essere in grado di eseguire i suoi compiti in modo sicuro,e dall'altra non deve poter mettere a rischio la sicurezza di altri. Esempi di domini di sicurezza sono i componenti del kernel,processi e macchine virtuali.

CIA : Triade della sicurezza

La sicurezza delle informazioni può essere divisa in tre componenti o obiettivi :

1. riservatezza;
2. integrità;
3. disponibilità.

Questi tre obiettivi costituiscono il nucleo centrale della sicurezza dei dati che dobbiamo proteggere. Nell'insieme i tre obiettivi prendono il nome di CIA (confidentiality,integrity and availability).
La **riservatezza** consiste nel fare in modo che i dati segreti restino tali. Più nello specifico,se il proprietario di alcuni dati ha deciso che tali dati devono essere resi disponibili solo ad alcuni e non ad altri,il sistema dovrebbe garantire che essi non siano mai rilasciati a chi non è autorizzato. Il proprietario dovrebbe essere in grado di specificare chi può vederli e il sistema dovrebbe applicare queste specifiche,il che dovrebbe idealmente avvenire per ogni singolo file.
L'**integrità** significa che gli utenti non autorizzati non dovrebbero mai essere in grado di modificare alcun dato senza il permesso del proprietario. La modifica dei dati,in questo contesto,comprende non solo la modifica vera e propria,ma anche la loro cancellazione e l'inserimento di dati fasulli. Se un sistema non è in grado di garantire che i dati che contiene saranno modificati solo quando lo deciderà il proprietario,non è un buon sistema di memorizzazione.
La **disponibilità** indica che nessuno deve poter disturbare il sistema per renderlo inutilizzabile. Questi attacchi di tipo **negazione del servizio (denial of service,DoS)** sono sempre più diffusi. Se per esempio un computer è un server internet,inondarlo di richieste può danneggiarlo,assorbendo tutto il tempo della CPU solo per esaminare e scartare le richieste in ingresso.
Sono disponibili modelli e tecnologia ragionevoli e adatti a gestire attacchi alla riservatezza e all'integrità;contrastare attacchi DoS è molto più difficile.

Domini di protezione

Un computer contiene molte risorse,od "oggetti",che hanno bisogno di essere protetti. Questi oggetti possono essere di tipo hardware (ad esempio le CPU,segmenti di memoria,unità disco ecc.) o di tipo software

(processi,file,database,semafori). Ogni oggetto ha un nome univoco tramite il quale vi si può fare riferimento e un insieme finito di operazioni che i processi possono eseguire. Le operazioni *read* e *write* sono indicate ad esempio per un file,*up* e *down* hanno senso se applicate a semafori. Serve un modo per proibire ai processi di accedere agli oggetti per i quali non sono autorizzati. Inoltre,questo meccanismo deve anche far si che,quando è necessario,si possano limitare i processi ad un sottoinsieme di operazioni (ad es. processo A ha diritto alla lettura del file F ma non alla scrittura). Un **dominio di protezione** è un insieme di coppie (**oggetto,diritti**),ognuna delle quali specifica un oggetto e un certo sottoinsieme di operazioni che possono essere eseguite su di esso. I domini di protezione e di sicurezza sono strettamente correlati. Ogni dominio di sicurezza,come un processo P o una macchina virtuale V,è in uno specifico dominio di protezione D che ne determina i diritti. In questo contesto un **diritto** è il permesso di eseguire una delle operazioni. Spesso un dominio di protezione corrisponde ad un singolo utente,e descrive ciò che l'utente può e non può fare,ma può essere anche più generale di un solo utente. Ad esempio i membri di un team di programmazione potrebbero appartenere allo stesso dominio di protezione per poter tutti accedere ai file del progetto. In alcuni casi,i domini di protezione sono organizzati in una gerarchia. Fintanto che una VM è in un dominio di protezione,nessun programma della VM può eseguire operazioni che esulino dal dominio di protezione. Questo però non significa che tutti i programmi della VM possano eseguire tutte le operazioni del dominio di protezione;alcuni avranno solo un sottoinsieme dei diritti di accesso. In altre parole,i domini di protezione di livello più alto saranno vincolati dal dominio di protezione di livello inferiore. L'allocazione degli oggetti ai domini dipende dalle specifiche : chi ha bisogno di eseguire quali operazioni su quali oggetti. In ogni momento,ciascun processo (o,in generale,dominio di sicurezza) è eseguito in qualche dominio di protezione. In altre parole,c'è un qualche insieme di oggetti al quale può accedere,e per ogni oggetto ha un certo set di diritti. I processi possono anche passare da un dominio di protezione ad un altro durante l'esecuzione;le regole per il passaggio tra domini di protezione sono dipendenti dai sistemi.

Ad esempio,in UNIX ogni processo ha una parte utente e una parte kernel. Quando il processo fa una chiamata di sistema,passa dalla parte utente a quella kernel. La parte kernel ha accesso ad un insieme di oggetti diverso da qualla utente e quindi una chiamata di sistema provoca uno scambio di domini di protezione.

Una questione importante è il modo in cui il sistema tiene traccia dell'appartenenza degli oggetti ad un dominio di protezione. Si utilizza una **matrice di protezione** con i domini come righe e gli oggetti come colonne. Ciascuna intersezione elenca i diritti,se ci sono,che quel dominio detiene per quell'oggetto. Data questa matrice e l'attuale numero di dominio,il sistema può indicare se sia autorizzato un accesso ad un dato oggetto,in una particolare modalità e da un dominio specificato. Lo scambio di dominio stesso può essere facilmente incluso in un modello a matrice capendo che il dominio è esso stesso un oggetto,con l'operazione *enter*. In questo caso abbiamo una matrice contenente anche i domini come oggetti. I processi di un

dominio possono passare ad un altro (con *enter*),ma una volta in quel dominio non possono tornare indietro.

Liste di controllo degli accessi (ACL)

Raramente si realizza una matrice contenente i domini come oggetti dato che è grande e sparsa. La maggior parte dei domini non ha alcun accesso alla maggior parte degli oggetti,quindi memorizzare una matrice così grande e per la maggior parte vuota è uno spreco di spazio del disco. Tuttavia,due metodi pratici sono memorizzare la matrice per righe o per colonne e poi memorizzare solo gli elementi non vuoti. La prima tecnica consiste nell'associare ciascun oggetto ad una lista ordinata,contenente tutti i domini che possono accedere all'oggetto e in che modo,detta **lista di controllo degli accessi (Access Control List ACL)**.
Supponiamo di avere tre processi ciascuno appartenente ad un dominio diverso e rispettivamente a proprietari A,B e C (quindi ciascun dominio corrisponde ad un utente). L'utente è spesso chiamato **soggetto** o **proprietario**,mentre ciò che gli appartiene è un **oggetto**. Nello spazio kernel abbiamo tre file F1,F2 e F3. A ciascun file è associata una ACL. Nell'esempio l'ACL di F1 ha due voci separate da un punto e virgola;la prima indica che qualsiasi processo di proprietà di A può scrivere e leggere F1 e la seconda che ogni processo di proprietà B può leggere F1. Tutti gli altri accessi da parte di questi due utenti e di altri utenti sono proibiti. I diritti sono assegnati all'utente e non al processo. Sempre dall'esempio vediamo che l'ACL di F2 ha tre voci : A,B,C ; tutte possono leggere il file e B può anche scriverlo. F3 ha invece come voci B e C ; entrambe possono leggere ed eseguire il file e B può anche scriverlo. Nella pratica le ACL sono più sofisticate. I diritti,infatti,potrebbero essere anche altri oltre a lettura,scrittura ed esecuzione. Alcuni possono essere generici,cioè applicabili a tutti gli oggetti (destroy object e copy object) altri possono essere specifici (append message).
Molti sistemi supportano il concetto di **gruppo di utenti**. I gruppi hanno dei nomi e possono essere inseriti nella ACL. Sono possibili due variazioni alla semantica dei gruppi. In alcuni sistemi ogni processo ha un ID utente (**UID**) e un ID gruppo (**GID**),e in questi casi un'ACL contiene voci della forma :

UID1 , GID1 : diritti1; UID2 , GID2 : diritti2 ; …

A queste condizioni,nel momento in cui viene effettuata una richiesta di accesso ad un oggetto si esegue un controllo usando l'UID e il GID del chiamante. Se sono presenti nell'ACL,i diritti elencati sono disponibili;se invece la combinazione (UID,GID) non è presente nella lista l'accesso non è consentito. Quest'uso dei gruppi introduce il concetto di **ruolo**. Consideriamo un computer in cui Tana sia amministratore di sistema,e quindi nel gruppo *sysadm*. Supponiamo che la società abbia anche alcuni club per i dipendenti e che Tana faccia parte di uno dei piccioni. I

membri del club appartengono al gruppo *pigfan* e hanno accesso ai computer della società per gestire il loro database di piccioni. Una parte dell'ACL potrebbe essere :

Password : tana,sysadm : RW
Pidgeon_data : bill,pigfan : RW ; tana,pigfan : RW
Dove *sysadm* e *pigfan* sono gruppi e,*tana* e *bill* sono utenti. Se Tana prova ad accedere ad uno di questi file il risultato dipenderebbe dal gruppo dal quale ha eseguito il login : al momento del login il sistema potrebbe chiederle quale dei suoi gruppi vuole usare,oppure potrebbero addirittura esserci un nome di login e/o una password diversi per tenere separati di due gruppi. Tramite questo schema si previene l'uso da parte di Tana del file delle password quando si collega al computer in veste di socia del fan club dei piccioni viaggiatori : può farlo solo quando è connessa come amministratrice di sistema. In alcuni casi un utente può avere accesso ad alcuni file indipendentemente dal gruppo dal quale ha eseguito il login;un caso del genere può essere gestito introducendo il concetto di **carattere jolly**,che significa "tutti". Ad esempio,la voce **tana, *:RW** per il file delle password darebbe a Tana accesso a questo file indipendentemente dal gruppo dal quale ha effettuato il login. Un'altra possibilità è consentire l'accesso ad un utente che appartiene a uno qualsiasi dei gruppi che hanno determinati diritti di accesso. Il vantaggio in questo caso è che un utente appartenente a più gruppi non deve specificare al momento del login quale gruppo usare : valgono tutti sempre. Lo svantaggio di questo approccio è che fornisce meno incapsulazione : Tana potrebbe modificare il file delle password durante una riunione del fan club dei piccioni. L'uso dei gruppi e dei caratteri jolly introduce la possibilità di bloccare selettivamente l'accesso ad un file di uno specifico utente. Ad esempio,la voce **anna, *: (none) ; * , *: RW** consente di accedere al file in lettura e scrittura a tutti tranne Anna. Funziona perché le voci sono analizzate in ordine ed è utilizzata la prima applicabile,mentre le successive non sono nemmeno esaminate. Viene trovata una corrispondenza per Anna sulla prima voce e i diritti di accesso,che in questo caso sono "none",vengono letti e applicati. A quel punto la ricerca termina;il fatto che il resto del mondo possa accedere non è nemmeno visto. L'altro modo per gestire i gruppi è non usare come voci delle ACL le coppie (UID,GID),un UID o un GID. Ad esempio,una voce per il file *pidgeon_data* (l'archivio dei piccioni di Tana) potrebbe essere **debbie : RW; emma : RW; pigfan :RW** che significa che Debbie,Emma e tutti i membri del gruppo *pigfan* hanno accesso al file in lettura e scrittura.
Accade a volte che un utente o un gruppo abbiano su di un file determinati permessi che ad un certo punto il proprietario del file intende revocare. Con le ACL è semplice revocare un diritto : basta modificarla. Se però l'ACL è verificata soltanto all'apertura del file,molto probabilmente la modifica avrà effetto solo alle successive chiamate *open* del file. Un file già aperto manterrà i diritti che aveva all'apertura,anche se l'utente non è più autorizzato ad accedervi.
Nei sistemi UNIX è possibile usare comandi come ***getfacl*** e ***setfacl*** rispettivamente per ispezionare e impostare la lista di controllo degli accessi. In pratica molti utenti si limitano a regolare l'accesso ai file usando i ben noti permessi UNIX di

lettura,scrittura,esecuzione per “utente” (proprietario),”gruppo” e “altro” (tutti gli altri),ma le liste di controllo degli accessi offrono un controllo più preciso sulla distribuzione degli accessi. Ad esempio,supponiamo di avere un file hello.txt con i seguenti permessi :

-rw-r- - - - - 1 herbertb staff 6 Nov 20 11:05 hello.txt

Il file ha permessi di lettura/scrittura per il proprietario,lettura per il gruppo *staff* e nulla per tutto il resto. Con le liste di controllo degli accessi,Herbert può dare ad un altro utente i permessi di lettura/scrittura sul file senza aggiungerlo al gruppo *staff* né rendere il file accessibile a tutti gli altri tramite comando :

setfacl -m u:utente:rw hello.txt

Crittografia

I SO usano soluzioni crittografiche in molti luoghi;ad esempio,alcuni file system cifrano tutti i dati sul disco,mentre protocolli come IPSec possono cifrare o firmare il contenuto dei pacchetti di rete. Lo scopo della crittografia è prendere un messaggio o un file,chiamato **testo in chiaro (plaintext)** e cifrarlo,ottenendo il **testo cifrato (ciphertext)** in un modo tale che solo coloro che sono autorizzati sappiano riconvertirlo in testo in chiaro. Per tutti gli altri,il testo cifrato è solo un’incomprensibile sequenza di bit. Gli algoritmi (funzioni) di cifratura e decifrazione dovrebbero essere sempre pubblici. Cercare di tenerli segreti non funziona quasi mai e dà un falso senso di sicurezza a coloro che cercano di mantenere la segretezza. Nel settore questa tattica è chiamata **security by obscurity (sicurezza tramite segretezza)**,ed è utilizzata dai dilettanti. La categoria dei dilettanti comprende anche grandi multinazionali. La segretezza dipende dai parametri dell’algoritmo,chiamati **chiavi**. Se P è il file di testo in chiaro,K_E è la chiave di codifica,C è il testo cifrato ed E è l’algoritmo di codifica,allora **C = E(P,K_E)**. Questa è la definizione di cifratura. Indica che il testo cifrato è ottenuto usando l’algoritmo di cifratura E (conosciuto) con il testo in chiaro P e la chiave di cifratura (segreta) K_E come parametri. L’idea che gli algoritmi debbano essere pubblici e che la segretezza risieda esclusivamente nelle chiavi è chiamata **principio di Kerckhoffs**. Analogamente, **P = D(C,K_D)** dove **D** è l’algoritmo di decifrazione e K_D è la chiave di decifrazione. Ciò indica che per ottenere il testo in chiaro P,con il testo cifrato C e la chiave di decifrazione K_D si deve eseguire l’algoritmo di decifrazione D con C e K_D come parametri.

Crittografia a chiave segreta

Consideriamo un algoritmo di crittografia dove ogni lettera viene sostituita da una lettera diversa,ad esempio le A sono sostituite dalle Q e così via. Questo sistema è detto **sostituzione monoalfabetica**,e la chiave è la stringa di 26 lettere

corrispondente all'intero alfabeto. Questa cifratura è facilissima da violare,ma illustra un'importante classe di sistemi crittografici. Quando è facile ottenere la chiave di decodifica del testo cifrato,come in questo caso,si parla di **crittografia a chiave segreta** o **crittografia a chiave simmetrica** (mittente e destinatario devono essere in possesso della chiave segreta condivisa). Sebbene i codici di sostituzione monoalfabetica siano completamente senza valore,vi sono altri algoritmi a chiave simmetrica relativamente sicuri,purché le chiavi siano abbastanza lunghe (256 bit).

Crittografia a chiave pubblica

I sistemi a chiave segreta sono efficienti perché la quantità di calcolo richiesto per cifrare o decifrare un messaggio è gestibile,ma ha un grande svantaggio : mittente e destinatario devono essere entrambi in possesso della chiave condivisa. Per aggirare questo problema si usa la **crittografia a chiave pubblica**. Questo sistema ha la peculiarità che per la cifratura e la decifrazione vengono usate due chiavi distinte e che,data una chiave di cifratura scelta oculatamente,è praticamente impossibile scoprire la chiave di decifrazione corrispondente.
La cifratura fa uso di un'operazione molto semplice mentre la decifrazione senza la chiave richiede l'esecuzione di un'operazione molto complessa.
La crittografia a chiave pubblica funziona nel modo seguente : si genera una coppia di chiavi (chiave pubblica,chiave privata) e si pubblica la chiave pubblica. La chiave pubblica è la chiave di cifratura,quella privata è la chiave di decifrazione. Solitamente la generazione della chiave è automatica,eventualmente con l'inserimento nell'algoritmo di una password utente come seme. Per spedire un messaggio segreto ad un utente,un corrispondente cifra il messaggio con la chiave pubblica del destinatario. Poiché solo il destinatario ha la chiave privata,solo questi sarà in grado di decifrarlo. La crittografia a chiave pubblica piace perché basta pubblicare la propria chiave pubblica : tutti potranno usarla e si è sicuri che solo il destinatario potrà interpretare il messaggio. Il problema di questo tipo di crittografia è che è molto più lenta di quella a chiave segreta (dove bisogna trovare il modo di trasmettere in modo sicuro la chiave a tutti i partecipanti della comunicazione).

Buffer overflow

Una fonte notevole di attacchi è dovuta al fatto che praticamente tutti i SO e la maggior parte dei programmi di sistema sono scritti in C o C++. Sfortunatamente,nessun compilatore C/C++ esegue un controllo dei limiti degli array. Per esempio,la funzione di libreria *gets* di C,che legge una stringa (di dimensione ignota) in un buffer di dimensione fissa,ma senza alcun controllo dell'overflow,è famosa per essere soggetta a questo tipo di attacchi (alcuni compilatori rilevano il suo impiego ed emettono un avviso). Di conseguenza la sequenza di codice seguente,per quanto non corretta,non è controllata :

```
void A() {
```

```
        char B[128];
        printf("Inserire messaggio di log: ");
        gets(B);
        writeLog(B);
}
```

La funzione A è una procedura di logging molto semplificata. Ogni volta che viene eseguita invita l'utente ad inserire un messaggio di log,quindi legge nel buffer B ciò che l'utente ha digitato usando la *gets*. Chiama quindi la funzione personalizzata *writeLog*,che dovrebbe scrivere la voce in un formato piacevole. Questo codice ha un bug molto grave,non evidente a prima vista : il problema è causato dal fatto che *gets* legge i caratteri dallo stdin fino a quando incontra un carattere di nuova riga,e non sa che il buffer B può contenere solo 128 byte. Supponiamo che l'utente scriva 256 caratteri. Dato che le violazioni dei limiti del buffer non vengono controllate,anche i 128 byte rimanenti saranno memorizzati nello stack,come se il buffer fosse di 256 byte. Tutto ciò che era memorizzato nelle posizioni di memoria successive alla fine del buffer viene sovrascritto.

Vediamo lo scenario del programma in esecuzione,con le variabili locali nello stack.

Ad un certo punto viene chiamata la procedura A. La sequenza di chiamata standard inizia spingendo sullo stack l'indirizzo di ritorno (che punta all'istruzione successiva alla chiamata),quindi trasferisce il controllo ad A,che riduce di 128 il puntatore allo stack per allocare spazio per la variabile locale (buffer B). Nel caso in cui l'utente inserisce più di 128 caratteri la funzione *gets* copia tutti i byte nello stack e sovrascrive l'indirizzo di ritorno;una parte della voce del log ora riempie quindi la locazione di memoria che,secondo il sistema,dovrebbe contenere l'indirizzo dell'istruzione a cui passare quando la funzione ritorna. Se l'utente inserisce un messaggio di log normale,i caratteri del messaggio probabilmente non rappresentano un indirizzo valido nel codice. Non appena la funzione A ritorna,il programma cercherà di saltare ad un indirizzo non valido,cosa che il sistema non gradisce e il programma andrà immediatamente in crash.

In questo modo,un attacker potrebbe inserire in input una stringa preparata in modo da sovrascrivere l'indirizzo di ritorno con l'indirizzo del buffer B. Il risultato è che,quando ritorna dalla funzione A,il programma torna all'inizio del buffer B ed esegue come codice i byte che contiene. Dato che l'attacker controlla il contenuto del buffer,può riempirlo con istruzioni macchina in modo che,nel contesto del programma originale,saranno eseguite istruzioni preparate dall'aggressore. Il programma è così sotto il controllo dell'attacker che può fare ora quello che vuole. Il trucco non funziona solo con i programmi che usano *gets*,ma per qualsiasi codice che copia in un buffer dati forniti da un utente senza controllare le violazioni dei limiti. I dati dell'utente possono essere costituiti da parametri della riga di comando,stringhe di ambiente,dati inviati su una connessione di rete o dati letti da un file.

Introduzione

Fino ad ora i processi visti erano indipendenti. Nella pratica,è molto comune che i processi interagiscono tra di essi per condividere dati,velocizzare elaborazioni,bilanciare il carico ecc. Ad esempio,in una pipeline della shell,l'output del primo processo deve essere passato al secondo processo e così via. C'è quindi bisogno che i processi comunichino,preferibilmente in un modo ben strutturato,non usando gli interrupt. Per fare in modo che due processi comunichino sono necessarie due tipologie di oggetti: un mezzo fisico per far parlare i processi e delle primitive per imporre dei vincoli nell'ordine in cui i processi si parlano.
Per far parlare due processi,una soluzione "scarsa" è ***mmap***,che ci consente di richiedere al kernel che nello spazio di indirizzamento logico venga creata un'area di memoria. Questa chiamata di sistema prevede una serie di parametri : indirizzo dell'area di memoria,dimensione,una serie di flag come flag_shared (area condivisa),prot_ (protezione lettura scrittura),map_anonymous che serve a dire se l'area di memoria deve essere mappata anche su un file … In questo modo,ad esempio,il codice di un processo che stampa un valore condiviso,non lo modifica ma ogni volta che gira trova cose diverse perché modificato da un altro processo. Se un processo sta utilizzando un dato condiviso,è necessario un meccanismo che,fino a quando non sono state effettuate tutte le operazioni di un processo su un dato,permetta che gli altri non possano operare su di esso,detto **mutua esclusione**.
In particolare la **sezione critica** è una serie di istruzioni che accedono ad una risorsa condivisa. Bisogna fare in modo che quando un processo si trovi nella sua sezione critica,nessun altro processo possa entrare in essa e quindi che l'esecuzione di uno escluda mutuamente l'altro. In questo modo,quando P1 esegue la propria sezione critica,P2 dovrà attendere che P1 termini per poter eseguire la sua sezione critica e questa tecnica è detta **sincronizzazione**.

Programmazione concorrente

E' l'insieme delle metodologie e degli strumenti necessarie per fornire il supporto all'esecuzione di applicazioni software come un insieme di attività svolte "simultaneamente". L'elaborazione concorrente è realizzabile anche su sistemi monoprocessori,dato che si avvia un processo,lo scheduler gli dà processore e dopo un po' viene tolto per dare spazio ad un altro processo prima che il precedente sia terminato. Se si hanno invece più processori si parlerà invece di **parallelismo**,con attività che sono sovrapposte nel tempo e anche in esecuzione nello stesso istante.
La programmazione concorrente prevede due macro-aree :

- primitive per definire attività indipendenti (processi,threads). Un esempio è la ***fork***;
- primitive per la **comunicazione** e **sincronizzazione** tra attività eseguite in modo concorrente (concorrenza non significa necessariamente parallelismo). Esse servono per fare in modo che se un processo sta scrivendo un'area di memoria condivisa,l'altro non possa accedervi.

Per la comunicazione dei processi in Linux,un approccio prevede una **memoria condivisa** (un esempio è la mmap),dove P1 e P2 condividono un pezzo di memoria. Un altro approccio è quello dei **messaggi**,dove P1 crea un pacchetto e lo manda a P2 con un messaggio.

Race condition

E' una situazione in cui due o più processi leggono o scrivono dati condivisi e il risultato finale di queste operazioni dipende dai tempi precisi in cui vengono eseguiti. Il debugging di programmi concorrenti che presentano race condition è molto complesso,dato che ad ogni esecuzione si produrrà un risultato diverso.

Processi concorrenti

- **processi indipendenti** : due processi P1 e P2 sono indipendenti se l'esecuzione di P1 non è influenzata da P2,e viceversa;
- **processi interagenti** : due processi P1 e P2 sono interagenti se l'esecuzione di P1 è influenzata da P2,e viceversa.

Tipi di interazione

- **competizione** : per l'uso di risorse comuni che non possono essere utilizzate contemporaneamente. L'unico modo per garantire che non succeda è fare in modo che quando un processo sta utilizzando una risorsa un altro non possa accedervi;
- **cooperazione** : nell'eseguire un'attività comune mediante scambio di informazioni. Ad esempio,nella memoria condivisa,oltre a doversi preoccupare che il dato sia scritto in modo opportuno, un esempio può essere che P2 si blocchi se non c'è il dato e che riprenda solo quando P1 abbia scritto qualcosa. Serve quindi che un processo possa segnalare all'altro processo che il dato è presente;
- **interferenza** : è una modalità di interazione tra processi non dovuta. E' dovuta a competizione tra processi per uso non autorizzato di risorse comuni,oppure ad un erronea soluzione di problemi di competizione e di cooperazione.

Sincronizzazione

Sincronizzare significa imporre dei vincoli (nell'ordine e nel tempo) nella esecuzione delle operazioni dei processi. Tra i vincoli per la sincronizzazione abbiamo :

- **competizione** : un solo processo alla volta deve avere accesso alla risorsa comune (sincronizzazione indiretta o implicita);

- **cooperazione** : le operazioni eseguite dai processi concorrenti devono seguire una sequenza prefissata (sincronizzazione diretta o esplicita).

Mutua esclusione

Lo scenario è quello in cui due (o più processi) vogliono utilizzare una risorsa (ad esempio memoria o file condiviso) ad uso esclusivo,denominata **risorsa critica**. La parte di programma che utilizza la risorsa (condivisa) è denominata **regione critica**. L'obiettivo è quello di fare in modo che non ci siano mai due processi che si trovano nelle loro regioni critiche allo stesso momento. In altre parole,ci serve una **mutua esclusione**,ovvero un sistema che,se un processo sta usando una risorsa critica,agli altri processi venga impedito di fare la stessa cosa.
Immaginiamo di avere un processo A,che ad un certo punto entra nella sua regione critica per operare con la variabile condivisa per un tot di tempo. Mentre A è in esecuzione,se abbiamo un processo B che vuole essere eseguito,il SO dovrà garantire un supporto per fare in modo che l'accesso di B sia ritardato fino a quando A non sarà uscito dalla sua regione critica. Quando A ha terminato l'esecuzione della sua regione critica,allora B potrà eseguire la sua regione critica. Ciò significa che se c'è un processo che tenta di accedere alla sezione critica quando un altro processo la sta eseguendo,il processo viene bloccato e quindi viene messo in attesa.
Ci sono delle condizioni che devono però essere rispettate:

1. Non devono mai esserci due processi contemporaneamente nelle loro regioni critiche;
2. Non può essere fatta alcuna supposizione sulle velocità o sul numero delle CPU. Ad esempio,se sappiamo che B è più lento ad abbiamo occupato una sezione critica,si potrebbe ipotizzare che B non acceda alla sezione critica prima che ho operato su A;
3. Nessun processo in esecuzione al di fuori della sua regione critica può bloccare altri processi;
4. Nessun processo dovrebbe restare per sempre in attesa di entrare nella sua regione critica. Ciò significa che prima o poi,un processo che ha fatto richiesta di accedere alla propria regione critica,vi acceda.

Possibili soluzioni al problema

Disabilitare gli interrupt

Mentre A è in esecuzione,potrebbe arrivare un interrupt e il SO inizia ad operare. Ogni volta che arriva un interrupt può intervenire anche lo scheduler,il quale può sottrarre processore al processo in esecuzione. Su un sistema a singolo processore la soluzione potrebbe essere che ogni processo disabiliti tutti gli interrupt appena entrato nella sua regione critica e li riabiliti quando ne esce. Con gli interrupt disattivati non può esservi alcun interrupt del clock. La CPU passa da processo a

processo solo con clock interrupt o altri interrupt,e con gli interrupt disabilitati la CPU non sarà passata ad altri processi. In questo modo,una volta che il processo ha disattivato gli interrupt,può esaminare e aggiornare la memoria condivisa senza temere che un interrupt di qualche processo intervenga. La soluzione prevede però che l'utente possa disabilitare gli interrupt e quindi non è una soluzione pratica,dato che ad esempio potrebbe dimenticarsi di riattivarli una volta terminata la sezione critica del processo. E' però spesso comodo per il kernel stesso disabilitare gli interrupt per poche istruzioni,mentre sta aggiornando variabili o liste critiche. Se per esempio avviene un interrupt mentre l'elenco dei processi pronti non è coerente,potrebbero verificarsi race condition. La soluzione potrebbe quindi essere utile all'interno del SO ma non è indicata come meccanismo di **mutua esclusione** per processi utente. Nel caso di multiprocessori,la disattivazione degli interrupt riguarda solo la CPU che ha eseguito l'istruzione ***disable*** mentre le altre continuerebbero l'esecuzione e potrebbero accedere alla memoria condivisa.

Variabili lock

Si tratta di una soluzione software. Supponiamo di avere una sola variabile condivisa,chiamata ***lock***,inizialmente posta a 0. Quando un processo vuole entrare nella sua regione critica,controlla prima il *lock*. Se è 0,il processo lo imposta ad 1 ed entra nella regione critica. Se è già ad 1,il processo aspetta fino a quando il lock vale 0. Quindi,0 significa che nessun processo è nella sua regione critica,1 che qualche processo è nella sua regione critica. Questo approccio presenta però un problema. Supponiamo che un processo legga il lock e veda che è 0. Prima che riesca a metterlo ad 1,viene eseguito un altro processo schedulato che imposta il lock ad 1. Quando il primo processo riparte,anch'esso lo imposterà ad 1 e i due processi sarebbero nella loro regione critica nello stesso momento. Si potrebbe risolvere il problema leggendo prima il valore del lock e poi controllandolo prima di salvarlo,ma ciò non aiuta dato che la race condition si verifica ora se il secondo processo modifica il lock subito dopo che il primo processo ha finito il secondo controllo. Il problema è che l'operazione di test e set del lock sono divisibili e non sono atomiche e quindi bisogna renderle atomiche.

Istruzione TSL

E' la soluzione per risolvere il problema delle variabili lock tramite supporto dell'hardware. Alcuni computer,specie quelli multiprocessore,hanno un'istruzione macchina come ***TSL RX,LOCK*** (Test and Set Lock) che funziona in questo modo : legge il contenuto della parola della memoria *lock* nel registro RX e poi salva un valore non zero all'indirizzo di memoria *lock* (tutto in un unico ciclo). Le operazioni di lettura della parola e del suo salvataggio sono garantite come indivisibili : nessun altro processore potrà accedere alla parola in memoria fino a quando l'istruzione è terminata. La CPU che esegue l'istruzione TSL blocca il bus di memoria per proibire ad altre CPU di accedere alla memoria fino a che non avrà finito.

Bloccare il bus della memoria è diverso da disabilitare gli interrupt. Disabilitare gli interrupt e poi eseguire una lettura su una parola di memoria seguita da una scrittura non impedisce ad un secondo processore di accedere alla parola fra la lettura e la scrittura. Infatti disabilitare gli interrupt sul processore 1 non ha effetto sul processore 2.
Per usare TSL si usa la variabile condivisa *lock* per coordinare l'accesso alla memoria condivisa. Quando *lock* è 0,qualsiasi processo può impostarla ad 1 usando l'istruzione TSL e poi leggere o scrivere la memoria condivisa. Quando ha finito,il processo imposta *lock* di nuovo a 0 usando una normale istruzione *move*.
L'accesso e l'uscita dalla regione critica,per evitare che due processi entrino contemporaneamente nelle loro regioni critiche funziona tramite due frammenti di codice :

```
enter_region:

        TSL REGISTER,LOCK
        CMP REGISTER,#0
        JNE enter_region
        RET
```

La prima istruzione,permette in un solo ciclo di clock e in modo indivisibile,di copiare il valore di *lock* corrente nel registro e imposta il *lock* ad 1. In questo modo il *lock* sarà bloccato per gli altri processi che provano ad accedere alla regione critica. Dopo aver fatto ciò,confronto il contenuto del registro con il valore zero e se il *lock* è 0 procedo con la sezione critica. Se il *lock* invece non è 0,il *lock* era stato impostato si riesegue il ciclo entrando in un test ciclico sulla variabile *lock* (si parla di attesa attiva/busy waiting).
Il frammento di codice per uscire dalla regione critica è invece il seguente :

```
leave_region:
        MOVE LOCK,#0
        RET
```

Si mette 0 nel *lock* e si ritorna al chiamante.
Quindi,prima di entrare nella sua regione critica,un processo chiama *enter_region*,la quale va in **busy waiting** (attesa attiva → testare continuamente una variabile finché non è valorizzata) finché il lock non si libera;poi acquisisce il *lock* e ritorna. Dopo la regione critica il processo chiama *leave_region*,che salva uno 0 in *lock*. Se i due frammenti vengono invocati in modo errato la mutua esclusione fallisce.
Oltre al problema del busy waiting c'è un problema molto più critico,ovvero la **priority inversion**. Supponiamo che ad entrare nella regione critica sia un processo con **priorità bassa (low = L)** che è arrivato a metà della sua regione critica. Supponiamo che arrivi un processo ad **alta priorità (high = H)** che vuole entrare

nella regione critica e che la politica di scheduling sia tale per cui il processo a priorità più bassa debba essere buttato fuori. A questo punto,se ad esempio all'istruzione i-esima della regione critica si butta fuori L e si dà processore ad H che vuole accedere alla regione critica,H troverà il lock ad 1 perché L non è riuscito a terminare la sua sezione critica per rimetterlo a 0. H entrerà quindi in un loop infinito di check sul *lock* ed L,a bassa priorità,non verrà mai selezionato per andare sul processore fino a che H è in esecuzione. Si verifica così una situazione di **stallo** chiamata **deadlock**.
Una istruzione alternativa alla TSL è **XCHG** che scambia ad esempio un registro e una parola in memoria in modo atomico.

Suspend (sleep) e wakeup

Il processo che trova LOCK ad 1 si sospende e restituisce volontariamente la CPU allo scheduler. Il processo che era in esecuzione sulla CPU andrà nella coda dei processi bloccati. Il processo che trova il LOCK bloccato effettua una ***suspend*** per lasciare il processore e diventare blocked. Chi invece sta utilizzando la regione critica deve avere a disposizione un'altra funzione chiamata ***wakeup*** che prenderà un processo in attesa sul LOCK e lo sposterà nella coda dei processi pronti,da dove prima o poi lo scheduler selezionerà un processo da mandare in esecuzione. La chiamata *wakeup* ha un parametro,il processo da risvegliare. In alternativa,le due chiamate,hanno un parametro,un indirizzo di memoria usato per far abbinare ogni *suspend* con la rispettiva *wakeup*. Nella pratica potrebbe servire sapere quanti processi sono in attesa su un certo LOCK e ciò porta ad una variabile detta **semaforo**.

Semaforo

Il semaforo è un tipo di dato caratterizzato da un **valore intero** (contatore) e da una **lista** (coda) di processi in **attesa** (processi in coda su quel semaforo).

```
typedef struct {

        int value;
        struct process *list;
} semaphore;
```

Il semaforo supporta tre operazioni :

1. Inizializzazione ad un valore non negativo;
2. **wait** : decrementa di 1 il valore del semaforo e quando questo valore diventa minore di 0 il processo viene bloccato.

```
wait (semaphore *S) {
```

```
        S → value–;

        if (S→value < 0) {

                add this process to S→list;
                block();
        }
}
```

Il valore iniziale a cui viene inizializzato il semaforo,può essere anche definito come il numero di processi che possono eseguire dopo la *wait* o contemporaneamente sul codice che seguirà. ***block()*** sospende il processo che la invoca. Le operazioni di controllo del valore,della sua modifica ed eventualmente del passaggio allo stato di *suspend* sono eseguite come singola **azione atomica** indivisibile. E' garantito che una volta che l'operazione sul semaforo è avviata,nessun altro processo può accedere al semaforo finché l'operazione non è completata o bloccata. Questa atomicità è assolutamente essenziale per risolvere i problemi di sincronizzazione ed evitare race condition. La *wait*,quindi,controlla se il valore del semaforo è maggiore di 0. Se è così,decrementa il valore (cioè consuma un *wakeup* memorizzato) e quindi continua. Se il valore è 0 il processo viene messo in *suspend*,per il momento senza completare la *wait*.

3. **signal** : aumenta il valore del semaforo coinvolto.

```
signal (semaphore *S) {

        S→value++;

        if (S→value <= 0) {

                remove a process P from S→list;
                wakeup(P);
        }
}
```

wakeup(P) pone in *pronto* un processo P bloccato. Se uno o più processi erano in *suspend* su quel semaforo,incapaci di completare una precedente operazione di *wait*,il sistema ne sceglie uno (ad esempio casualmente) e gli permette di completare la sua *wait*. Dopo una *signal* su un semaforo contenente più processi in *suspend*,il semaforo avrà sempre valore 0,ma ci sarà un processo in meno nello stato di *suspend*. Anche l'operazione di incremento del semaforo e di risveglio del processo è indivisibile. Nessun processo si blocca nel fare una *signal*,così come nessun processo si blocca nel fare un *wakeup* nel momento precedente.

nota : i valori del semaforo sono anche chiamati **permessi** in java.

Modello concettuale di un semaforo

Supponiamo di inizializzare il semaforo ad 1 → S = 1. Nella coda processi pronti abbiamo C,D,B. A esegue una wait e quindi S = 0 ed A viene aggiunto alla coda dei processi pronti. Immaginiamo che con approccio FIFO,B venga scelto dal processore ed esegua anch'esso una wait. Quindi avremo S = -1 e B viene aggiunto alla coda dei processi sospesi. A questo punto,D prende possesso del processore ed esegue una signal. Quindi S = 0 e dato che è <= 0 uno dei processi che si trova nella coda dei processi sospesi,in questo caso abbiamo solo B,passa nella coda dei processi pronti.
L'aggiunta-eliminazione di processi dalla coda del semaforo può essere basata su strategia FIFO.
Il valore del semaforo non è più solo una variabile di conteggio ma una variabile di conteggio e una coda. Se abbiamo inizializzato il semaforo ad 1,avremo P1/P2 che esegue una wait (prima istruzione della sua sezione critica è wait),decrementa S di 1 e prende possesso della risorsa condivisa ovvero ciò che c'è dopo la wait. Se P2 esegue la wait mentre P1 è ancora in esecuzione,S = -1 e P2 viene messo in attesa. Si risolve quindi il busy waiting e il priority inversion. Prima o poi P1 effettuerà la signal,rilascerà il permesso sul semaforo,P2 sarà spostato nella coda dei pronti per poter eventualmente essere eseguito.
E' necessario garantire che due processi non possano eseguire contemporaneamente ***wait*** e ***signal*** sullo stesso semaforo : le operazioni sui semafori sono eseguite come singola azione **atomica** indivisibile. Il SO,prenderà quindi una serie di precauzioni per le istruzioni che effettueranno il test e l'aggiornamento per eseguirle in modo atomico. Nei sistemi monoprocessore verranno ad esempio disabilitati brevemente l'interrupt.

Mutua esclusione con semafori

Per risolvere il problema della mutua esclusione con i semafori basterà ricorrere ad un semaforo inizializzato ad 1. Esso prende il nome di **semaforo binario**. Tutti i processi che utilizzano una regione critica dovranno fare *wait* su questo semaforo (anche chiamato **mutex** : mutual exclusion) e la *signal* sempre su questo semaforo. Anche se abbiamo visto questo concetto con due codici diversi (padre e figlio),si ottiene lo stesso effetto prendendo lo stesso programma ed eseguendolo *n* volte : possiamo avere *n* processi che eseguono lo stesso programma e che avranno la stessa sezione critica ed accederanno alla stessa variabile. Questa è la soluzione per la **competizione**.

Cooperazione con semafori (produttore-consumatore)

Due processi condividono un buffer comune,di dimensione fisse. Uno dei due,il produttore,mette informazioni nel buffer e l'altro,il consumatore,le preleva. La cosa si complica quando il produttore vuole mettere un nuovo elemento nel buffer e questo è già pieno.
Ci sono quindi dei vincoli :

- il produttore non può produrre se il buffer è pieno;
- il consumatore non può prelevare da un buffer vuoto.

Questo problema può essere risolto per mezzo dei semafori. La soluzione impiega tre semafori :

- **full** : è inizializzato a 0 e serve per il conteggio del numero di posti pieni. Serve quindi a bloccare il consumatore quando il buffer è vuoto;
- **empty** : è inizializzato ad N,ovvero il numero di posti del buffer,e serve per il conteggio del numero di posti vuoti. Serve quindi a bloccare il produttore quando il buffer è pieno;
- **mutex** : è inizializzato ad 1 e serve per assicurarsi che produttore e consumatore non accedano al buffer nello stesso istante (serializza gli accessi al buffer condiviso).

Supponiamo a questo punto di avere due funzioni,un produttore e un consumatore.
Il produttore ad un certo punto fa inserire un intero (item) e il consumatore rimuove l'item e lo salva in una variabile intera. Bisogna accertarsi che le due operazioni siano effettuate in maniera mutuamente esclusiva e abbiamo quindi alla base un problema di **competizione**. Esso si risolve tramite il semaforo **mutex = 1** per proteggere l'accesso alla parte condivisa. Si utilizza poi il semaforo **full = 0** (il buffer è vuoto) : in particolare il consumatore farà la *wait* sul semaforo full e si bloccherà se full = 0. Dopo che il produttore avrà prodotto,si effettua la *signal* nel produttore per svegliare il processo in attesa. Il produttore dovrà essere invece bloccato se il buffer è pieno : ciò si realizza tramite semaforo **empty = N** , che viene utilizzato con una *wait* dal produttore e dal consumatore con una *signal* dopo che il consumatore ha prelevato. Se arrivano 10 processi insieme,avremo potenzialmente 10 processi che possono produrre ed accedere alla risorsa condivisa (l'11 si bloccherà). Il semaforo mutex è utilizzato per la mutua esclusione e cioè per garantire che un solo processo alla volta stia leggendo o scrivendo il buffer e le variabili associate.

Esercitazione programmazione IPC

L'IPC (Inter-process communication) tratta la comunicazione tra processi tramite strutture dati rese disponibili dal kernel,chiamate **risorse IPC**. Esse sono :

- memoria condivisa o shared memory (**shm**);
- semafori (**sem**);
- code di messaggi (**msg**).

Quando serve una risorsa IPC,la richiesta deve essere formulata al kernel tramite una system call specifica. Il kernel tiene traccia di questi oggetti che gli vengono richiesti. Nello specifico,ogni risorsa IPC già esistente è identificata da un **valore univoco** nel sistema,denominata **chiave (IPC key)**. Un IPC key può essere "cablata" nel codice oppure generata dal SO tramite l'invocazione di ***ftok***.
Un comando critico per l'IPC,è ***ipcs*** : esso visualizza tutte le strutture allocate (o solo shm,sem,msg) mostrandone anche l'identificatore e l'utente proprietario.
Un altro comando è ***ipcrm***,utilizzato per rimuovere una data struttura,noto il suo identificatore.

Come creare segmento di memoria condivisa

Bisogna innanzitutto includere **<shm/sys.h>**. Per creare un segmento di memoria condivisa si utilizza la funzione ***shmget***. Essa restituisce un intero : se è -1 non è andata a buon fine altrimenti avrà un valore intero e cioè l'identificatore numerico per la SHM (descrittore).
La *shmget* utilizza tre argomenti :

- **key** : identifica la SHM in maniera univoca nel sistema. E' un intero,ma lo indicheremo in esadecimale per una migliore visibilità (es. 0xaa);
- **size** : dimensione (in byte) della SHM,ad esempio se si condivide un intero sarà pari a 4;
- **flag** : modalità di creazione e permessi di accesso (IPC_CREAT,IPC_EXCL,permessi). Utilizzeremo **IPC_CREAT | 0666**. Esso ci permette di creare un segmento di memoria con una determinata chiave se non esisteva già. Se non venisse specificato questo flag,la procedura ritorna con codice -1. **IPC_EXCL** (exclusive),se trova quella chiave già esistente,ritorna con codice -1. Lo **0** davanti a **666** ci dice che che 666 va interpretato come un ottale. In particolare i **permessi** vanno letti come triple di bit,con 1 che indica che il permesso c'è e 0 che non c'è. I permessi sono rappresentati in questo modo : ***rwx rwx rwx*** (r = read , w = write , x = execute),con i primi 3 per l'utente,i centrali per il gruppo a cui appartiene l'utente e gli ultimi 3 per il resto. Quindi 666 corrisponderà a : 110 110 110 , e quindi l'utente,il gruppo e il resto hanno permessi per leggere e scrivere.

Dopo aver richiesto il segmento di memoria tramite *shmget* bisognerà poi annetterlo nello spazio di indirizzamento del processo tramite ***shmat*** (shm attach),la quale

collega il segmento in memoria. Restituisce l'indirizzo del segmento collegato e -1 in caso contrario. La *shmat* ha tre argomenti :

- **shmid** : identificatore SHM,restituito dalla *shmget*;
- **shmaddr** : indirizzo dell'area di memoria del processo chiamante al quale collegare la SHM. E' tipicamente impostato a NULL (lasceremo fare al kernel);
- **flag** : IPC_RDONLY per collegare in sola lettura altrimenti **0** per lettura e scrittura.

Per scollegare il segmento si può usare ***shmdt (detach)*** e per eliminarlo si può usare ***shmctl*** ma noi useremo ***ipcrm***.
La shm viene inizializzata a 0 in automatico.
Rilanciando il programma che incrementa un valore condiviso,ripartirà dall'ultimo valore salvato. Eseguendo lo stesso programma su due prompt,ci sarà una perdita di aggiornamenti : entrambi provano ad incrementare di 100,vanno in conflitto e si avrà solo un incremento di 100 (e non 200). Risolveremo questo problema tramite altre implementazioni sul codice.
A questo punto chiamando *ipcs* troveremo la nuova memoria condivisa con ID x0aa,con permessi 666,dimensione 4,nella colonna **nattach** avremo il numero di processi collegati a quel determinato segmento (in quel determinato momento) e lo **stato**. Per la memoria creata da noi non c'è ma in genere è **dest** : Se si richiede la cancellazione di un segmento di memoria la risorsa non viene cancellata immediatamente ma solo quando non ci sono più processi collegati.
Per rimuovere il segmento si usa **ipcrm -m *id*** (**-m** indica che si deve rimuovere un segmento di memoria).

Semafori

L'idea è quella di programmare la mutua esclusione sul segmento di memoria creato. Bisogna innanzitutto chiedere il semaforo. Si utilizza la funzione ***semget***,la quale restituisce un identificatore numerico per l'array di semafori e -1 in caso contrario. I parametri sono :

- **key** : come nella shmget;
- **nsems** : numero di semafori → ogni volta che si crea un semaforo,il SO crea un array di semafori. Ciascun semaforo avrà il proprio indice in un array;
- **semflag** : modalità di creazione e permessi (come nella shmget);

Bisognerà poi inizializzare il semaforo. Si utilizza la chiamata ***semctl (sem control)***. Essa ha come parametri :

- **semid** ;
- **indice** del semaforo nell'array;

- **flag** : **SETVAL** → imposta valore;
- **valore iniziale** : 1 per mutua esclusione.

Per chiedere i permessi e quindi decrementare il semaforo si usa ***wait*** mentre per restituire i permessi si usa ***signal***. Queste due operazioni non possono essere effettuate direttamente tramite chiamate di sistema. Vengono implementate due procedure **sem_wait** e **sem_signal**.
In particolare abbiamo una system call chiamata **semop**,la quale permette di eseguire N operazioni sui semafori. Essa chiede su quale semaforo applicare N operazioni (semid). Il numero di operazioni da eseguire è specificato dall'ultimo parametro (unsigned nsops). Ciascuna operazione è descritta da una *struct* chiamata **sembuf** :

```
struct sembuf {

        short sem_num; // identifica l'indice del semaforo nell'array
        short sem_op; // operazione da applicare
        short sem_flg; // modalità con cui è applicata l'operazione
}
```

Se diamo un valore negativo a sem_op,il semaforo viene incrementato della quantità ops.sem_op ed equivale ad una *wait* (si blocca se sem_val <= 0).
Se diamo un valore positivo a sem_op,il semaforo viene incrementato della quantità ops.sem_op ed equivale ad una *signal*.
Se diamo un valore uguale a 0 a sem_op,si mette un processo in attesa che il valore del semaforo diventi 0.

I valori che possono invece essere assegnati a **sem_flg** sono :

- **IPC_NOWAIT** : abbiamo visto che i semafori possono essere utilizzati facendo in modo che si effettui una *wait* e se non si può procedere il processo viene bloccato. Nella pratica,può essere utile che se non si hanno i permessi il processo non si blocchi per poi riprovare il futuro (comodo per il polling);
- **SEM_UNDO** : ripristina il vecchio valore quando termina.

Implementazione wait

```
void sem_wait (int id_sem,int numsem) {

        struct sembuf sem_buf;

        sem_buf.sem_num = numsem;
        sem_buf.sem_op = -1;
        sem_buf.sem_flg = 0;

        semop(id_sem,&sem_buf,1);

}
```

La sem_wait istanzia un sem_buf,ovvero il descrittore dell'operazione,dove imposta il flag in modo che venga fatta una *wait* di 1. Avremo **numsem** (id del semaforo nell'array),sem_op = -1 e i flag pari a 0 per indicare che è un'operazione classica (né non bloccante né col ripristino dei valori iniziali). Viene poi chiamata la semop(id_sem,&sem_buf,1) → 1 perché stiamo eseguendo una sola operazione.

Implementazione signal

è la stessa cosa solo che sem_op è pari ad 1 e non a -1.
Quindi per entrare nella sezione critica facciamo sem_wait sul semaforo e sem_signal sullo stesso semaforo quando usciamo dalla sezione critica.

Problema inizializzazione

Implementando in questo modo il programma,le cose non cambieranno comunque : entrambi i processi impostano il semaforo ad 1 → ogni processo si alza,si attacca alla memoria,si attribuisce un permesso ed accede. Il problema è che ogni processo che arriva si autosblocca e inizia a lavorare. Questo perché il codice di inizializzazione è eseguito da tutti i programmi e ciò si deve evitare : chi arriva dopo si deve attaccare ad un semaforo già esistente e già inizializzato,senza reinizializzare. Solo il primo processo deve inizializzare il semaforo.
La soluzione che vedremo è semi-corretta e si "gioca" col flag IPC_CREAT : con esso chiediamo che se non c'è la risorsa con quella chiave deve essere creata ; se non lo specifichiamo,se non c'è quella risorsa essa non viene creata. Effettuiamo quindi una prima invocazione in cui non chiediamo di fare anche la creazione. In questo modo un processo ha modo di sapere che è il primo che sta provando a creare il semaforo. Quindi la creazione e inizializzazione del semaforo avvengono solo se il semaforo non c'è. Quando il secondo processo prova a chiedere quella chiave per quel semaforo,se c'è gli viene restituito l'identificativo e parte con l'utilizzo del semaforo.

Questa soluzione può generare una **race condition** : tra la semget e la semctl se più processi accedono al semaforo con una chiave comune ed eseguono il codice di creazione-inizializzazione del semaforo. La creazione e inizializzazione non sono atomiche mentre la semop è atomica. Si può risolvere la race assumendo che ci sia un processo incaricato di fare l'inizializzazione e che ciò avvenga prima che altri processi inizino ad utilizzare il semaforo. Un'altra soluzione è utilizzare due semafori per inizializzarne uno : uno che assicura mutua esclusione sulla chiave e l'altro per assicurare mutua esclusione sulla get. L'ultima soluzione è quella di mettere in polling (fino a un certo numero di tentativi) e verifica se il semaforo è stato creato : se non è stato creato non viene inizializzato e se dopo un tot di tempo non è stato ancora creato,il processo termina.

Produttore consumatore

Assumiamo che i processi non condividano un solo intero ma che condividano un buffer di N interi. Il buffer ha spazio per **N+1** interi,dove l'ultimo intero memorizza l'indice della prima posizione libera (più a destra) in cui produrre. Il consumatore può leggere da esso e ripulire una posizione. Se non si hanno posizioni piene il consumatore deve attendere e se tutte le posizioni sono occupate il produttore dovrà attendere. Si dovrà chiedere un segmento di memoria condiviso di dimensione pari a **4*N** (4 = dimensione intero e N numero interi). Creiamo il semaforo con semget e nella dimensione metteremo 4*N + 4 , dove il +4 serve per la locazione dell'intero dove si trova l'indice della prima posizione libera in cui il produttore può scrivere e il consumatore consumare. Utilizziamo il buffer come se fosse una pila e quindi memorizzo in una variabile la prima posizione libera in cui scrivere ed eventualmente consumare. Il buffer ha logica LIFO. Il codice rimane poi,fino ad ora,uguale con la differenza che creeremo 3 semafori invece di 1.
Se il semaforo non esiste,dovremo inizializzare 3 semafori : MUTEX , FULL , EMPTY.
Il buffer,come detto,ha N posizioni : da 0 ad N-1 sono le posizioni occupabili mentre N contiene l'indice della prima posizione libera. Quando si crea il segmento di memoria condivisa,ci saranno tutti 0 e quindi 0 è la prima posizione libera che c'è.
Il produttore quindi accede all'ultimo intero per scoprire qual è la posizione libera in cui scrivere. Dovrà poi mettere in quella posizione il valore dato da tastiera,incrementare l'indice (si dovrà scrivere in un altra posizione successivamente).
Il consumatore,invece,dovrà capire dove consumare e quindi farà idx-1 (prima casella occupata essendo idx = buffer[N]) e metterà il valore letto in a. Infine si aggiorna la posizione per permettere di scrivere a chi arriverà dopo.
Bisognerà poi marcare i due pezzi di codice con la sezione critica sem_wait e sem_signal (ad esempio consumatore deve acquisire,bloccare e sommare il semaforo). MUTEX sarà sempre mutuo condiviso e quindi chi esegue il produttore esclude il consumatore. Utilizzare solo la mutua esclusione non è sufficiente a risolvere il problema.

Prima di bloccarsi la propria sezione critica,il produttore e consumatore devono fare dei check. Il consumatore deve controllare il semaforo FULL per vedere quante posizioni sono libere tramite sem_wait e all'inizio sarà 0. A questo punto,quando il consumatore ha consumato e svuotato una posizione,metterà un permesso su EMPTY per indicare al produttore che una posizione è stata svuotata.
In maniera duale il produttore si deve sospendere se non ci sono caselle vuote e quindi EMPTY è diventato 0. Dopo la signal sul MUTEX farà una signal su FULL per avvisare di aver riempito una posizione.
Se lanciamo il programma iniziale inserendo **0** per far partire il consumatore,esso si blocca : ciò si verifica perché FULL = 0 e ci dice quante caselle sono libere. Dato che è 0,facendo la wait su FULL verrà spostato dal processore nella coda dei processi in attesa sul semaforo. Questo rimarrà in attesa fino a quando non arriva un produttore.
Se facciamo sforare il produttore,ad esempio se il buffer ha 3 posizioni e noi ne produciamo 4,alla 4a il produttore si mette in attesa fino a quando non arriva un consumatore,il quale preleva dall'ultima posizione. Questa posizione rimarrà libera e verrà occupata dalla 4a scrittura che era in attesa.