

SISTEMA DI ELABORAZIONE

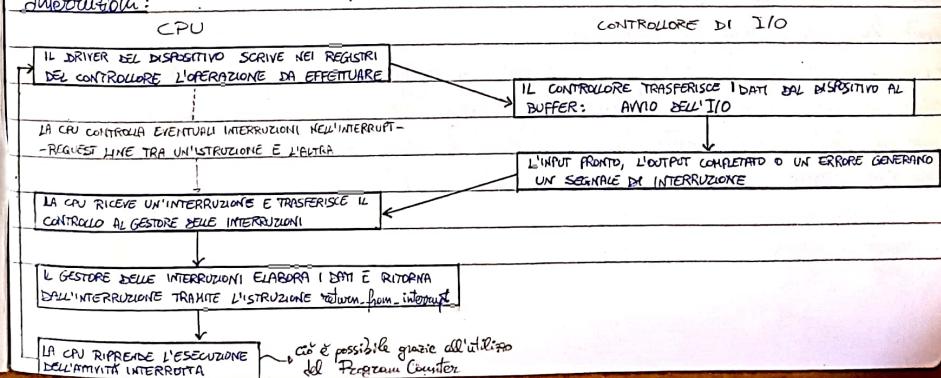
Sottiviso in:

- HARDWARE, composto da:
 - CPU (unità centrale d'elaborazione)
 - memoria
 - I/O (dispositivi d'ingresso e uscita dei dati)
 - ↳ fornisce al sistema le risorse elaborate fondamentali.
- PROGRAMMI APPLICATIVI → definiscono il modo in cui si usano queste risorse per risolvere determinati problemi computazionali.
- SISTEMA OPERATIVO → controlla l'hardware e ne coordina l'utilizzo da parte dei programmi applicativi; in pratica fornisce un ambiente nel quale i programmi possono lavorare in modo utile.

Organizzazione di un sistema di elaborazione:

- Una o più CPU
 - CONTROLLORI di dispositivi connessi con un canale di comunicazione comune (BUS) che permette l'accesso alla memoria condivisa del sistema; talvolta, anziché il canale di comunicazione comune, si hanno degli switch, che permettono ai dispositivi fisici di interagire con varie parti del sistema concorrentemente.
In effetti, i controllori si occupano di un particolare dispositivo fisico e possono gestire una o più unità connesse a esso.
 - DRIVER del dispositivo (uno per ogni controllore), che gestiscono le specificità del controllore e fungono da interfaccia uniforme col resto del sistema.

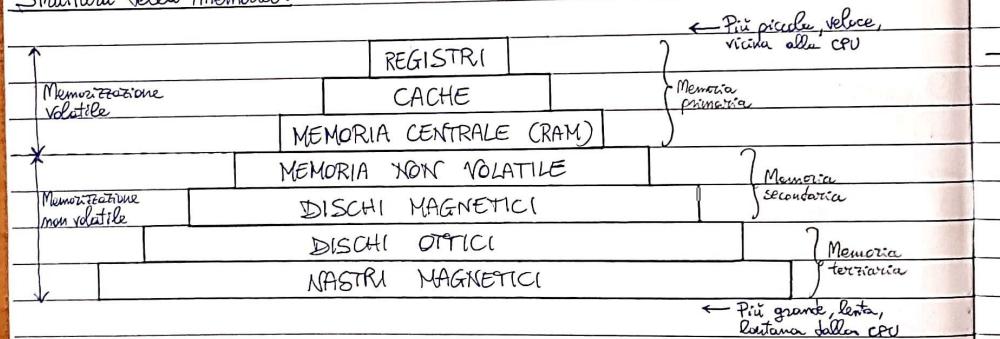
Interruzioni:



GESTIONE DELLE INTERRUZIONI: Windows e UNIX usano una zona di memoria per il vettore delle interruzioni, che contiene i puntatori alle teste di elenchi di gestori di interruzioni.
 L'effettivamente ci sono più gestori di interruzioni che inizia al vettore quando viene sollevata un'interruzione, i gestori dell'elenco corrispondente vengono chiamati a uno a uno, finché non viene trovato quello in grado di servire la richiesta (INTERRUPT CHAINING).

→ IL MECCANISMO DI INTERRUPT IMPLEMENTA ANCHE UN SISTEMA DI LIVELLI DI PRIORITÀ DELLE INTERRUZIONI
 Ciò consente che le interruzioni ad alta priorità siano servite prima di quelle a bassa priorità.

Struttura della memoria:



L'I/O guidato dalle interruzioni è attutto al trasferimento di piccole quantità di dati ma, in caso di trasferimenti massicci, può generare un pesante sovraccarico (OVERHEAD). Per risolvere il problema, si utilizza la tecnica dell'ACCESSO DIRETTO ALLA MEMORIA (DMA): una volta impostati i buffer, i puntatori e i contatori necessari al dispositivo di I/O, il controllore trasferisce un intero blocco di dati dal proprio buffer direttamente nella memoria centrale (o viceversa), senza alcun intervento da parte della CPU. In questo modo, l'azione richiede una sola interruzione per ogni blocco di dati trasferito, piuttosto che per ogni byte. Così, mentre il controllore del dispositivo effettua le operazioni descritte, la CPU rimane libera di occuparsi di altri compiti.

Evoluzione dei sistemi di calcolo:

→ ELABORAZIONE SERIALE

- È un sistema di calcolo riservato per l'esecuzione di un singolo job per volta.
- L'output di ogni esecuzione di job viene fornito tramite la perforazione delle schede.
- Esistono solo il hardware e i programmi applicativi: non c'è nessun sistema operativo.
- Messioni a carico dell'utente:

- Caricamento del compilatore e del programma sorgente nella memoria
- Salvataggio del programma compilato
- Collegamento con moduli predefiniti
- Caricamento e avvio del programma eseguibile

→ SISTEMI OPERATIVI BATCH

In una zona di memoria vengono precaricati dei moduli software che verificano lo stato del dispositivo. Tali moduli costituiscono un MONITOR (che è una prima forma di sistema operativo), il quale effettua il caricamento del programma in memoria e lo avvia. Il programma restituisce il controllo al monitor in caso di terminazione / errore e, in caso di interazione coi dispositivi.

Anche qui si ha un singolo job in esecuzione per volta \Rightarrow sottoutilizzo della CPU

→ SPOLLING (SIMULTANEOUS PERIPHERAL OPERATION ON-LINE)

Da: dispositivi di ingresso, non si va più direttamente in memoria (e viceversa).
Viene aggiunto un intermediario, la MEMORIA DISCO, che è più veloce dei dispositivi:
l'input viene anticipato su disco, l'output viene ritornato da disco \Rightarrow riduzione della percentuale di attesa della CPU grazie alla contemporaneità di input e output di job distanti.

→ SISTEMI BATCH MULTIPROGRAMMATI (MULTITASKING)

Nei momenti di stalli della CPU nell'esecuzione di un'attività, può essere avviata la gestione di altri job. Così aumenta il coefficiente di utilizzazione della CPU e, quindi, l'efficienza. Ciò, però, può comportare un sovraccarico della memoria, per cui non sarebbe più possibile caricare i job più ampi \Rightarrow UTILIZZO DELLA MEMORIA ESTERNA

Tuttavia, il vero problema è che il controllo viene restituito al monitor solo in caso di richiesta verso un dispositivo, terminazione o errore
l'esecuzione di job con frequenti richieste di dispositivo può essere penalizzata
incapacità di gestire applicazioni interattive

Mi
JL
ti

→ SISTEMI OPERATIVI TIME-SHARING

Utilizzano uno schema di condivisione temporale dell'utilizzo della CPU.

È possibile che l'esecuzione di un job venga interrotta intempestivamente dal fatto che esso effettui una richiesta verso un dispositivo (PREEMPTION = pre-rilascio).

Il time-sharing shuttle le INTERRUZIONI, in cui viene aggiornato il Program Counter per iniziare l'esecuzione di un nuovo job ancor prima che venga terminata l'esecuzione del job precedente.

NB: Una TRAP è simile a un INTERRUPT ma si ha quando si verifica un errore.

→ SISTEMI REAL-TIME

Sono sistemi operativi che devono far sì che un programma esegua determinati task entro tempi prestabiliti (DEADLINE). Ne esistono due tipologie:

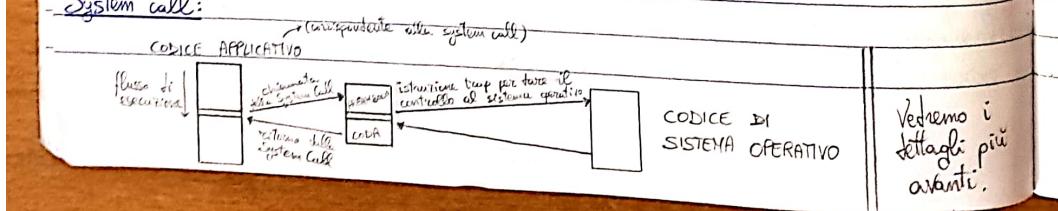
- SOFT REAL-TIME: le deadline dovrebbero essere rispettate; in casi limite se ne può comunque eliminare qualcuna.
- HARD REAL-TIME: le deadline devono assolutamente essere rispettate, altrimenti si potrebbero avere effetti catastrofici. L'hard real-time non si ha mai in sistemi puramente software e non è supportato dai sistemi operativi time-sharing.

Con l'introduzione del time-sharing, i job diventano più precisamente PROCESSI.

Un processo è un programma in esecuzione con associati:

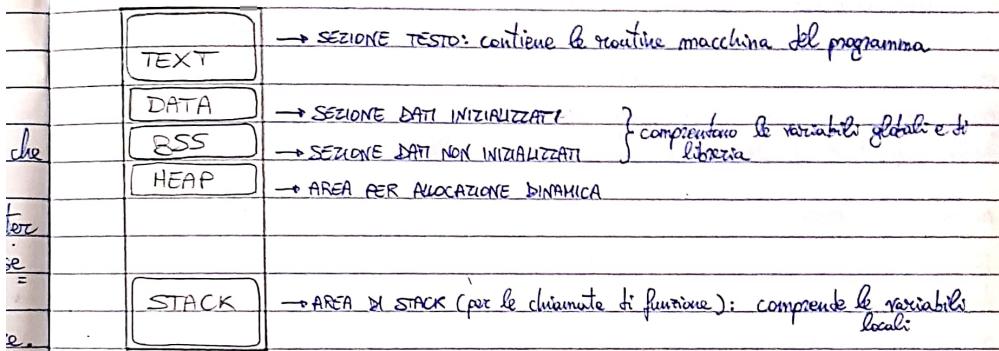
- i dati su cui esso opera;
- un contesto di esecuzione, che costituisce le informazioni necessarie al sistema operativo per schedularne il processo stesso.

System call:



Memoria accessibile ai programmi applicativi:

Il sistema operativo mette a disposizione un contenitore di memoria in cui sono salvate le tipi di informazioni differenti:



Il contenitore di memoria viene anche denominato ADDRESS SPACE (= spazio per gli indirizzi validi per l'applicazione)

può indicare differenti discriminano elementi differenti
indirizzo = offset dall'inizio del contenitore

Esistono una zona di indirizzi all'interno del contenitore (tra HEAP e STACK) che il sistema operativo non dà in uso ma è potentialmente disponibile per un'eventuale espansione dell'heap. Se si tenta di accedere a questa zona, si ha un SEGMENTATION FAULT.

FORMATO DEGLI ESEGIBILI:

→ UNIX: ELF (Executable and Linkable Format); attraverso di esso viene specificato come mandare in setup l'applicazione nel momento in cui il sistema operativo richiede di attivarla.

→ WINDOWS: EXE; all'interno di esso ci sono i dati, il codice macchina da mandare in esecuzione e i metadati che indicano quali sezioni del contenitore di memoria devono essere attive.

Variabili puntatore:

Sono delle variabili che contengono degli indirizzi, ovvero degli offset della locazione di memoria a cui stanno puntando all'interno dell'address space.

In alcuni linguaggi di programmazione (come C), i puntatori a int, a float o a double necessitano dello stesso numero di byte per essere memorizzati. Perciò, è per esempio possibile assegnare il valore di un puntatore a double a un puntatore a int causando solo un warning in compilazione; per inhibire questa possibilità, su gcc basta compilare il programma con la sintesi -Werror.

ARITMETICA DEI PUNTATORI:

```
int *x;
```

```
double *y;
```

```
void function(void) {
```

```
    x = y;
```

→ È ANCORA UN PUNTATORE; IDENTIFICA UN INDIRIZZO CHE, RISPETTO A QUELLO PUNTATO DA x, SI TROVA PIÙ AVANTI DI UN NUMERO DI BYTE pari alla taglia di un int (4 byte)

```
    if (x+1 == y+1) return 1;
```

```
    return 0; }
```

→ IDENTIFICA UN INDIRIZZO CHE, RISPETTO A QUELLO PUNTATO DA y, SI TROVA PIÙ AVANTI DI UN NUMERO DI BYTE pari alla taglia di un double (8 byte)

→ PERCIÒ x+1 ≠ y+1

I puntatori vengono utilizzati per:

- comunicare a una funzione (come scanf()) dove deve operare in memoria;
- accedere alle informazioni presenti nella sezione heap che, in effetti, non contiene dati memorizzati nelle sezioni apposite (data / bss).

DATA UN'ESPRESSIONE INDIRIZZO, GLI OPERATORI PER ACCEDERE AL CONTENUTO EFFETTIVO DELLA CELLA DI MEMORIA CORRISPONDENTE SONO:

.* INDIREZIONE (prefisso all'espressione puntatore)

.[] SPIAZZAMENTO E INDIREZIONE (suffisso all'espressione puntatore)

>[] è parametrico: all'interno delle parentesi si può inserire un codice numerico che indica di quanto ci si deve spostare nella memoria per accedere all'elemento.

Array e puntatori:

Il nome di un array corrisponde a un'espressione indirizzo (= posizione in memoria), come i puntatori. La differenza sta nel fatto che gli array sono RVALUES (right values), ovvero non possono comparire a sinistra di un'assegnazione, perché non sono delle variabili (solo i contenuti degli array rappresentano delle variabili).

ESEMPIO:

```
int *x;  
int y[128];  
  
x=y;  
x=y+10; } ASSEGNAZIONI  
*x = *(y+10); } LECITE  
*(y+10) = *x;  
  
y=x; ~~~~~> ASSEGNAZIONE NON LECITA
```

Richiami su scanf():

È una funzione che vuole come parametro una stringa di formato che identifica un numero generico di informazioni da consegnare al chiamante (per esempio "%d" è una stringa di formato che indica che si deve memorizzare un certo numero di byte corrispondente alla taglia di un intero), e l'indirizzo in cui salvare il valore che si sta acquisendo (per esempio &x).

NB: Se si passa il puntatore a una zona di memoria in cui non si ha spazio sufficiente per un intero (se si ha "%f"), la funzione scanf() genererà un side-effect in memoria in cui si sovraffanno a sovrascrivere informazioni che non si vorrebbero sovrascrivere; in questo caso si ha un overflow.

scanf() ha un valore di ritorno che corrisponde al numero di argomenti, tra quelli chiesti in input, che sono stati effettivamente consegnati in memoria.

ESEMPIO:

Supponiamo di avere scanf("%d %d", p, p1); e vediamo cosa accade quando tale funzione riceve determinati input:

- ciao a tutti → IL VALORE OUTPUT DI scanf() È 0 → si ha un'incompatibilità che causa un segnale
- 10 ciao a tutti → IL VALORE OUTPUT DI scanf() È 1
- abc 10 → IL VALORE OUTPUT DI scanf() È 0
↳ il primo parziale blocca l'intera esecuzione

In caso di mancata acquisizione di `scanf()`, i dati che avranno dovuto essere acquisiti vengono BUFFERIZZATI, cioè mantenuti temporaneamente in specifiche aree di memoria sotto il controllo della libreria `<stdio.h>`.

Più nello specifico avviene questo: il flusso di dati che dovrebbero essere acquisiti da `scanf()` non effettua un passaggio diretto dalla sorgente alla libreria di `scanf()`, bensì viene inizialmente acquisito dal software del sistema operativo tramite un meccanismo di interrupt. Successivamente, la libreria di `scanf()` può richiedere questi dati al software del sistema attraverso specifiche system call, per poi mantenerli bufferizzati in specifiche aree di memoria. Tra l'altro, questi bufferizzati sono esattamente i dati di cui, quando viene invocata successivamente `scanf()`, viene verificato il matching con il tipo di dato che va consegnato.

ESEMPI:

1) ...
 `scanf("%d", p);`
 ...
→ `scanf()` INVOCÀ UNA SYSTEM CALL (chiamata al sistema operativo, il quale deve fornire i dati presi in input da tastiera)

2) ...
 `scanf("%d", p);`
 ...
→ SE L'INPUT È "ciao a tutti", IL DATO NON VIENE ACQUISITO E VIENE BUFFERIZZATO
 `scanf("%d", p);`
 ...
→ VENGONO CONTROLLATI I DATI IN BUFFER PER VEDERE SE POSSONO ESSERE ACQUISITI; IN QUESTO CASO NO E QUINDI RIHANGLIÒ BUFFERIZZATI

Richiami su `printf()`:

È una funzione che agisce in modo duale: quando viene chiamata, mostra all'utente i dati forniti come parametro. Ma essi vanno effettivamente a finire sul dispositivo di output (per esempio il terminale)? Non è detto.

In effetti, inizialmente questi dati vengono acquisiti dalla libreria di `printf()`, collocati all'interno di un'area di memoria e mantenuti lì temporaneamente. Per far sì che i dati vengano effettivamente messi in output, possono essere chiamate altre `printf()` in modo tale che il buffer si saturi e l'ultima `printf()` chiavi una system call per consegnare i dati al software del sistema e far sì che il sistema operativo emette i dati verso il dispositivo. Un'altra possibilità è aggiungere alla stringa da fare a `printf()` i caratteri '\n', che indicano la terminazione di una riga e portano di conseguenza a chiamare la system call per consegnare i dati al software del sistema.

Anche printf() ritorna un valore di output (è possibile scrivere $x = \text{printf}()$).
Tale valore di ritorno è il numero di byte che sono stati aggiunti nel buffer intermedio (ovvero il numero di byte costituenti il messaggio protetto).

È possibile emettere in qualsiasi momento i dati bufferizzati tramite printf() al software del sistema? Sì, grazie alla funzione fflush() (nella libreria <stlib.h>),

che accetta un parametro che può essere uno dei due identificatori globali:

• std::out → porta all'OUTPUT BUFFER → serve a svuotare il buffer di output (svuota a fine linea gli spaziati in caso di '\n')

• std::in → porta all'INPUT BUFFER → in realtà non ha effetto nel buffer di input

Per modificare il comportamento di default di fflush(std::in), si ricorre a una #define:

#define FLUSH

#define flush(std::in) while (getchar() != '\n')

ACQUISIZIONE DI UN CARATTERE PER VOLTA NELLA SECONDA DI CARATTERI DATI IN INPUT
FINCHÉ NON VIENE ELIMINATO TUTTO CIÒ CHE COMPARTE SULLA STESSA LINEA (DIVERÒ FINCHÉ NON SI INCANTA '\n')

In questo modo, chiamando fflush(std::in), viene eliminata un'intera linea dal buffer e si passa ad analizzare la linea successiva.

N.B.: In compilazione deve essere aggiunta la flag -DFLUSH.

(ulteriormente la parola chiave compresa tra #define FLUSH ed #endif non viene considerata)

Strutture (struct):

Sono delle tavole di informazioni. La grande differenza dagli array è che possono contenere collezioni di datieterogenei (di tipo diverso).

• ACCESSO AI CAMPI DELLA STRUCT TRAMITE PUNTATORE: operatore '→' (suffisso all'espresione di indirizzo), che effettua uno spostamento e un accesso alla variabile.

• ACCESSO AI CAMPI DELLA STRUCT TRAMITE NOME DELLA VARIABILE: operatore '.' (punto)

ESEMPIO:

```
typedef struct _table_entry {  
    int x;  
    float y;  
} table_entry;  
table_entry t;  
table_entry *p = &t;
```

→ t.x EQUIVALENTE A p->x EQUIVALENTE A (&t)->x

Type casting:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
char *p;  
int i;  
p = (char *)a + sizeof(a);  
for (i=1; i<=10; i++) {  
    printf("%d", ((int *)p)[-i]);  
}  
?  
ELANK  
→ 'm' 't'  
SONO SEI SEPARATORI DI VALORI  
Se fatti in input a una scanf(), quest'ultima li salta, eliminandoli dal buffer  
di input.
```

Stringhe:

Sono sequenze di caratteri che terminano con il carattere speciale '\0'.
Quando viene chiamata una printf() o una scanf(), il primo parametro è un puntatore a carattere e la qualche parte in memoria sarà presente una stringa che rappresenta il contenuto della stringa di formato fatto a printf() o a scanf(); essa viene appunto riconosciuta tramite l'inizio di memoria del primo carattere.
NB: All'interno di una stringa in memoria possono esserci anche dei \n, dei '\0' o dei '\t'.

ESEMPIO: ciao a tutti:\0
Perciò: ' ' 'm' 't'
→ NELL'INPUT (eg. scanf()) SONO SEPARATORI DI VALORI
IN UNA STRINGA NON GIOCANO ALCUN RUOLO PARTICOLARE

Consideriamo il tipo di dato char *text [3]
text è una variabile di tipo array e ha tre elementi; ciascuno di essi è un puntatore a carattere (char *) e, quindi, può identificare una stringa in memoria.

System call (più nel dettaglio):

È una funzione invocabile a cui si possono passare alcuni parametri. Viene mostrata al codice applicativo come una semplice funzione di libreria ma ha un blocco di istruzioni macchina corrispondente. È strutturata quindi in una maniera machine-dependent in modo da raccontare il codice applicativo con le funzionalità che offre il sistema operativo.

All'interno del codice macchina che caratterizza la system call si trova una particolare istruzione che dà il controllo al sistema operativo. Prima di tale istruzione c'è unsembla che va a scrivere dei valori in alcuni registri di processore (R, R', R''), e questi valori orfano delle relazioni con i parametri passati alla system call. Nel momento in cui si passa il controllo al sistema operativo, e quindi ai moduli esterni all'applicazione, tali moduli osservano il valore dei registri di processore. Dopo aver compiuto il suo lavoro, il software del sistema operativo scrive dei valori di uscita sempre all'interno dei registri del processore e restituisce il controllo alla funzione chiamante. A questo punto la routine macchina può eseguire la coda delle istruzioni della system call, che serve a ripetere il valore dei registri di processore per determinare se il lavoro del sistema operativo è stato svolto correttamente ed eventualmente che quali sono le informazioni (i valori ritornati) che devono essere date al chiamante.

In particolare, quando viene eseguita l'istruzione speciale che passa il controllo al sistema operativo, la CPU cambia stato e viene cambiato anche il valore dello stack pointer il quale si sposta su uno stack differente.

Standard di linguaggio:

Definisce tutti i servizi standard che dovrebbero essere disponibili quando si programmano applicazioni in una data tecnologia (per esempio, se si programma in C, lo standard è ANSI-C). Questi servizi, in termini di software, sono codificati in librerie standard che hanno una specifica di interfaccia e una semantica di esecuzione e che possono chiamare delle system call.

ESEMPIO:

→ UNIX: printf() CHIAMA LA SYSTEM CALL write()
→ WINDOWS: printf() CHIAMA LA SYSTEM CALL WriteFile()

Standard di sistema:

- Definisce tutti i servizi offerti da uno specifico sistema per programmare applicazioni secondo una certa tecnologia software (per esempio C). È fatto da tutte le system call e da altre funzioni che servono a programmare applicazioni su uno specifico sistema o su un altro sistema.
 - Per sistemi UNIX abbiamo lo standard Posix
 - Per sistemi Windows abbiamo le Windows-API (Win-API)

Vediamo tre codici differenti che semplicemente stampano ciò che viene fornito in input tramite una scanf():

1) UTILIZZO DELLA LIBRERIA STANDARD (COMPATIBILE SIA CON UNIX CHE CON WINDOWS)

```
#include <stdio.h>
int main (int x, char** y) {
    char c;
    while (1) {
        scanf ("%c", &c);
        printf ("%c", c);
    }
}
```

2) UTILIZZO DELLO STANDARD DI SISTEMA POSIX (COMPATIBILE CON UNIX)

```
#include <unistd.h>
int main (int x, char** y) {
    char c;
    while (1) {
        read (0, &c, 1);           → canale standard di ingresso = tastiera
        write (1, &c, 1);          → numero di byte coinvolti nell'operazione
                                    di lettura o di scrittura
    }
}
```

area di memoria coinvolta nell'operazione
destinatione standard per l'operazione di output = terminale

3) INTERAZIONE DIRETTA (SCRITTURA DI UN PROGRAMMA MACHINE DEPENDENT)

start: sub \$0x4, %rsp → RISERVO 4 BYTE IN CIMA ALLO STACK PER MANTENERE I CARATTERI PER READ / WRITE
loop: mov \$0x0, %rax → INDICO AL SOFTWARE DEL SO CHE LA SYSTEM CALL DA ESEGUIRE È READ()
ste = mov \$0x0, %rdi → 1° PARAMETRO CHE LA SYSCALL DEVE RICEVERE (CANALE DI INPUT=TASTIERA)
mov \$rsp, %rsi → 2° PARAMETRO CHE LA SYSCALL DEVE RICEVERE (PUNTATORE SU DATI CHE DEVONO ESSERE CONSEGNATI)
mov \$0x1, %rdx → 3° PARAMETRO CHE LA SYSCALL DEVE RICEVERE (NUMERO DI BYTE COMINOLTI)
syscall → IL CONTROLLO VA AL SO → CAMBIO DI STACK / ESECIZ. ISTRUZ. PRIVILEGIATE (CHE NON SI POSSONO SCRIVERE NEI PROGR. APPLICATIVI)
it mov \$0x1, %rax → INDICO AL SOFTWARE DEL SO CHE LA SYSTEM CALL DA ESEGUIRE È WRITE()
mov \$0x1, %rdi → 1° PARAMETRO CHE LA SYSCALL DEVE RICEVERE (CANALE DI OUTPUT=TERMINALE)
syscall → GLI ALTRI PARAMETRI SONO GIÀ A POSTO
jmp loop

Kernel del sistema operativo:

È l'insieme dei moduli software presenti in memoria quando il sistema operativo è attivo; si tratta di moduli software accessibili tramite system call.

Il Kernel include anche tutti i moduli che servono per la gestione degli interrupt e delle trap.

In particolare, si ha una trap quando c'è la necessità che il flusso di esecuzione corrente venga interrotto o varcato e, di conseguenza, il controllo del modulo sia passato al Kernel. Ciò, per esempio, si può verificare in caso di errore.

MicroKernel:

È un Kernel di dimensioni ridotte. Perciò, ha un insieme ristretto di funzioni rispetto al Kernel vero e proprio, e cioè:

→ GESTIONE DELLE INTERRUZIONI

→ GESTIONE BASICA DELLA MEMORIA

→ COMUNICAZIONE (SCAMBIO DI DATI) TRA PROCESSI (in cui il microkernel può fare da intermettore)

→ SCHEDULING DI CPU (= riassegnazione della capacità di lavoro della CPU alle varie applicazioni che sono attive)

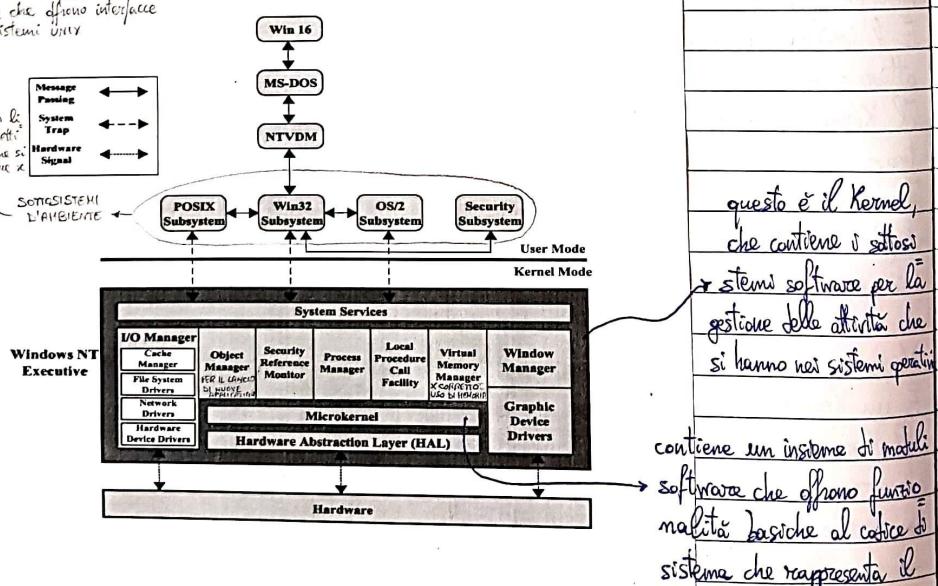
Windows NT:

Nel 1993 ci fu il lancio del primo sistema operativo vero e proprio da parte della Microsoft: Windows NT, che fu il primo a implementare il Kernel e il concetto di modalità d'esecuzione differenziata (User = non privilegiata vs Kernel = privilegiata).

Windows NT è un'architettura MULTITASKING ma non MULTI-USER, per cui è possibile mantenere tante applicazioni attive che, tuttavia, possono essere eseguite solo per conto di un singolo utente. Può inoltre supportare applicazioni sviluppate anche per altri sistemi operativi.

L'architettura Windows è microKernel modificata ed è orientata agli oggetti (è stata sviluppata con il linguaggio C++, che è object-oriented (il C non lo è)).

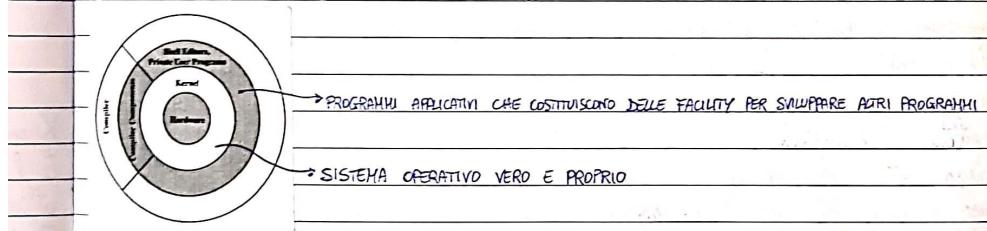
Architettura di NT



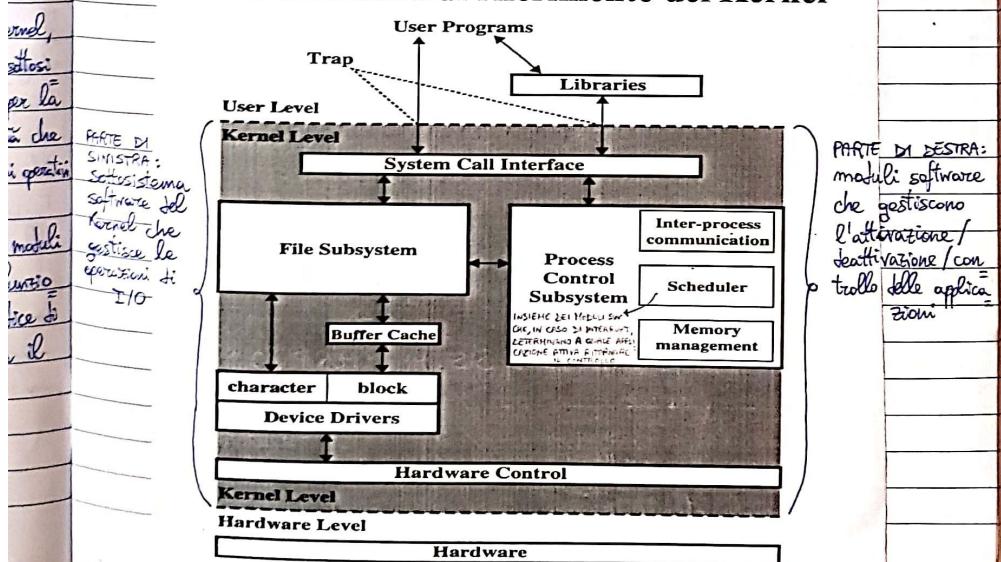
UNIX:

alt: Nel 1969 ci fu il lancio della prima versione di UNIX, che venne scritta interamente in
 ecuzio linguaggio C, appositamente definito (il linguaggio precedente era il B ma non era adatto).
 I sistemi UNIX, a differenza di quelli Windows, sono molto variabili e non seguono una
 man evoluzione lineare nel tempo, per cui può risultare difficile riconoscerli. E per
 i su questo motivo che esiste il comitato POSIX, che stabilisce il set minimo dei servizi che
 ni avranno essere presenti nel Kernel di un sistema operativo affinché esso possa essere riconosciuto come sistema UNIX.

stata Vediamo la struttura dell'architettura di UNIX:



Architettura di riferimento del Kernel



È un'architettura modulare, cioè formata da più sottosistemi fatti per fare cose; un'eventuale interazione tra moduli di uno stesso tipo viene tramite interfaccie pubbliche, bensì, per esempio, chiamando una funzione dell'altro sottosistema.

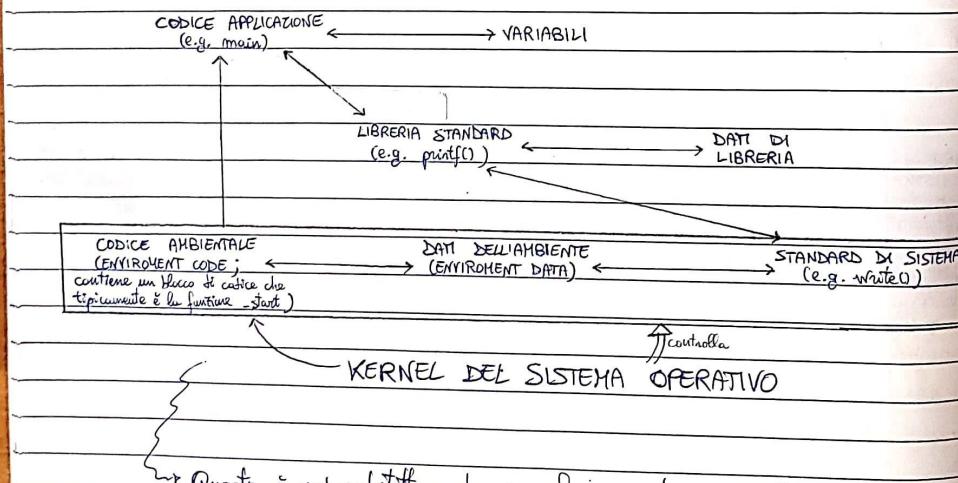
Ambiente di un'applicazione:

È un set di moduli software che fanno parte di librerie e interagiscono con i moduli applicativi.

Quando viene lanciata un'applicazione, il primo a prenderne il controllo non è la funzione main (che è un modulo applicativo), bensì è uno dei moduli ambientali, offre una funzione speciale denominata `start`. Tale funzione e quella da essa invocata hanno il compito di organizzare alcune informazioni in memoria in modo tale che siano facilmente accessibili al main.

L'ambiente, oltre ai moduli software, è costituito anche da dati, che accompagnano le variabili globali e locali che già conosciamo.

Uno degli utilizzi dell'ambiente è raccordare le attività eseguite a livello Kernel per mandare in startup l'applicazione e le informazioni da fornire a essa. Un altro uso è pilotare l'esecuzione di specifiche funzioni di libreria: quando un'applicazione chiama una system call, le operazioni che tale system call può eseguire possono essere funzioni di alcuni dati presenti in memoria, i quali potrebbero essere stati mandati in setup dal Kernel del sistema operativo.



Questa è un'architettura che viene fuori quando si compila secondo
recide standard

È possibile prescindere dall'ambiente, restituendo ex-novo i moduli software che controllano il funzionamento di un programma; per esempio si possono compilare programmi C senza main; nella compilazione bisognerebbe aggiungere la voce -nostartfiles, che è un flag che ridefinisce chi è la funzione start del programma, al posto di quella presente nel codice ambientale. In particolare, in questo caso, la prima istruzione a essere eseguita è la prima all'interno della sezione .text del programma.

Makefile:

Le è un particolare file che supporta dei comandi di shell (per esempio make). make è un programma che legge quel che c'è scritto nel Makefile ed esegue le attività che gli sono state specificate.

ESEMPIO DI MAKEFILE, DOVE moduleA.c, moduleB.c SONO MODULI/FUNZIONI QUALSIASI:

Compile:

introduction
setup
order 1: compile
order 2: compile

gcc -c moduleA.c } COMANDI CHE COSTITUISCONO L'OBBIETTIVO (COMANDI SHELL)
gcc -c moduleB.c } OGNI COMANDO DEVE ESSERE SU UNA NUOVA LINEA E PRECEDUTO DA UN TAB

order 1: compile → COMANDO CHE INDICA CHE PRIMA DEVE ESSERE COMPILATO l'obiettivo "compile"

(ld) moduleA.o moduleB.o

order 2: compile
(ld) moduleB.o moduleA.o → QUESTO COMANDO INDICA CHE NEL PROGRAMMA IL CODICE DI moduleB COMPIRE FRAIA NELLA SEZIONE TESTO RISpetto AL CONCIO DI moduleA, E I DATI DI moduleB COMPIANO FRAIA NELLA SEZIONE DATI RISpetto AI DATI DI moduleA.
→ VICE CHE SERVE A LINKARE TRA LORO QUESTI DUE OGGETTI

→ compile, order 1 e order 2 sono obiettivi; in particolare, quando lancio il comando make, che va ad analizzare il contenuto di questo Makefile, potrei dirgli di eseguire uno solo degli obiettivi specificati.

→ COMANDO retq (ASSEMBLY): prende lo stack pointer e cerca di installare il valore del Program Counter sul registro di puntamento delle istruzioni sul processore. Se però non c'è un chiamante che riporta il controllo, si ha un segmentation fault. In pratica, retq è una RETURN al livello di codice macchina.

→ Analizziamo ora un esempio di codice:

// please compile with -nostartfiles

```
#include <stdio.h>
#include <stdlib.h>
void _start(void) {
    printf("Hello world!\n");
    exit(0);
}
```

FUNZIONE APPARTENENTE ALLO STANDARO DI SISTEMA `<stdio.h>` E ALLA LIBRERIA STANDARD `<stdlib.h>` CHE CHIAMA LA SYSTEM CALL CHE CHIEDE AL KERNEL DI SOSPETARE L'ESECUZIONE DELL'APPPLICAZIONE LA QUALE NON PRENDERÀ PIÙ IL CONTROLLO E NON ESEGUEGGIA TUTTI; SENZA `exit(0)` SI ATREBBE UN SEGMENTATION FAULT PERCHÉ APERTO NON C'È alcuna funzione che abbia chiamato `_start()` E CHE POSSA RIFREDERRE IL CONTROLLO

Funzionalità principali dell'ambiente:

Le informazioni ambientali fungono da raccordo tra il Kernel del sistema operativo e il main per quel che riguarda il passaggio dei parametri al main stesso.

PARAMETRI DELLA FUNZIONE MAIN: `main(int argc, char* argv[])`

* La prima stringa (`argv[0]`) è il nome del programma

NUERO DI LOCATORI DI MEMORIA VALIDE ALL'INTERNO DELL'ARRAY `argv[]`

ARRAY DI PUNTATORI A CARATTORE
(= ARRAY DI STRINGHE)*

Tipicamente questi parametri si utilizzano quando viene lanciata un'applicazione da shell, in cui si specificano le stringhe che l'applicazione stessa dovrà rendere disponibili al modulo main. Il passaggio di queste stringhe all'interno del contenitore di memoria del programma avviene appunto tramite l'ambiente.

PROCESSO A

la cui applicazione contiene una system call che chiede al Kernel del sistema operativo di lanciare un altro programma X con una serie di parametri (in particolare puntatori)

Queste informazioni non vengono definite a tempo di compilazione, bensì a tempo di startup

Il software di `_start` convenzionale riceve dal Kernel l'indirizzo di memoria di specifiche stringhe (presenti fra i dati di partenza) che determineranno i parametri da passare al main (`argc, argv[]`)

dove la possibilità al Kernel di andare all'interno della memoria dell'applicazione attiva nel processo A, di prelevare al suo interno la stringa che il process A aveva depositato in memoria prima di chiudere il sistema, e di riportarla nello spazio di indirizzi interno del programma X (in particolare nella parte bassa del stack)

KERNEL DEL SISTEMA OPERATIVO

SETUP DEI DATI DI PARTENZA DEL CONTENITORE DI MEMORIA DEL PROGRAMMA X

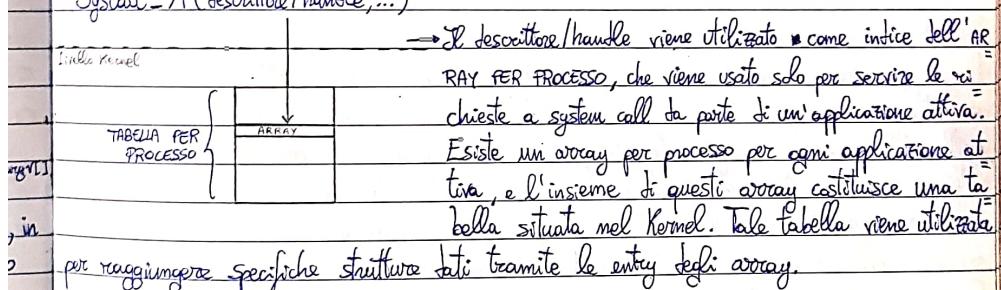
I dati di partenza includono anche i dati ambientali che vengono utilizzati dalle funzioni dello standard di sistema.

Descrittori ed handle:

I puntatori non costituiscono l'unico tipo di parametri di input delle system call: una seconda tipologia consiste in particolari codici numerici che sono i DESCRIPTORI nei sistemi UNIX e gli HANDLES nei sistemi Windows. Essi vengono usati dal Kernel per identificare in modo veloce il posizionamento in memoria di specifiche strutture dati su cui deve essere effettuato il lavoro da parte del Kernel stesso.

Vediamo l'uso dei descrittori/handle all'interno del Kernel, sia Syscall_A() una qualunque system call.

Syscall_A (descrittore/handle, ...)



Oltre alle tabelle per processo, esistono anche le TABELLE GLOBALI, che hanno le medesime caratteristiche ma sono raggiungibili da qualsiasi processo.

Output dal Kernel:

Ci sono due modi per il Kernel fornire output alle applicazioni:

- 1) Side-effect in memoria forniti al passaggio di puntatori come parametri di system call
- 2) Valore di ritorno di una system call

Un output dal Kernel è di fatto una maschera di bit manipolabile dal blocco "tail" di uno stub di accesso al Kernel. La tail è composta da istruzioni macchina che analizzano il valore di ritorno del Kernel scritto in specifici registri di processore (eax nell'x86) e restorano di conseguenza un valore alla funzione chiamante (dopo aver eventualmente aggiornato tali registri).

In realtà, prima di restituire il controllo, la tail va in memoria per adempiere ad

altri compiti. Per esempio, se si è verificato un errore, va a scrivere la motivazione di tale errore, per poi restituire al chiamante un valore speciale (tipicamente -1 in UNIX, 0 in Windows).

→ Su Posix, la motivazione di un errore si trova nella variabile globale errno, che è unica e contiene il codice d'errore dell'ultima system call che ha fallito.

→ Su Windows, continua a esistere una variabile globale contenente un codice di errore, ma stavolta non può essere letta direttamente; per conoscere il codice di errore, deve essere chiamata una specifica funzione dello standard di Windows:

GetLastError() che, appunto, restituisce in output il codice di errore.

Aspetti basici di sicurezza:

Linguaggi come C permettono un "controllo "assoluto" sullo stato della memoria riservata per le applicazioni: tramite i puntatori è possibile muoversi liberamente nell'address space e leggere/aggiornare le informazioni lì presenti.

Alcune funzioni standard accedono alla memoria tramite uno schema di PUNTAMENTO + SPIAZZAMENTO (lo spiazzamento si ha per eseguire delle attività, per esempio la scrittura, su più celle di memoria a partire dall'indirizzo indicato).

Spesso lo spiazzamento non è deterministico \Rightarrow RISCHIO DI BUFFER OVERFLOW (scrivuta in un'area di memoria che non doveva essere modificata).

ESEMPI:

1) scanf() è una delle funzioni più pericolose dal punto di vista della sicurezza perché porta facilmente a corrompere dati in memoria tramite un buffer overflow e, quindi, a corrompere il flusso di esecuzione dell'applicazione.

2) gets() è anche peggiore poiché, anziché una stringa, acquisisce un'intera linea, per cui è ancora più probabile cadere in un buffer overflow.

La causa è la ragione per cui è una funzione deprecata e, se deve essere utilizzata, bisogna farla dichiarare (es. int gets(char *));

Per questo motivo, esistono delle funzioni orientate a prevenire questo tipo di problema. Sulla standard di programmazione C per ambiente Windows c'è una variante di scanf() che si chiama scanf-s() (ste per scanf secure) in cui, per ogni dato che si cerca di far acquisire in memoria, deve essere specificato il numero massimo di byte che devono essere coinvolti nell'acquisizione.

Anche gets() ha una variante più sicura di questo tipo: fgets(), che accetta tre parametri:

• la stringa lata in input

• la dimensione massima della stringa (inclusa la '\0') \rightarrow se eccede, si ha un taglio della stringa

• un puntatore a file / struttura dati che in questo caso acquisisce la stringa data in input \rightarrow la tastiera è identificata con struttura

Esistono altri modi per proteggersi dai buffer overflow: per esempio, `scanf()` mette a disposizione il MEMORY MODIFIER (mutilatore di memoria) "m", prefisso ad tipo di dato da acquisire.

Data una variabile puntata a carattere `p` precedentemente dichiarata, il memory modifier alloca dinamicamente tramite `malloc()` un buffer in base alla taglia dell'input fornito da tastiera, e scrive all'interno di `p` l'indirizzo di memoria della stringa che è stata acquisita.

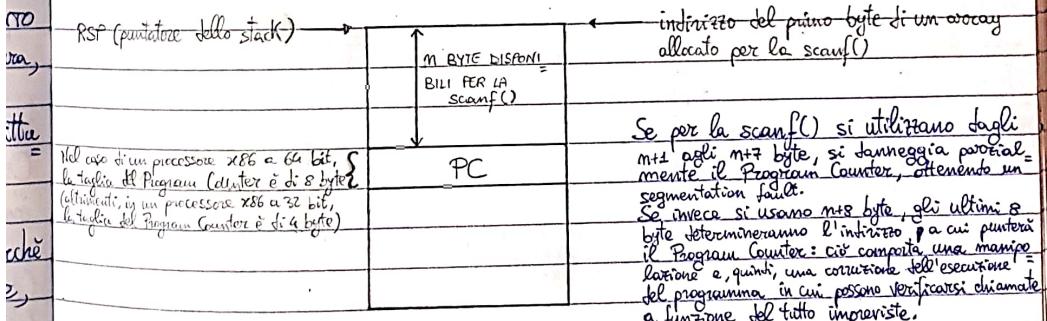
ESEMPIO PRATICO:

```
char *p;  
scanf ("%ms", &p);
```

NB: In tal caso, nel momento in cui l'area di memoria puntata da `p` non serve più, è bene eseguire il comando `free(p)` per evitare poi `p = NULL`.

Alternativamente, si può utilizzare un valore numerico al posto di "m" per specificare il numero di byte da trattare nell'operazione.

APPROFONDIMENTO (STACK OVERFLOW):



Segmentation fault:

È un tipo di errore che può avere cause di varia natura:

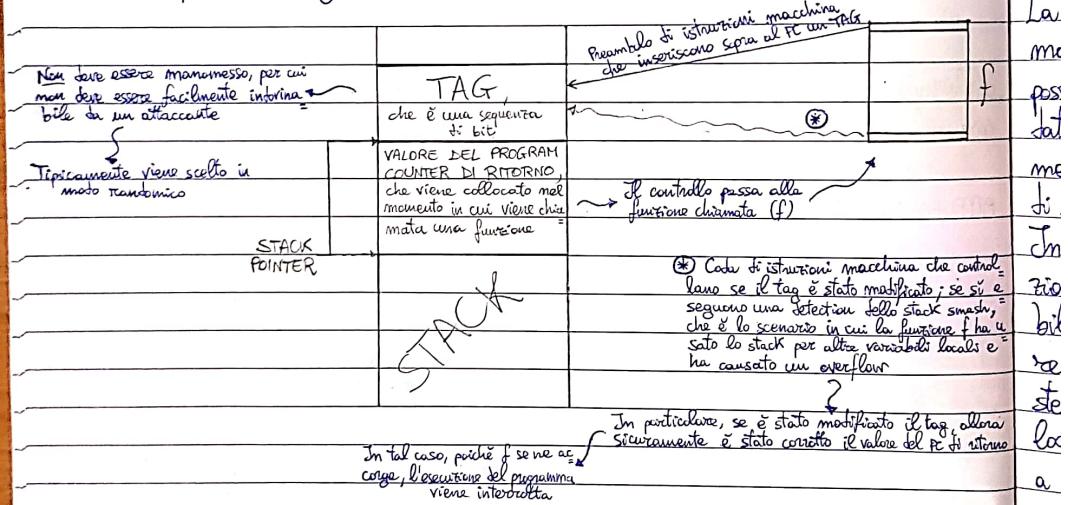
→ Accesso a porzioni di spazio di indirizzamento correntemente non valide (comprese tra l'heap e lo stack).

→ Fetch: si un istruzione puntata dal rip (=register di instruction pointer, che corrisponde al registro di Program Counter) all'interno dello stack: si tratta di un accesso a una tipologia correntemente non permessa per lo stack. È possibile modificare

questa caratteristica digitando `-f execstack` sulla linea di compilazione.
 → Tentativo di sovrascrivere una zona di memoria all'interno della sezione testo, che è configurata per essere read-only.

Stack protector:

Esiste una variante di compilazione dei programmi che si chiama `-fstack-protector` e rende più sicura le applicazioni più soggette allo stack overflow. Infatti, se viene invocata, si presenta il seguente scenario:



ALTRA POSSIBILE CAUSA DEI BUFFER OVERFLOW:

Dato un array `r`, è possibile muoversi al suo interno con una variabile puntatore `p`, che può essere inizializzata con un valore corrispondente a un indirizzo pari a `r+K`, dove `K` è una qualsiasi variabile che rappresenta lo spostamento rispetto al primo elemento dell'array. Se `K` è tale per cui `p` punta al di fuori dell'area di memoria riservata all'array, nel momento in cui si tenta di effettuare un'operazione sull'elemento puntato da `p`, si ha un buffer overflow.

Randomizzazione dell'address space:

È una tecnica che rende la posizione randomica (e quindi infinita) determinate aree di memoria all'interno del programma; fa sì che un attaccante non conosca la posizione in memoria di determinate funzioni.

La randomizzazione dell'address space è nata con una generazione di processori più moderni in cui è stato introdotto l'**INDIRIZZAMENTO RIP-RELATIVE**, il quale costituisce la possibilità di identificare una specifica locazione di memoria della zona text o della zona dati a partire da un'altra locazione di memoria della zona text mediante uno spostamento pari a Δ all'interno dell'address space (che può essere generato da un'istruzione di salto o da un'istruzione che comporta l'accesso nella zona dati).

In realtà, un fenomeno di randomizzazione si poteva avere anche prima dell'introduzione dell'indirizzamento rip-relative: la zona di memoria intrinsecamente randomizzabile è lo stack, poiché tutte le informazioni al suo interno vengono accedute a partire dallo stack pointer e, quindi, secondo una regola di spostamento dello stack pointer stesso. Di fatto, quando viene attivata una funzione, il suo stack frame può essere collocato in un punto arbitrario dello stack, perché dipende dalla sequenza di chiamate a funzioni precedenti.

Per compilare un programma con un address space le cui zone di memoria abbiano un offset randomico:

• UNIX: `gcc -pie, -fPIE` (Position Independent Executable)

• WINDOWS: `cl /DYNAMICBASE`

In particolare, su Linux si può abilitare/disabilitare la randomizzazione, a prescindere dalla presenza delle voci `-pie, -fPIE` in compilazione, utilizzando lo pseudo-file `/proc/sys/Kernell/randomize_va_space` (IL VALORE 0 DISATTIVA LA RANDONIZZAZIONE, MENTRE IL VALORE 2 LA ATTIVA).

PROCESSI

Il concetto di processo è strettamente correlato al concetto di SISTEMA OPERATIVO TIME-SHARING, il quale offre la possibilità di mantenere nella memoria principale più programmi, oltre che al software e alle strutture dati del Kernel. È possibile controllare l'esecuzione dei vari processi secondo delle regole di SCHEDULING (o assegnazione) del processore decise dal Kernel, per cui un'applicazione non può monopolizzare l'utilizzo della CPU.

TRACIA DI ESECUZIONE DI UN'APPLICAZIONE = sequenza di istruzioni che caratterizzano l'esecuzione di quella specifica applicazione all'interno del sistema.

su un sistema operativo time-sharing, l'esecuzione di più tracce avviene secondo uno schema INTERLEAVED (interallacciato): il processore viene "migrato" nell'esecuzione di applicazioni differenti durante l'evoluzione del tempo.

ESEMPIO DI ESECUZIONE INTERLEAVED:

$\alpha+0$ } ESECUZIONE DEL
 $\alpha+1$ } PROCESSO A
 $\alpha+2$
 $\alpha+3$ TIMEOUT → interrupt da timer che dà il controllo al software del Kernel
 $\beta+0$ } ESECUZIONE DEL KERNEL
 $\beta+K$
 $\beta+1$ → il Kernel decide che il processo B deve ora prendere il controllo, installando sulla CPU delle informazioni
 $\beta+2$ che la prima a eseguire l'istruzione macchina iniziale di B
 $\beta+3$ I/O REQUEST → interrupt dovuto a un'istruzione ($\beta+2$) che ha volutamente lasciato il controllo al Kernel
 $\gamma+0$
 $\gamma+1$
 $\gamma+2$
 $\gamma+3$ TIMEOUT
 $\delta+0$
 $\delta+K$
 $\delta+4$
 $\delta+5$ PROCESSO A TERMINATO
 $\delta+x$
 $\delta+y$
 $\beta+2$
 $\beta+3$ PROCESSO B TERMINATO
 $\delta+x$
 $\delta+y$
 $\gamma+4$
 $\gamma+5$ PROCESSO C TERMINATO

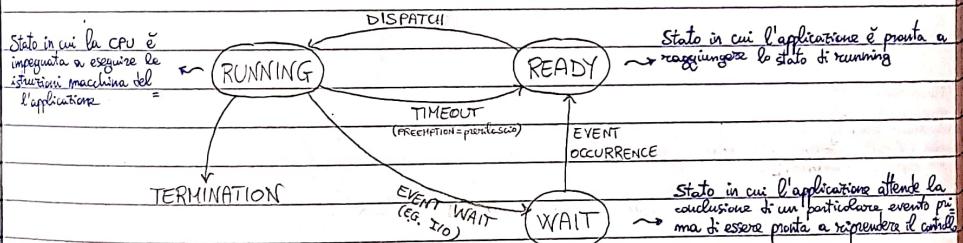


→ Un processo, dopo aver ceduto il controllo a seguito di un interrupt da timer, è pronto a riprendere l'esecuzione.

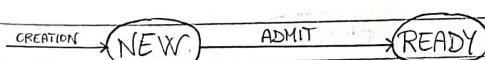
→ Se invece un processo ha volontariamente dato il controllo al Kernel tramite un'istruzione speciale (e.g. una system call), non è detto che sia subito pronto a riprendere l'esecuzione: può darsi che l'istruzione faccia sì che il Kernel attivi un dispositivo che però concluderà la sua attività più avanti.

Di conseguenza, il Kernel ha la necessità di tenere traccia istante per istante di quali processi possono o non possono riprendere il controllo della CPU.

Stati fondamentali dei processi:



Stati aggiornati:



Stato in cui il sistema alloca e inizializza i dati per la gestione dell'esecuzione del processo → questo stato è necessario perché non è detto che il Kernel abbia risorse disponibili per gestire una nuova applicazione attiva (requisito necessario affinché essa possa trovarsi nello stato di ready)

qui non lavora il software dell'applicazione, bensì il software del Kernel



questo è vero finché qualcuno (e.g. qualche processo) viene a richiedere il controllo di terminazione di tali applicazioni → rimane quindi una traccia delle applicazioni terminate, che vengono a questo punto definite **APPLICATION ZOMBIE**

il Kernel continua a mantenere per l'applicazione (che ormai ha terminato l'esecuzione) dei metadati che indicano che l'applicazione è vissuta nel software del sistema e ha terminato con un certo codice di terminazione → MA

Il processore è molto più veloce dei sistemi di I/O \Rightarrow un processo potrebbe rimanere bloccato nello stato di wait per molto tempo \Rightarrow c'è il rischio che moltissime applicazioni rimangano contemporaneamente nello stato di wait \Rightarrow in alcuni istanti potrebbe ad esempio non esserci nessuna applicazione nello stato ready mentre il Kernel rilascia il controllo, con conseguente sottoutilizzo della CPU.

SOLUZIONE: mantenere attivi contemporaneamente più processi possibili. Per aggirare il limite della quantità di memoria a disposizione, si adotta la tecnica dello SWAPPING, che ha lo scopo di eliminare momentaneamente dalla RAM i contenuti di memoria relativi ai processi che si trovano nello stato di wait, e posizionarli in uno spazio esterno (e.g. i hard disk).

Diagramma a stati di processi e swapping:

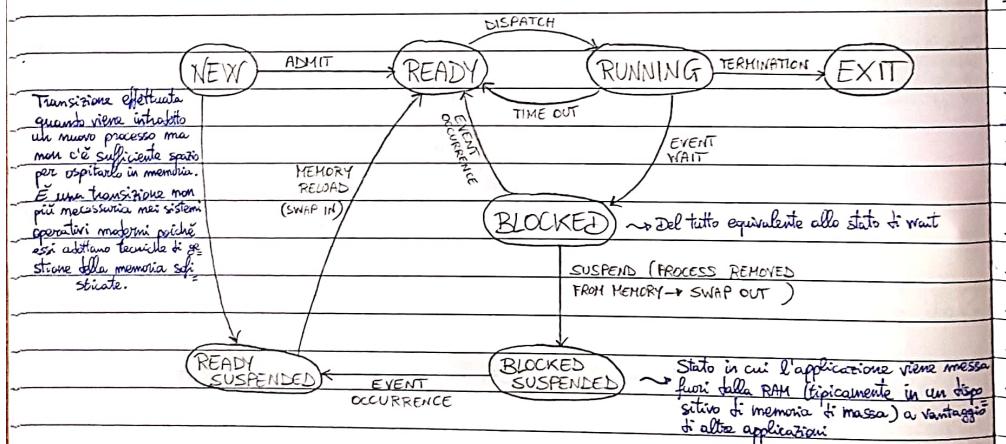


Tabella dei processi:

Il sistema operativo deve mantenere una serie di metadati che portino il software del Kernel a conoscere in ogni istante di tempo:

- LO STATO DELLA MEMORIA, descritto mediante delle tabelle di memoria
- LO STATO DEI DISPOSITIVI
- IL CONTENUTO DEI FILE
- LE INFORMAZIONI RELATIVE AI PROCESSI

Queste ultime sono raccolte in una TABELLA DEI PROCESSI, che ha una entry per ciascun processo che può essere gestito. Inoltre, per ogni processo si hanno delle informazioni specifiche accessibili a partire dalla corrispondente entry della tabella.

N.B.: La tabella dei processi occupa uno spazio fisso (ha delle entry per un numero limitato di processi), per cui su un sistema operativo reale non è possibile mantenere attivo un numero arbitrariamente grande di processi.

ai Tabella di memoria:

sch) Ne esistono di due tipi:

→ CORE MAP: mantiene le informazioni riguardo le aree di memoria libere e occupate della memoria principale.

→ SWAP MAP: mantiene le informazioni riguardo le aree di memoria libere e occupate della memoria secondaria (ovvero dell'area di swap).

Immagine di un processo:

È costituita dalle informazioni che devono essere mantenute affinché un'applicazione sia eseguita e gestita correttamente da un sistema time-sharing, ovvero:

→ IL PROGRAMMA

→ I DATI

→ UNO STACK

→ UNO STACK DI SISTEMA, utilizzato quando vengono eseguite istruzioni in modalità Kernel.

→ UNA COLLEZIONE DI ATTRIBUTI, che fungono da informazioni basiche per la gestione dell'applicazione; tale collezione viene denominata PROCESS CONTROL BLOCK (PCB - blocco di controllo del processo).

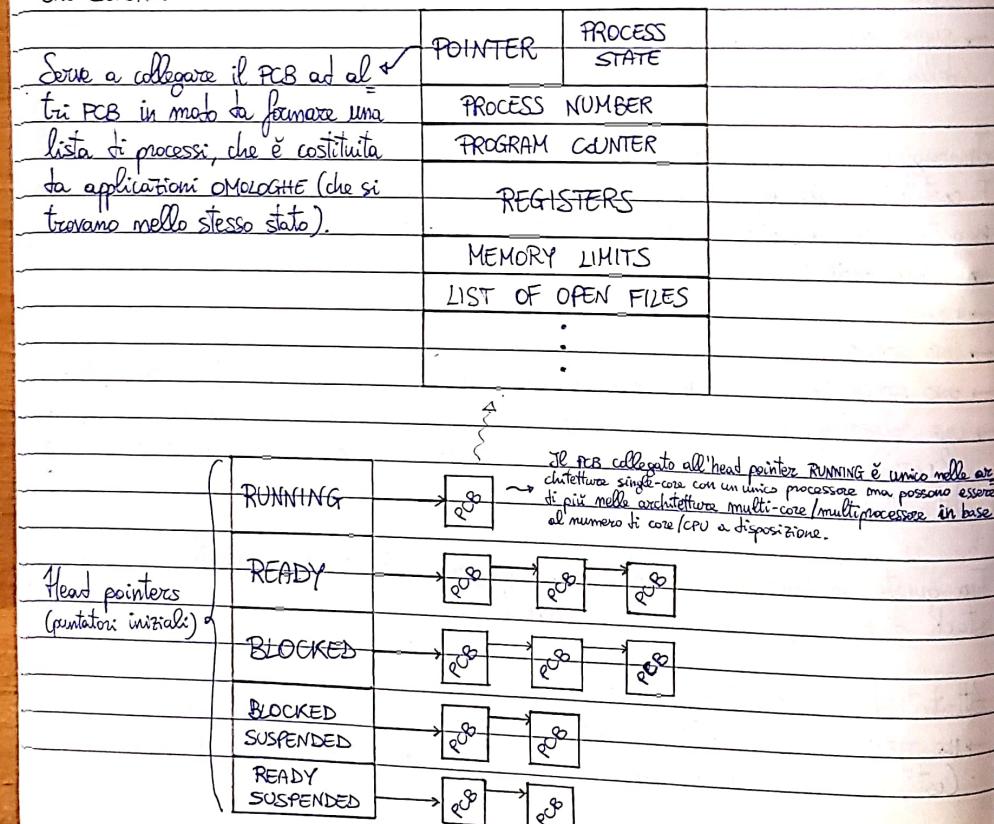
Tra queste, le informazioni che sono sempre mantenute in RAM sono il PCB e lo stack di sistema; il resto è swappabile.

Attributi basici del PCB:

→ IDENTIFICATORI: sono codici numerici che vengono associati alle applicazioni attive. Costituiscono il nome dell'applicazione e i nomi degli eventuali processi figli.

- **STATO DEL PROCESSO:** è la posizione corrente nel diagramma a stati dei processi.
- **PRIVILEGI:** indicano a quali utenti è consentito l'accesso a servizi e risorse.
- **REGISTRI:** insieme delle informazioni che devono essere emesse nei registri di processo
per far sì che un'applicazione possa riprendere correttamente l'esecuzione esattamente dal punto in cui aveva perso il controllo.
- **INFORMAZIONI DI SCHEDULING:** indicano per esempio quanta priorità ha un'applicazione rispetto ad altre applicazioni.
- **INFORMAZIONI DI STATO:** indicano il tipo di evento che deve accadere affinché il processo possa uscire dallo stato di blocco.

UNO SCHEMA:



Questa costruzione è utile per il Kernel: per esempio, quando deve selezionare il processo a cui dare il controllo, può farlo agevolmente accedendo alla lista dei PCB collegati a partire dall'head pointer READY.

• Quale processo deve passare dallo stato ready allo stato running? → PROBLEMA DEL SCHEDULING DELLA CPU (SCHEDULING DI BASSO LIVELLO)

• Quale processo deve passare dallo stato ready suspended allo stato ready? → PROBLEMA DELLA GESTIONE DELLO SWAPPING (SCHEDULING DI ALTO LIVELLO)

È possibile che le applicazioni nello stato di blocco non siano organizzate in un'unica lista: può infatti esserci una distinzione tra i processi in attesa di dati provenienti da uno specifico dispositivo, i processi in attesa di eventi provenienti dal terminale, e così via.

Cambio di modo di esecuzione:

Si ha quindi la modalità con cui il processore sta operando viene modificata e in particolare la CPU ha accesso a un privilegio di livello superiore, passando dalla modalità user alla modalità Kernel (o viceversa). Ciò che è modificato sono le impostazioni del processore, non della singola applicazione!

Sui processori moderni, le informazioni che indicano il grado di privilegio di un processo si chiamano CPL (Current Privilege Level) e vengono rappresentate con dei bit di stato del processore.

Cambio di contesto:

• PRIMA DEL CAMBIO: il processore lavorava su delle attività di una specifica applicazione.

• DOPO IL CAMBIO: il processore lavora su delle attività di un'altra applicazione.

Il cambio di contesto porta a manipolare i PCB: nel PCB del vecchio processo viene salvata la fotografia di CPU da ripristinare quando il processo riprenderà il controllo; poi questo PCB deve essere spostato nella lista adeguata; infine, deve essere aggiornato in maniera analoga anche il PCB del nuovo processo, il cui contesto deve essere riaperto affinché l'esecuzione sia ripresa correttamente.

NB: Non ci sono implicazioni dirette tra cambio di modo e cambio di contesto (sono eventi indipendenti).

ESEMPIO:

È possibile avere uno scenario in cui un processo P è nello stato di running finché non si ha un interrupt o una system call che passa il controllo al Kernel, il quale poi restituirà il controllo allo stesso processo P. In questo caso si sono verificati due cambi di modo ma nessun cambio di contesto.

Cause del cambio di modo di esecuzione:

- ATTIVAZIONE DI UNA FUNZIONE DEL KERNEL
- GESTIONE DI UNA ROUTINE DI INTERRUZIONE

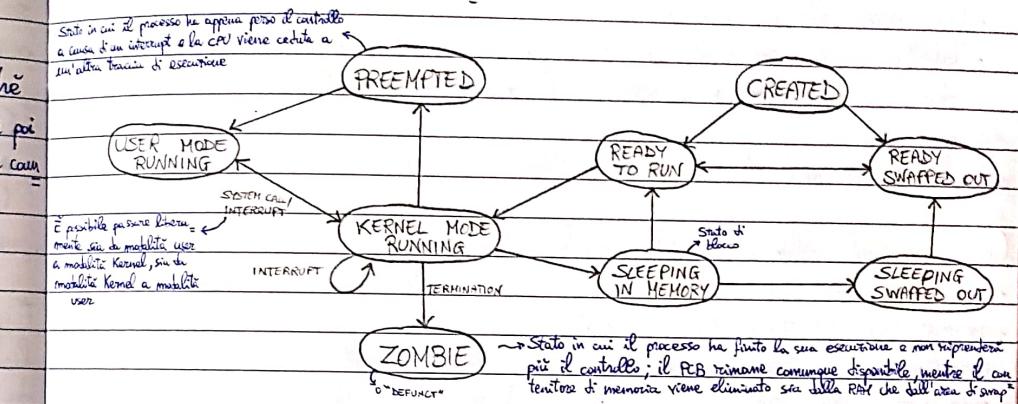
Cause del cambio di contesto:

- INTERRUZIONE DI CLOCK: non c'è più tempo per occuparsi di una specifica applicazione e viene attivato lo scheduler per cedere il controllo a un altro processo.
- INTERRUZIONE DI I/O, in cui in particolare modo si può presentare il seguente scenario:
Si ha un processo A in esecuzione e un processo B nello stato di blocco mentre attende che un dispositivo di I/O termini le sue attività. Quando ciò avviene, il dispositivo lancia un interrupt che assegna il controllo al Kernel e porta il processo B nello stato ready. Dopotutto, il Kernel dovrà scegliere a quale processo restituire il controllo (sono entrambi ready); in particolare, se B ha una priorità più alta, sarà lui a essere affidato alla CPU con conseguente cambio di contesto.
- FAULT DI MEMORIA (nei sistemi a memoria virtuale), in cui si ha un processo nello stato running, ma solo una porzione del suo address space si trova in RAM (l'altra è swapped out); nel momento in cui serve una zona di memoria swapped out, il processo è costretto ad cedere il controllo.

Demoni di sistema:

Sono tracce di esecuzione che vivono esclusivamente all'interno del software del Kernel (in modalità Kernel).

Diagramma a stati di riferimento in UNIX: → SISTEMI UNIX DIVERSI POTREBBERO PRESENTARE ALCUNE DIFFERENZE



Classica immagine di processo in sistemi UNIX:

È divisa in tre parti:

- 1) CONTESTO UTENTE: contiene le informazioni che caratterizzano la parte user dell'applicazione, e cioè:
 - Testo
 - Dati
 - Stack utente
 - Memoria condivisa, in cui più applicazioni condividono una stessa porzione di address space.
- 2) CONTESTO DEI REGISTRIS: contiene le informazioni sullo stato del processore che deve essere ripristinato quando la traccia di esecuzione associata a una specifica applicazione deve riprendere il controllo, e cioè:
 - Program Counter
 - Registro di stato del processore
 - Stack pointer
 - Register generali
- 3) CONTESTO DI SISTEMA: contiene i dati mantenuti a livello di sistema utili a gestire una specifica applicazione, e cioè:

- Entry della tabella dei processi (oltre il PCB)
- User area, che contiene informazioni aggiornate rispetto al PCB.
- Tabella di indirizzamento, che indica quali parti del contenitore di memoria di un'applicazione sono caricate in memoria (e dove risiedono esattamente in memoria) e quali no.
- Stack del modo Kernel

Campi principali dell'entry della tabella dei processi:

- STATO DEL PROCESSO
- IDENTIFICATORI D'UTENTE (reale ed effettivo)
- IDENTIFICATORI DI PROCESSO (pid (=process id), id del parent)
- DESCRITTORE DEGLI EVENTI (valido in stato di sleeping)
- AFFINITÀ DI PROCESSORE: se si hanno più processori, è l'informazione che indica quali sono le CPU che possono effettivamente eseguire l'applicazione.
- PRIORITÀ
- STATO DELLA MEMORIA (swap in / swap out)
- INFORMAZIONI DI TIMING, che possono essere date da cronometri che indicano quanti cicli di macchina un'applicazione ha eseguito in modalità user piuttosto che in modalità Kernel, o anche per quanti cicli di macchina si è trovata nello stato di ready (quest'ultima informazione in particolare può costituire un parametro aggiuntivo per la scelta del prossimo processo che dovrà prendere il controllo della CPU).

Campi principali della User area (U area):

- VALORE DI RITORNO DELLE SYSTEM CALL
- TABELLA DEI DESCRITTORI DI FILE, ovvero una tabella associata a una specifica applicazione che indica quali canali I/O sono validi per lei.
- TERMINALE ASSOCIATO ALL'APPLICAZIONE
- ARRAY PER I GESTORI DI SEGNALI
- PARAMETRI DI I/O (e.g. indirizzi dei buffer)

Avvio tradizionale dei sistemi UNIX: → I PASSAGGI DESCRITTI SONO RELATIVI AI SISTEMI PIÙ ANTICHI E TRADIZIONALI MA, DI FATTO, MOLTO DEI SISTEMI UNIX PIÙ ATTUALI ANCHE IN MOLTI ANALOGI

1) Il Kernel viene caricato in memoria e il controllo viene passato a qualche suo modulo.

Viene dunque attivata una prima traccia d'esecuzione del Kernel, che viene identificata con pid=0 ed effettua il setup delle strutture dati del Kernel in memoria.

2) Viene attivata una prima applicazione, init (più recentemente nota come systemd), che viene identificata con pid=1 ed esegue degli ulteriori step di inizializzazione, che consistono in:

→ leggere alcuni file di configurazione (/etc/inittab ; /etc/ttysab);

→ lanciare altre applicazioni in base alle informazioni presenti in questi file.

In particolare, /etc/ttysab è un file che mantiene la lista dei terminali che devono essere gestiti dall'istanza di sistema UNIX. Si parla di lista di terminali perché, agli albori, i sistemi UNIX erano progettati per supportare più postazioni operative per gli utenti, ciascuna delle quali era costituita da una tastiera e un terminale e, quindi, aveva canali per input/output a sé stanti.

Dunque, nello specifico l'applicazione init lancia un'applicazione getty per ogni terminale da gestire.

3) Lasciata applicazione getty gestisce il suo terminale e lancia a sua volta un'applicazione di login, in cui bisogna inserire le credenziali per accedere al sistema.

4) L'applicazione di login (che è effettivamente un'applicazione di I/O) riceve in input la password inserita dall'utente e la confronta con quella presente all'interno del file delle password (/etc/passwd).

5) Se le credenziali inserite sono corrette, l'applicazione di login ammette l'utente al sistema e fa partire una shell (=terminale, che è anch'esso un'applicazione).

Le shell (le più tipiche sono C, Bash, Bourne e Korn) vengono utilizzate per far avviare altri programmi e per compiere attività.

COMANDI DI SHELL: nome-comando [arg1, arg2, ..., argm]

sono i parametri da passare al main (se esiste)

C creazione di un processo tramite fork():

Se un dato processo (che denominiamo processo padre) ha all'interno della sezione testo la system call fork(), nel momento in cui quest'ultima viene invocata, viene generato

→ così come sono stati appena creati prima di eseguire la fork()

un processo figlio, che sostanzialmente è un clone del padre: all'interno dell'address space, sono identici sia il testo, sia i dati, sia l'heap, sia lo stack. In particolare, il processo padre e il processo figlio, nel momento in cui saranno assegnati alla CPU, riprenderanno il controllo dalla stessa istruzione macchina (quella successiva alla chiamata del Kernel per effetto della fork()).

Consideriamo il blocco di codice macchina che nell'eseguibile rappresenta la routine fork().

Sappiamo che la coda può analizzare il valore dei registri di processore eventualmente aggiornati dal Kernel. È vero che entrambi i processi

riprendono la loro esecuzione dalla prima istru-

zione di questa coda, ma è anche vero che la fotografia di CPU che viene vista nella coda del padre è diversa dalla fotografia di CPU che viene vista nella coda del figlio. In particolare, a essere diverso è il valore del registro %rax, che sarebbe il valore ritorno del Kernel e, di conseguenza, il valore di ritorno della system call: il processo figlio ha 0 come valore di ritorno, mentre il processo padre ha un intero positivo (che corrisponde al pid del figlio).

Può succedere anche che il processo figlio non venga generato. Questo scenario può verificarsi per esempio quando il Kernel non ha più risorse per gestire un nuovo processo. In tal caso la system call fallisce e ha -1 come valore di ritorno.

Differenza tra return 0 ed exit(0) nel main (ma vale per qualsiasi funzione):

• exit(0) → il main ritorna il controllo al Kernel passandogli il codice di terminazione specificato, il quale viene salvato nel PCB. Il processo passa allo stato di zombie, terminando l'esecuzione all'interno del modulo main.

• return 0 → il main ritorna il controllo agli starters (o più in generale al chiamante), i quali invocano una exit() con codice di terminazione pari al valore restituito dal main.

Famiglia di chiamate exec.

L'attivazione di un programma eseguibile generico (non un clone dello stato del pro-



ISTRUZIONE CHE PASSA IL CONTROLLO AL KERNEL PER GENERARE PER IL PROCESSO FIGLIO

Vedi
Sup
a
nel
la:
spa
ma
zare
cerri
In
bisc
(tip
→!
→!
Per
pre
Com
per
fun
ESE
Qu
ma
bie

programma correntemente in esecuzione) avviene tramite la famiglia di chiamate exec, le quali sono tutte specificate nello standard di sistema Posix.

exec
execp
execle
execr
execvp
execve

} FUNZIONI DELLA LIBRERIA STANDARD DI SISTEMA CHE INTERNAMENTE CHIAMANO LA SYSTEM CALL execve

execve → L'UNICA VERA SYSTEM CALL DELLA FAMIGLIA

Vediamo come opera execve:

Supponiamo di avere un processo attivo P con il suo address space e i suoi metadati a livello Kernel, alcuni dei quali sono mantenuti nel PCB. Supponiamo inoltre che nel contenitore di memoria sta girando un programma PROG che, prima o poi, chiama la system call execve, che passa il controllo al Kernel, il quale dismette l'address space relativo a PROG, inizializza un address space diverso relativo a un nuovo programma ma PROG è assegnata a quest'ultimo il PCB e tutti i metadati che prima caratterizzavano PROG. Di conseguenza, execve non genera un nuovo processo ma più semplicemente modifica le informazioni del processo corrente all'interno dell'address space. In particolare, nel momento in cui viene chiamata una funzione della famiglia exec, bisogna passare al Kernel delle informazioni da salvare all'interno dell'address space (tipicamente nella parte bassa dello stack) del nuovo programma, ovvero:

- le stringhe che dovranno essere utilizzate dal main;
- le stringhe che dovranno essere utilizzate dalle funzioni ambientali.

Perciò, su una chiamata di exec, è classico dover specificare non solo il nome del programma che deve partire, ma anche queste altre informazioni appena descritte. Comunque sia, avere a disposizione delle funzioni differentiate nella famiglia di exec permette in taluni casi di risparmiare il numero di informazioni da comunicare alla funzione invocata.

ESEMPIO:

Quando si invoca execl o execvp, è sufficiente specificare il nome del nuovo programma e i parametri del main: le stringhe che dovranno essere usate dalle funzioni ambientali saranno ereditate dal vecchio programma.

PERCHÉ QUANDO UN PROGRAMMA ESEGUE UNA exec /E VARIABILI D'AMBIENTE CHE SI TROVANO SULLA SHELL RESTANO INTATTE?

Quando l'utente digita sulla shell un comando per far partire un programma, la shell stessa, in quanto processo, esegue una fork generando un processo figlio che fa girare il programma. Se quest'ultimo contiene una funzione della famiglia exec, è lo stesso programma figlio a sacrificare se stesso e a sostituire il suo address space, mentre la shell conserva tutte le informazioni iniziali.

APPROFONDIMENTO:

Il comportamento di default della shell è mettersi nello stato di blocco mentre il processo figlio esegue il programma. Se si vuole mantenere il controllo della shell anche quando il child è in esecuzione, basta aggiungere sulla linea di comando per lanciare il programma il simbolo "&", che sta per esecuzione in background; per uscire poi la shell nello stato di blocco, basta digitare sul terminale "fg", che sta per foreground.

DIFFERENZA TRA execl ED execr:

- execl → tra i suoi parametri c'è un vero e proprio elenco di stringhe da passare al main del programma che deve essere attivato; il numero di tali parametri corrisponde al limite superiore della quantità di stringhe che possono essere passate al main.
- execr → ha come secondo parametro un array di puntatori a carattere; anch'esso indica le stringhe da passare al main del programma che deve essere attivato, ma non pone un limite superiore alla loro quantità.

Shell:

Accetta dei comandi in input forniti dall'utente e li analizza per stabilire se si tratta di comandi INTERNI o ESTERNI.

→ COMANDI ESTERNI: portano la shell a eseguire una fork() per far girare un'applicazione la cui struttura dipenderà dal comando (e i suoi eventuali parametri) dati in input.

→ COMANDI INTERNI: sono tali che il software che serve per eseguire le attività associate al comando dato in input è già presente nella shell, la quale invoca internamente

te i moduli per poter eseguire queste attività interne.

ESEMPI DI COMANDI INTERNI:

- aggiornamento o acquisizione dei valori delle informazioni ambientali
- Exit

Comando strtok():

TOKENIZZAZIONE DI UNA STRINGA S = creazione, a partire da s, di una serie di porzioni (=token) tramite la funzione strtok().

PARAMETRI:

- Puntatore a s
- Puntatore alla stringa t contenente i caratteri da tokenizzare, ovvero da sostituire con '\0' (e.g. BLANK)

OUTPUT: puntatore al primo token.

Per ottenere il puntatore al token successivo, bisogna invocare strtok(NULL, t).

In effetti, strtok() (così come printf() e scanf()) è una funzione STATEFUL: quando viene eseguita una chiamata, questa aggiorna il contenuto delle variabili libere (in questo caso specifico i puntatori ai token), il cui valore può essere riasserrato in chiamate successive.

NB: Se si tenta di usare la strtok() su una stringa corrispondente a una variabile globale, si ha un segmentation fault, poiché tale stringa è memorizzata all'interno della porzione read-only della sezione dati dell'address space.

Gestione basica delle variabili d'ambiente da shell:

Vediamo come sono fatti i comandi interni per le due classi di shell tradizionali (bash e tcsh):

• BASH SHELL

→ export NAME=VAL : comando per cui la shell va nel suo stesso ambiente per impostare il valore VAL alla variabile di nome NAME (se la variabile non esisteva allora viene inserita, altrimenti ne viene semplicemente aggiornato il valore).

→ unset NAME : comando che permette alla shell di chiamare l'API per eliminare la variabile di nome NAME dall'ambiente.

→ \$NAME : comando per cui la shell emette in output il valore della variabile di nome NAME.

TCSH SHELL

- setenv NAME VAL : comando equivalente a export NAME=VAL .
- unsetenv NAME : comando equivalente a unset NAME .
- \$NAME : comando perfettamente uguale a \$NAME per la bash shell.

Di fatto, le variabili d'ambiente sono informazioni mantenute a livello applicativo.
A livello Kernel ne vengono riflesse soltanto due: la PWD (directory corrente della mia applicazione) e la root (il punto più alto dell'archivio che può essere osservabile dalla mia applicazione). Deve esserci pertinenza tra le variabili d'ambiente a livello applicativo e quelle a livello Kernel: se per esempio viene cambiato a livello applicativo il valore della PWD, allora la shell si fa carico di aggiornare esplicitamente la sua controparte Kernel mediante la system call chdir().

Oggetti su Windows:

Su Windows, il software del Kernel, oltre alle applicazioni, gestisce ENTITÀ (=OGGETTI), che si distinguono per:

- TIPOLOGIA (e.g. processo)
- ATTRIBUTI: sono informazioni gestionali dell'oggetto, quindi sono analoghi ai metadati.
- SERVIZI: definiscono ciò che può essere richiesto al sistema operativo riguardo la gestione dell'oggetto.

Oggetti di tipo processo - attributi:

- ID DEL PROCESSO
- PRIORITÀ DI BASE PER L'ASSEGNAZIONE ALLA CPU
- AFFINITÀ DI PROCESSORE
- TEMPO DI ESECUZIONE (in particolare il tempo speso nello stato running, il tempo speso nello stato ready, ecc..)
- STATO DI USCITA (è l'equivalente dello stato zombie in UNIX con relativo codice di terminazione del processo)
- DESCRITTORE DELLA SICUREZZA, che specifica quali risorse possono essere accedute o non accedute dall'applicazione (in particolare è utile per evitare che applicazioni attive differenti interfacciano l'una con l'altra).

Oggetti di tipo processo - servizi principali:

→ CREAZIONE DI PROCESSI

→ TERMINAZIONE DI PROCESSI

→ APERTURA DI PROCESSI: è un servizio che dà la possibilità di creare dinamicamente delle relazioni tra oggetti.

Supponiamo di avere due processi P e P' , e che P chiami il Kernel per richiedere l'apertura di P' . Se quest'ultima va a buon fine, quando il Kernel restorna, viene data a P la possibilità di avere un riferimento (quindi un codice) che permette di chiamare ulteriori servizi che andranno a lavorare sullo stato di P' .

In taluni casi, questo avviene per default nel momento in cui il processo P crea il processo P' .

→ RICHIESTA/MODIFICA DELLE INFORMAZIONI DI GESTIONE DEI PROCESSI con cui ESISTE UNA RELAZIONE (per esempio, un processo P può cambiare la priorità di base di un processo P' , a patto che P' sia aperto e abbia una relazione con P).

Handle e tabella degli oggetti:

Il codice numerico che permette a un'applicazione attiva di invocare un servizio del Kernel per operare sullo stato di un altro oggetto è definito HANDLE (= maniglia).

In particolare, quando viene invocato il Kernel per lavorare su un oggetto di cui si possiede l'handle, il Kernel va a verificare se, attraverso questa maniglia, all'interno di una tabella degli oggetti validi per il processo chiamante, c'è o no la possibilità di reperire informazioni che consentono di lavorare su questo specifico oggetto.



Struttura SECURITY_ATTRIBUTES:

Su Windows si utilizzano spesso dei tipi di dato un po' più complessi rispetto a quelli che si vedono su UNIX (e.g. puntatori a particolari strutture dati).

Una struttura molto importante che possiamo utilizzare sfruttando le Win-API è il struct `SECURITY_ATTRIBUTES`, che specifica la regola di gestione di sicurezza che deve essere attuata dal Kernel su un particolare oggetto. Vediamo come è fatta questa struttura:

```
typedef struct _SECURITY_ATTRIBUTES {
```

VARIABLE INTERA CHE SPECIFICA LA DIMENSIONE DI QUESTA STESSA STRUTTURA (È UTILE IN GENERALE PER DISTINGUER VERSIONI DIVERSE DELLA STRUCT, CHE PUÒ SUBIRE MODIFICHE)

PUNTATORE A UNA LOCAZIONE DI MEMORIA IN CUI SONO SPECIFICATE LE REGOLE DI GESTIONE DI SICUREZZA DELL'OGGETTO IN QUESTIONE

BOOL bInheritHandle; → VARIABLE BOLEANA CHE INDICA SE L'HANDLE DELL'OGGETTO IN QUESTIONE PUÒ ESSERE EREDITATO DA ALTRI PROCESSI

```
} SECURITY_ATTRIBUTES;
```

→ `DWORD` è una ridefinizione del tipo di dato `int`.

→ `LPVOID` è una ridefinizione del tipo di dato `void *` e sta a indicare un LONG POINTER TO VOID (puntatore a 64 bit a una locazione void di cui non è specificato il tipo di dato).

Altre strutture dati:

Ulteriori struct che danno altre informazioni su un dato processo sono `PROCESS_INFORMATION` e `STARTUPINFO`.

```
typedef struct _PROCESS_INFORMATION {
```

HANDLE hProcess; → HANDLE AL PROCESSO E

HANDLE hThread; → ALLA TRACCIA DI ESECUZIONE .

DWORD dwProcessId; → CODICI NUMERICI DEL PROCESSO E

DWORD dwThreadId; → DELLA TRACCIA DI ESECUZIONE

```
} PROCESS_INFORMATION;
```

```
typedef struct _STARTUPINFO {
```

DWORD cb; → VARIABLE CHE SPECIFICA LA DIMENSIONE

DI QUESTA STESSA STRUTTURA
(cb STA PER CONTROL BLOCK)

```
...  
} STARTUPINFO
```

Chiaramente, per poter usare tutte queste struct, è necessario includere nell'ambiente di programmazione lo standard di sistema Windows (<windows.h>).

Creazione di un processo:

Su Windows, la creazione di un nuovo processo non avviene tramite una chiamata del API, bensì richiede che il contenitore di memoria dell'applicazione che viene a essere lanciata sia definito ex-maro.

Inoltre, all'atto della creazione, vengono in realtà generate due entità: un processo e un thread. In particolare, il programma con l'address space e alcuni metadati a livello Kernel costituiscono il processo, mentre la traccia delle istruzioni macchina è definita come thread.

ASCII vs UNICODE:

Consideriamo una generica chiamata di CreateProcess(). Il suo primo parametro, che è un puntatore a una stringa di caratteri e identifica il nome di un programma, può essere scritto in due modi differenti:

→ CODIFICA ASCII: è la codifica che siamo abituati a utilizzare e che associa ogni carattere della stringa a un singolo byte in memoria.

→ CODIFICA UNICODE: è la codifica utilizzata a livello Kernel in particolar modo per i nomi di file / programmi, e associa ogni carattere della stringa a una coppia di byte in memoria.

Perciò, nel nostro esempio, nel caso in cui si adotta la codifica ASCII, è necessario l'intervento di un intermediario tra il software a livello applicativo e il software a livello Kernel per convertire la stringa data come parametro a CreateProcess() nella stringa UNICODE corrispondente. Se invece si utilizza la codifica UNICODE, si passa semplicemente il puntatore di questa stringa al Kernel senza necessità di intermediazione.

Di conseguenza, CreateProcess() è un placeholder: non esiste effettivamente all'interno dello standard di sistema Windows la sua reale implementazione, ma esistono le implementazioni di CreateProcessW() e CreateProcessA().

→ CreateProcessW() accetta come primo parametro un puntatore a una stringa UNICODE, per cui, quando prende il controllo, non deve effettuare alcuna conversione.

- CreateProcess() accetta come primo parametro un puntatore a una stringa ASCII e internamente è in grado di raccordare la rappresentazione ASCII a una rappresentazione UNICODE, la quale verrà poi passata al Kernel.
- Lavorare con CreateProcess(), invece, permette di stabilire in modo esplicito se si sta utilizzando la codifica UNICODE o ASCII, dettaglio che verrà specificato in fase di compilazione. Quindi, il tipo di dato del parametro di questa funzione deve essere un puntatore a una stringa di cui non si può dire a priori se sia in codifica UNICODE o ASCII. I caratteri di tale stringa sono identificati col tipo di dato TCHAR. In particolare:
 - Se in fase di compilazione viene specificata la macro UNICODE, il tipo di dato TCHAR diventerà wchar_t (wide char type), che codifica le stringhe in UNICODE.
 - Altrimenti, TCHAR diventerà char, che codifica le stringhe in ASCII.
 In ogni caso, al programmatore è consentito usare direttamente i tipi di dato char e wchar_t, anche in concordanza.
- ESEMPIO DI INIZIALIZZAZIONE DI char E wchar_t:


```
char oneChar = 'x';
wchar_t oneChar = L'x';
```

→ ITEM CON LE STRINGHE
- Vediamo altre facility sulla codifica delle stringhe:
 - Macro TEXT → se applicata a una stringa, permette di salvarla in memoria sia in codifica ASCII che in codifica UNICODE, in modo tale che la sua rappresentazione sia poi reaccordata in tempo di compilazione.
 - ESEMPIO DI UTILIZZO:


```
TCHAR message[] = TEXT ("Ciao a tutti");
```
 - Funzioni wprintf(wchar_t*), wscanf(wchar_t*, ...) → accettano come parametri stringhe in codifica UNICODE.
 - Funzione _tcslen(TCHAR*) → calcola la lunghezza di una stringa di TCHAR.

→ TYPEDEF PER LA GESTIONE DELLE STRINGHE IN WIN-API:

	TYPEDEF	DEFINIZIONE
CHAR		char
PSTR o LPSTR		char*
PCSTR o LPCSTR		const char*
PWSTR o LPWSTR		wchar_t*
PCWSTR o LPCWSTR		const wchar_t*



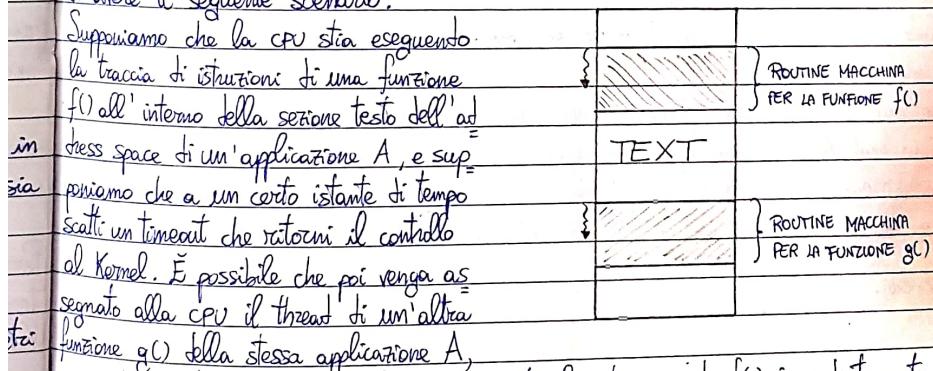
Thread:

Sono percorsi di esecuzione di istruzioni macchina tracciati tramite l'operazione di fetch. Nei sistemi operativi moderni i concetti di spazio di indirizzamento e di traccia di istruzioni sono disaccoppiati e, quindi, non sono più inglobati nella nozione di "processo". In particolare:

- La traccia di istruzioni (thread) è l'unità base per il dispatching: in CPU non vengono più schedulati processi, bensì percorsi di esecuzione.
- Il processo in senso classico è solo l'unità base proprietaria delle risorse (contenitore di memoria, routine macchina usate per comporre percorsi, dati/metadati che indicano per conto di quale utenza l'applicazione è attiva, ecc.).

Di conseguenza, ogni processo può essere strutturato come un insieme di thread: si è passati così ai sistemi operativi basati su processi a sistemi operativi multithreading.

Il fatto che siano proprio i percorsi di esecuzione a essere assegnati al processore permette di avere il seguente scenario:



e quindi non devono essere per forza processate le istruzioni di $f()$ immediatamente successive all'ultima che era stata eseguita, nel caso in cui il controllo non sia restituito a un thread di un'applicazione diversa. In ogni caso, viene comunque salvato per un secondo momento lo snapshot di CPU della prossima istruzione di $f()$ da eseguire. Quello che di fatto si ha è un time-sharing tra più tracce di istruzioni dello stesso programma e non più esclusivamente tra processi differenti.

N.B.: Le tracce di esecuzione di un programma non vengono universalmente definite a tempo di compilazione: dipendono per esempio dall'esito di eventuali istruzioni macchina di salto condizionato presenti nel codice.

Affinché il tutto funzioni correttamente, però, bisogna avere delle accortezze.

Consideriamo due percorsi di esecuzione X, Y concorrenti e relativi alla stessa applicazione, e supponiamo che, in un certo istante di tempo:

1) La traccia X sia in esecuzione.

2) Il Kernel prende il controllo per poi darlo alla traccia Y.

3) Il Kernel prende di nuovo il controllo per poi restituirlo alla traccia X.

Inizialmente la traccia X stava lavorando all'interno del contenitore di memoria utilizzando una certa area di stack: le informazioni che stanno sullo stack per X non devono essere corrette quando X riprende il controllo la volta successiva. È evidente quindi che Y non possa utilizzare la stessa area di stack durante la sua esecuzione.

CONSEGUENZA: quando si lavora in multithreading, l'organizzazione delle informazioni all'interno del contenitore di memoria è leggermente diversa.

In particolare, per ogni thread abbiamo bisogno di:

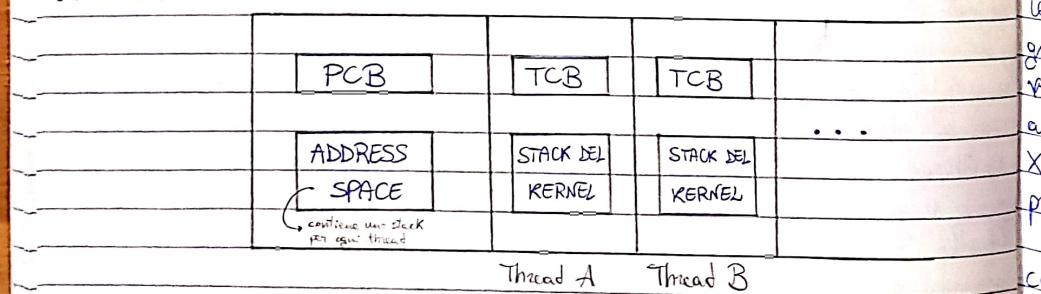
→ Un diagramma a stati che indica in quale stato si trova il thread.

→ Uno stack di esecuzione in modo user all'interno dell'address space, in modo tale che, quando il thread è attivo, non interferisca con gli stack utilizzati da altri thread della stessa applicazione.

→ Uno stack di esecuzione in modo Kernel, che viene utilizzato ogni volta che il thread è tra in modo Kernel durante la sua esecuzione.

→ Un Thread Control Block (TCB) che contiene metadati specifici di gestione del thread (e.g. fotografia di CPU, informazioni di identificazione del thread, ecc.).

UNO SCHEMA:



Variabili TLS:

qui abbiamo visto due tipi di variabili: LOCALI e GLOBALI.

Nel multithreading le variabili locali non danno problemi perché, appunto, sono raccolte in tanti stack quanti sono i thread concorrenti per una specifica applicazione.

Le variabili globali, invece, si trovano nella sezione dati dell'address space che è un'isola unica (e quindi condivisa). Ciò potrebbe comportare dei problemi di sincronizzazione e di consistenza.

ESEMPIO:

Supponiamo di avere una variabile globale V di tipo struct e di avere due thread X , concorrenti e relativi alla stessa applicazione. Le variabili di tipo struct, a causa della loro complessità, per essere aggiornate richiedono molteplici istruzioni macchina. È quindi possibile che, a un certo punto, il thread X , mentre è in esecuzione, perdi il controllo della CPU dopo aver iniziato (ma senza aver completato) un aggiornamento della variabile V : in tal caso, se il Kernel farà poi il controllo alla traccia Y , quest'ultima si ritroverà la variabile V aggiornata in modo parziale, per cui si creerà un'inconsistenza.

Esiste una terza categoria di variabili, ovvero le VARIABILI PER THREAD (o THREAD LOCAL STORAGE - TLS), che vengono viste dalle funzioni di una specifica traccia X come variabili globali, per cui qualunque loro aggiornamento effettuato da una funzione di X può essere osservato da qualunque altra funzione di X . Se però esiste un altro thread che chiama le stesse funzioni che lavorano su variabili TLS, non si ha alcuna interazione. Ciò è dovuto al fatto che esiste un'istanza diversa di variabili TLS per ogni thread concorrente: se un'applicazione ha in sé n tracce, il suo address space è diviso in sezioni diverse per queste variabili. Quando uno dei thread è in esecuzione, va a toccare soltanto la sua istanza del TLS. Ciò non esclude comunque che la traccia X , mediante meccanismi di puntamento e spostamento, possa accedere alla sezione TLS propria di una traccia differente (quindi attenzione!).

COME DICHIARARE UNA VARIABILE TLS:

- POSIX → `thread` prima della dichiarazione della variabile
- WINDOWS → `declspec(thread)` prima della dichiarazione della variabile

* A differenza delle variabili locali, le variabili TLS sopravvivono alla chiusura delle funzioni.

Oggetti: tipi thread su Windows - attributi:

→ ID DEL THREAD

→ CONTESTO DEL THREAD, offerto le informazioni da ripristinare in CPU nel momento in cui la traccia riavrà il controllo

→ PRIORITÀ BASE E PRIORITÀ DINAMICA DEL THREAD

→ AFFINITÀ DI PROCESSORE

→ TEMPO DI ESECUZIONE

→ STATO DI USCITA (NB: la terminazione di un thread non implica la terminazione di un'applicazione soprattutto se quest'ultima è composta da più tracce concorrenti)

Vantaggi

1) Se:

la s

2) Cons

co (

trac

Thread

Di seg

il mu

Oggetti: tipi thread su Windows - servizi principali:

→ CREAZIONE DI THREAD

→ APERTURA DI THREAD

→ RICHIESTA/MODIFICA DI INFORMAZIONI DI THREAD

→ TERMINAZIONE DI THREAD

→ LETTURA DI CONTESTO

→ MODIFICA DI CONTESTO

→ BLOCCO

Non ci sono servizi relativi alla memoria poiché si tratta di servizi di gestione del l'address space, che è un'entità diversa dal thread.

INFORM.

MAPPO

ALLOC

DE

Utilizzo di scanf() su thread concorrenti:

Se più tracce di esecuzione concorrenti chiamano una scanf(), è opportuno fare le seguenti considerazioni:

→ Le varie scanf() lavorano sullo stesso buffer di input che, di fatto, è un buffer della libreria e, quindi, una variabile globale. C'è dunque un problema di consistenza che può viene gio risolto dalla libreria scanf(), la quale al suo interno include dei meccanismi di sincronizzazione.

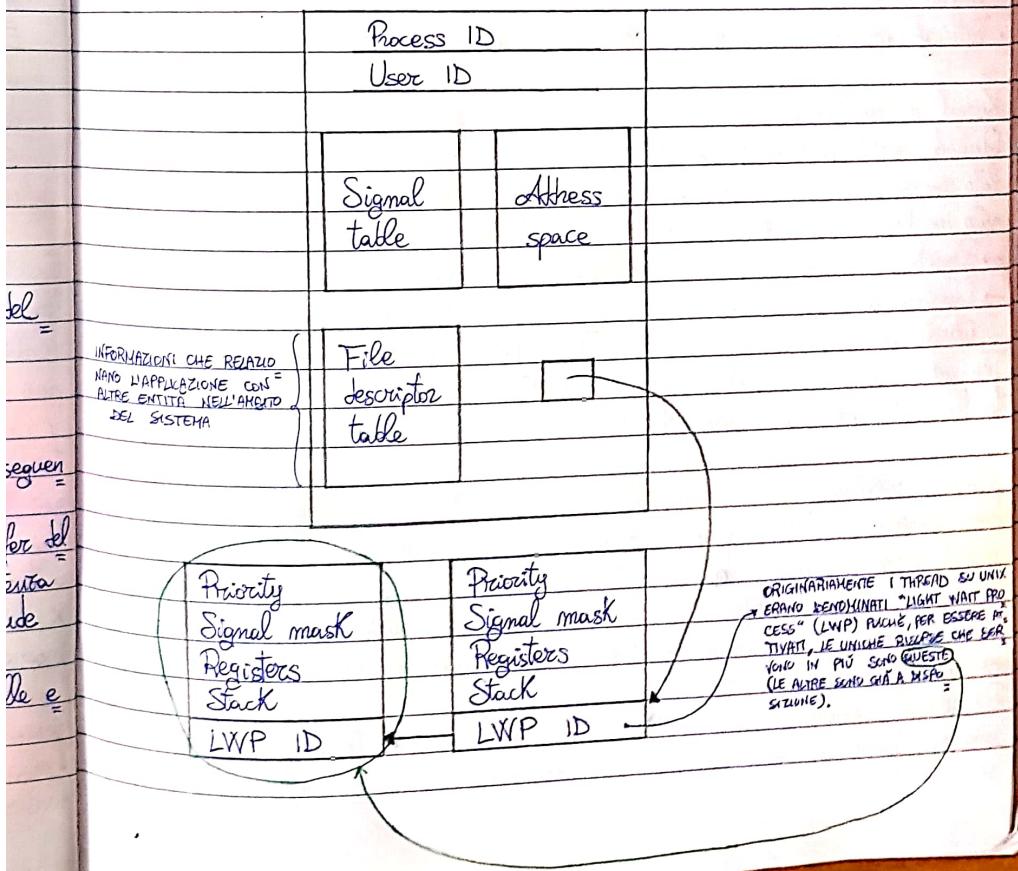
→ Lo stato del buffer di input all'atto dell'esecuzione di una scanf() dipende dalle eventuali chiamate di scanf() precedenti.

Vantaggi del multithreading:

- 1) Se si hanno a disposizione più CPU, è possibile farle lavorare tutti all'interno del in la stessa applicazione, assegnando a ciascuno di esse uno specifico thread.
- 2) Consente di avere applicazioni che, nello stesso istante di tempo, sono nello stato di blocco (per esempio in attesa di dati in input) con un thread, e utilizzano una o più CPU tramite altri thread.

• Thread in sistemi UNIX:

Di seguito viene riportato uno schema per Solaris, che è il primo sistema UNIX ad adottare il multithreading.



CF

Cons:

un'

giù

Jm

→:

→:

→:

→:

→:

→:

→:

→:

→:

→:

→:

Costrutto syscall ():

- La funzione dello standard Posix void exit(int) chiama la system call void exit_group(int) che chiede al Kernel di terminare tutti i thread attivi e, quindi, l'intero processo. Però, è una chiamata che non va fatta se si vuole invocare l'omonima system call void exit(int) che invece chiede al Kernel di terminare solo il thread corrente. Questo fatto potrebbe ingannare il programmatore; per essere sicuri di ciò che viene effettivamente eseguito a livello Kernel su sistemi UNIX, si ricorre a una API universale: syscall(), che accetta come parametri:
- Un codice numerico che corrisponde al numero del servizio del Kernel che deve essere invocato (per esempio, 60 corrisponde proprio alla richiesta di terminazione del solo thread chiamante).
 - Tutti gli argomenti che devono essere passati al servizio del Kernel.

Kernel e multithreading:

Quando inizialmente il Kernel viene caricato in memoria e sul processore viene installato uno snapshot di CPU per cominciare a eseguire le istruzioni del Kernel, viene attivato un thread detto idle-process che lavora esclusivamente a livello Kernel per:

- Permettere al Kernel stesso di inizializzare le sue stesse strutture dati
- Lanciare altri thread che lavorano solo sulle strutture dati e sul codice del Kernel.

Queste particolari tracce sono i cosiddetti DEMONI DI SISTEMA.

Tra questi demoni è molto importante il Kswapd, che si fa carico di spostare i contenitori delle applicazioni dalla RAM all'area di swap e viceversa.

Librerie rientranti:

Sono librerie che possono essere utilizzate in ambito multithreading in modo tale che, quando vengono chiamate dalle loro API da più thread concorrenti della stessa applicazione, non si hanno problemi di consistenza delle informazioni. Il loro unico svantaggio è il costo temporale per la sincronizzazione delle tracce concorrenti.

ESEMPI: printf, scanf, malloc.

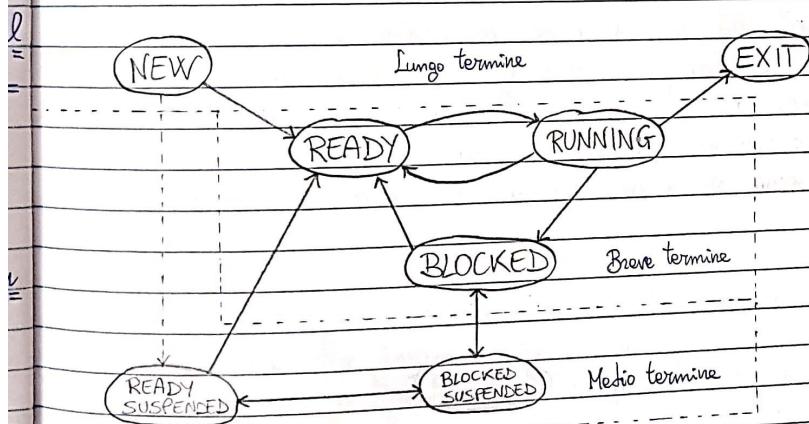
Per alcune funzioni di libreria esistono due varianti: una thread safe (rientrante) e una no; hanno entrambe lo stesso nome ma la versione thread safe ha in più il suffisso -r.

CPU-SCHEDULING

consiste nella pianificazione dell'uso della CPU: poiché i thread attivi all'interno di un'applicazione possono essere davvero tanti, è necessario di volta in volta fare la scelta giusta su quale traccia deve essere assegnata alla CPU.

In realtà, esistono complessivamente quattro tipi di scheduling:

- A LUNGO TERMINE: consiste nella decisione su se aggiungere subito oppure dopo un nuovo processo all'insieme dei processi attivi.
- A MEDIO TERMINE: consiste nelle decisioni da prendere sull'inserimento totale o parziale di un processo attivo all'interno della memoria di lavoro.
- A BREVE TERMINE: è il CPU-scheduling (o dispatching).
- I/O SCHEDULING: consiste nelle decisioni su come sequenzializzare le richieste da servizio verso i dispositivi di I/O.



Scheduling a lungo termine:

Permette al software del sistema di:

- Mantenere uno specifico livello di multiprogrammazione (ovvero un determinato numero di processi attivi) che conviene non eccedere per evitare di impegnare troppo le risorse.
- Avere attivo un numero giusto di processi: I/O BOUND e CPU BOUND.

- **PROCESSO CPU BOUND:** tenta a utilizzare prioritariamente la CPU: è tipico che la relazione spontaneamente ad altri processi. Generalmente perde il controllo quanto scatta il timeout.
- **PROCESSO I/O BOUND:** quando viene assegnato alla CPU, tenta a eseguire poche istruzioni macchina per poi rilasciare spontaneamente il controllo al Kernel per entrare nello stato di blocco in attesa di un evento relativo all'I/O.

Lo scheduling a lungo termine viene messo in funzione per attivare eventualmente un processo che era stato messo in attesa per mancanza di risorse, e ciò è possibile:

- Alla terminazione di un processo
- Su richiesta (per esempio da parte di un timer che periodicamente invia segnali al Kernel)
- Quando la percentuale di utilizzo della CPU scende al di sotto di valori specifici

N.B.: Lo scheduling a lungo termine non può essere usato per le applicazioni interattive, che sono quelle con cui l'utente ha una vera e propria interazione: esse devono essere attivate immediatamente, dato che l'utente non può aspettare per un tempo indeterminato. Di conseguenza, la pianificazione a lungo termine ora è praticamente in disuso: era ragionevole utilizzarla sui sistemi batch multiprogrammati (antecedenti ai sistemi time-sharing), in cui le applicazioni dovevano semplicemente fare calcoli, acquisire dati dai dispositivi o fornire dati in output, senza avere la possibilità di interagire con l'utente.

Scheduling a breve termine - criteri per il dispatching:

- **CRITERI ORIENTATI ALL'UTENTE:** la CPU viene assegnata alle applicazioni attive funzionalmente a come l'utente percepisce il comportamento del sistema.
- **CRITERI ORIENTATI AL SISTEMA:** la CPU viene assegnata alle applicazioni attive in modo tale da ottimizzare il comportamento del sistema nella sua globalità (e.g. utilizzo delle risorse).
- **CRITERI ORIENTATI A METRICHE PRESTAZIONALI:** il sistema misura quantitativamente gli effetti di determinate regole di scheduling e, in base alle misurazioni, ripianifica eventualmente l'utilizzo della CPU.

→ CRITERI ORIENTATI A METRICHE NON PRESTAZIONALI: il sistema sfrutta le caratteristiche di qualche applicazione (non basate su misure precise) per ripianificare eventualmente l'utilizzo della CPU.

CRITERI ORIENTATI ALL'UTENTE E A METRICHE PRESTAZIONALI:

→ TEMPO DI RISPOSTA: tempo necessario affinché il processo inizi a produrre un nuovo output che può essere letto dall'utente interattivo. In altre parole, è il tempo di intervallo tra un output e l'altro.

→ TEMPO DI TURNAROUND: tempo totale che intercorre tra l'istante di creazione e l'istante di completamento dell'applicazione.

→ SCADENZA: deadline per il completamento di specifiche attività all'interno dell'applicazione (viene matchata con un booleano: scadenza rispettata/non rispettata).

CRITERI ORIENTATI ALL'UTENTE E A METRICHE NON PRESTAZIONALI:

→ PREVEDIBILITÀ: possibilità di supporre l'esecuzione dell'applicazione più o meno con la stessa efficienza indipendentemente dal livello di carico del sistema.

CRITERI ORIENTATI AL SISTEMA E A METRICHE PRESTAZIONALI:

→ THROUGHPUT: numero di processi che si riescono a completare nell'unità di tempo.

→ UTILIZZAZIONE DEL PROCESSORE: percentuale di tempo in cui la CPU risulta impegnata.

CRITERI ORIENTATI AL SISTEMA E A METRICHE NON PRESTAZIONALI:

→ FAIRNESS: capacità del sistema di essere equo (ovvero che non sfavoreisce sempre un'applicazione a vantaggio di un'altra nell'utilizzo della CPU, evitando così il fenomeno dello STARVATION).

→ PRIORITÀ: capacità di stabilire quali processi sono più importanti e quali lo sono meno nel momento in cui la CPU deve essere riassegnata (è un criterio molto utile anche per gestire i demoni di sistema).

→ BILANCIAZIONE DELLE RISORSE: capacità di equilibrare l'utilizzo delle risorse al fine di aumentarne lo sfruttamento.

Riassumendo, il sistema operativo deve essere in grado di:

- Discriminare livelli di priorità diversi delle varie applicazioni.
- Eseguire prelascio della CPU per effetto di meccanismi di interrupt.
- Stabilire il livello di priorità da assegnare a un'applicazione nel momento in cui entra nello stato ready (che può dipendere anche dallo stato da cui proviene, che può essere di blocco o running).

Analizziamo ora l'evoluzione storica del problema dello scheduling di CPU tenendo conto che è nato già nell'ambito delle macchine uniprocessore.

Scheduling FCFS (First Come First Served):

Vengono eseguiti prima i processi che entrano nello stato ready antecedentemente rispetto agli altri.

Non vi è prelascio: ogni processo rimane in esecuzione finché non termina oppure finché non viene chiamato il software del sistema per invocare un servizio bloccante.

Svantaggi:

- Non viene minimizzato il tempo di attesa (= quantità di tempo che l'applicazione spende nello stato ready) e, di conseguenza, neanche il tempo di turnaround, soprattutto nel caso in cui le applicazioni che prendono per prime il controllo della CPU sono le più lente.
- I processi I/O bound vengono sfavillati a causa di qualche applicazione CPU bound che monopolizza il processore, per cui si potrebbe avere un sottoutilizzo dei dispositivi di I/O.

Scheduling Round-Robin (o a carosello o time-slicing):

I processi nello stato ready vengono mandati in esecuzione a turno per un quanto di tempo Δ specifico dopo il quale avviene un prelascio (onde evitare la monopolizzazione del processore da parte di un'applicazione).

È possibile che una frazione S del quanto non venga sfruttata (per esempio a causa dell'invocazione di un servizio bloccante).

Nel momento in cui un processo entra nello stato ready, viene messo in coda e sarà

l'ultimo a essere riassegnato poi alla CPU.

Svantaggi:

- I processi I/O bound vengono sfavoriti rispetto ai processi CPU bound: spesso e volentieri non sfruttano l'intero quanto di tempo a disposizione a causa di chiamate a servizi bloccanti.
- Di conseguenza vengono sfavorite le applicazioni interattive (che sono un sottoinsieme delle applicazioni I/O bound), per cui si potrebbe avere uno sottoutilizzo dei dispositivi di I/O.
- La scelta del time-slice (= quanto di tempo) è critica:
 - Se il quanto di tempo è troppo elevato, a maggior ragione le applicazioni I/O bound sfruttano una frazione piccola del tempo totale a disposizione e, inoltre, una volta che entrano nello stato ready, devono attendere a lungo prima di poter riprendere il controllo della CPU.
 - Se il quanto di tempo è troppo piccolo, le applicazioni I/O bound rischiano di non riuscire a eseguire le poche istruzioni macchina che prececano la chiamata a un servizio bloccante prima del timeout. In tal caso, possono utilizzare un servizio di I/O soltanto al turno successivo.

le Scheduling Round-Robin virtuale:

Si hanno due livelli di priorità (bassa, alta).
Se un'applicazione è viene prelazionata (ovvero ha utilizzato tutto il suo quanto di tempo) oppure viene ammessa per la prima volta, allora le viene assegnata la priorità bassa.
Se un'applicazione è BACK FROM I/O (ovvero entra nello stato ready a partire dallo stato di blocco; il concetto di "back from I/O" viene esteso successivamente anche alla presenza dello stato ready suspended), allora le viene assegnata la priorità alta.

CRITERIO PER L'ASSEGNAZIONE DELLA CPU:

Se la coda ad alta priorità non è vuota, il prossimo processo da schedulare sarà il più in coda di questa coda. Altrimenti, il controllo verrà dato al primo processo della coda a bassa priorità.

TEMPO DI ASSEGNAZIONE DELLA CPU:

→ APPLICAZIONI CON PRIORITÀ BASSA: quanto di tempo Δ di time-slice

→ APPLICAZIONI CON PRIORITÀ ALTA: $\Delta-X$, dove X è la porzione effettivamente consumata del quanto precedente \Rightarrow alle applicazioni back from I/O viene data la possibilità solo di recuperare il tempo che non è stato sfruttato nel turno precedente.

Schedulazione
Tra i processi
(= rapporto
RR = $\frac{w_i}{w_j}$)

Scheduling SPN (Shortest Process Next):

Tra i processi nello stato ready, viene scelto quello col CPU BURST (= quantità di tempo in cui l'applicazione vuole utilizzare la CPU in maniera continuativa prima della terminazione o dell'invocazione di un servizio bloccante) più basso.

Neanche questo scheduler è adatto per le applicazioni interattive: anche se hanno un CPU burst basso (perché tendono ad avere per poco tempo il controllo della CPU), se entrano nello stato ready quando un altro processo (magari con un CPU burst molto alto) è nel bel mezzo dell'esecuzione, devono attendere che tale esecuzione termini.

Per questo motivo esiste una variante di tale algoritmo di scheduling: SRTN (Shortest Remaining Time Next), in cui l'assegnazione della CPU viene effettuata ogni volta che l'insieme dei processi ready cambia. In tale occasione non vengono valutati più i CPU burst, bensì i remaining time (= porzioni dei CPU burst che non sono state ancora processate) delle applicazioni nello stato ready.

VANTAGGI:

- Viene minimizzato il tempo di attesa e, di conseguenza, il tempo di turnaround.

- I processi interattivi non vengono sfavillati nel caso in cui c'è prelazione (vero per lo scheduling SRTN), per cui, in generale, non si ha un sottoutilizzo dei dispositivi di I/O.

SVANTAGGI:

- È necessario avere dei meccanismi di gestione della lunghezza dei CPU burst (o dei remaining time).

- Si può verificare una STARVATION a discapito dei processi con CPU burst più elevati.

STIMA DEI CPU BURST:

→ MEDIA ARITMETICA: $S_{m+1} = \frac{1}{n} \sum_{i=1}^m S_i$

→ MEDIA ESPONZIALE: $S_{m+1} = \alpha T_m + (1-\alpha) S_m$

S_i = media dei CPU burst all'i-esimo turno

T_i = CPU burst effettivamente osservato all'i-esimo turno

α = parametro da scegliere con attenzione ($0 < \alpha < 1$)

VANTAGGI:
• I processi interattivi non si sfavillano.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

VANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

SVANTAGGI:
• È meno efficiente rispetto alle varie varianti di SJF.

Scheduling HRRN (Highest Response Ratio Next):

Tra i processi nello stato ready, viene selezionato quello col RESPONSE RATIO RR (= rapporto di risposta) maggiore.

$$RR = \frac{W+S}{S}$$

W = tempo speso nello stato ready (tempo di attesa)

S = tempo speso nell'utilizzo della CPU (tempo di servizio)

VANTAGGI:

- I processi I/O bound (caratterizzati da valori di S piccoli) non vengono sfavortiti, per cui non si ha un sottoutilizzo dei dispositivi di I/O.

Svantaggi:

- È necessario effettuare misurazioni precise e aggiornamenti periodici dei valori RR delle varie applicazioni, il che implica un costo non indifferente.

Scheduling Multi-Level Feedback-Queue:

Vengono utilizzati n livelli di priorità diversi (da 0 a n-1), che vengono gestiti nel seguente modo: se la coda a priorità 0 (la più alta) non è vuota, il prossimo processo a schedulare sarà il primo di questa coda. Altrimenti si guardano nell'ordine le code a priorità 1, a priorità 2, e così via.

I processi partono tutti dalla coda a priorità 0. Se un'applicazione utilizza tutto il quanto di tempo, la sua priorità viene declassata di un livello, altrimenti viene confermata. In questo modo, le applicazioni I/O bound non vengono sfavortite, mentre quelle I/O bound vengono sempre declassate senza possibilità di recupero (FEEDBACK NEGATIVO).

Per ovviare parzialmente a questo problema di STARVATION, si fa in modo che i quanti di tempo non siano uguali per tutte le applicazioni, bensì obbediscano alla seguente legge:

$$\text{QUANTO DI TEMPO} = 2 \cdot \Delta$$

i = priorità

Δ = quanto di tempo di un processo a priorità 0

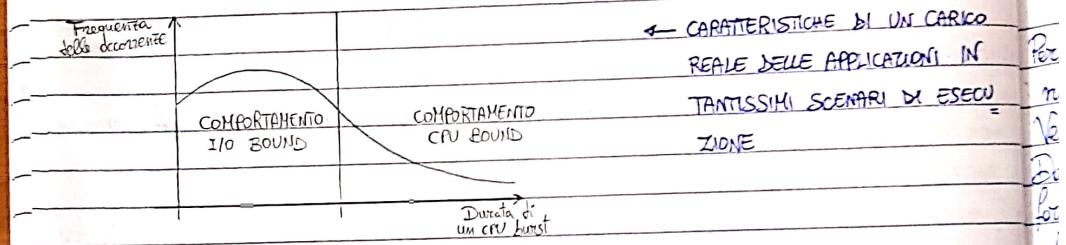
VANTAGGI:

- Non c'è necessità né di predire né di monitorare informazioni particolari.

Svantaggi:

- Il feedback dei processi è solo negativo, per cui la loro priorità può solo essere declassata.

sata. In questo modo, vengono svantaggiate le applicazioni che possono essere alternativamente CPU bound e I/O bound (esempio tipico: I/O bound \rightarrow CPU bound \rightarrow I/O bound).



→ Come già accennato, possono esserci applicazioni che, col tempo, passano da essere CPU bound a essere I/O bound e viceversa: bisogna essere in grado di rispondere correttamente a queste variazioni di comportamento per evitare di sfavorire applicazioni che, sfortunatamente, durante una specifica fase di esecuzione, hanno dimostrato un comportamento CPU bound.

Scheduling UNIX tradizionale (SVR3 - 4.3 BSD): → Nelle vecchie sistemi UNIX per scegliere processi mai più utilizzati per schierarli fuori

Vengono ancora utilizzate code multiple di priorità con feedback che porta alla variazione di priorità in base al comportamento che dimostra un processo: stavolta, però, il feedback può essere sia positivo che negativo.

Lo schema di esecuzione dei processi segue la logica Round-Robin, in cui il quanto di tempo dipende dalla priorità.

Uno dei criteri per stabilire il livello di priorità di un'applicazione che entra nello stato ready è lo stato da cui essa proviene (running, blocked o swapped out).

I livelli di priorità sono 40: vanno da -20 (priorità più alta) a +19 (priorità più bassa).

ELENCO DELLE APPLICAZIONI IN ORDINE DI PRIORITÀ DECREScente:

→ Provenienti dallo stato ready swapped out (priorità -20)

→ Provenienti dallo stato blocked (priorità -19)

→ Provenienti dallo stato running (priorità di riferimento per l'applicazione, che è variabile)

c La priorità di riifornimento viene calcolata con la seguente formula:

$$P = \text{base} + \frac{\text{CPU}}{2} + \text{nice} \rightarrow \text{VALORE COMPRESO TRA -20 E -19 CHE PUÒ ESSERE IMPOSTATO}$$

FATTO INIZIALMENTE A TE TEMPO DI UTILIZZO DELLA CPU NORMALIZZATO: VIENE CALCOLATO FERIODICAMENTE (MAGARI OGNI 100 TURNI)

Per modificare la niceness del processo corrente, si può digitare sulla shell il comando nice -n [+number] [command] (PER IMPORTE LA NICENESS DI UN ALTRO PROCESSO: nice -m [+number] -p [pid])

Vediamo perché questa istruzione funziona:

Dato che si tratta di un comando esterno, quando viene lanciato, la shell esegue una fork() nel processo figlio, prima che esso chiami una exec, viene invocata la system call nice() per impostare il valore della niceness in funzione di quanto viene specificato sulla command line. Perciò, la niceness rimane settata sia prima che dopo la exec.

a Auto-grouping:

È una facility di LINUX che serve a rendere più agevoli le attività di gestione delle priorità (e quindi delle niceness) delle applicazioni attive da parte di un amministratore di sistema. Classifica tutti i processi lanciati da uno stesso terminale (e i loro thread) in un unico gruppo. Se un processo cambia la sua niceness, la modifica è osservabile solo dagli altri processi dello stesso gruppo: questo serve a garantire che più gruppi diversi vengano trattati egualmente nell'assegnazione della CPU. La niceness diventa quindi un concetto non globale, bensì locale.

Per escludere l'auto-grouping, basta impostare il valore 0 nel file di configurazione del sistema /proc/sys/Kernel/sched-autogroup-enabled.

s) Lo scheduling UNIX tradizionale è ottimo per pianificare l'utilizzo di un'unica CPU da parte di processi. Con l'evoluzione dei computer, tuttavia, si è iniziato a schedulare thread anziché processi e a disporre più di un processo. Di conseguenza, sono nate delle nuove esigenze per quanto riguarda i criteri di assegnazione delle CPU, e lo scheduling UNIX tradizionale non ha conservato del tutto la sua efficacia.

Sistemi multiprocessore:

CARATTERISTICHE ARCHITETTURALI:

Si hanno unità di calcolo (CPU / CPU-core) multiple che condividono una memoria principale comune: è dunque un sistema FORTEMENTE ACCOPPIATO (tightly coupled).

PROBLEMATICA 1: QUANTE E QUALI CPU CONVIENE ASSEGNIARE A UN DETERMINATO PROCESSO?

E QUALE ALGORITMO SPECIFICO SI DEVE USARE PER L'ASSEGNAZIONE DI UNA SPECIFICA CPU A UN PROCESSO?

Ci sono principalmente due politiche di assegnazione di processi/thread alle unità di calcolo:

→ **POLICY STATICÀ:** l'assegnazione di un'applicazione alle CPU che potranno processarla avviene una volta sola, per cui l'overhead è ridotto. D'altra parte, si rischia che i processori vengano sottoutilizzati a causa di uno sbilanciamento nelle loro assegnazioni.

→ **POLICY DINAMICA:** l'overhead è superiore a causa delle riassegnazioni multiple delle applicazioni alle CPU, che richiedono l'esecuzione di algoritmi decisionali. Infatti, viene garantito un'imbilanciamento nelle assegnazioni delle varie CPU quando, per esempio, inizialmente qualcuna di esse processa solo applicazioni I/O bound e qualcun'altra processa solo applicazioni CPU bound.

PROBLEMATICA 2: QUALI CPU POSSONO REALMENTE ESEGUIRE SOFTWARE IN MODO KERNEL?

Nel momento in cui si hanno più processori, la zona di memoria contenente le informazioni a livello Kernel (e.g. PCB e metadati vari) è condivisa dalle varie CPU, per cui potrebbe verificarsi un'inconsistenza nel momento in cui i processori entrano in modo Kernel contemporaneamente. Per affrontare questa problematica è possibile avere due approcci differenti:

→ **APPROCCIO MASTER/SLAVE:** il sistema operativo (in particolare il Kernel) viene eseguito solo su una specifica CPU. Quando su un'altra CPU viene richiesto un servizio in modo Kernel, tramite un meccanismo di INTER PROCESSOR INTERRUPT viene avvertito il processore che effettivamente può eseguirlo.

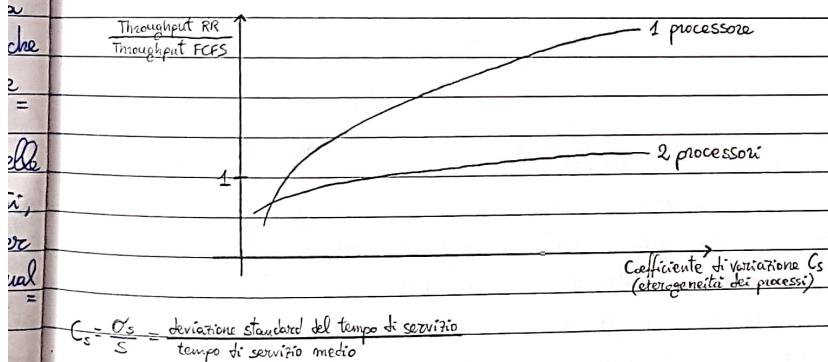
→ **APPROCCIO PEER (CONVENTIONALMENTE USATO NEI SISTEMI MODERNI):** il Kernel viene eseguito

unto su tutte le unità di calcolo, per cui, all'interno del software del Kernel, c'è la necessità di gestire in maniera esplicita la coerenza dei dati: per esempio tramite gli stessi algoritmi di scheduling.

PROBLEMATICA 3: SI DEVONO UTILIZZARE LE POLITICHE SOFISTICATE DI MULTIPROGRAMMAZIONE ADOTTATE PER I PROCESSORI SINGOLI?

Quanto sono disponibili molte CPU/CPU-core che lavorano su processi, il loro livello di utilizzo non è più un fattore così critico, dato che il costo di un processo è proporzionalmente ridotto rispetto a quello dell'intera architettura.

CONFRONTO TRA SCHEDULING ROUND-ROBIN E SCHEDULING FCFS SUI PROCESSI:

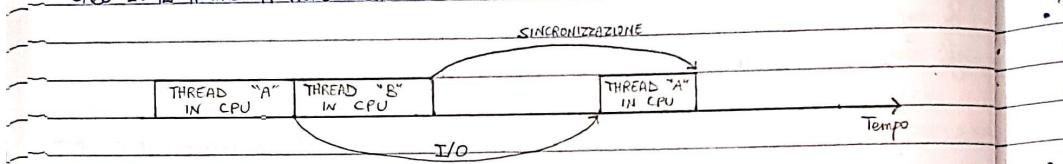


Tuttavia, lo scheduling sui thread è più complicato perché, in tal caso, è più probabile che le attività siano correlate tra loro e, quindi, abbiamo dei punti di sincronizzazione.

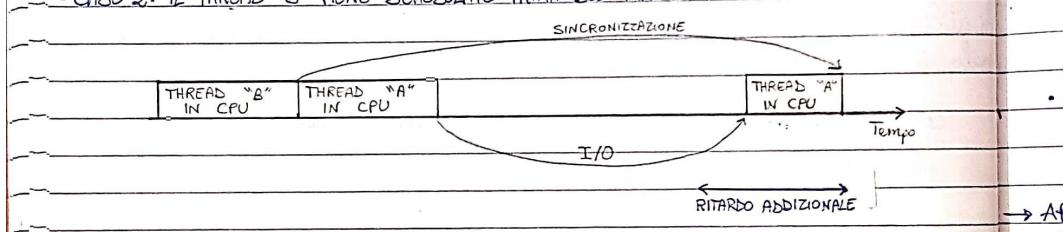
ESEMPIO:

Supponiamo di avere un thread A e un thread B concorrenti nella stessa applicazione, e supponiamo che A vada nello stato di blocco in attesa che qualche dispositivo di I/O termini il suo lavoro, e che B, dopo aver eseguito delle attività, chiami un servizio di sincronizzazione per attendere che A finisca la sua esecuzione. In questo caso, l'ordine con cui i due thread vengono mandati in esercizio influenza parecchio sul tempo di turnaround dell'intera applicazione.

CASO 1: IL THREAD "A" VIENE SCHEDULATO PRIMA DEL THREAD "B"



CASO 2: IL THREAD "B" VIENE SCHEDULATO PRIMA DEL THREAD "A"



Un altro fattore molto è che un processo può a questo punto essere sia CPU bound che I/O bound, dipendendo dal comportamento dei singoli thread che lo compongono. In particolare, l'I/O boundness non dipende soltanto dalla tendenza del thread nell'invocare i dispositivi di I/O, ma anche dalla tendenza nell'entrare nello stato di blocco per motivi diversi (e.g. per attendere che un'altra traccia completi una determinata attività), e questo è uno scenario che si verifica più frequentemente con i thread che con i processi. Di conseguenza, risulta più destande sbagliare a schedolare un thread piuttosto che sbagliare a schedolare un processo: nasce quindi la necessità di politiche avanzate per la gestione di assegnazione della CPU ai thread.

Scheduling di thread:

Esistono due politiche principali dello scheduling di thread: il LOAD SHARING e il LOAD BALANCING.

→ APPROCCIO LOAD SHARING:

Si ha una coda globale (anche multilivello) di thread pronti a eseguire, da cui si può pescare qualsiasi traccia da assegnare a qualsiasi CPU libera, per cui si ha

Sched

Ha

• PRIO

ent

• PRIO

a

• PRIO

1fa

una distribuzione uniforme del carico.

Problema di scalabilità: se ci sono più CPU, è possibile che, a un tratto, alcune di loro accedano contemporaneamente all'unica coda dei thread in attesa di essere eseguiti. In tal caso, i processori devono essere in grado di sincronizzarsi per evitare incosistenze.

Efficienza ridotta dell'architettura di caching: se inizialmente un thread viene processato dalla CPU A e successivamente viene assegnato alla CPU B, nella cache della CPU B (che NOL è continua) non si trovano i dati d'interesse per il processo in esecuzione in questione.

Possibilità di gestire vari livelli di priorità con politiche disparate: FCFS, SJFF (Smallest Number of Threads First), ecc.

→ APPROCCIO LOAD BALANCING:

• I thread pronti a eseguire sono mantenuti su pool (code) separati, uno per ogni CPU. Però, si ha una migliore scalabilità delle operazioni di scheduling, dato che non è necessaria una fase di sincronizzazione tra CPU.
• È necessario uno spostamento periodico di thread da una coda di CPU a un'altra, in caso di sbilanciamento di carico tra i vari processori. Inoltre, nel momento in cui avviene una migrazione, il software che la esegue deve attivare dei meccanismi di sincronizzazione nell'accesso sia al pool che il thread abbandona, sia a quello in cui il thread viene inserito, in modo tale che la migrazione non vada a interferire con le attività di scheduling.

Scheduling UNIX SPIN:

Esso divide i livelli di priorità in tre classi:

• ESEGUITE LEGGO: classe Real Time, in cui le tracce di esecuzione devono essere eseguite entro un certo limite di tempo.

• ESEGUITE SCRIVO: classe Kernel, in cui le attività hanno un'importanza paragonabile a quella eseguita dai demoni di sistema.

• ESEGUITE SCRIVO: classe Time-Sharing, che comprende le applicazioni convenzionali, le quali vengono schedolate secondo schemi classici.

Il Kernel è PREEMPTABLE: se sprovviste di un'applicazione della classe Real Time meno spese
tra vengono eseguiti dei moduli a livello Kernel (identificati come safe places), questi ulteriori non si
timi potrebbero perdere il controllo per favorire le attività più prioritarie.

N.B.: Questo non può accadere nel momento in cui in esecuzione ci sono dei moduli a livello Kernel che non sono marcati come safe places, perché magari devono portare a termine il prima possibile delle attività critiche.

In questo scheduler viene utilizzato un BITMAP che permette di determinare rapidamente i disponibili senza doverli scartare uno per uno.

Inoltre, il quanto di tempo varia in funzione delle classi di priorità (a priorità più alta corrispondono quanti di tempo più elevati).

Scheduling adottato dallo standard Posix:

Ha 100 livelli di priorità, che vanno da 0 a 99. Il livello 0 costituisce una vera e propria classe di priorità, la Time-Sharing, che a sua volta è suddivisa in 40 sottolivelli. Gli altri 99 livelli, invece, sono destinati alla classe Real Time.

Anche in questo scheduler la priorità delle applicazioni può essere modificata reimpennando la nice ness.

Per modificare la priorità di una specifica applicazione nello schema Real-Time, si può digitare sulla shell il comando chrt -p [priorità] [pid].

→ VAI
Tuttavia, tale operazione, in particolar modo se posta a un miglioramento della priorità, è permessa solo all'utente "root".
Se invece sulla shell si digita chrt -f [priorità] [pid], si sta cominciando la priorità di un thread nello schema Real Time e secondo una policy FCFS: tale traccia sarà assegnata a una CPU finché non la rilascerà spontaneamente oppure finché non sprovvisterà un altro thread con un livello di priorità più elevato.

Epoche di scheduling in LINUX:

Con un numero così considerabile di livelli di priorità, sorge il problema di quanto spesso i thread meno prioritari riescono a prendere il controllo di una CPU. Viene così introdotto il concetto di EPOCA.

Un'epoca inizia quando il Kernel riassegna ai thread attivi dei quanti di tempo da restituire.

men spendere in CPU in un determinato lasso di tempo, e finisce quanto tutti i thread han
i ul no speso tutti i loro quanti di tempo a disposizione.

Ovviamente, più un'applicazione è prioritaria, più quanti di tempo può sfruttare in una ce
a determinata epoca. In ogni caso, questa tecnica consente ai thread nei livelli più bassi
fare di spendere in CPU un numero, seppur minimale, di quanti di tempo per ogni epoca.
amen Il concetto di quanto di tempo viene meno solamente per le applicazioni schedolate secon
de una policy FCFS.

ii Supponiamo ora che un thread abbia a disposizione per una specifica epoca x quanti di tempo, che, durante la sua esecuzione generi un thread figlio e che, all'atto della genera
zione, gli rimangano x' quanti di tempo da spendere. Di questi x' quanti, al child
ne vengono donati y , mentre al parent ne restano $x'-y$.

era Si adotta questa tecnica di partizionamento e non si assegnano dei quanti di tempo nuove
to ai thread figli per evitare lo scenario in cui l'epoca non finisce più a seguito di
una generazione continua e ininterrotta di tanti thread child.

npo Scheduling in sistemi Windows:

È naturalmente orientato a schedulare thread in sistemi multi-core, per cui non è stato
può soggetto a un'evoluzione storica.

Ci sono code multiple di priorità divise in due fasce:

→ VARIABLE: livelli 0 → 15

→ REAL-TIME: livelli 16 → 31

Si hanno dunque 32 livelli di priorità diversi gestiti con la politica Round-Robin.

Il prelascio è basato anche sulla priorità: se una CPU è assegnata a un thread A
e subentra un thread B più prioritario, allora A cede il controllo a B anche nel caso
in cui il suo quanto di tempo non è stato del tutto consumato.

to Inizialmente viene assegnata una priorità base ai processi da cui possono dipendere le
co priorità dei loro singoli thread.

Nella fascia Variable, se un thread rilascia la CPU allo scadere del quanto, la sua prio
rità viene decalata di un livello (con un limite inferiore dato dalla priorità di riferimen

to, che è funzione della priorità base del processo in cui la traccia rive). Se invece un thread rilascia la CPU in anticipo, la sua priorità viene aumentata di un livello (con un limite superiore dato dal livello 15).

Ciò non è possibile all'interno della fascia Real-Time, in cui le priorità rimangono fisse a meno che non si chiama una system call apposita.

Affinità di processore:

Un thread si dice affine a uno specifico processore se fa parte del pool di thread associati a quella CPU (in altre parole, se ha la possibilità di essere assegnata a quel processore).

Le affinità sono temporanee: possono essere modificate nel tempo per esempio tramite un bilanciamento di carico. È possibile tuttavia rendere un'affinità duratura escludendo la possibilità che uno specifico thread venga migrato nel pool di thread associati a qualche altra CPU.

È una facility molto utile per:

- Vedere come si comporterebbe un'applicazione ipotizzando di avere un numero di processori più basso di quelli effettivamente presenti nell'architettura hardware (scale down).
- Lasciare libere determinate CPU per riservarle magari alle attività più critiche e urgenti.

In particolare, all'interno del Kernel, esiste per ogni thread una maschera di bit tra i metadati di gestione per quella specifica traccia. Tale maschera di bit indica quali sono le CPU permesse ($bit=1$) e quali non sono permesse ($bit=0$) per quel thread. Può comunque essere modificata con system call apposite.

Nel momento in cui un thread A genera un halt, quest'ultimo eredita da A questa maschera di bit.

Per modificare le affinità di uno specifico thread, si può digitare sulla shell il comando `taskset -p [bitmask] [pid]`.