

## SINCRONIZZAZIONE

Nel momento in cui si hanno più thread che condividono la stessa area di memoria, essi molti ben spesso, per non avere inconsistenti, devono accedervi seguendo delle regole di sincronizzazione ben precise. Vediamo subito un esempio.

Anchetipo del produttore/consumatore:

Supponiamo che un thread P (il produttore) abbia accesso in scrittura a un buffer con N entry e che un altro thread C (il consumatore) abbia accesso in lettura al medesimo buffer. Supponiamo inoltre che C debba leggere una sola volta dall'area di memoria tutte le informazioni scritte da P. Una regola che si può impostare è la circularità, per cui, una volta che uno dei due thread ha attraversato buffer [N-1], dovrà tornare più a buffer [0].

Questo modello, tuttavia, presenta dei grossi problemi: può accadere infatti che C inizi la sua computazione prima di P, per cui legge delle informazioni mai scritte dal produttore; d'altra parte, non è neanche da escludersi che P possa trovarsi oltre un giro avanti rispetto a C, il che per cui sovrascrive dati che non sono ancora stati consumati.

Per ovviare a questo inconveniente, viene aggiunto al buffer un contatore che tiene traccia del numero di item che sono stati scritti da P ma che non sono stati letti da C. In questo modo, quando il contatore è pari a 0, C deve fermarsi perché momentaneamente non ha più informazioni da leggere; analogamente, quando il contatore vale N, P deve bloccarsi onde evitare di sovrascrivere dati che devono ancora essere letti.

PSEUDOCODICE:

PRODUTTORE:

Repeat

```
<produce X>;  
while counter=N do no-op;  
buffer[in]←X;  
in←(in+1)mod(N);  
counter←counter+1;  
until false
```

CONSUMATORE:

Repeat

```
while counter=0 do no-op;  
Y←buffer[out];  
out←(out+1)mod(N);  
counter←counter-1;  
<consume Y>;  
until false
```

Questo meccanismo di sincronizzazione non funziona: consideriamo i comandi

1  $\text{counter} \leftarrow \text{counter} + 1$ ;  $\text{counter} \leftarrow \text{counter} - 1$ . Non sono atomici, bensì sono composti,  
di ben tre istruzioni macchina ciascuno:

2  $\text{Reg1} \leftarrow \text{counter}$   
 $\text{counter} \leftarrow \text{counter} + 1 \Rightarrow \text{Reg1} \leftarrow \text{Reg1} + 1$   
 $\text{counter} \leftarrow \text{Reg1}$

3  $\text{Reg2} \leftarrow \text{counter}$   
4  $\text{counter} \leftarrow \text{counter} - 1 \Rightarrow \text{Reg2} \leftarrow \text{Reg2} - 1$   
5  $\text{counter} \leftarrow \text{Reg2}$

- 2 Di conseguenza, sapendo che il codice del produttore e quello del consumatore costituiscono due  
3 thread differenti, in un contesto di time sharing (o anche con più processori) può presentarsi  
4 il seguente scenario:

5  $\text{Reg1} \leftarrow \text{counter}$   
6  $\text{Reg1} \leftarrow \text{Reg1} + 1 \leftarrow \text{INTERRUPT DA TIMER}$   
7  $\text{Reg2} \leftarrow \text{counter}$   
8  $\text{Reg2} \leftarrow \text{Reg2} - 1$   
9  $\text{counter} \leftarrow \text{Reg2} \leftarrow \text{INTERRUPT DA TIMER}$   
10  $\text{counter} \leftarrow \text{Reg1}$

In questo caso, il thread del produttore è stato interrotto da quello del consumatore e il decremento del contatore da parte di C non viene minimamente osservato, dando luogo così a un'inconsistenza.

#### Sezioni critiche:

Sono dei blocchi non atomici di istruzioni macchina dove:

- Un thread può leggere/scrivere dati condivisi con altri thread.
- La correttezza del risultato dipende dall'interleaving delle tracce in esecuzione.

La sequenza di istruzioni

Reg1 ← counter

Reg1 ← Reg1+1

counter ← Reg1

è di fatto una sezione critica.

Al

Sot

L'idea per cercare di risolvere tale problematica legata alla sincronizzazione è quella di permettere l'esecuzione della porzione di traccia relativa alla sezione critica come se fosse un'azione atomica. Per far ciò, esistono tre possibili approcci:

• Algoritmi di mutua esclusione

• Approcci hardware e istruzioni Read-Modify-Write (RMW)

• Mutex / semafori

Prv

var

PRI

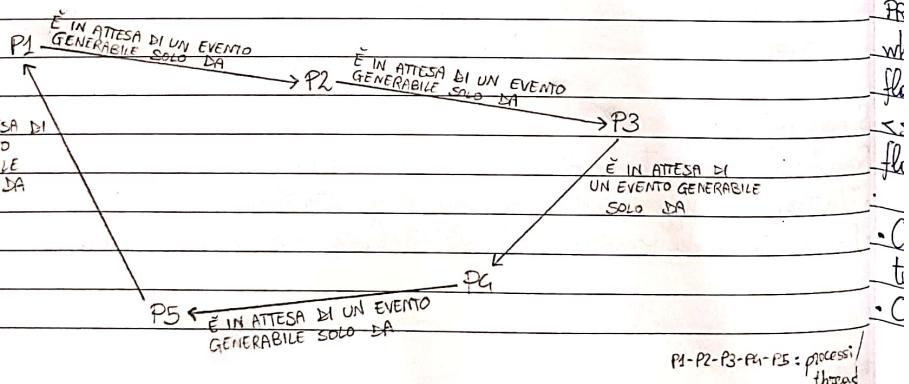
wh

Moltre, si può dire che il problema della sezione critica viene risolto in maniera soddisfacente se sono rispettate le seguenti tre condizioni:

→ MUTUA ESCLUSIONE: quando un thread accede a una sezione critica, nessun'altra traccia può eseguire la stessa sezione critica contemporaneamente.

→ PROGRESSO: un thread che lo richiede deve essere ammesso alla sezione critica senza ritardi nel caso in cui nessun'altra traccia si trovi nella medesima sezione critica.

→ ATTESA LIMITATA: un thread che lo richiede deve in ogni caso essere ammesso alla sezione critica in un tempo limitato: in particolare, non devono presentarsi né STARVATION né STALLI. Questi ultimi si verificano in scenari come il seguente:



Algoritmi di mutua esclusione:

Sono caratterizzati dalla seguente struttura:

	PREAMBOLO	→ Istruzioni che permettono la sincronizzazione con altri thread
	CORPO CENTRALE	→ Istruzioni della sezione critica
	CODA	→ Istruzioni che fanno sì che qualche altro thread progetta nelle attività di accesso alla sezione critica

Primo tentativo (algoritmo di Dekker):

var turno: int;

PROCESSO X

while turno ≠ X do no-op;

rit  $\langle$ sezione critica $\rangle$ ;

turno  $\leftarrow$  Y;

cia.

PROCESSO Y

while turno ≠ Y do no-op;

rit  $\langle$ sezione critica $\rangle$ ;

turno  $\leftarrow$  X;

• C'è garanzia di mutua esclusione.

• Non c'è garanzia di progresso: X deve attendere anche quando turno = Y ma Y non è in sezione critica.

me

Secondo tentativo:

var flag: array [1, n] of boolean;

PROCESSO X

while flag[Y] do no-op;

flag[X]  $\leftarrow$  TRUE;

rit  $\langle$ sezione critica $\rangle$ ;

flag[X]  $\leftarrow$  FALSE;

PROCESSO Y

while flag[X] do no-op;

flag[Y]  $\leftarrow$  TRUE;

rit  $\langle$ sezione critica $\rangle$ ;

flag[Y]  $\leftarrow$  FALSE;

• Ciascun processo si assicura che nessun altro si sia prenotato scorrendo l'array di flag, eventualmente alza la bandierina e poi entra in sezione critica.

• C'è garanzia di progresso.

- Non c'è garanzia di mutua esclusione; può infatti presentarsi il seguente scenario:
  - 1) X si accerta che nessun altro processo si è prenotato, esce dal ciclo while ma viene immediatamente deschedulato.
  - 2) Y si accerta che nessun altro processo si è prenotato, alza la bandierina e, mentre ge fl ra in sezione critica, viene deschedulato.
  - 3) X prende nuovamente il controllo ma ormai è uscito dal ciclo while, perciò alza anche lui la bandierina ed entra in sezione critica, dove ora sono in due.

Terzo tentativo:

var flag : array [1..n] of Boolean;

PROCESSO X

flag[X] ← TRUE;

while flag[Y] do no-op;

<sezione critica>;

flag[X] ← FALSE;

PROCESSO Y

flag[Y] ← TRUE;

while flag[X] do no-op;

<sezione critica>;

flag[Y] ← FALSE;

- È una variante del secondo tentativo in cui i processi prima alzano la bandierina e solo poi scortano l'array flag per assicurarsi che nessun altro si sia prenotato.

- Non c'è garanzia di attesa limitata a cause di possibili deadlock; può infatti presentarsi il seguente scenario:

1) X alza la bandierina ma viene immediatamente deschedulato.

2) Y alza la bandierina ed entra nel ciclo while, dove si accorge di dover attendere che X termini il suo lavoro.

3) X prende nuovamente il controllo ed entra anche lui nel ciclo while, dove si accorge di dover attendere che Y abbassi la sua bandierina. In questo modo, X, Y si stanno aspettando a vicenda, generando un vero e proprio deadlock.



Quarto tentativo:

var flag: array [1, n] of boolean;

PROCESSO X

```
flag[X] ← TRUE;  
while flag[Y] do {  
    flag[X] ← FALSE;  
    <pausa>;  
    flag[X] ← TRUE;  
}  
<sezione critica>;  
flag[X] ← FALSE;
```

PROCESSO Y

```
flag[Y] ← TRUE;  
while flag[X] do {  
    flag[Y] ← FALSE;  
    <pausa>;  
    flag[Y] ← TRUE;  
}  
<sezione critica>;  
flag[Y] ← FALSE;
```

• È una variante del terzo tentativo mirata a prevenire il deadlock.

• Tuttavia, non c'è garanzia di attesa limitata a causa di possibili starvation: infatti, ogni volta che un processo non riesce a entrare in sezione critica perché qualcun altro ha la bandiera alzata, cede il passo, rischiando di non ottenere mai il suo turno.

Cinque tentativo (algoritmo del forno - Lamport, 1974):

var choosing: array [1, n] of boolean; → PROBLEMI CHE INDICANO SE I RISPECTIVI PROCESSI SONO IN FASE DI FRELEVO DEL NUMERO D'ORDINE

number: array [1, n] of int; → NUMERI D'ORDINE; SE UGUALI A ZERO, VOLU' DIRE CHE I RISPECTIVI PROCESSI NON SONO MENTANEAEMENTE INTERESSATI A ENTRARE IN SEZIONE CRITICA

repeat {

choosing[i] ← TRUE;

number[i] ← <max in array number> + 1; → QUESTO COMANDO NON È ATÔMICO: L'AGGIORNAMENTO DI number[i] RICHIESTA 3 ISTRUZIONI MACCHINA (LOAD - OPERAZ. ARITMETICA - STORE).

choosing[i] ← FALSE;

for j=1 to n do {

while choosing[j] do no-op; → PER CUI PUÒ SUCCEDERE CHE UN PROCESSO PIÙ VENGA DESCHEDULATO DEDO.

choosing[j] ← TRUE; → LA LOAD HA PRIMA L'ELA STORE; IL PROCESSO CHE FRENDERÀ IL CONTROLLO.

number[j] ← number[i]; → NON VEDENDO L'ARRAY AGGIORNATO LA Pj, FRENDERÀ LO STESSO SUO NUMERO D'ORDINE.

choosing[j] ← FALSE;

while number[j] ≠ 0 and (number[j], j) < (number[i], i) do no-op;

number[j] ← 0; → ATTESA CHE TUTTI I PROCESSI, ALMENO PER UN ATTIMO, NON STIRNO AGGIORNANDO IL LORO NUMERO D'ORDINE, IN modo da non scavalcare a fringu eventuali concorrenti che stanno per ricevere lo stesso numero.

} → ENTRENDONO DI TRASFERIRSI IN SEZIONE CRITICA IN DUE PASSI AVANTI IL PROCESSO COL PIÙ MINORE.

<sezione critica>;

number[i] ← 0;

until FALSE

Anche questo algoritmo, seppur funzionante, presenta delle problematiche non indifferenti:

→ Dà luogo a BUSY WAITING: quando un processo entra in attesa di un evento da parte di qualcun altro, lo fa impiegando la CPU.

→ È consistente solo se l'architettura hardware sottostante ha un modello di memoria STRONG. Spieghiamo che le scritture vengono osservate in ordine anche da altri thread, per cui risulta in modo chiaro che la sequenza di operazioni effettiva è:

- 1) choosing[i]  $\leftarrow$  TRUE;
- 2) number[i]  $\leftarrow$  <max in array number[I + 1]>;
- 3) choosing[i]  $\leftarrow$  FALSE;

ecc.

ogni

scritt.

PSEL

Istruzioni RMW:

Un secondo approccio per affrontare il problema della sincronizzazione è l'introduzione di istruzioni che effettuano operazioni come l'aggiornamento di una variabile in maniera del tutto atomica: queste istruzioni sono dette RMW (Read-Memory-Write) e hanno tre fasi:

1) Prelevo di un valore  $x$  da un'area di memoria e inserimento di  $x$  all'interno di un registro del processore.

2) Esecuzione di operazioni su  $x$  (e.g. confronto con valori di altri registri del processore, aggiornamento).

3) Inserimento del valore aggiornato all'interno dell'area di memoria; nel frattempo, a bordo della CPU possono essere effettuate altre attività (e.g. settaggio di alcuni bit del registro di stato per indicare quale tipo di manipolazione è stata eseguita oppure se qualche particolare condizione è stata matchata).

Le istruzioni RMW richiedono quindi diversi cicli di clock per essere portate a termine, per cui, da sole, non garantiscono un'esecuzione senza interferenze da parte di altri processori che eseguono thread concorrenti. Per questo motivo, quando una RMW effettua un'operazione su una locazione di memoria M, a livello hardware (in particolare mediante una logica a bordo della gestione dell'architettura di memoria) avviene un arbitraggio che blocca qualunque altra CPU che tenti di accedere a M finché la RMW non avrà effettivamente completato il suo lavoro.

Nella pratica, non vengono bloccati solo gli accessi mirati a M, ma anche quelli tirati

a un qualunque punto della cache line (che per esempio può occupare 64 byte di memoria),  
a cui  $M$  appartiene.

### Spinlock:

Con una particolare istruzione RMW è possibile implementare una logica di SPINLOCK che funziona così:

Si ha una variabile serratura che, se impostata a 0, viene portata a 1 dal primo thread  $T$  che arriva, il quale entra poi in sezione critica. Tutte le altre tracce che s'aggirano nel momento in cui serratura = 1 vengono bloccate finché  $T$  non sarà uscito dalla sezione critica e avrà resettato la variabile a 0.

### PSEUDOCODICE:

var serratura : int;

PROCESSO X

while ! test-and-set (serratura) do no-op;

<sezione critica>;

serratura  $\leftarrow$  0;

```
function test-and-set (var z: int) : boolean {  
    if (!z) {  
        z  $\leftarrow$  1;  
        test-and-set  $\leftarrow$  TRUE;  
    }  
    else test-and-set  $\leftarrow$  FALSE;  
}
```

L'istruzione macchina RMW in questione è CAS (Compare-And-Swap) che sui processori x86 prende il nome di CMPXCHG (Compare-And-Exchange) e funziona nel seguente modo:  
→ Compare il valore di una locazione di memoria con quello del registro EAX.  
→ Se i due valori sono uguali, la locazione viene aggiornata col contenuto di un altro registro indicato come operando (e.g. RBX).

In effetti, questi sono esattamente i passaggi effettuati nella logica di spinlock sulla variabile serratura. D�K SIM

Comunque sia, anche lo spinlocking presenta il problema per cui i thread che attendono per accedere a una sezione critica lo fanno utilizzando la CPU in maniera massiva. Di conseguenza, questa tecnica va bene per sezioni critiche composte da poche istruzioni type macchina ma è molto inefficiente negli altri casi.

Union:

Sono delle tabelle di informazioni. La grande differenza dalle struct è che hanno associata a essi un'area di memoria capace di contenere non tutti i campi della tabella ma temporaneamente, bensì un'unica variabile alla volta: la grandezza di tale locazione, infatti, è solo pari alla dimensione del tipo di dato che occupa più byte all'interno della union.

Semafori:

Sono delle strutture dati di sincronizzazione gestite a livello Kernel. Rispetto agli algoritmi procedurali di mutua esclusione e alle istruzioni RMW, hanno il grosso vantaggio di non presentare busy waiting: quando un thread è in attesa di un particolare evento da parte di qualcun altro (come per esempio l'uscita da una sezione critica), viene messo in stato di blocco; nel momento in cui questo evento avviene, lo stato del semaforo cambia e il thread viene rischedulato. I semafori nella pratica possono essere visti come dei distributori di gettoni e sono caratterizzati da tre operazioni:

→ INIZIALIZZAZIONE con relativa assegnazione di un valore intero non negativo (= numero di gettoni) al nuovo semaforo. Sys

→ OPERAZIONE DI WAIT, che tenta il decremento di un'unità del valore del semaforo, e induce un'attesa sul chiamante nel caso in cui il valore di partenza è pari a 0. Semafori

→ OPERAZIONE DI SIGNAL, che incrementa di un'unità il valore del semaforo ed eventualmente libera dall'attesa un processo P che ha eseguito un'operazione di wait la quale lo ha portato nello stato di blocco. In tal modo, P consuma subito il gettone appena messo a disposizione. stat. inc. IPC vett.

Questa è possibile gestire una sezione critica con un semaforo a un solo gettone (SEMAFORO BINARIO).  
I `printf()` e `scanf()` utilizzano internamente degli approcci basati su spinlock o semafori per sincronizzare le attività sull'unico buffer di cui dispongono.

tb  
ivà IMPLEMENTAZIONI IN PSEUDOCODICE DEI SEMAFORI:  
i type semaphore < struct {  
 value : int; → numero di gettoni  
 L : list of processes; // or threads → processi/thread in attesa di un rilascio di gettoni  
};  
 ↗ PCB ↗ TCB

ocia  
n procedura Wait (var s: semaphore):  
if (s.value - 1) < 0 {  
 add current process to s.L;  
 block current process;  
} else s.value --;

itmi procedura Signal (var s: semaphore):  
s.value ← s.value + 1;  
if s.L not empty {  
 delete a process P from s.L;  
 unblock P;  
 s.value ← s.value - 1;  
}

Queste sono brevi attività del Kernel in sezione critica e possono essere eseguite con il supporto dello spinlock.

istuce System V semaphore in UNIX:  
Sono degli array in cui ciascuna entry corrisponde a un semaforo. Un thread può tentare un prelievo di un qualsiasi numero n di gettoni da n entry diverse del vettore. Se anche uno solo dei valori associati a questi n semafori è inizialmente uguale a 0, il thread viene messo in blocco finché la sua richiesta non diventerà globalmente soddisfacibile.  
Inoltre, quando viene istanziato un array semaforico, viene inizializzata una entry nella IPCI (Inter Process Communication Table) con un descrittore che permette l'utilizzo del nuovo vettore.

### Liberie semaforiche Posix:

Per utilizzare i semafori su UNIX è possibile sia ricorrere direttamente alle system call (semget(), semctl(), semop()) ma anche delle funzioni di libreria più semplici che, dati una struttura dati semaforica a livello user e un suo puntatore, operano nel seguente modo:



Tra le informazioni di questa struttura dati c'è un identificatore del canale di I/O che viene utilizzato dalla libreria in modo trasparente al programmatore per andare a operare su un oggetto di I/O a livello Kernel. Tale oggetto è incorporato in uno specifico driver che gli permette di eseguire le operazioni di sincronizzazione.

Tra le API di libreria troviamo sem-open(), sem-init(), sem-wait(), sem-post(), sem-getvalue(), sem-unlink() .

### Struttura sem-undo:

Se per un certo semaforo S il flag SEM\_UNDO è attivato, ogni volta che un'applicazione termina, il Kernel deve revocare le sue operazioni eseguite su S. Affinché ciò sia possibile, il sistema operativo mantiene una struttura sem-undo in cui sono registrate le operazioni di sincronizzazione di ogni processo.

Il valore di tale struttura non viene ereditato da un child generato tramite fork(), ma viene comunque mantenuto in caso di sostituzione di codice tramite exec: si tratta infatti di un'informazione che riguarda direttamente la singola applicazione.

### Sincronizzazione con più produttori e più consumatori:

Di norma, nel caso più generale, una sezione critica può ospitare o esclusivamente uno scrittore o esclusivamente un certo numero di lettori (fino a un massimo di N, che può

varicare in base alle esigenze). Affinché questa condizione sia rispettata, si utilizzano due distributori:

1 GETTONE	N GETTONI
-----------	-----------

Distributore per lo scrittore - Distributore per i lettori

→ Gli scrittori, affinché siano effettivamente fuori all'interno della sezione critica, devono prelevare sia il gettone del primo distributore, sia tutti gli N token del secondo.

→ Ai consumatori basterebbe richiedere un unico gettone al distributore per i lettori ma, in tal modo, provocherebbe starvation a discapito degli scrittori: se per esempio c'è un produttore in attesa e solo una parte dei token è disponibile nel distributore per i lettori, nel momento in cui sognaggino altri consumatori, essi possono entrare direttamente in sezione critica passando avanti al produttore. Per evitare ciò e adottare una politica di accesso FIFO, i lettori devono prima prelevare il gettone del primo distributore, poi richiederne un altro al secondo e, infine, rilasciarne subito uno al primo.

### 1. Mutex:

È una variante del semaforo che ha solo due stati:

→ 0: MUTEX NON DISPONIBILE

→ 1: MUTEX DISPONIBILE

Su Unix è supportato dalla libreria pthread e funziona allo stesso modo di una struttura semaforica di livello user.

Tra le API di libreria troviamo `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`.

NB: C'è una differenza tra un mutex e un semaforo binario:

• MUTEX: l'unico gettone può essere rilasciato esclusivamente dal thread che precedentemente lo aveva prelevato.

• SEMAFORO BINARIO: chiunque può rilasciare un token dopo che è stato prelevato da uno specifico thread.

## EVENTI

- Sono particolari condizioni rilevabili dal software del Kernel, ma sono molto spesso di interesse anche per il software applicativo: il sistema operativo fa da ponte, tramite un meccanismo basato su **SEGNALAZIONE** (o **NOTIFICA**), tra gli eventi e il software applicativo per permettere a quest'ultimo di poter "reagire" all'accadimento degli eventi stessi.
- In particolare, inviare una segnalazione a un'applicazione implica aggiornare i suoi metadati di gestione, il cui nuovo valore le consente di eseguire delle attività reactive.
- Gli eventi possono essere di natura differente:
- Possono essere causati dall'esecuzione dell'applicazione stessa per poi essere rilevati dal software del Kernel (e.g. segmentation fault).
  - Possono essere generati da componenti software esterni all'applicazione (e.g. dai demoni).
  - Possono essere attivati tramite una system call inclusa nel processo stesso.

### Gestione degli eventi a livello applicativo:

Quando viene generato un evento, il Kernel fa sì che l'applicazione reagisca a esso nel momento in cui nell'interno del software applicativo viene registrata ed eseguita un'apposita funzione, nota come **GESTORE DELL'EVENTO**.

Le modalità secondo cui l'attivazione del gestore avviene sono differenti: a seconda della tipologia e delle caratteristiche del sistema operativo: per esempio, un fattore molto incisivo è se il multi-threading fosse supportato fin dall'inizio oppure no.

### → MODELLO UNIX

Supponiamo che, mentre un thread T è in esecuzione, avviene un evento: il Kernel prende il controllo e, quando lo rilascia a T, installa sulla CPU delle informazioni, tra cui un particolare valore del Program Counter che non porta T a riprendere l'esecuzione da dove era stata interrotta, bensì in qualche altra zona di codice che permette a T di reagire all'evento.

SOLO A SEGUITO DELLA TERMINAZIONE DI QUESTA ZONA DI CODICE L'ESECUZIONE DI T VIENE EFFETTIVAMENTE RIPRISTINATA

Questo modello si basa sul concetto di **INTERRUPT A LIVELLO SOFTWARE** (di fatto funziona come un interrupt vero e proprio).

### → MODELLO WINDOWS CON VIRTUAL POLLING

Supponiamo che, mentre un thread T di un processo P è in esecuzione, avviene un evento: il Kernel lo modifica e genera un nuovo thread T' all'interno di P. Sarà proprio T' a eseguire il blocco di codice del gestore dell'evento.

### → MODELLO WINDOWS CON REAL POLLING

Supponiamo che un processo P è in esecuzione. Allora esiste un thread T che vive in P e che chiede periodicamente al Kernel, tramite apposite system call, se qualche evento si è verificato. Se la risposta è sì, T va a eseguire il blocco di codice del gestore dell'evento.

Questo modello, fatto, funziona in modo analogo alle API `WaitForSingleObject()` e `WaitForMultipleObjects()`, con due differenze sostanziali:

- Un thread che chiama `WaitForSingleObject()` o `WaitForMultipleObjects()`, dopo essere stato eventualmente messo in stato di blocco, riprende le sue attività a livello user da dove erano state interrotte; dall'altra parte, T chiede al Kernel di essere portato in una zona del testo differente dopo che si è verificato un evento.
- `WaitForSingleObject()` e `WaitForMultipleObjects()` richiedono come parametro di handle agli oggetti per cui viene invocata l'attesa, mentre il thread T non specifica nulla al Kernel riguardo le entità che possono scatenare un evento.

### Eventi in sistemi UNIX:

In un sistema UNIX, un processo risiede in uno dei seguenti tre stati in relazione con la segnalazione di un evento:  
→ l'intero processo, non un singolo thread!

→ SEGNALAZIONE IGNORATA IMPLICITAMENTE: è lo scenario di default, in cui non viene specificato nulla al Kernel riguardo l'accadimento di eventi. Perciò, nel momento in cui il Kernel ne rileva uno, emette una segnalazione che verrà ignorata, per cui, in maniera del tutto autonoma, terminerà l'applicazione.

→ SEGNALAZIONE IGNORATA ESPlicitamente: al Kernel viene specificato in modo esplicito che le eventuali segnalazioni che verranno emesse non sono di interesse per l'applicazione.

→ SEGNALAZIONE CATTURATA: nel momento in cui qualche evento accade, il Kernel prende il controllo per potere il processo a eseguire un gestore dell'evento.

È possibile portare l'applicazione da uno stato all'altro in modo del tutto arbitrario mediante apposite system call.

Inoltre, ciascuna segnalazione è associata a uno specifico codice numerico, che è uno dei metadati di gestione dell'applicazione.

Segnalazioni UNIX più comuni:

- SIGHUP (1) → hangup: il processo riceve questo segnale quando il terminale a cui era associato viene chiuso.
- SIGINT (2) → interrupt: il processo riceve questo segnale quando l'utente preme la combinazione di tasti di interrupt (solitamente Control+C).
- SIGQUIT (3) → quit: è simile a SIGINT ma in più, in caso di terminazione del processo, il sistema registra lo stato attuale dell'address space su un particolare file detto CORE DUMP. Solitamente SIGQUIT viene generato premendo i tasti Control+\.
- SIGILL (4) → illegal instruction: il processo riceve questo segnale nel caso in cui tenti di eseguire un'istruzione proibita o inesistente.
- SIGSEGV (11) → segmentation violation: il processo riceve questo segnale nel caso in cui tenti di eseguire un accesso non supportato all'interno dell'address space, che porta a un segmentation fault.
- SIGTERM (15) → termination: il Kernel invia questo segnale al processo se vuole invitarlo a terminare l'esecuzione il prima possibile (si tratta di una richiesta non forzata di terminazione).
- SIGALRM (14) → alarm: il processo riceve questo segnale al termine del countdown dell'orologio di allarme.
- SIGCHLD (17) → child death: il processo riceve questo segnale quando uno dei suoi figli termina. Se SIGCHLD viene ignorato implicitamente, il parent non termina; inoltre, se non viene ignorato esplicitamente, il Kernel mantiene il controllo di terminazione del child.
- SIGKILL (9) → Kill: se il processo riceve questo segnale, a prescindere dello stato di partizione in cui si trova, è costretto a terminare immediatamente (viene di fatto ucciso).
- SIGUSR1, SIGUSR2 (10, 12) → user defined: sono segnali che non hanno un significato ben preciso: il Kernel non li emette in maniera spontanea, bensì solo se l'utente lo richiede.

### Eredità degli stati di relazione:

Il comportamento di un processo associato alla ricezione di un segnale (e quindi il suo stato o la relazione con quel segnale) viene sempre ereditato dai child.

Per quanto riguarda la famiglia exec, solo le impostazioni SIG\_IGN e SIG\_DFL vengono mantenute; la cattura, invece, viene convertita in ignoramento implicito poiché, in generale, a seguito del cambiamento dell'address space, il gestore dell'evento viene perso.

### Mecanismo di consegna dei segnali:

Di norma, quando si verifica un evento, il Kernel invia la relativa segnalazione al thread di interesse la volta successiva in cui quest'ultimo viene rischedulato in user mode.

Supponiamo ora che, in un certo istante di tempo, un thread T chiavi una system call bloccante e che, prima del completamento del servizio specificato nella chiamata di sistema, T venga colpito da una segnalazione (per esempio tramite una kill()). Allora T viene subito portato nello stato ready per poter reagire in tempi brevi all'evento e, di conseguenza, la system call fallisce (si dice che viene abortita).

In questo caso specifico, la variabile errno viene settata a EINTR.  
NB: Attualmente, col multi-threading, errno non è più una semplice variabile, ma una macro che invoca una funzione che va a leggere il codice d'errore per uno specifico thread in un'apposita variabile TLS.

### Signal mask:

È una maschera di bit che indica per uno specifico thread quali segnalazioni sono arrivate (e stanno per essere servite) e quali no:



BIT = 1 → la segnalazione è arrivata.  
BIT = 0 → la segnalazione non è arrivata.  
In particolare, una volta che il gestore di uno specifico evento viene attivato (o più in generale la segnalazione è stata acquisita), il bit corrispondente nella signal mask viene resettato automaticamente a 0.

Le consegne multiple dello stesso segnale nell'arco di poco tempo possono andare perse: Blocc  
ogni entry della signal mask può assumere soltanto due valori, per cui non è un contatore. Perme  
Di conseguenza, se sopraggiunge una segnalazione ~~per~~ nel momento in cui il bit corrispon~~po av~~  
dente della maschera è già settato a 1, il valore di questo bit non cambia, per cui alla  
fine il segnale verrà acquisito solo una volta.

Questo  
caso, re

#### Non atomicità dei gestori:

In generale, un gestore di segnale è costituito da diverse istruzioni macchina e può essere a struttura  
sua volta interrotto dall'arrivo di un'altra segnalazione: questo implica che l'esecuzione È una  
dei gestori non atomica, il che porta a una serie di problemi: così:

→ Ogni gestore necessita della sua area di stack all'interno dell'address space, per cui la  
ricezione di una quantità immensa di segnali porta a un'allocazione massiva dello stack, struc  
rischiando così uno stack overflow.

→ Supponiamo che la segnalazione  $x$  porti all'esecuzione del gestore  $H$  il quale contiene una  
funzione di sincronizzazione con un mutex. Supponiamo inoltre che il thread  $T$ , mentre  
sta già eseguendo il codice di  $H$  (e ha già prelevato l'unico token del mutex), venga  
nuovamente colpito dal segnale  $x$ . Allora  $T$  viene interrotto e, quando riprende il control  
lo, deve ricominciare da capo l'esecuzione di  $H$ : in tal modo, troverà il mutex non disponibile, per cui entrerà nello stato di blocco in attesa che l'oggetto di sincronizzazione  
si sblocca, cosa che però può essere fatta solo per opera di  $T$  stesso: viene generato  
così un deadlock.

→ Spesso le attività del gestore sono critiche e, se non vengono portate a termine in maniera atomica, si possono avere degli effetti collaterali.

Per ovviare a tutti questi inconvenienti, lo standard di sistema Posix mette a disposizione delle API appropriate (e.g. `sigprocmask()`, `sigpending()`, `sigaction()`) che consentono di effettuare le seguenti operazioni:

- Bloccare la consegna di determinati segnali e, quindi, impedire che essi colpiscano il thread nel momento in cui vengono inviati. Il bloccaggio viene comunque ignorato e bypassato per i segnali scatenati da un accesso in memoria scorretto e per SIGKILL.

In particolare, le segnalazioni inviate ma non effettivamente consegnate a causa di un

Varie sono dette PENDENTI.

- Permettere al thread di reagire ai segnali pendenti in un secondo momento (magari dopo aver effettuato una query sulla loro esistenza).

Questo tipo di attività è molto utile per eseguire un gestore di memoria in maniera atomica, rendendo immune da ulteriori segnalazioni.

Struttura sigaction:

È una strutt di UNIX che include informazioni di gestione di uno specifico segnale ed è fatta così:

```
struct sigaction {  
    void (*sa_handler)(int); ~> SIG_DFL OPPURE SIG_IGN OPPURE PUNTATORE A UNA FUNZIONE DI GESTIONE DEL  
    int sa_flags; ~> SEGNALE CHE PUÒ ACCETTARE UN INTERO COME UNICO PARAM. (IL CODICE DELLA SEGNALEAZ.)  
    void (*sa_sigaction)(int, siginfo_t *, void *); ~> PUNTATORE A UNA FUNZIONE DI GESTIONE  
    sigset_t sa_mask; ~> DEL SEGNALE CHE ACCETTA ANCHE INFORMAZIONI A GRANA+FINE  
    SIGNAL MASK CHE SPECIFICA I SEGNALI DA BLOCCARE DURANTE L'ESECUZIONE DEL GESTORE  
    int sa_flags; ~> FLAG CHE INDICA SE LE INFORMAZIONI A GRANA FINE SU UN EVENTO DEVONO ESSERE INVOLATRE O MENO DAL KERNEL;  
    void (*sa_restorer)(void); ~> TALI INFORMAZIONI SONO EVENTUALMENTE UTILI COME PARAMETRI DI "sa_sigaction"  
};
```

## Informazioni per la gestione del segnale

```
siginfo_t {  
    int si_signo; /* Signal number */  
    int si_errno; /* An errno value */  
    int si_code; /* Signal code */  
    pid_t si_pid; /* Sending process ID */  
    uid_t si_uid; /* Real user ID of sending process */  
    int si_status; /* Exit value or signal */  
    clock_t si_utime; /* User time consumed */  
    clock_t si_stime; /* System time consumed */  
    sigval_t si_value; /* Signal value */  
    int si_int; /* POSIX.1b signal */  
    void * si_ptr; /* POSIX.1b signal */  
    void * si_addr; /* Memory location which caused fault */  
    int si_band; /* Band event */  
    int si_fd; /* File descriptor */  
}
```

Tipico per la gestione di SIGSEGV

### Trattamento del segnale SIGHUP:

Esistono due modi per far sopravvivere un'applicazione P al segnale SIGHUP (e quindi In alla chiusura della shell parent):

→ Impostazione dello stato di relazione  $\rightarrow$  di ignoramento esplicito con SIGHUP tramite una chiamata a signal() o sigaction().

→ Aggiunta sulla command line della voce nohup che consente a P di cambiare lo stato di relazione con SIGHUP dopo la fork() eseguita dalla shell e prima della exec.

In tal modo, se la shell parent viene chiusa mentre P è in esecuzione, poiché è previsto che qualunque processo attivo deve avere un parent, P diventa automaticamente un child di init, che è l'applicazione capostipite di tutte le altre.

### Determinazione del modo di terminazione di un processo:

A questo punto della trattazione possiamo dire che un processo può terminare principalmente in due modi:

• Tramite una chiamata a exit(c), dove c è il codice di terminazione.

• Per effetto di una segnalazione.

Per ciò, una variabile intera che contiene le informazioni di terminazione di un processo ha ch i due byte più significativi che sono delle don't care e i due meno significativi che sono organizzati in uno dei seguenti modi:

EXIT CODE	0	0	SIGNAL NUMBER
-----------	---	---	---------------

Per determinare ad esempio il modo di terminazione di un child, si può ricorrere ai seguenti comandi:

```

int status;
wait(&status);
if((status & 255)==0) {
    printf("Process regularly exited with exit status %d \n", (status>>8) & 255);
} else if((status>>8 & 255)==0) {
    printf("Process abnormally terminated by signal %d \n", status & 255);
}
  
```

### Eventi in Sistemi Windows:

In Windows si distinguono due categorie di eventi:

→ **EVENTI DI SISTEMA**: come in UNIX, fanno la possibilità di eseguire un gestore, ma comunque viene adottato il meccanismo secondo cui il thread interrompe l'esecuzione per trasferirsi nell'handler: in particolare, ciò avviene quando è impossibile mandare avanti l'esecuzione senza prima aver reagito all'evento (per esempio, in caso di segmentation fault). In altri casi, viene utilizzato uno schema di polling virtuale.

→ **MESSAGGI-EVENTO**: sono oggetti utilizzati per notificare e comunicare all'applicazione un cambio di stato di una delle finestre gestite dal processo stesso. In particolare, una finestra è un oggetto attraverso cui l'utente può interagire con l'applicazione (per esempio, può essere un pulsante, una casella di testo, ecc.).

Per i messaggi-evento viene adottato uno schema di polling reale.

### Eventi di sistema:

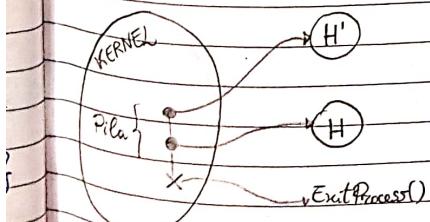
Sono simili a quelli visti per UNIX; alcuni esempi sono: CTRL-C-EVENT (0), CTRL-BREAK-EVENT (1), CTRL\_CLOSE\_EVENT (2), CTRL\_LOGOFF\_EVENT (5), CTRL\_SHUTDOWN\_EVENT (6).

Un gestore è una funzione del tipo `BOOL WINAPI HandlerRoutine (-In- DWORD dwCtrlType)`, che accetta come parametro il codice di un segnale e restituisce TRUE se, dal punto di vista del Kernel, la sua esecuzione è stata effettivamente portata a termine, FALSE altrimenti.

Analizziamo ora il meccanismo con cui funzionano gli handler:

Al livello Kernel viene mantenuta una pila di indirizzi di memoria che corrispondono ai punti dell'address space in cui iniziano i gestori. Ogni volta che viene attivato un handler tramite un'apposita system call, viene aggiunto il relativo indirizzo in cima alla pila.

Consideriamo un esempio con due gestori validi: H (attivato per primo) e H' (attivato dopo).



Nel momento in cui avviava una qualunque se  
gnalazione, il Kernel attiva un nuovo thread  
(se previsto), il quale esegue gli handler con  
una logica Last-In-First-Out, quindi partendo  
da H'. Se quest'ultimo restituisce TRUE, vuol  
dire che è stato effettivamente pensato per rispon-

→ Nello schema di virtual polling può essere attivato al più un thread per volta per eseguire gli handler.

dere a questa segnalazione; in caso contrario, il thread va a eseguire H. Se anche coloro questo secondo gestore restituisce FALSE, e non ce ne sono altri da poter eseguire, il thread in Kernel aggiorna lo snapshot di CPU affinché il thread chiami l'handler di default, ha i ExitProcess(), che invoca la terminazione dell'intera applicazione.

Tuttavia, esistono degli scenari in cui tutto ciò non accade ed è proprio lo stesso thread che esegue il codice dell'applicazione a interrompere il suo flusso ed eventualmente ad andare a girare l'handler (stabilito con la system call signal()) nel momento in cui avviene una segnalazione. Questo è il caso in cui l'evento è relativo a un'anomalia e, quindi, è uno dei seguenti:

- SIGABRT → terminazione anomala
- SIGFPE → errore a virgola mobile
- SIGILL → istruzione non valida
- SIGSEGV → accesso alla memoria non valida
- SIGTERM → richiesta di terminazione

#### Messaggi-evento:

Sono una forma di comunicazione asincrona. Rispetto agli eventi di sistema, sono caratterizzati, oltre dal codice numerico, da dei veri e propri messaggi (codificati con altri due numeri) contenenti informazioni a gamma fine sull'evento.

Qui l'utilizzo delle finestre è cruciale: esse sono il mezzo per la trasmissione dei messaggi-evento, i quali sono correlati al cambio di stato di una data finestra e possono provenire sia dall'utente interattivo (uso di finestre visibili) che dal sistema (uso di finestre non visibili).

Ciascuna finestra è associata a una specifica classe C, che contiene i metadati di gestione e di caratterizzazione di tutte le finestre appartenenti a essa. Tra queste informazioni figura un puntatore all'handler che viene eseguito per gestire gli eventi che colpiscono una finestra appartenente a C: di conseguenza, le finestre della medesima classe hanno il gestore in comune.

Inoltre, i messaggi-evento sono caratterizzati dallo schema di polling reale: gli handler vengono eseguiti da un thread T attivo già da prima dell'accadimento dell'evento. In particolare,

classe, l'invia periodicamente al Kernel se è sopraggiunto un messaggio-evento; nel momento in cui la risposta è sì, l'invia richiedere di essere portato a eseguire il gestore relativo alla finestra che è stata colpita.

Handler relativi a messaggi - evento:

CALLBACK indica che la chiamata alla funzione non avviene in modo diretto, bensì da parte del Kernel

Sono funzioni del tipo LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam), dove:

- HWND hwnd → handle verso la finestra colpita dall'evento
- UINT uMsg → codice numerico dell'evento
- WPARAM wParam & LPARAM lParam → codici numerici che costituiscono parte del payload del messaggio generato dal Kernel per discriminare ulteriori informazioni riguardo l'evento.

Struttura WNDCLASS:

È una struttura di Windows che include informazioni di gestione e di caratterizzazione di una specifica classe di finestre, ed è fatta così:

```
typedef struct _WNDCLASS {  
    UINT style;  
    WNDPROC lpfnWndProc; → PUNTATORE ALLA FUNZIONE DI GESTIONE DEGLI EVENTI  
    int cbClsExtra;  
    int cbWndExtra;  
    HANDLE hInstance;  
    HICON hIcon;  
    HCURSOR hCursor;  
    HBRUSH hbrBackground;  
    LPCTSTR lpszMenuName;  
    LPCTSTR lpszClassName; → NOME DELLA CLASSE DI FINESTRE  
} WNDCLASS;
```

→ Gli altri parametri riguardano perlopiù gli aspetti grafici delle finestre

### Meccanismo del polling reale:

Un thread, per effettuare polling reale, esegue i seguenti comandi:

SI ESCO DAL CYCLE GUANDO IL MESSAGE() RESTITUISCE ZERO; CIÒ AVVIENE GUANDO VIENE INVIATO IL MESSAGGIO-EVENTO WM\_QUIT CHE, TIPICAMENTE, PORTA AL TERMINE DELLA PIATTAFORMA.

```
while (GetMessage (&msg, NULL, 0, 0)) {
    TranslateMessage (&msg);
    DispatchMessage (&msg);
}
```

È anche possibile organizzare più thread, ciascuno dei quali si occupa di uno specifico range di codici numerici dell'messaggio-evento (oppure di una specifica classe di finestra)

### Gestire i messaggi-evento di default:

Nello standard di sistema Windows esistono già impostati:

- Un gestore di default, DefWindowProc()
- Moltissimi tipi di messaggi-evento di default

Tuttavia, non è raro creare dei messaggi-evento "custom" da inserire all'interno di un particolare processo. A tal proposito, per progettare un'applicazione con finestre, è possibile definire ex-novo un handler che, tramite un costrutto switch{} (analogo a if/else), chiama la funzione DefWindowProc() solo se il messaggio-evento che si è verificato è di default.

Un esempio di messaggio-evento di default è WM\_CREATE (Window Manager Create), che consiste nella creazione di una nuova finestra e porta quindi all'esecuzione del rispettivo handler, il quale, per exemplificare la vita del programmatore, può appunto invocare il gestore di default, che a sua volta restituisce il codice numerico 0 per comunicare l'avvenuta creazione della finestra.

### Classi di finestre di default:

Come ausilio di programmazione, WINAPI offre anche delle classi di finestre preconfigurate, tra cui:

- BUTTON, che viene colpita da messaggi-evento ogni volta che viene cliccato dall'utente.
- EDIT, che è una finestra in grado di contenere testo e viene colpita da messaggi-evento ogni volta che l'utente digita un carattere da tastiera.

In particolare, esistono due messaggi-evento fondamentali che possono colpire una finestra di

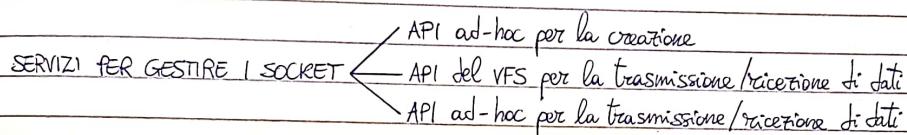
tipi box:

- WM\_GETTEXT, che fornisce su un buffer il testo contenuto nella finestra.
- WM\_SETTEXT, che mostra sulla finestra il testo attualmente presente in un buffer.

~ È possibile utilizzare il MENU per permettere all'utente interattivo di inviare ulteriori messaggi-vento a una specifica finestra.

## SOCKET

È uno strumento dei sistemi operativi che serve a costruire applicazioni che siano per esempio in grado di sfruttare interfacce di rete. In particolare, è un oggetto di I/O associato a un canale di I/O, il quale viene acceduto tramite descrittori/handle.



Le reali operazioni associate alle system-call, come per gli altri oggetti di I/O, vengono implementate da un driver o una pila di driver di livello Kernel.

Per ogni socket devono essere definiti i seguenti elementi:

→ DOMINIO = ciò che determina il modo in cui un socket S è identificato; costituisce quindi le regole con cui i metadati di S sono strutturati. Nella pratica, è un insieme di identificatori di socket (detti INDIRIZZI) che possono essere:

• LOCALI RISPETTO A UNO SPECIFICO SISTEMA OPERATIVO

• LOCALI RISPETTO A UNA SOTTORETE DI SISTEMI (e.g. LAN - Local Area Network)

• GLOBALI (universalmente validi)

→ TIPO DI COMUNICAZIONE = modalità con cui le informazioni sono emesse/ricevute tramite socket; può essere:

• STREAM (flusso di informazioni)

• BLOCK (pacchetto di informazioni)

→ PROTOCOLLO = istanza del driver di I/O da associare a un socket.

Domini classici:

DOMINIO	EQUIVALENTE A:	DESCRIZIONE	NOTE
AF_INET (Address Family Internet)	PF_INET (Protocol Family Internet)	Dominio Internet	Sockets vengono identificati su scala globale.
AF_UNIX	PF_UNIX	Dominio UNIX	Sockets sono utilizzabili solo nell'istanza di sistema UNIX.
AF_NS	PF_NS	Dominio Xerox	
AF_IMPLINK	PF_IMPLINK	Dominio IMP (Interface Message Passes)	

sono delle macro che identificano i codici numerici associati a questi domini

L'equivalenza tra AF\_INET e PF\_INET (e così via) è dovuta al fatto che tipicamente c'è una relazione molto stretta tra dominio e protocollo: l'utilizzo di uno specifico dominio per un socket  $S$   $\rightarrow$  vincola la scelta del protocollo per  $S$ .

Formato degli indirizzi:

Per identificare un socket  $S$  viene utilizzata una struttura dati che specifica:

- Il dominio di  $S$
- L'identificatore di  $S$  all'interno del dominio

La struttura più generale è la seguente:

```
struct sockaddr {  
    u_short sa_family; /* address family */  
    char sa_data[14]; /* up to 14 bytes of protocol specific address */
```

• A)

In ogni caso, ciascun dominio supporta una sua struttura ad-hoc per l'identificazione dei socket.

Vediamo qualche esempio:

- AF\_UNIX  $\rightarrow$  struct sockaddr\_unix

	FAMILY	Tip
	FINO A 108 BYTE PER IL PATHNAME	

- AF\_INET  $\rightarrow$  struct sockaddr\_in

	FAMILY	
	2 BYTE PER IL PORT NUMBER	→ Codice numerico dello specifico socket all'interno del sistema
	4 BYTE PER L'IP NUMBER (HOST-ID)	→ Istruzione di sistema operativo in cui il socket è ospitato
	ZONA INUTILIZZATA	Pri ca toc più

```

struct sockaddr_in {
    short sin_family;           /* Domain */
    u_short sin_port;          /* 2-bytes port number */
    struct in_addr sin_addr;   /* 4-bytes host-id */
    char sin_zero[8];          /* unused */
}

```

↳ struct in\_addr {  
 u\_long s\_addr; } → I 4 byte dell'host-id devono essere scritti secondo una regola  
 universale di ordinamento per ovviare al problema delle endianess  
 differenti che si possono avere su processori diversi.

• AF\_NS → struct sockaddr\_ns

FAMILY	
4 BYTE PER IL NET-ID	→ Sottorete di appartenenza del sistema operativo
6 BYTE PER L'HOST-ID	→ Istanza di sistema operativo in cui il socket è ospitato
2 BYTE PER IL PORT NUMBER	→ Codice numerico dello specifico socket all'interno del sistema
ZONA INUTILIZZATA	

↳ Tipi di comunicazione classici:

TIPO DI COMUNICAZIONE	DESCRIZIONE
SOCK_STREAM	Comunicazione basata sullo streaming
SOCK_DGRAM	Comunicazione basata sullo scambio di datagrammi (particolari: puoi, ti informa)
SOCK_RAW	Comunicazione basata sullo scambio di pacchetti informativi non elaborati
SOCK_SEQPACKET	Comunicazione basata sullo scambio di pacchetti sequenziali
SOCK_RDM	

↳ SONO DELLE MACRO CHE IDENTIFICANO I CODICI NUMERICI ASSOCIATI A QUESTI TIPI DI COMUNICAZIONE

↳ Protocolli di default:

↳ A ciascuna coppia DOMINIO-TIPO DI COMUNICAZIONE può essere associato nessuno, uno o tanti protocolli differenti in base alla compatibilità. Nella pagina seguente sono mostrate le combinazioni più importanti.

TIPO DI COMUNICAZ.	DOMINIO	AF_UNIX	AF_INET	AF_NS
SOCK_STREAM		Esegue un unico protocollo di default che però non ha un acronymo	TCP	SPP
SOCK_DGRAM		Esegue un unico protocollo di default che però non ha un acronymo	UDP	IDP
SOCK_RAW			IP	Esegue un unico protocollo di default che però non ha un acronymo
SOCK_SEQPACKET				SPP

Combinazioni ammesse per AF\_INET:

DOMINIO	TIPO DI COMUNICAZIONE	PROTOCOLLO	
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
AF_INET	SOCK_RAWR	IPPROTO_RAW	

Connessione tra socket su comunicazione STREAM:

Un socket rappresenta una MACCHINA A STATI pilotata dall'istanza di sistema che lo ospita: tale macchina può quindi cambiare stato per effetto di una chiamata a un'API apposita.

Supponiamo ora di avere due socket A, B (anche in sistemi operativi diversi). Se relative macchine TA, TB possono essere strettamente accoppiate: in tal caso, in maniera trasparente al programmatore, se per esempio viene chiamata una system call che porta TB a cambiare stato, allora viene inviata una segnalazione all'istanza di sistema contenente A affinché anche TA cambi stato.

Questo scenario può essere attuato tramite il tipo di comunicazione di streaming e, in particolare, attraverso l'operazione di CONNESSIONE ESPlicita tra i due socket. Riconosciamo A, B e supponiamo di connettere B con A specificando l'indirizzo di quest'ultimo. Le macchine che vengono realmente accoppiate sono TB, TA', dove A' è una copia di A, ovvero una nuova istanza di socket generata dinamicamente. In particolare, i flussi di byte prodotti da B potranno essere estratti dalla copia di A e viceversa (si ha un vero e proprio tubo di connessione tra B e A').

### Backlog di connessioni:

Affinché un socket A sia in grado di accettare le connessioni verso di lui, deve trovarsi nello stato di listening, il che rende impossibile per lui entrare effettivamente in comunicazione con altri socket (cosa che invece faranno i socket che verranno generati dinamicamente).

L'accettazione di una connessione vera e propria avviene tramite il servizio accept(), che deve essere invocato periodicamente per poter instaurare più comunicazioni. In ogni caso, tra una chiamata e l'altra di questa funzione, intercorre un intervallo di tempo  $\Delta$  (speso dal thread chiamante per effettuare particolari attività, come per esempio generare una nuova traccia di esecuzione che si occupi della comunicazione tra il socket che ha lanciato la connessione e la nuova copia di A) nel quale è possibile che arrivino altre richieste di connessione. In tal caso, queste ultime possono rimanere momentaneamente pendenti finché A non sarà nuovamente disponibile oppure finché non scorrerà un timeout (che dipende dal protocollo utilizzato). In particolare, il numero massimo consentito di richieste contemporaneamente pendenti è detto backlog e viene tipicamente scelto dal programmatore.

### Connessione tra socket su comunicazione DGRAM:

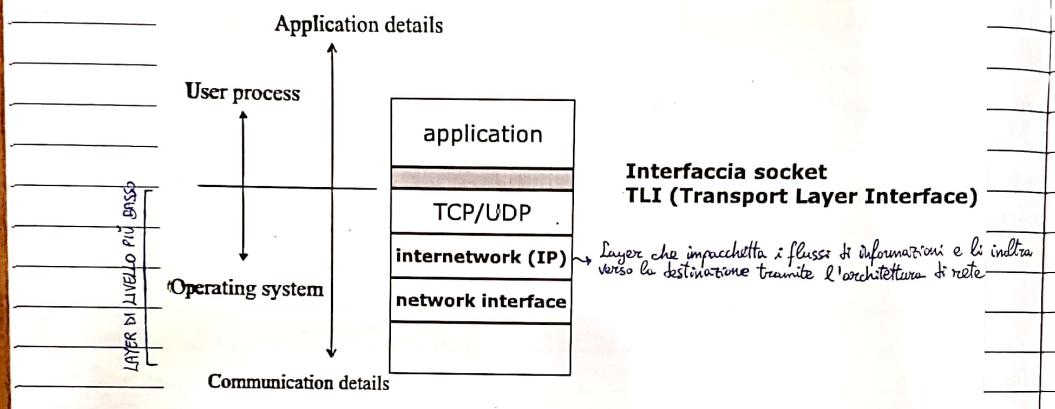
Supponiamo di avere due socket A, B caratterizzati dal tipo di comunicazione DGRAM che è basato sullo scambio di datagrammi. Stavolta, affinché B possa inviare un messaggio ad A (e viceversa), non è necessaria una connessione preliminare: i datagrammi possono essere spediti a chiunque in qualsiasi momento, per cui è necessario specificare di volta in volta l'indirizzo del socket destinatario.

Il discorso cambia nel momento in cui, per esempio, viene instaurata una connessione tra A e B: in tal caso, a meno di modifiche successive, tutti i messaggi prodotti da B arriveranno come destinazione A.

N.B.: Il modello coi datagrammi è meno affidabile del modello streaming: mentre nel secondo tutti i flussi di informazioni raggiungono la destinazione solo sotto rottura del tubo di comunicazione, nel primo i messaggi possono andare persi in modo non deterministico.



### Socket AF\_INET:



### Protocolli standard di AF\_INET:

→ TCP (TRANSMISSION CONTROL PROTOCOL)

▷ Orientato alla connessione

▷ Connessione full duplex (bidirezionale)

▷ Consegna affidabile e ordinata delle informazioni

→ UDP (USER DATAGRAM PROTOCOL)

▷ Non orientato alla connessione

▷ Consegna non affidabile e non ordinata delle informazioni

### Port number:

Nei sistemi BSD (una particolare versione di UNIX) esistono tre fasce di port number:

- 0-1023 → port number riservati ai socket delle applicazioni che girano per conto del root.

ESEMPI:

SERVER	PORT NUMBER	PROTOCOLLO
ftp	21	TCP
telnet	23	TCP
HTTP	80	TCP
snmp	161	UDP

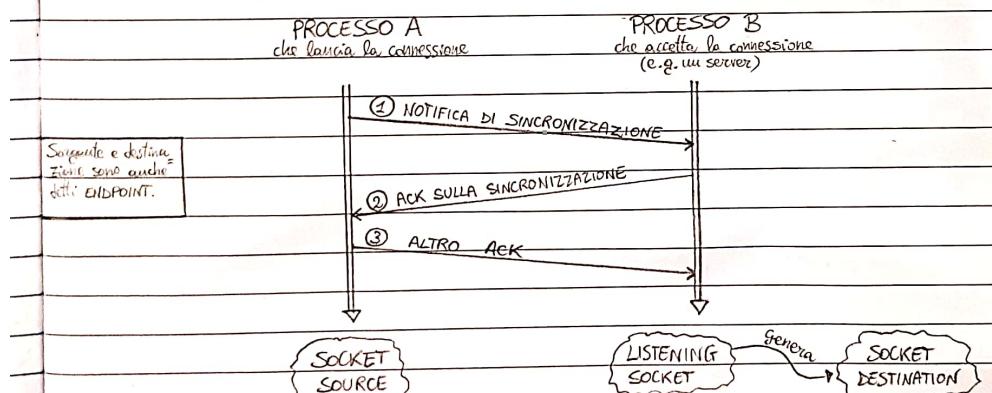
► ftp (File Transfer Protocol) → basato su un'organizzazione composta da un server e tanti client; questi ultimi possono richiedere al server di scaricare dei file tramite un modello streaming.

► HTTP (Hyper Text Transfer Protocol) → utilizzato per acquisire dati tramite browser.

- 1024-5000 → port number effimeri; vengono assegnati a un socket S automaticamente dal sistema operativo nel momento in cui S viene utilizzato senza prima essere stato associato a un traffico.
  - 5001-65535 → port number non privilegiati; la loro assegnazione non richiede particolari requisiti.

Connessione tra socket AF\_INET su comunicazione STREAM:

Arrivare in tre fasi (THREE-WAY HANDSHAKE):



- HA STESSO PORT NUMBER E STESSO IP NUMBER NEL LISTENING SOCKET.
  - A DIFFERENZA DEL LISTENING SOCKET, NON È IN ATTESA DI NUOVE CONNESSIONI, PENSY È IN COMUNICAZIONE COL SOCKET SOURCE.
  - TRA I SUOI METADATI DI GESTIONE MANTIENE L'IDENTIFICATORE DEL SOCKET SOURCE E IL PROTOCOLLO IN USO.

Tutti i flussi informativi che viaggiano tra il SOCKET SOURCE e il SOCKET DESTINATION sono caratterizzati dalla seguente quintupla:

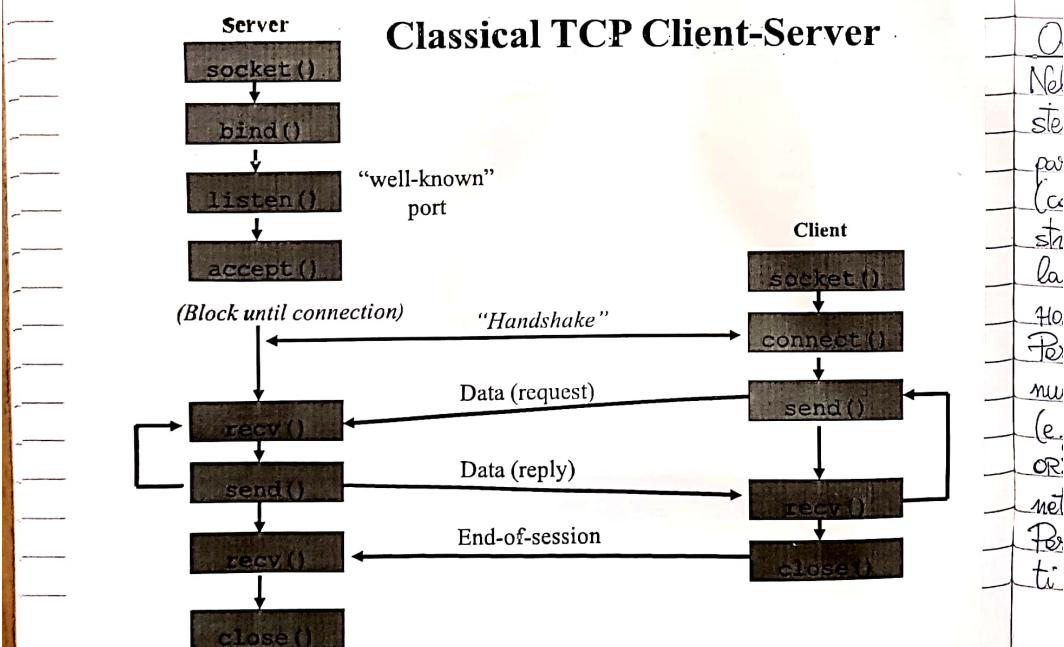
<SOURCE-PORT, SOURCE-IP, DESTINATION-PORT, DESTINATION-IP, PROTOCOL>

## Macro INADDR\_ANY:

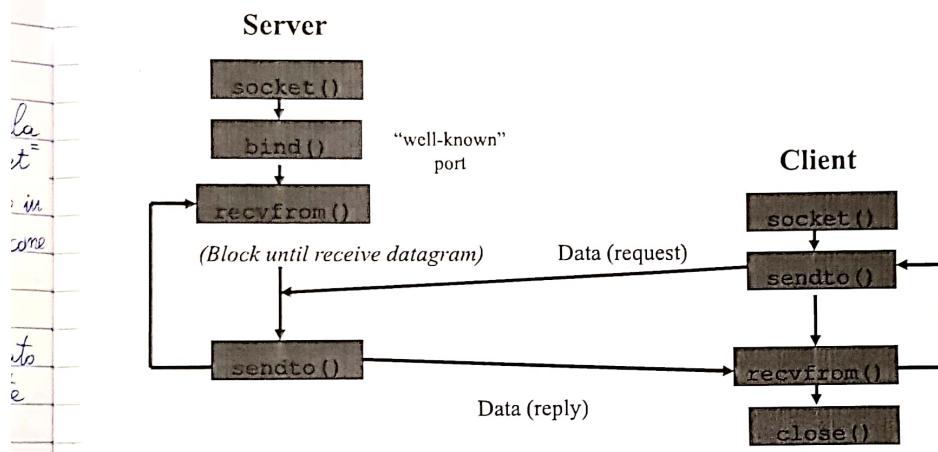
Nell'identificazione di un socket è possibile assegnare la macro INADDR\_ANY al campo relativo all'IP number. In particolare, INADDR\_ANY è un multi-indirizzo che associa il socket a tutti gli IP number del sistema sottostante. Ciò è possibile ad esempio nel momento in cui la medesima istanza di sistema è raggiungibile da più parti del networking differenti, come se ci fossero più destinazioni distinte (perché magari il sistema dispone di più schede di rete).

Un indirizzo IP particolare è 127.0.0.1, tramite cui l'host (il sistema) viene identificato all'interno di se stesso. Perciò, un socket con questo IP number è visibile esclusivamente all'interno del sistema in cui si trova, anche se il suo dominio è AF\_INET.

→ 127.0.0.1 è sempre compreso nella macro INADDR\_ANY.



## Classical UDP Client-Server



### Orientamento dei byte:

Nel dominio AF\_INET è frequente avere una comunicazione tra socket appartenenti a due sistemi operativi diversi, per cui possono essere coinvolte macchine che codificano i numeri (in particolare gli IP number e i port number) in modo differente. Infatti, esistono processori (come x86) che rappresentano i numeri con la codifica little endian (in cui il bit più a destra è il meno significativo) e, viceversa, ce ne sono altri che rappresentano i numeri con la codifica big endian. In ogni caso, i formati utilizzati sulle singole macchine sono detti HOST ORDER.

Per mantenere una coerenza e una consistenza delle informazioni come IP number e port number, è necessario che, prima dell'invocazione di API che utilizzano indirizzi di socket (e.g. `bind()`, `connect()`), questi dati siano convertiti in un formato canonico che è il NETWORK ORDER (= big endian); analogamente, il destinatario, ove necessario, deve trasformare dal network order all'host order i valori ricevuti.

Per effettuare queste conversioni, lo standard di sistema mette a disposizione le seguenti API:

- `uint16_t htons(uint16_t host16bitvalue)` → TRASFORMA UN INTERO IN HOST ORDER A 16 BIT NELLO STESSO VALORE IN NETWORK ORDER
- `uint32_t htonl(uint32_t host32bitvalue)` → TRASFORMA UN INTERO IN HOST ORDER A 32 BIT NELLO STESSO VALORE IN NETWORK ORDER
- `uint16_t ntohs(uint16_t network16bitvalue)` → TRASFORMA UN INTERO IN NETWORK ORDER A 16 BIT NELLO STESSO VALORE IN HOST ORDER
- `uint32_t ntohl(uint32_t network32bitvalue)` → TRASFORMA UN INTERO IN NETWORK ORDER A 32 BIT NELLO STESSO VALORE IN HOST ORDER

The end.