Sistemi operativi

2. Processi e thread

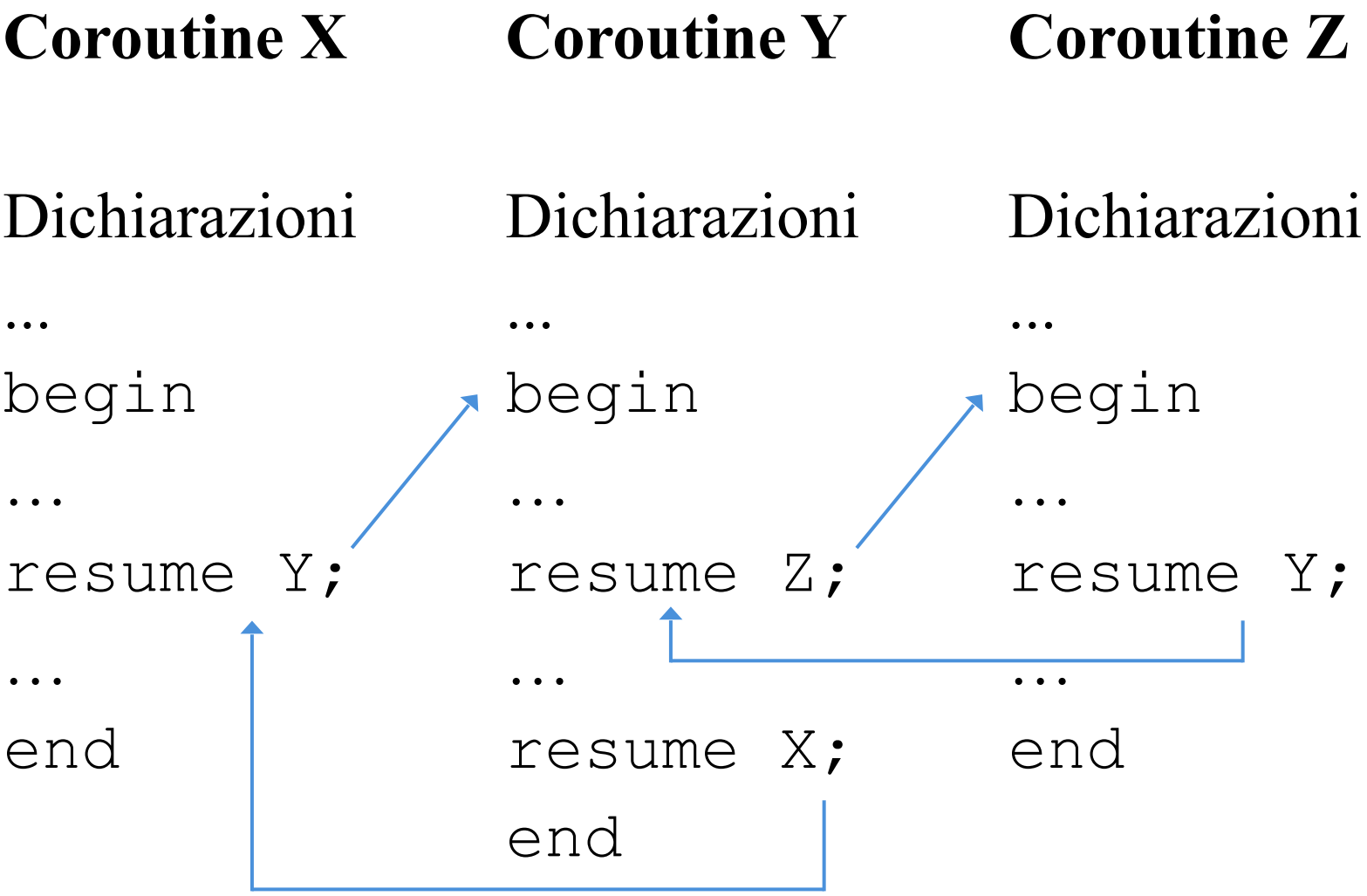
Mauro Fiorentini

2. Processi e thread

Coroutines

- Una forma primitiva, oggi poco usata, di pseudo-multiprogrammazione.
  - Il programmatore individua i punti nei quali un thread deve cedere il controllo e a chi cederlo.
  - Quando un thread riceve il controllo, riparte da dov'era arrivato.
  - Serve un costrutto per la dichiarazione di coroutine e uno per la cessione di controllo.
- Introdotte da Simula (Dahl 1968) e riprese da BLISS (1971) e Modula 2 (1977).

Esempio di coroutine



Caratteristiche delle coroutine

- Il passaggio di controllo esplicito ricorda quello delle chiamate di funzione, ma il controllo potrebbe non ritornare mai a chi lo cede.
  - Non esiste un meccanismo a stack.
- Non esprime un vero parallelismo.
- Riprese in tempi recenti col nome “fiber”.

Processo

- Un processo è un programma in esecuzione.
  - Ha un flusso di controllo, determinato dalla sequenza di istruzioni eseguite.
  - Ha uno stato, costituito dallo stato della memoria, dei registri e delle risorse di sistema allocate.
- Può coesistere con altri processi sulla stessa macchina.

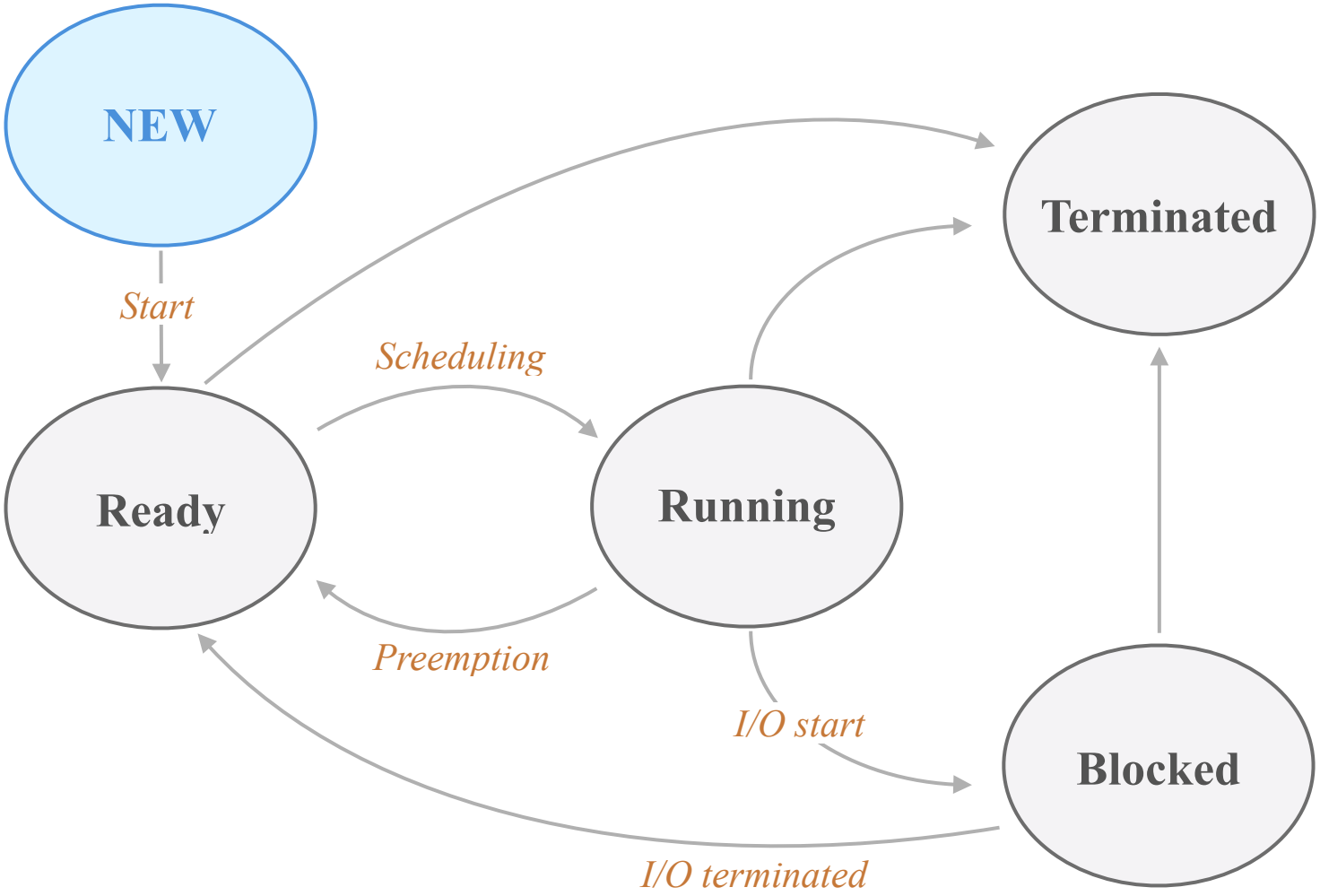
Stati di un processo (1)

- Un processo in ogni istante si trova in uno dei seguenti stati:
  - **new**: creato, ma non ancora fatto partire;
  - **running**: utilizza la CPU;
  - **ready**: in attesa della CPU; non attivo solo per mancanza di risorse;
  - **waiting**: in attesa di un evento che gli permetta di ripartire;
  - **terminated**: terminato irreversibilmente.

Stato di un processo (2)

- Il numero di processi running non può superare il numero di CPU presenti.
- Il numero massimo di processi negli altri stati dipende dalle risorse del sistema.

Transizioni tra gli stati



Transizioni tra gli stati (1)

- Un processo viene creato nello stato **new**, passa in stato **ready** con una *start*.
- Un processo passa nello stato **terminated** quando viene ucciso da una *kill*, dopodiché viene distrutto.
- In molti sistemi queste coppie di azioni sono eseguite da una sola primitiva per ciascuna.

Transizioni tra gli stati (2)

- Un processo **running** passa nello stato **blocked** quando:
  - inizia un'operazione di I/O o che comunque implichi l'attesa di un evento;
  - richiede di essere sospeso per un certo tempo;
  - un altro processo lo sospende.
- Un processo **blocked** diventa **ready** quando:
  - l'operazione di I/O termina o l'evento atteso si verifica.
  - il tempo prefissato è trascorso;
  - un altro processo lo sblocca.

Transizioni tra gli stati (3)

- Un processo **running** diventa **ready** quando il sistema decide di togliergli il controllo della CPU.
- Un processo **ready** diventa **running** quando il sistema gli assegna il controllo della CPU.
  - La cessione della CPU a un altro processo è sempre innescata da un interrupt, da un trap o da una system call.

Altri stati

- In un sistema reale sono possibili altri stati:
  - **sleeping**: una sorta di blocked “a tempo”; il processo diventa ready dopo un tempo determinato.
  - **waiting**: il processo attende che un semaforo, non necessariamente connesso all’I/O, gli dia via libera.
- Si tratta di varianti dello stato waiting, con le relative transizioni.

Immagine di un processo

- Consiste di:
  - codice;
  - valori dei dati;
  - contenuto dello stack;
  - valori dei registri;
  - risorse di sistema allocate e loro stato;
  - ogni altra informazione necessaria all'esecuzione del processo.

Process table

- Struttura dati gestita dal sistema operativo, contenente le informazioni relative a tutti i processi esistenti.
  - Di solito è un vettore o una lista.
- Per ogni processo viene allocata una struttura identica.
  - I dati contenuti dipendono dal sistema in esame, anche se una parte è comune a tutti i sistemi.

Dati nella process table

- Dati relativi al processo:
  - Es.: stato, registri, flag, ora di inizio, tempo di CPU utilizzato, id, priorità.
- Dati relativi alla memoria:
  - quantità di memoria allocata, indirizzo di inizio o della page table.
- Dati relativi al file system:
  - Es.: id utente, file aperti.
- Dati relativi alle risorse allocate (eventualmente condivise):
  - Es.: semafori.

Race condition

- Situazione nella quale l'esito dipende dalla sequenza temporale di eventi asincroni.
- Esempio:
  - un processo scrive una cella di memoria e un altro la legge;
  - in generale è impossibile prevedere se il secondo leggerà prima o dopo che il primo abbia scritto;
  - esecuzioni successive, sulla stessa macchina, possono dare esiti differenti.

Comunicazione tra processi

- Può essere effettuata:
  - con memoria condivisa, alla quale i processi accedono liberamente;
  - con messaggi, trasferiti da un processo all'altro dal sistema.
- In ogni caso è necessaria una forma di sincronizzazione, per evitare pericolose "race condition".

Problemi classici

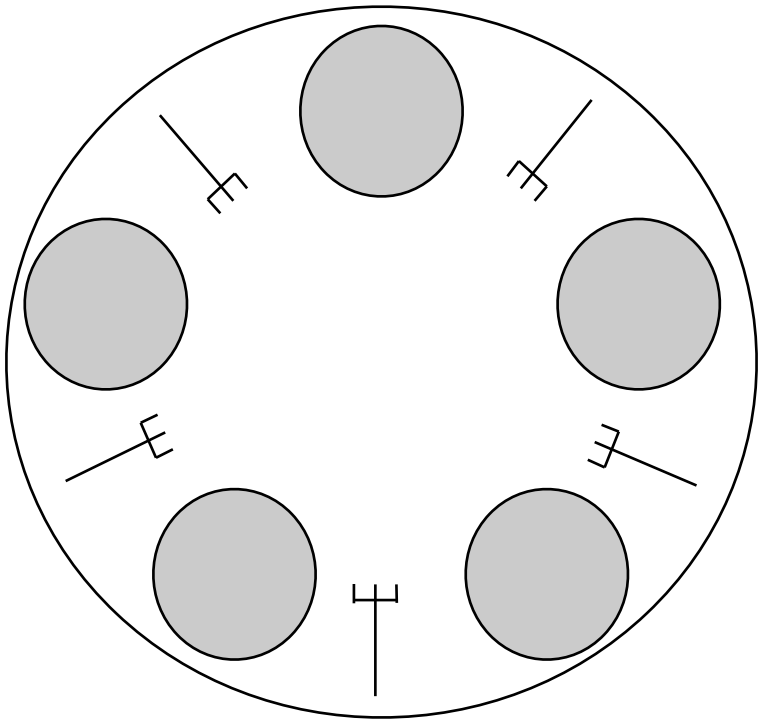
- Alcuni problemi, versione idealizzata di problemi reali, sono divenuti famosi e sono usati come banco di prova delle primitive di sincronizzazione:
  - produttore e consumatore.
  - cinque filosofi;
  - lettori e scrittori;
  - il negozio del barbiere;
  - i fumatori.

Produttore e consumatore

- Noto anche come “problema del buffer limitato”.
- Un classico problema di sincronizzazione.
- Due processi, che operano a velocità diverse e imprevedibili, condividono un buffer:
  - uno genera dati nel buffer;
  - l’altro li utilizza.
- Se il buffer è pieno o vuoto, uno dei processi va fermato.

Cinque filosofi

Problema proposto da Dijkstra nel 1965



- Cinque filosofi, seduti a un tavolo rotondo, alternano pranzo e meditazione.
- Sul tavolo 5 piatti si alternano a 5 posate.
- Servitori discreti mantengono i piatti sempre pieni.
- Ogni filosofo necessita, per poter mangiare, delle due posate adiacenti al suo posto.

Condizioni del problema

- I filosofi decidono quando mangiare in momenti asincroni e imprevedibili.
- Se nessuno sta mangiando, il primo che lo desideri deve poter mangiare.
- Bisogna evitare situazioni di **deadlock**.
  - Es.: ciascuno prende la forchetta alla sua destra e attende che quella alla sinistra sia libera.
- Bisogna evitare la **starvation**.
  - Un filosofo non riesce a mangiare per un tempo indefinito.

Lettori e scrittori

- Numerosi processi cercano di accedere simultaneamente a un database.
- Alcuni desiderano scrivere, altri leggere.
- Le letture possono avvenire contemporaneamente.
- Le scritture bloccano ogni altra operazione fino al loro termine.
  - Come alternativa, si possono permettere letture a record differenti durante le scritture.

Condizioni del problema

- Si deve permettere il massimo parallelismo possibile.
- Nessuna operazione deve essere ritardata indefinitamente.

Il negozio del barbiere

- Nel negozio di un barbiere vi sono *n* sedie per attendere e una per farsi radere.
- Il barbiere rade i clienti; in loro assenza dorme.
- Quando un cliente arriva:
  - se è il primo, deve svegliare il barbiere e farsi radere;
  - altrimenti si siede su una delle sedie;
  - se tutte sono occupate, esce senza farsi radere.

I fumatori

- Per fumare servono tabacco, carta e fiammiferi.
- Un tabaccaio ha 3 clienti fumatori.
- Ciascun fumatore dispone di un diverso oggetto (in quantità infinita) e deve acquistare gli altri due.

Condizioni del problema

- Il tabaccaio pone in vendita due diversi oggetti scelti a caso.
- Il fumatore che necessita di quella coppia deve essere sbloccato.
- Dopo l'acquisto, il fumatore va a fumare per un tempo casuale e il tabaccaio pone in vendita un'altra coppia di oggetti.

Mutua esclusione

- Meccanismo che permette di evitare race condition, impedendo a più processi un accesso simultaneo a risorse condivise.
- Il programmatore identifica regioni critiche e azioni atomiche.
- Il sistema fornisce gli strumenti per implementarle.
  - Il sistema operativo le può utilizzare al suo interno.

Regioni critiche

- Sono porzioni di codice, identiche in più processi, nelle quali in ogni istante può trovarsi al massimo un processo.
  - Il processo che si trova in una regione critica costringe gli altri che vogliano entrarvi ad attendere che ne esca.
  - Da una regione critica si può accedere a un'altra, sempreché libera.

Azioni atomiche

- Sono porzioni di codice eseguite da un processo senza subire interruzioni da parte degli altri.
  - In pratica sono equivalenti a singole istruzioni macchina.

Condizioni per una corretta sincronizzazione

- Due processi non possono trovarsi contemporaneamente in una regione critica.
- Non deve essere fatta alcuna ipotesi sulle velocità di esecuzione dei processi.
- Nessun processo che si trovi all'esterno di una sezione critica ne può bloccare altri.
- A nessun processo dev'essere negato indefinitamente l'accesso a una regione critica.

Caratteristiche di una soluzione corretta

- Al massimo un processo per volta può trovarsi nella sezione critica.
- Se due o più processi tentano di entrare nella sezione critica e nessuno la occupa, a uno dei richiedenti deve essere permesso l'accesso.
- A un processo non deve essere negato indefinitamente l'accesso a una sezione critica.

Implementazioni software delle regioni critiche

- Esistono soluzioni puramente software al problema delle regioni critiche:
  - disabilitazione degli interrupt;
  - alternanza;
  - lock;
  - algoritmi di Dekker e Peterson.
- Si basano su due funzioni o blocchi di codice, da chiamare all'entrata e all'uscita delle regioni critiche.
  - Se un processo bara, tutte falliscono.

Disabilitazione degli interrupt

- Un processo disabilita gli interrupt entrando in una sezione critica, li riabilita uscendo.
- Nulla può interrompere l'esecuzione della sezione critica e nessun altro processo può prendere il controllo della CPU.
  - Si devono assolutamente evitare trap nelle regioni critiche.
- Soluzione semplicissima da implementare.

Svantaggi della disabilitazione degli interrupt

- Concedere ai processi il controllo degli interrupt è pericoloso.
- Un solo processo mal progettato può mettere in crisi il sistema.
- Si rischia di inibire l'I/O per tempi troppo lunghi.
- Soluzione attuata solo in sistemi molto semplici.
- Soluzione inadatta ai multiprocessori.

Sistemi e disabilitazione degli interrupt

- I sistemi utilizzano frequentemente questa soluzione al loro interno, per realizzare piccole regioni critiche, che accedono a variabili e tabelle interne.
- Il codice eseguito nelle regioni critiche deve essere affidabile e veloce.

Lock

- Si utilizza una variabile booleana di lock per ogni regione critica.
- Prima di entrare in una regione critica, ogni processo esamina la variabile di lock:
  - se falsa, la pone a vera ed entra;
  - se vera, riprova.
- Uscendo, i processi rimettono a falsa la variabile di lock.

Svantaggi dei lock

- Idea non utilizzabile nella forma più semplice.
- Si verifica una race condition nell'accesso alla stessa variabile di lock.

Implementazione hardware dei lock

- Per implementare le regioni critiche molte CPU dispongono di istruzioni apposite:
  - test and set (tset): esamina il valore di una cella di memoria e le assegna comunque un valore noto;
  - swap: scambia un registro con una cella di memoria.
- Permettono di implementare i lock senza disabilitare gli interrupt.

Lock con test and set

```
volatile        unsigned int    lock = 0;

void    enter(void)
        {
        while (tset(&lock) != 0);
        }

void    leave(void)
        {
        lock = 0;
        }
```

Lock con swap

```
volatile        unsigned int    lock = 0;

void    enter(void)
        {
        while (swap(&lock, 1) != 0);
        }

void    leave(void)
        {
        lock = 0;
        }
```

Controindicazioni di tset e swap

- Non tutti i processori dispongono di tset o swap.
- La loro implementazione è complicata in presenza di cache.
- La necessità di bloccare il bus per due accessi consecutivi crea complicazioni.
  - In particolare nei sistemi multiprocessori.

Ottimizzazione con swap

- In ambiente multiprocessore può convenire una versione leggermente diversa:

```
void        enter(void)
            {
            while (lock != 0);
            while (swap(&lock, 1) != 0);
            }
```

- In questo modo si riducono le scritture e quindi gli aggiornamenti alle cache degli altri processori.
  - Generalmente il secondo ciclo, che scrive, verrà eseguito una sola volta.

Implementazione software di tset e swap

- Se non disponibili in hardware, tset e swap possono essere efficacemente simulate via software, disabilitando gli interrupt.
  - Normalmente implementate in assembler.
- Soluzione non valida in macchine multiprocessore.

Test and set in software

```
unsigned int   tset(unsigned int *p)
        {
        unsigned int     value;
        status_type      status;

        status = disable();
        value = *p;
        *p = 1;
        restore(status);
        return value;
        }
```

Swap in software

```
unsigned int    swap(unsigned int *p,
                     unsigned int new_value)
        {
        unsigned int      value;
        status_type       status;

        status = disable();
        value = *p;
        *p = new_value;
        restore(status);
        return value;
        }
```

Alternanza

- Si utilizza una variabile che stabilisce chi abbia diritto di entrare nella regione critica.
- Solo il processo che ne ha diritto può entrare.
- All'uscita la variabile di controllo viene modificata, per permettere l'ingresso del prossimo processo.

Caratteristiche dell'alternanza

- Soluzione corretta.
- Se un processo richiede più accessi alla regione critica degli altri o è più veloce, viene penalizzato.
- Un processo può avere diritto d'accesso quando non gli serve, bloccandone invece un altro che tenta di entrare nella regione critica.

L’algoritmo di Dekker

- Dekker fu il primo a proporre una soluzione interamente software nel 1965.
  - La versione riportata è semplificata e trascritta in un linguaggio strutturato.
  - L’idea base è stabilire con una variabile (`turn`) a chi tocchi entrare in caso di conflitto.
- Valido solo per due processi.
- Ciascuno deve conoscere il proprio ID (`me`) e quello dell’altro (`other`).

L'algoritmo di Dekker: dichiarazioni

```
typedef enum { COMPETING, OUT } status_type;

status_type     status [2] = { OUT, OUT };
volatile        sig_atomic_t turn;
unsigned int    me;
unsigned int    other;
```

L'algoritmo di Dekker: il codice

```
status [me] = COMPETING;
while (status [other] == COMPETING)
        {
        if (turn == other)
                {
                status [me] = OUT;
                while (turn != me)
                   wait();
                status [me] = COMPETING;
                }
        }
critical_section();
turn = other;
status [me] = OUT;
```

L’algoritmo di Peterson

- Peterson propose nel 1981 una soluzione molto più semplice, basata su due funzioni da chiamare all’ingresso e all’uscita.
  - L’idea base è stabilire con una variabile quale processo sia in coda e debba aspettare;
  - in caso di esecuzione simultanea della funzione `enter`, l’ultimo processo a scrivere nella variabile di controllo attende.

L'algoritmo di Peterson: ingresso

```
volatile        sig_atomic_t    waiting;
volatile        unsigned int    interested [2] =
                    { FALSE, FALSE };

void    enter(unsigned int process)
        {
        interested [process] = TRUE;
        waiting = process;
        while (waiting == process &&
                interested [1 - process]);
        }
```

L'algoritmo di Peterson: uscita

```
void    leave(unsigned int process)
        {
        interested [process] = FALSE;
        }
```

Note sugli algoritmi di Dekker e Peterson

- Non generalizzabili a più processi.
- Richiedono accessi atomici alla variabile che controlla il turno di accesso.
  - Si possono implementare correttamente in pochi linguaggi.
    - Può essere necessario scrivere una parte in assembler.
  - In C e C++ le variabili `turn` e `waiting` devono essere dichiarate di tipo `volatile sig_atomic_t`, le altre variabili condivise devono essere dichiarate `volatile`.

Busy waiting

- Lock, alternanza e algoritmo di Peterson implicano "busy waiting":
  - quando un processo si vede rifiutato l'ingresso, ritenta immediatamente;
  - in questo modo si spreca inutilmente molto tempo macchina.
- Il busy waiting è ammissibile solo per tempi molto brevi.

Inversione di priorità

- Un problema che può presentarsi col busy waiting.
- Se un processo ad alta priorità diventa ready mentre un altro a priorità minore è in una regione critica, il primo acquista il controllo della CPU, impedendo all'altro di uscire dalla regione critica.
  - Inizia un ciclo infinito di busy waiting.

Mutua esclusione senza busy waiting (1)

- Sono stati proposti vari metodi per evitare che il processo che attende di entrare in una regione critica consumi tempo di CPU:
  - sleep e wakeup;
  - semafori;
  - contatori di eventi;
  - monitor;
  - scambio di messaggi.

Mutua esclusione senza busy waiting (2)

- La caratteristica comune a tutte le soluzioni è che utilizzano primitive implementate **all'interno del sistema**, che può quindi mettere in stato blocked i processi che devono attendere e cedere il controllo della CPU ad altri.
  - Si utilizza di solito una coda di processi in attesa per ogni risorsa di sincronizzazione.

Sleep e wakeup

- Soluzione basata su due primitive:
  - `sleep` sospende il processo che la esegue;
  - `wakeup` riattiva il processo designato.
- Le primitive devono essere atomiche.

Produttore e consumatore con sleep e wakeup

- Il produttore:
  - quando trova il buffer pieno, si sospende;
  - quando ha prodotto un dato, sveglia il consumatore, se questi era sospeso.
- Il consumatore:
  - quando trova il buffer vuoto, si sospende;
  - quando ha consumato un dato, sveglia il produttore, se questi era sospeso.

Produttore

```
void    producer(void)
        {
        data_type  item;

        for (;;)
                {
                item = produce();
                if (count == MAX)
                    sleep();
                put_item(&item);
                if (count++ == 0)
                    wakeup(consumer_id);
                }
        }
```

Consumatore

```
void    consumer(void)
        {
        data_type  item;

        for (;;)
                {
                if (count == 0)
                   sleep();
                item = get_item();
                if (count-- == MAX)
                   wakeup(producer_id);
                consume(&item);
                }
        }
```

Note all'implementazione

- Gli accessi al contatore `count` vanno comunque protetti entro regioni critiche per evitare race condition.
- L'implementazione suggerita ha un difetto: se un processo esegue una `wakeup` quando l'altro ha determinato di dover eseguire una `sleep`, **ma non l'ha ancora eseguita**, la `wakeup` non ha effetto e si arriva a un blocco dei due processi.

Soluzione corretta con sleep e wakeup

- Bisogna aggiungere un flag (`wakeup_pending`) che tiene traccia di eventuali `wakeup` eseguite prima della corrispondente `sleep`.
  - La `sleep` esamina il flag e, se lo trova vero, lo mette a falso invece di sospendere il processo.
- La generalizzazione a *n* processi si complica.
  - Servono *n* flag di wakeup pendenti per ogni processo.

I semafori

- Nel 1965 Dijkstra propose di utilizzare un contatore intero di wakeup pendenti per ogni processo.
- Introdusse nella sua proposta il concetto di semaforo e due primitive: `up` e `down`, generalizzazioni di `sleep` e `wakeup`.
  - `down` (o `wait`) decrementa il contatore; se il risultato è negativo, sospende il processo che la esegue;
  - `up` (o `signal`) incrementa il contatore; se prima dell'operazione era negativo, riattiva un processo in attesa.

Caratteristiche dei semafori

- A ogni semaforo sono associati un contatore intero e una coda di processi.
- Il contatore indica (in valore assoluto):
  - se negativo, il numero di processi in attesa;
  - se positivo, il numero di risorse libere.
- Le primitive devono essere atomiche.
- Possono risolvere sia i problemi di sincronizzazione che quelli di mutua esclusione.

P e V

- I nomi proposti da Dijkstra erano P, dall'olandese *passeren* (passare) e V, dall'olandese *vrygeren* (rilasciare), abbreviazioni rimaste nella letteratura, anche se pochi ne ricordano l'origine.

Semafori binari

- Caso particolare, nel quale il contatore assume solo i valori vero o falso.
  - Il booleano indica la disponibilità della risorsa.
  - Servono a impedire a due o più processi di entrare simultaneamente in una regione critica.
- La soluzione si basa su due primitive:
  - `wait` se trova il valore vero, lo mette a falso, altrimenti sospende il processo che la esegue;
  - `signal` riattiva un processo in attesa; se non ve ne sono altri, mette a vero il valore.

Mutua esclusione con up e down

```
semaphore mutex;

        down(&mutex);
                // regione critica
        up(&mutex);
```

Produttore e consumatore con up e down

- Si usano 3 semafori:
  - `empty`, che conta il numero di posti disponibili nel buffer, inizializzato con la dimensione del buffer;
  - `full`, che conta il numero di posti occupati nel buffer, inizializzato a zero;
  - `mutex`, semaforo binario inizializzato a 1, che evita che i due processi accedano simultaneamente al buffer e al contatore di elementi presenti.

Produttore

```
void    producer(void)
        {
        data_type  item;

        for (;;)
                {
                item = produce();
                down(&empty);
                down(&mutex);
                put_item(&item);
                up(&mutex);
                up(&full);
                }
        }
```

Consumatore

```
void    consumer(void)
        {
        data_type  item;

        for (;;)
                {
                down(&full);
                down(&mutex);
                item = get_item();
                up(&mutex);
                up(&empty);
                consume(&item);
                }
        }
```

Scelta del processo da riattivare

- Può essere scelto casualmente.
- Può essere il processo a massima priorità.
- Più comunemente è il processo in coda da più tempo.
  - La coda è implementata con una lista e gestita con tecnica FIFO.

Contatori di eventi

- Proposti da Reed e Kanodia nel 1979.
- Soluzione basata su di un contatore, inizialmente azzerato e poi solo incrementato.
- Soluzione basata su tre primitive:
  - `read` legge il contatore;
  - `advance` lo incrementa di 1;
  - `await` sospende il processo che la esegue, finché il contatore raggiunge o supera l'argomento.

Soluzione con contatori di eventi

- Si usano 2 contatori:
  - `in`, che conta il numero di elementi messi nel buffer;
  - `out`, che conta il numero di elementi tolti dal buffer.
- Produttore e consumatore lavorano su elementi differenti del buffer e non necessitano ulteriori sincronizzazioni.

Produttore

```
void    producer(void)
        {
        data_type           item;
        unsigned int        in_count = 0;

        for (;;)
                {
                item = produce();
                ++in_count;
                await(out, in_count - MAX);
                put_item(&item, in_count % MAX);
                advance(&in);
                }
        }
```

Consumatore

```
void    consumer(void)
        {
        data_type           item;
        unsigned int        out_count = 0;

        for (;;)
                {
                ++out_count;
                await(in, out_count);
                item = get_item(out_count % MAX);
                advance(&out);
                consume(&item);
                }
        }
```

Note all'implementazione

- I contatori non vengono mai decrementati.
- Si devono usare interi, non negativi, di dimensione tale da non avere problemi di overflow.
  - Utilizzando 64 bit, con un milione di incrementi al secondo, l'overflow si verifica dopo oltre mezzo milione di anni.
- L'implementazione può essere scomoda in molti linguaggi.

Monitor

- Proposti da Hoare nel 1974 e Brinch Hansen nel 1975.
- Soluzione basata su una collezione di procedure, variabili, strutture dati e lock, chiamata “monitor”.
  - Una classe ante litteram.

Caratteristiche di un monitor

- Per entrare nel monitor un processo deve acquisire una chiave (lock).
  - Solo un processo alla volta può entrare in una qualunque delle procedure del monitor; gli altri vengono bloccati e attendono.
  - Si risolve il problema della mutua esclusione.
- Entro un monitor si possono dichiarare variabili di condizione, sulle quali sono ammesse le operazioni `wait` e `signal`.
  - Si risolve il problema della sincronizzazione.

Implementazione dei monitor

- L'implementazione proposta è a livello di linguaggio, opportunamente esteso.
  - I programmatori hanno meno problemi, possono commettere meno errori.
    - Non è necessario preoccuparsi della corretta collocazione, sequenza e inizializzazione delle funzioni di sincronizzazione.
  - I processi non possono "barare", se scritti nel linguaggio esteso.
    - Il compilatore riconosce i monitor e genera le necessarie chiamate al sistema.

I monitor nei linguaggi

- Soluzione complessa da implementare.
  - Bisogna scrivere o modificare un compilatore.
- Pochi linguaggi l'hanno adottata:
  - Concurrent Pascal, Brinch Hansen 1975.
  - Concurrent Euclid, Holt 1983.
  - Java, 1996.
- Le altre soluzioni sono implementate mediante chiamate a funzioni, che il compilatore può trattare come chiamate qualsiasi.

Wait e signal

- Analoghe a `sleep` e `wakeup`:
  - `wait` blocca il processo che la esegue, mettendolo in attesa che una condizione si verifichi;
  - `signal` sblocca il processo eventualmente in attesa su di una condizione.
- La `wait` rilascia il lock, per permettere a un altro processo di entrare.
- Il processo risvegliato da una `signal` acquisisce automaticamente il lock.

Le condizioni

- Non sono semafori.
  - Se si esegue una `signal` su di una condizione sulla quale nessun processo è in attesa, il segnale non ha effetto.
  - Non hanno memoria.
- Nessun problema per il programmatore, che può tener traccia dello stato dei processi con variabili interne al monitor e quindi ad accesso atomico.

Signal secondo Hoare

- Nella proposta di Hoare la `signal` rende immediatamente running il processo riattivato.
  - Soluzione complessa, perché il processo potrebbe comunque essere sospeso subito dopo.
  - Vi potrebbero essere due processi contemporaneamente attivi nel monitor.

Signal secondo Brinch Hansen

- Nella proposta di Brinch Hansen la `signal` provoca atomicamente anche l'uscita dal monitor.
  - Concettualmente più semplice.

Differenza tra wait e sleep

- La `wait` permette a un altro processo di entrare nel monitor.
- L'errore si verifica con `sleep` e `wakeup` se un processo esegue una `wakeup` prima che l'altro abbia eseguito una `sleep`.
- Con i monitor il pericolo non sussiste, perché solo un processo alla volta può trovarsi attivo nel monitor.

Produttore e consumatore con monitor (1)

```
monitor ProducerConsumer
        condition  full, empty;
        integer count;

        procedure enterItem;
        begin
                if count = Max then
                   wait(full);
                putItem;
                count := count + 1;
                if count = 1 then
                   signal(empty);
        end;
```

Produttore e consumatore con monitor (2)

```
        procedure removeItem;
        begin
                if count = 0 then
                   wait(empty);
                getItem;
                count := count - 1;
                if count = Max - 1 then
                   signal(full);
        end;

        count := 0; { inizializzazione }
end monitor;
```

Produttore con monitor

```
procedure producer;
begin
        while true do
                begin
                    produceItem;
                    ProducerConsumer.enterItem;
                end;
end;
```

Consumatore con monitor

```
procedure consumer;
begin
        while true do
                begin
                    ProducerConsumer.removeItem;
                    consumeItem;
                end;
end;
```

Note sull'implementazione

- Il linguaggio dev'essere esteso, aggiungendo le parole chiave: `monitor`, `condition`, `wait`, `signal`.
- Nei linguaggi a oggetti il monitor può essere implementato in modo naturale con una classe (es.: Java), aggiungendo una parola chiave per identificare i metodi all'interno dei quali può trovarsi un solo processo per volta.

Implementazione dei monitor con semafori (1)

- Servono 4 funzioni, corrispondenti alle operazioni chiave:
  - ingresso nel monitor,
  - uscita dal monitor,
  - `wait`,
  - `signal`.
- Il compilatore provvede a inserire le relative chiamate.
- Bisogna considerare la possibile ricorsione.

Implementazione dei monitor con semafori (2)

```
   // Generic monitor function
int function(int argument)
        {
        unsigned int     flag;

        flag = enter_monitor();
                ...
                   // function body
                ...
        exit_monitor(flag);
        }
```

Implementazione dei monitor con semafori (3)

```
semaphore       lock = 1;
unsigned int    pid = NONE;

unsigned int    enter_monitor(void)
        {
        if (pid == get_pid())
                return FALSE;
        down(&lock);
        pid = get_pid();
        return TRUE;
        }
```

Implementazione dei monitor con semafori (4)

```
void    exit_monitor(unsigned int flag)
        {
        if (flag)
                {
                pid = NONE;
                up(&lock);
                }
        }
```

Implementazione dei monitor con semafori (5)

```
void    wait(semaphore *s)
        {
        pid = NONE;
        up(&lock);
        down(s);
        pid = get_pid();
        down(&lock);
        }

void    signal(semaphore *s)
        {
        up(s);
        }
```

Problemi con i monitor

- I programmatori tendono a implementare gran parte della soluzione del loro problema all'interno dei monitor.
  - Poiché un solo processo alla volta può trovarsi in un monitor, si possono bloccare gli altri per tempi lunghi.
  - L'implementazione tende a essere monolitica e pesante, con scarso parallelismo.
- Bisogna cambiare linguaggio di programmazione.
  - Spesso il nuovo linguaggio è presentato come estensione di uno esistente.

Difetti delle varie soluzioni

- Semafori e soluzioni simili sono a basso livello, scomode, adattabili a macchine con memoria condivisa, ma non a macchine multiprocessore senza memoria condivisa.
- I monitor richiedono un adattamento del linguaggio.
- Nessuna soluzione fornisce un modo per scambiare informazioni tra processori differenti.

Scambio di messaggi

- Soluzione basata su due primitive:
  - `send` invia un messaggio a un processo;
  - `receive` riceve un messaggio o sospende il processo che la esegue, fino all'arrivo di un messaggio.
- Il messaggio può essere un dato semplice (intero, puntatore) o una intera struttura di dati (record).

Prima soluzione con scambio di messaggi

- Il produttore:
  - quando ha prodotto un dato, spedisce un messaggio, costituito dal dato stesso.
- Il consumatore:
  - quando non trova messaggi, si sospende;
  - quando ha consumato un dato, acquisisce un altro messaggio, costituito dal prossimo dato.

Produttore

```
void    producer(void)
        {
        data_type  item;
        message m;

        for (;;)
                {
                item = produce();
                build_message(&m, &item);
                send(consumer_id, &m);
                }
        }
```

Consumatore

```
void    consumer(void)
        {
        data_type  item;
        message m;

        for (;;)
                {
                receive(producer_id, &m);
                item = extract_item(&m)
                consume(&item);
                }
        }
```

Note all'implementazione

- Se i processi si trovano su calcolatori differenti e il consumatore è più lento, il produttore ne satura la coda di messaggi.
- In molte implementazioni alcuni messaggi andrebbero persi.

Soluzione corretta con scambio di messaggi

- Il produttore:
  - riceve messaggi, costituiti da “scatole vuote”;
  - quando ha prodotto un dato, spedisce un messaggio, costituito dal dato stesso.
- Il consumatore:
  - quando non trova messaggi, si sospende;
  - quando ha consumato un dato, rimanda al produttore il messaggio, come “scatola vuota” e acquisisce un altro messaggio, costituito dal prossimo dato.

Produttore

```
void    producer(void)
        {
        data_type  item;
        message m;

        for (;;)
                {
                item = produce();
                receive(consumed_id, &m);
                build_message(&m, &item);
                send(consumer_id, &m);
                }
        }
```

Consumatore (1)

```
void    consumer(void)
        {
        data_type  item;
        message m;

        initialize();
        for (;;)
                {
                receive(producer_id, &m);
                item = extract_item(&m)
                send(producer_id, &m);
                consume(&item);
                }
        }
```

Consumatore (2)

```
void    initialize(void)
        {
        unsigned int     n;
        message m;

                // send MAX empty slots
        for (n = 0; n < MAX; ++n)
                send(producer_id, &m);
        }
```

Varianti sullo scambio di messaggi

- Il mittente può essere ignorato dalla `receive` (manca il primo argomento).
- I messaggi possono avere lunghezza fissa oppure variabile.

Mailbox

- Una mailbox è un contenitore per una coda di messaggi.
  - I messaggi vengono prelevati in ordine di arrivo.
- Ha capienza finita, specificata alla creazione.
  - Quando un processo tenta di accodare messaggi oltre la capienza viene sospeso, finché un messaggio viene prelevato.

Mailbox e processi

- Le mailbox non appartengono a uno specifico processo.
  - Qualsiasi processo può inviare o prelevare messaggi.
  - I meccanismi di protezione possono consentire di imporre restrizioni.
- Possono essere dinamicamente create e distrutte.

Produttore con mailbox

```
void    producer(void)
        {
        data_type  item;
        message m;

        for (;;)
                {
                item = produce();
                receive(empty_id, &m);
                build_message(&m, &item);
                send(full_id, &m);
                }
        }
```

Consumatore con mailbox (1)

```
void    consumer(void)
        {
        data_type  item;
        message m;

        initialize();
        for (;;)
                {
                receive(full_id, &m);
                item = extract_item(&m)
                send(empty_id, &m);
                consume(&item);
                }
        }
```

Consumatore con mailbox (2)

```
void    initialize(void)
        {
        unsigned int     n;
        message m;

        empty_id = create_mailbox(MAX);
        full_id = create_mailbox(MAX);

                // send MAX empty slots
        for (n = 0; n < MAX; ++n)
                send(empty_id, &m);
        }
```

Pipe

- Un pipe è una mailbox, a capienza finita, di messaggi di lunghezza arbitraria, nella quale non vengono conservati i confini tra i messaggi.
- I messaggi vengono scritti come sequenze di byte; il ricevente deve sapere quanti byte prelevare.
- Se la scrittura riempie il pipe, il mittente viene sospeso.

Produttore con pipe

```
void    producer(void)
        {
        data_type  item;

        pipe_id = create_pipe();
        for (;;)
                {
                item = produce();
                write_pipe(pipe_id, &item,
                    sizeof(item));
                }
        }
```

Consumatore con pipe

```
void    consumer(void)
        {
        data_type  item;

        for (;;)
                {
                read_pipe(pipe_id, &item,
                    sizeof(item));
                consume(&item);
                }
        }
```

Identificazione

- Produttore e consumatore devono scambiarsi un identificatore per poter inviare messaggi:
  - identificatore di processo, per lo scambio diretto;
  - identificatore di mailbox, per lo scambio via mailbox;
  - identificatore di pipe, per lo scambio via pipe.
- Un antenato comune può allocare le risorse e distribuire gli identificatori.

Communicating Sequential Processes

- I CSP, proposti da Hoare, sono un meccanismo di scambio messaggi integrato nel linguaggio.
  - Nato per monoprocessori, il meccanismo si presta anche alla comunicazioni in processori paralleli.
- I processi utilizzano primitive di I/O o molto simili per leggere e scrivere i messaggi su canali di comunicazione.
- Il sistema si occupa del trasferimento dei messaggi e della sincronizzazione.

Caratteristiche dei CSP

- I processi hanno spazi di indirizzamento disgiunti.
- La sintassi è molto semplice.
  - Integrata nel linguaggio come nel caso dei monitor.
  - Aggiungono ai linguaggi due tipi di primitive:
    - lettura e scrittura, per comunicare con altri processi;
    - guarded-command.
- Varianti molto simili ai CSP sono state implementate in Occam e Ada.

Comunicazione tra CSP

- Lettura e scrittura sospendono il processo, sino all'avvenuta comunicazione.
  - Si ottiene comunicazione e sincronizzazione.
  - Falliscono solo se non esistono altri processi disponibili a comunicare nel sistema.
- I messaggi hanno un tipo e la comunicazione avviene solo quando i processi coinvolti utilizzano lo stesso tipo.

Guarded command

- I guarded-command sono insiemi di istruzioni, ciascuna preceduta da un'espressione booleana ("guardia").
  - Le espressioni sono valutate in ordine non specificato: quando una è vera, viene eseguita l'istruzione corrispondente.
- Permettono tra l'altro a un processo di comportarsi da "server", rispondendo al primo messaggio in arrivo, tra più messaggi di tipi differenti.

Transputer

- Processore prodotto a partire dai primi anni ‘80 dalla INMOS (UK).
  - Nato per costruire matrici bidimensionali di processori identici.
    - Adattabile a configurazioni differenti.
    - Ogni processore ha memoria (dati e codice) privata.
  - Implementa in hardware la comunicazione.
    - Ogni processore dispone di 4 porte bidirezionali veloci e mentre elabora può comunicare con altrettanti gemelli.
  - Il context switch è molto veloce, perché implementato direttamente in hardware.

Occam

- Linguaggio creato per il transputer.
  - Linguaggio di basso livello, con strutture dati e di controllo simili a quelle del C, ma sintassi differente.
  - Dispone di semplici primitive per leggere e scrivere sulle 4 porte bidirezionali.
  - Le stesse primitive permettono la comunicazione tra processi sulla stessa macchina.
    - Chi invia o riceve messaggi può ignorare su quale processore si trovi l'interlocutore.

Produttore e consumatore in Occam

- Il produttore esegue ciclicamente l'invio di dati:
  `consumer ! dato_1`
- Il consumatore esegue ciclicamente la ricezione di dati:
  `producer ? dato_2`
- L'effetto è equivalente a un assegnamento di `dato_1` a `dato_2`, eseguito tra processi diversi (eventualmente su processori diversi).
  - La sincronizzazione è gestita dall'hardware.

Ada

- I processi (“task”) sono tipi predefiniti, che possono essere utilizzati nelle normali strutture dati.
- Dispone di primitive di invio/ricezione di messaggi come i CSP.
- Dispone di un’istruzione che implementa i guarded-command.

Vantaggi dello scambio di messaggi

- Efficace per:
  - sincronizzazione e comunicazione tra processi su sistemi multiprocessore o su calcolatori connessi in rete;
  - architetture client-server.
- Generalmente semplice da comprendere.
  - Riduce gli errori rispetto ad altri meccanismi.

Svantaggi dello scambio di messaggi

- Può essere complesso da implementare, soprattutto su macchine connesse in rete.
  - Serve un raffinato protocollo di comunicazione.
- Su un monoprocessore il messaggio viene copiato inutilmente.

Comunicazione tra processi (1)

- Vi sono due tipi di meccanismi:
  - memoria condivisa e sincronizzazione;
  - scambio di messaggi.
- Needham e Lauer dimostrarono che sono equivalenti.

Comunicazione tra processi (2)

- Molti sistemi li mettono a disposizione entrambi.
  - E' poco sensato usarli insieme.
- La memoria condivisa di solito è più efficiente, ma più complessa da utilizzare.
- Lo scambio di messaggi è preferibile se i processi hanno spazi di indirizzamento disgiunti.

Caratteristiche di un meccanismo di IPC

- Il canale di comunicazione tra processi può essere:
  - diretto (intrinsecamente bidirezionale);
  - tramite mailbox (unidirezionale o bidirezionale).
- Una comunicazione diretta di solito richiede che ogni processo coinvolto conosca in anticipo l'identità dell'altro.
  - Inadatto ai server.

Memoria di un meccanismo di IPC

- Può essere nulla o con buffer, di capacità limitata e stabilita a priori.
- Vantaggi dei meccanismi a memoria nulla.
  - Il mittente viene sospeso sino all'avvenuta comunicazione.
    - Non servono sincronizzazioni aggiuntive per sapere se il messaggio è stato letto.
- Vantaggi dei meccanismi con buffer.
  - Meno rigidi, maggiore parallelismo.

Natura dei messaggi

- Possono essere:
  - a lunghezza fissa;
    - caso particolare: byte stream;
  - a lunghezza variabile;
  - con tipo:
    - legati al linguaggio;
    - permettono di discriminare in ricezione tra messaggi diversi.

Gestione delle eccezioni

- In caso di errore il sistema può:
  - ignorare il problema:
    - si è spesso costretti a implementare un sistema di “messaggi con ricevuta”;
  - avvertire il mittente:
    - può essere complicato identificare il messaggio non trasmesso e gestire l’errore;
  - garantire comunque la consegna, ma in un tempo non specificato:
    - complesso da implementare, inadatto a sistemi interattivi.

Protocollo di comunicazione

- Insieme di regole, che stabiliscono come inviare i messaggi e come trattare gli errori.
- Con calcolatori connessi in rete il protocollo può diventare molto complicato.

Problemi da risolvere nei protocolli (1)

- Un processo può non consumare i messaggi abbastanza velocemente.
  - Bisogna poter sospendere e riattivare i processi che accodano messaggi su di una coda piena.
- Meccanismi di identificazione di risorse (calcolatori, processi, mailbox) remote.
- I messaggi possono arrivare con errori.
- I messaggi possono andare persi sulla rete.

Problemi da risolvere nei protocolli (2)

- Guasti permanenti, che rendono impossibile l'invio di messaggi a una macchina.
  - Potrebbero essere scoperti dopo che il processo mittente ha eseguito la `send`.
- Il destinatario potrebbe non esistere (più).
- Autenticazione del mittente.
- Sicurezza contro intercettazione abusiva e invio di messaggi falsi.
- Prestazioni.

Errori nei messaggi

- Il livello inferiore dei sistemi di comunicazione è in grado di identificare e scartare, con probabilità molto vicina al 100%, i messaggi errati.
  - Si utilizzano varie forme di CRC e checksum.
- Al livello superiore del protocollo bisogna scoprire quando un messaggio non è stato ricevuto.
  - Un messaggio può andare perduto senza lasciare traccia, a causa di disturbi o guasti.

Sincronizzazione dei messaggi

- Un metodo comune per assicurare la corretta ricezione dei messaggi è inviare un messaggio di conferma.
  - Se il mittente non riceve la conferma entro un tempo prefissato, invia di nuovo il messaggio.

Messaggi di conferma

- Se va persa la conferma, il ricevente si vede recapitare due copie dello stesso messaggio.
  - Come rimedio standard si numerano progressivamente i messaggi, in modo che chi riceve due volte un identico messaggio possa ignorare la copia.
  - Analogamente chi riceve un messaggio fuori sequenza può avvisare il mittente che qualche messaggio è andato perso.

Altre primitive di sincronizzazione

- Sono state inventate numerose altre primitive, varianti più o meno complesse di quelle elencate.
- Sono tutte equivalenti, nel senso che con un gruppo qualsiasi è possibile implementare tutte le altre.
- I sistemi spesso offrono numerose varianti delle primitive base.

Implementazione di primitive atomiche

- Le primitive che devono essere atomiche sono implementate come parte del sistema operativo.
  - Dato che sono molto veloci, sospendere gli interrupt durante la loro esecuzione è una soluzione comune.

Primitive atomiche e multiprocessore

- Nelle macchine multiprocessore bisogna proteggere gli accessi alle variabili comuni alle varie copie di sistemi (es.: i contatori dei semafori) da accessi simultanei.
  - Si usa una variabile di lock, gestita con test and set o swap.

Thread

- Si dicono “thread” o processi leggeri (lightweight process) più flussi di controllo indipendenti nello stesso processo.
  - Simili a mini-processi, appartenenti allo stesso utente.
- Molti sistemi operativi consentono a ogni processo di suddividersi in più thread.
- Utilizzano meno risorse di sistema (page table, process table) dei processi.

Thread e risorse

- I thread appartenenti allo stesso processo condividono tutte le risorse di sistema, in particolare:
  - lo spazio di indirizzamento, codice e dati;
  - file aperti, semafori, timer, segnali ecc..
- Hanno stack separati, all'interno di un unico spazio di indirizzamento.
- Un processo termina quando sono terminati tutti i suoi thread.

Thread e protezioni

- Il sistema non protegge i thread di uno stesso processo da interferenze indesiderate:
  - sarebbe impossibile o molto dispendioso;
  - si suppone cooperazione tra i thread dello stesso processo;
  - il sistema fornisce primitive di sincronizzazione (semafori, mailbox ecc.).
- Thread di processi diversi sono separati dai meccanismi di protezione dei processi.

Thread e sistema

- Per ogni thread il sistema conserva solo poche informazioni:
  - registri,
  - stato,
  - priorità,
  - relazioni di parentela.
- Le altre informazioni, comuni ai thread dello stesso processo, sono conservate nella process table, un'unica copia per processo.

Vantaggi della suddivisione in thread di un processo

- Su di una macchina multiprocessore si può avere reale parallelismo tra thread dello stesso processo.
- La comunicazione tra thread è molto semplice.
  - Utilizzando un unico spazio di indirizzamento, condividono implicitamente tutte le variabili globali e possono passarsi gli indirizzi di quelle automatiche e dinamiche.

Thread scheduling e switching

- I thread sono schedulati come i processi.
  - Hanno stato e priorità.
  - Per passare da un thread a un altro dello stesso processo il sistema deve solo salvare e ricaricare i registri, senza agire su cache, paginazione e segmentazione.

Thread e linguaggi

- Alcuni linguaggi supportano costrutti per la creazione di thread, semplificandone la gestione.
  - fork - join,
  - cobegin - coend;
  - thread come classi.
- In alternativa i thread possono essere creati con primitive del sistema operativo.

Simulazione dei thread

- Se il sistema non rende disponibili i thread, si possono realizzare con una libreria.
- Il sistema continua a vedere un unico processo.
  - Se viene sospeso, si bloccano tutti i thread.
- Il time-sharing tra i thread è gestito dalla libreria.
  - I thread ricevono ciascuno una frazione del tempo che il sistema alloca al processo.
  - Non si possono sfruttare più CPU, se presenti.

Simulazione dei thread

- Per implementarla bisogna:
  - far chiamare una funzione a intervalli fissi, per attivare uno scheduler;
  - allocare stack separati;
  - gestire il context switch tra i thread.
- Bisogna inoltre implementare almeno:
  - uno scheduler;
  - le funzioni di creazione e distruzione;
  - le primitive di sincronizzazione e mutua esclusione.

Fork - join

- Meccanismo, proposto da Conway nel 1963, basato su due primitive:
  - **fork**, simile a una chiamata di funzione, che crea e lancia un nuovo thread;
  - **join**, che attende la fine del thread specificato.
    - Non necessariamente figlio del thread che esegue la join.
- Usato nel linguaggio Mesa.
- Adottato, con qualche modifica, per la creazione di processi in UNIX.

Cobegin - coend

- Meccanismo, proposto da Dijkstra nel 1968, per rendere strutturata la creazione e distruzione di thread.
  - Non permette la stessa flessibilità di fork - join, ma è più controllato e sicuro.
- Si basa su due parole chiave, `cobegin` e `coend`, che racchiudono un blocco di istruzioni, da eseguirsi tutte in parallelo.
  - Ogni istruzione origina un thread.
- Usato nel linguaggio Edison (Brinch Hansen 1983).

Linguaggi e parallelismo (1)

- Vi sono linguaggi che integrano il parallelismo, con un modello a memoria condivisa (variabili comuni ai processi).
  - Alcuni integrano i monitor:
    - es.: Concurrent Pascal, Modula, Mesa, Edison.
  - Relativamente pochi integrano costrutti per il parallelismo come istruzioni:
    - es.: Edison.
  - Altri integrano il thread (spesso chiamato task) tra i tipi fondamentali:
    - es.: Ada, Java.

Linguaggi e parallelismo (2)

- Pochi linguaggi esprimono il parallelismo con comunicazione a messaggi.
  - Ottimo modello per architetture multiprocessore.
  - Es.: PLITS, Occam.
- Alcuni linguaggi forniscono sia memoria condivisa, che scambio di messaggi.
  - Es.: CHILL, Ada.

Concurrent Pascal

- Estensione del Pascal, definita da Brinch Hansen nel 1973.
  - Il primo linguaggio concorrente.
  - Introdusse i monitor e le classi.
    - Tipi di dati astratti, ai quali accede un solo processo per volta.
    - Dispone di variabili che fungono da semaforo (di tipo “condition”) e due primitive, “delay” e “continue”, analoghe a wait e signal.
  - Semplice, ma poco flessibile.
    - Le risorse condivise vanno allocate staticamente.
  - Supporto run-time relativamente grande.

Modula

- Simile al Pascal, definito da Wirth nel 1977.
  - Dispone di “interface module” (monitor), variabili di tipo “signal” (semafori) e di due primitive “wait” e “send” (equivalente a signal).
- Supporto run-time estremamente ridotto.
  - Wirth riuscì a farne una versione che usava solo 98 word su PDP 11/45.

Modula 2

- Simile al Modula, definito da Wirth nel 1982.
  - Versione semplificata, di maggior successo.
    - Utilizzato per applicazioni commerciali.
  - Rende disponibile un meccanismo a coroutine, col quale il programmatore può costruire uno scheduler.
    - Dispone di una libreria con scheduler già pronti.

Mesa

- Nato allo Xerox PARC nel 1977, come linguaggio di sistema.
  - Creazione dinamica di processi tramite fork.
  - Semafori e primitive di sincronizzazione.
    - Possibile specificare un tempo massimo di attesa.
  - Potente meccanismo di gestione eccezioni.

Edison

- Definito da Brinch Hansen nel 1983 per la programmazione di multi-microelaboratori.
  - Offre “moduli” (monitor) e un costrutto per il parallelismo (cobegin - coend).

Ada (1)

- Fortemente voluto dal Dipartimento della Difesa USA come linguaggio unificante.
- Scelto tra numerosi candidati con un processo di selezione durato anni.
- Il linguaggio più usato in campo militare e aerospaziale.

Ada (2)

- Task (thread) come costruttore di tipi.
  - I singoli task sono definiti come variabili e possono entrare in strutture dati arbitrarie.
    - Sono possibili vettori di task, record contenenti task ecc..
- Sincronizzazione mediante rendez-vous.
  - Un “punto di incontro” tra due task.
  - Il primo che arriva aspetta l’altro.
  - Uno dei due richiede un servizio e passa all’altro le informazioni necessarie.
  - Realizza un passaggio di messaggi.

Java

- Definito dalla Sun nel 1996.
- Thread come tipo di dati astratto.
  - Una delle classi predefinite.
  - I singoli thread sono oggetti e possono essere creati dinamicamente.
  - Sincronizzazione mediante `wait` e `notify` (equivalente alla `signal`) e possibilità di dichiarare regioni critiche nel codice.

PLITS

- Definito da Feldman nel 1976.
- Dispone di send e receive a livello di linguaggio
  - Si può associare un codice (chiave) a ogni messaggio.
  - Nella receive si può specificare che si accettano solo i messaggi da un processo specifico e/o con una chiave fissata.

Occam (1)

- Definito dalla Inmos nel 1983 per architetture multiprocessore basate su transputer.
  - Il processore ha una struttura interna fatta apposta per implementare lo scambio di messaggi in modo diretto ed efficiente.
- Implementa il modello dei Communicating Sequential Processes di Hoare (1978).

Occam (2)

- Dispone di send e receive come istruzioni.
  - Nella forme:

    <destinatario> ! <espressione>

    <sorgente> ? <variabile>.
  - Il fatto che il processo col quale si comunica sia sullo stesso processore o su un altro è completamente trasparente.
  - Nell'ultima versione del processore (T9000) l'instradamento dei messaggi è dinamico.

Chill

- Definito dal CCITT nel 1980, per applicazioni nelle telecomunicazioni.
- Permette di definire regioni critiche, con variabili locali alla regione.
- Dispone di variabili “evento”, simili ai semafori, ma molto flessibili.
  - E’ possibile specificare priorità e sospensione in attesa di più eventi.
- Send e receive agiscono su buffer (variabili simili a mailbox).

Scheduler

- Componente del sistema operativo che decide quale, tra i processi (o thread) ready, debba avere il controllo della CPU.
- Obiettivo:
  - minimizzare le attese;
  - massimizzare il throughput.

Attivazione dello scheduler

- Interviene ogni volta che il processo running cambia stato:
  - per una system call che lo blocca (es.: wait);
  - perché è scaduto il tempo assegnatogli;
  - perché commette un errore che provoca un trap;
  - perché termina la sua esecuzione.
- Viene quindi attivato da una system call o da un interrupt.

Preemption

- Si dice non preemptive uno scheduler che viene attivato solo quando il processo corrente passa da running a waiting o muore.
- Si dice preemptive uno scheduler che viene attivato anche:
  - quando il processo corrente passa da running a ready, per scadenza del tempo assegnato;
  - quando un altro processo diventata ready.
- Se lo scheduler è preemptive, la CPU può essere tolta ai processi in qualsiasi momento.

Tecniche di scheduling

- Run to completion (senza preemption).
  - La CPU resta assegnata allo stesso processo finché questo termina o esegue una system call che lo blocca.
  - Adatto solo ai sistemi batch.
  - Un processo in loop può bloccare il calcolatore.
  - Es.: MAC/OS, MS Windows 3.0.

Varianti principali di run to completion non preemptive

- FIFO:
  - non riduce i tempi di attesa medi o massimi.
- Shortest job first:
  - minimizza i tempi di attesa medi;
  - richiede una previsione a priori del tempo di esecuzione;
  - può essere adattato a scheduler preemptive.

Scheduling a tempo

- Un componente hardware (clock) invia interrupt a intervalli regolari (programmabili), da 50 a 10000 volte al secondo.
- A ogni interrupt il sistema decide se lasciare la CPU al processo interrotto o cederla ad altri.

Scheduling preemptive round robin

- A ogni processo viene assegnato un intervallo di tempo, detto “quanto”; al suo esaurimento il processo diventa ready e la CPU viene ceduta a un altro.
- I processi ready formano una coda FIFO:
  - i processi che diventano ready (incluso quello al quale scade il quanto) vengono agganciati in coda;
  - quando la CPU si libera, il processo in testa diventa running.

Durata del quanto

- Se troppo corta, la CPU spende una frazione eccessiva di tempo per eseguire l'algoritmo di scheduling e il context switch.
- Se troppo lunga, i tempi di risposta per gli utenti interattivi diventano molto lunghi.
- Il compromesso dipende dalla macchina; di solito tra 5 e 100 ms.
- Non è necessariamente la stessa per tutti i processi.

Round robin con priorità (1)

- A ogni processo può essere assegnata una priorità alla creazione:
  - dall'utente;
  - dal sistema, in funzione dell'utente;
  - dal processo padre.
- Nel controllo di processo può dipendere dall'urgenza o importanza.

Round robin con priorità (2)

- Quando la CPU si libera, viene reso running il processo ready a massima priorità.
  - Un meccanismo round robin semplice può essere utilizzato se vi sono più processi alla stessa priorità.
- Normalmente si usa un vettore di code o una lista di code, una per ogni livello di priorità.

Priorità

- La priorità può essere:
  - statica, assegnata alla creazione e immutabile;
    - se un solo processo ha la priorità massima, diventa un run to completion;
    - i processi a bassa priorità rischiano di non ricevere quasi mai la CPU;
  - dinamica, stabilita alla creazione e variata poi:
    - dal sistema, per evitare che pochi processi si accaparrino la CPU;
    - dai processi, a seconda delle loro esigenze.

Priorità statica

- Se si conosce la durata dei processi, dare priorità ai più brevi minimizza il tempo di attesa medio (shortest job first).
- Possibile solo in alcuni casi di sistemi batch.
  - Numero e durata dei processi noti in anticipo.

Priorità dinamica

- In mancanza di informazioni precise sui processi, bisogna accontentarsi di stime.
- Una ragionevole approssimazione dell'algoritmo ottimale si ottiene stimando la durata dei processi sulla base del loro comportamento e variando di conseguenza la priorità.
- Si può basare la stima:
  - sull'utilizzo totale di tempo di CPU;
  - sugli intervalli tra operazioni di I/O.

Calcolo della priorità (1)

- In un sistema interattivo bisogna privilegiare i processi che eseguono molto I/O.
  - Ricevuta la CPU, la cederanno rapidamente.
  - Aumenta il throughput del sistema.
- I processi che utilizzano molta CPU, viceversa, rallentano tutti gli altri.
  - Impediscono a un'operazione di I/O di un altro processo di iniziare per tutto il loro quanto.

Calcolo della priorità (2)

- Sono stati provati molti algoritmi basati su queste idee.
- Una tecnica possibile è cambiare la priorità di un processo che perde il controllo della CPU in:

  (durata del quanto) / (tempo usato).
- I processi che eseguono I/O utilizzano una piccola frazione del quanto e vengono portati a una priorità alta; quelli che lo usano completamente scendono a priorità 1.

Priorità basata su utilizzo totale di CPU

- Vi sono varie possibilità.
  - Usare valori iniziali di priorità molto alti e diminuire la priorità ad ogni quanto consumato, senza farla diventare negativa. Quando il processo running non è più il più prioritario, la CPU viene ceduta a un altro.
  - Ridurre la priorità logaritmicamente all'aumentare del tempo totale di CPU utilizzato (UNIX).
    - Penalizza i processi che non terminano mai.

Priorità basata sul tempo di CPU tra operazioni di I/O

- Una possibile alternativa è variare la priorità sulla base della stima del tempo di CPU che il processo userà prima di bloccarsi.
  - Si può calcolare una stima S_n del tempo T_n che sarà utilizzato prima della prossima operazione di I/O.

    Es.: $S_n = a\, S_{n-1} + (1- a)\, T_{n-1}$, con $0 < a < 1$
- Non penalizza i processi che non terminano mai.

Priorità basata sull'equità

- Si calcola il tempo di CPU che spetterebbe a ciascun processo, suddividendo equamente il tempo tra i processi ready.
- La priorità dei processi che hanno usato più del tempo spettante viene ridotta, la priorità di quelli che hanno usato meno del tempo spettante viene aumentata.

Un esempio (1)

- Il CTSS, su IBM 7094 (1962) aveva il problema di un context switch estremamente lento, perché richiedeva lo swap su disco dell'intero processo.
- Con tale vincolo conviene che i processi che usano molto la CPU ricevano raramente quanti lunghi, per ridurre i tempi persi nel context switch.

Un esempio (2)

- Si ricorse a una gerarchia di classi.
  - Viene reso running il primo processo ready di classe minima.
  - I processi in classe n ricevono quanti di durata 2^n.
  - Quando un processo usa interamente un quanto viene spostato nella classe successiva.
  - Quando un processo effettua un'operazione di I/O viene riportato in classe 0.

Priorità e deadline

- In un sistema real-time un processo può dichiarare la durata prevista e la propria deadline (istante entro il quale dev'essere completato).
- All'avvicinarsi della deadline la priorità viene aumentata.
- Il processo più prossimo alla scadenza diventa running.

Round robin con e senza priorità

- Possono coesistere:
  - il round robin è generalmente utilizzato tra processi alla stessa priorità;
  - la priorità può essere usata in un round robin puro per determinare la lunghezza del quanto;
  - si può usare un approccio misto, con processi ad alta priorità, per gestire eventi importanti, ma occasionali, che hanno precedenza sui processi “normali”, tra i quali si utilizza un round robin semplice.

Scheduling a due livelli

- Utilizzato in assenza di memoria virtuale.
- Uno scheduler alloca la CPU ai processi in memoria.
- Un secondo scheduler decide, a intervalli più lunghi, quali processi caricare da disco.
- Gli algoritmi dei due scheduler sono indipendenti.