Lezione 5 [04/10/22]

Operazioni: terminazione di un processo

Un processo termina quando finisce l'esecuzione della sua ultima istruzione

Con un'apposita system call (`exit`) il processo chiede al sistema di essere cancellato:

- Il processo figlio può restituire l'output al processo padre (un'informazione di stato, ricevuto attraverso la chiamata di sistema `wait`)

Si può avere anche una terminazione spuria del processo attraverso una system call (`abort`) che è gestita dal padre del processo

Si arriva a questa soluzione ad es. nei seguenti casi:

- Il processo figlio ha ecceduto nell'uso delle risorse che gli erano state assegnate
- Il padre non ha più bisogno del compito che il figlio svolgeva
- Il padre termina e il sistema operativo non consente al figlio di continuare l'esecuzione
- La maggior parte dei sistemi non consentono a processi figli di esistere dopo la terminazione del padre
    - Quest'azione porta all'effetto di una **terminazione a cascata**

Per illustrare un esempio di esecuzione e terminazione di un processo, si consideri che nel sistema operativo Linux come in UNIX un processo può terminare per mezzo della chiamata di sistema `exit()`, fornendo uno stato di uscita **(exit status)** come parametro:

```
// Uscita con stato 1
exit(1);
```

Di fatto, in caso di terminazione normale, `exit()` può essere chiamato direttamente (come mostrato sopra) o indirettamente (tramite un'istruzione `return` nel main)

Un processo genitore può attendere la terminazione di un processo figlio utilizzando la chiamata di sistema `wait()`, a cui viene passato un parametro che permette al genitore di ottenere lo stato di uscita del figlio. Questa chiamata di sistema restituisce anche l'ID del processo figlio terminato in modo che il genitore possa sapere di quale dei suoi figli si tratta:

```
pid_t pid;
int status;
```

```
pid = wait(&status);
```

Quando un processo termina, le sue risorse vengono deallocate dal sistema operativo

Tuttavia la sua voce nella tabella dei processi deve rimanere fino a quando il padre chiama `wait()`, perché la tabella dei processi contiene lo stato di uscita del processo

Un processo che è terminato, ma il cui genitore non ha ancora chiamato la `wait()`, è detto processo **zombie**

Tutti i processi passano in questo stato quando terminano, ma in genere rimangono zombie solo per un breve tempo. Una volta che il genitore chiama la `wait()` il PID del processo zombie e la sua voce nella tabella dei processi vengono rilasciati

Consideriamo ora che cosa accadrebbe se un genitore terminasse senza invocare la `wait()`, lasciando così orfani i suoi figli

Linux e UNIX affrontano questa situazione assegnando al processo init il ruolo di nuovo genitore dei processi orfani. Il processo init invoca periodicamente `wait()`, consentendo in tal modo di raccogliere lo stato di uscita di qualsiasi processo orfano e rilasciando il suo PID e la relativa voce nella tabella dei processi

Processi cooperanti

I processi concorrenti nel sistema operativo possono essere **indipendenti** o **cooperanti**

Un processo è indipendente se non può influire su altri processi, né può subirne l'influsso, durante la sua esecuzione

Un processo cooperante può influire su altri processi, e può subirne l'influsso, durante la sua esecuzione

Vantaggi della cooperazione tra processi:

- Condivisione di informazioni:
  - Più utenti possono essere interessati alle stesse informazioni
- Accelerazione del calcolo:
  - Se vi sono più CPU o canali di I/O
- Modularità
  - Può essere più pratica ed efficiente la costruzione di un sistema modulare
- Convenienza

- Anche un solo utente può avere la necessità di compiere più operazioni contemporaneamente

I modelli fondamentali della comunicazione tra processi sono due:

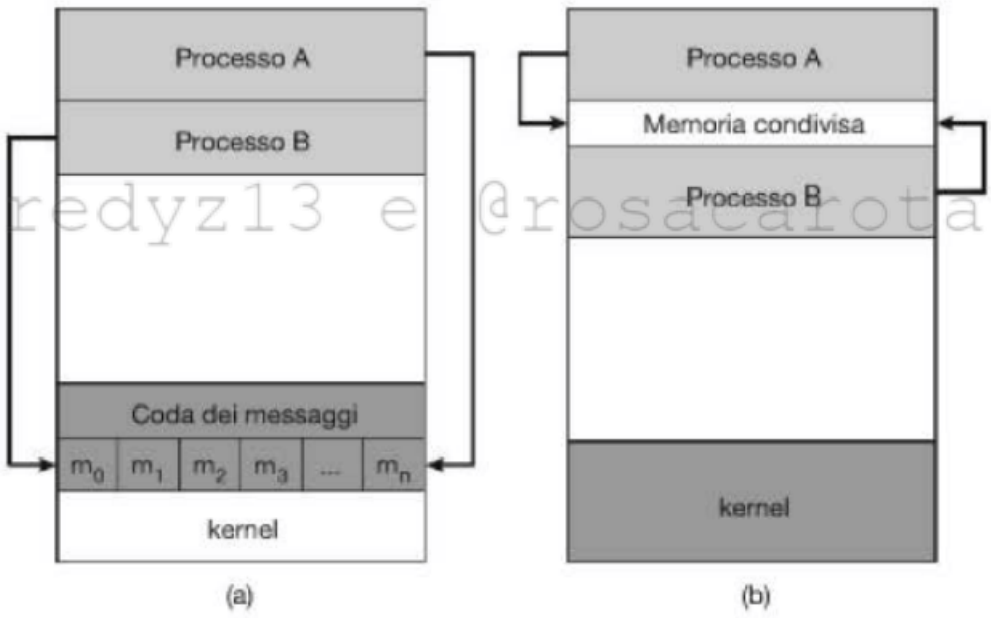
- A **memoria condivisa**
- A **scambio di messaggi**

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di **comunicazione tra processi** (**IPC**, interprocess communication)

Nel modello a memoria condivisa, si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona

Nel secondo modello la comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti

A: Scambio di messaggi **B**: Memoria condivisa



Nei sistemi operativi sono diffusi entrambi i modelli; spesso coesistono in un unico sistema

Lo scambio di messaggi è utile per trasmettere piccole quantità di dati, non essendovi bisogno di evitare conflitti, esso è anche più facile da implementare, rispetto alla memoria condivisa, in un sistema distribuito

La memoria condivisa può essere più veloce dello scambio di messaggi, che è solitamente implementato tramite chiamate di sistema che impegnano il kernel; la memoria condivisa, invece, richiede l'intervento del kernel solo per allocare le regioni di memoria condivisa, dopo di che tutti gli accessi sono

gestiti alla stregua di ordinari accessi in memoria che non richiedono l’assistenza del kernel

Recenti ricerche su sistemi con più core di elaborazione indicano che su tali sistemi lo scambio di messaggi fornisce prestazioni migliori rispetto alla memoria condivisa, che, soffre inoltre di problemi di coerenza della cache che sorgono a causa della migrazione dei dati condivisi tra le varie cache

Sistemi a memoria condivisa

La comunicazione tra processi basata sulla condivisione della memoria richiede che i processi comunicanti allochino una zona di memoria condivisa, di solito residente nello spazio degli indirizzi del processo che la alloca; gli altri processi che desiderano usarla per comunicare dovranno annetterla al loro spazio degli indirizzi

Si ricordi che, normalmente, il sistema operativo tenta di impedire a un processo l’accesso alla memoria di altri processi.

La condivisione della memoria richiede che due o più processi raggiungano un accordo per superare questo limite, in modo da poter comunicare tramite scritture e letture dell’aria condivisa

Il tipo e la collocazione dei dati sono determinati dei processi, e rimangono al di fuori del controllo del sistema operativo

I processi hanno anche la responsabilità di non scrivere nella stessa locazione simultaneamente

Il problema del produttore e del consumatore

Paradigma per processi cooperanti

Un processo **produttore** “produce” informazioni, che sono “consumate” da un processo **consumatore**

Vi è necessità di un buffer su cui:

- Chi produce può immagazzinare informazioni
- Chi consuma può prelevarle

Si possono avere due tipi di memoria:

- Memoria illimitata **(unbounded-buffer)** in cui non ci sono limiti alla dimensione del buffer
- Memoria limitata **(bounded-buffer)** in cui la dimensione del buffer è fissata

Il buffer dovrà risiedere in una zona di memoria condivisa dai due processi.

I due processi devono essere sincronizzati in modo tale che il consumatore non tenti di consumare un’unità non ancora prodotta

Se il buffer è illimitato il consumatore può dover attendere nuovi oggetti, ma il produttore può sempre produrne

Se il buffer è limitato il consumatore deve attendere se il buffer è vuoto; viceversa, il produttore deve attendere se il buffer è pieno

Memoria limitata: soluzione con memoria condivisa

Consideriamo più attentamente in che modo il buffer limitato illustra la comunicazione tra processi con memoria condivisa

Soluzione che permette di memorizzare (in un buffer) fino a `BUFFER_SIZE - 1` elementi

Il produttore e il consumatore condividono le seguenti variabili in una zona di memoria condivisa:

```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
item next_produced;

while (1) {
    while (((in + 1) % BUFFER_SIZE) == out); // Non fa niente
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

item next_consumed;

while (1) {
    while (in == out); // Non fa niente
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```


Il buffer condiviso è realizzato come un array circolare con due puntatori logici: `in` e `out`

La variabile `in` indica la successiva posizione libera nel buffer; `out` indica la prima posizione piena del buffer

Il buffer è vuoto se `in == out`; è pieno se `((in + 1) % BUFFER_SIZE) == out`

Il processo produttore ha una variabile locale `next_produced` contenente il nuovo elemento da produrre

Il processo consumatore ha una variabile locale `next_consumed` in cui si memorizza l'elemento da consumare

Sistemi a scambio di messaggi

Lo scambio di messaggi è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio di indirizzi

Attraverso le funzionalità della comunicazione tra processi, i processi sono in grado di comunicare tra loro e di sincronizzare le proprie azioni:

- **Inter Process Communication** - IPC

Un IPC fornisce almeno due operazioni, **send** e **receive:**

- send(message)
- receive(message)

I messaggi possono essere di dimensione fissa o variabile

Se P e Q vogliono comunicare tra loro, devono:

- Stabilire il **canale di comunicazione**
- Scambiare messaggi via send/receive

Realizzazione del canale di comunicazione:

- **Realizzazione fisica** (ad es. memoria condivisa, bus, etc.)
- **Realizzazione logica** (comunicazione diretta o indiretta, gestione automatica o esplicita del buffer etc.)

Implementazione

Per implementare la comunicazione tra processi è importante sapere quanto segue:

- Come vengono stabiliti i canali
- Se un canale può essere associato a due o più processi
- Quanti canali possono esistere tra ogni coppia di processi
- Conoscere la capacità di un canale
- Sapere se un canale può supportare messaggi a dimensione fissa o variabile
- Se la comunicazione sullo specifico canale è unidirezionale o bidirezionale

Esistono inoltre diversi modi per effettuare l’implementazione logica di un canale:

- Comunicazione diretta o indiretta
- Comunicazione simmetrica o asimmetrica
- Capacità zero, limitata o illimitata della coda di messaggi
- Invio per copiatura o per riferimento
- Messaggi di dimensione fissa o di dimensione variabile

Comunicazione diretta

Con la comunicazione diretta, ogni processo che intenda comunicare deve nominare esplicitamente il ricevente o il trasmittente della comunicazione

In questo schema le funzioni primitive send() e receive() si definiscono come segue:

- send(P, messaggio) - invia messaggio al processo P
- receive(Q, messaggio) - ricevi messaggio dal processo Q

Un canale di comunicazione ha le seguenti caratteristiche:

- I canali sono stabiliti automaticamente
- Un canale è associato esattamente a due processi
- Esiste esattamente un canale tra ciascuna coppia di processi
- Il canale può essere unidirezionale, ma generalmente è bidirezionale

Comunicazione indiretta

Con la comunicazione indiretta i messaggi vengono inviati a delle **porte** o **mailbox** (considerabile in modo astratto come un oggetto in cui i processi possono introdurre o prelevare messaggi), che li ricevono:

- Ciascuna porta ha un identificativo unico
- I processi possono comunicare solo se hanno una **porta in comune**

Proprietà dei canali di comunicazione:

- Tra una coppia di processi si stabilisce un canale solo se entrambi i processi condividono una stessa porta
- Un canale può essere associato a più di due processi
- Tra ogni coppia di processi comunicanti può esserci un certo numero di canali diversi, ciascuno corrispondente a una porta
- Un canale può essere unidirezionale o bidirezionale

Il S.O. offre un meccanismo che permette al processo di:

- Creare una nuova porta
- Inviare e ricevere messaggi tramite la porta
- Rimuovere la porta

Le primitive di comunicazione sono:

- send(A, messaggio) - invia un messaggio alla porta A
- receive(A, messaggio) - ricevi un messaggio dalla porta A

Condivisione delle porte:

- P_1, P_2 e P_3 condividono la porta A

- P_1 invia un messaggio ad A; Sia P_2 che P_3 eseguono una receive() da A
- Quale processo riceverà il messaggio?

Soluzioni:

- Fare in modo che un canale sia associato al più a due processi
- Consentire a un solo processo alla volta di eseguire l'operazione di receive()
- Consentire al sistema di scegliere arbitrariamente (anche tramite algoritmi di selezione, secondo il quale i processi ricevono i messaggi a turno) quale processo deve riceve il messaggio (che sarà ricevuto da P_2 o da P_3 ma non da entrambi)
- Il sistema comunicherà l'identità del ricevente al trasmittente

Sincronizzazione

Lo scambio di messaggi può essere **sincrono** (o bloccante) oppure **asincrono** (o non bloccante)

- **Invio sincrono** - il processo che invia il messaggio si blocca nell'attesa che il processo ricevente, o la porta, riceva il messaggio
- **Invio asincrono** - il processo invia il messaggio e riprende la propria esecuzione
- **Ricezione sincrona** - il ricevente si blocca nell'attesa dell'arrivo di un messaggio
- **Ricezione asincrona** - il ricevente riceve un messaggio valido o un valore nullo

Le primitive send e receive possono essere sincrone o asincrone

Se esse sono entrambe sincrone si parla di **rendezvous** tra mittente e ricevente

Code di messaggi

Un canale ha una capacità che determina il numero dei messaggi che possono risiedere al suo interno

Questa caratteristica può essere immaginata come una coda di messaggi legata al canale

Esistono tre tipi di code:

1. **Capacità zero** - la coda ha capacità zero, per cui il canale non può avere messaggi in attesa al suo interno. Con questa specifica il trasmittente deve attendere che il ricevente abbia ricevuto il messaggio per inviarne un altro (rendezvous)
2. **Capacità limitata** - la coda ha lunghezza finita **n**, quindi al suo interno non può contenere più di **n** messaggi. Finché la coda non è piena il trasmittente può inviare messaggi

3. **Capacità illimitata** - la coda ha lunghezza infinita, per cui il trasmittente può sempre inviare messaggi e non resta mai in attesa

Nel caso 1 si parla di **gestione del buffer esplicita** o assente, negli altri due casi si dice che la **gestione del buffer è automatica**