Appunti di sistemi operativi

Appunti per il corso universitario di sistemi operativi, riferito a sistemi Unix/Windows.
Si discute su problemi di sincronizzazione, memoria e scheduling dei processi.

ARGOMENTI

INTRODUZIONE

INPUT/OUTPUT

GESTIONE DEI PROCESSI

ALGORITMI DI SCHEDULING

SCHEDULING MULTI CPU

SISTEMI REAL TIME

SCHEDULING SU LINUX

SCHEDULING SU WINDOWS

OPERAZIONI SUI PROCESSI

COMUNICAZIONE TRA PROCESSI

THREAD

SINCRONIZZAZIONE TRA PROCESSI

GESTIONE MEMORIA

COMUNICAZIONE TRA PROCESSI

PROCESSI COOPERANTI

un processo è cooperante se influenza o può essere influenzato da altri processi in esecuzione nel sistema (un processo che condivide dati con altri processi)

utile per ottenere:

- **parallelizzazione dell'esecuzione** (es. multi-cpu, cluster)
- **replicazione** (es. connessioni di rete, servizi)
- **modularità** (diversi thread per funzioni diverse di una stessa applicazione; es. correttore ortografico word: posso continuare a scrivere mentre corregge; posso compiere più azioni contemporaneamente)
- **condivisione delle informazioni**

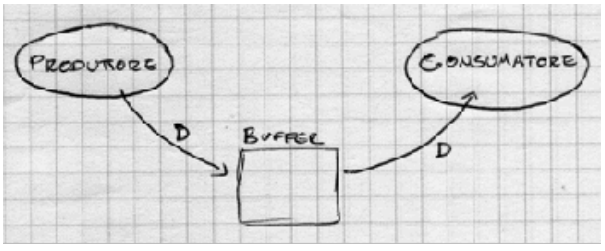
PROCESSI IN COMPETIZIONE

due processi sono in competizione se potrebbero evolvere indipendentemente ma entrano in **conflitto sulla ripartizione delle risorse**

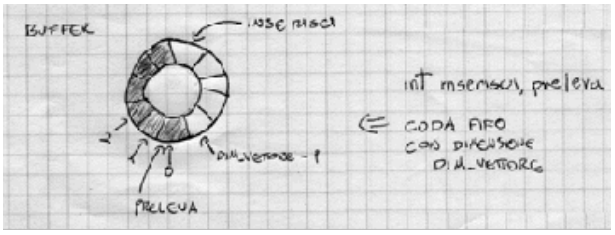
es. scheduling = **tutti i processi competono per la CPU**; coda di stampa (competizione per la risorsa stampante)

MODELLO A MEMORIA CONDIVISA (PRODUTTORE / CONSUMATORE UTILIZZANDO MEMORIA CONDIVISA)

tipologia di comunicazione **produttore/consumatore** = interazione tra 2 processi, uno produce un dato e lo scrive in un buffer (quindi non devo necessariamente sincronizzare ogni singolo dato), l'altro lo deve utilizzare (es. stampa: invio dei dati al processo che li va a stampare)



implementiamo il buffer con un array circolare di dimensione DIM_VETTORE



abbiamo 2 puntatori: inserisci e preleva che si spostano attorno all'array

si tratta di una coda FIFO con una dimensione massima prefissata (DIM_VETTORE), altrimenti si ingrandirebbe fino a sfruttare tutta la memoria

codice del processo produttore:

```
while (1){
   <produce un dato D>

   buffer[produci] = D;
   produci = produci + 1;
   // bisogna stare attenti perché quando arriviamo a

   // DIM_VETTORE bisogna tornare a 0, quindi:
   produci = (produci + 1) % DIM_VETTORE;
}
```

codice del processo consumatore:

```
while(1){
   <consuma il dato D>

   D = buffer[preleva];
   preleva = (preleva + 1) % DIM_VETTORE;
}
```

- inserisci = indica la successiva posizione libera del vettore
- preleva = indica la prima posizione occupata del vettore

problemi:

- produttore deve partire prima del consumatore
- consumatore deve aspettare che sia stato prodotto il dato
- produttore deve aspettare che dato sia stato letto prima di sovrascriverlo

soluzioni:

- controllo i valori di preleva e produci
- in base a questi posso decidere se attendere o meno (utilizzando un ciclo while)

per verificare se il buffer è vuoto guardo quando inserisci==preleva mentre guardo se è pieno quando (inserisci+1)%DIM_VETTORE=preleva; in questo modo funziona ma “spreco” una posizione del buffer, altrimenti dovrei usare un'altra variabile ma ci sono altri problemi

codice del processo produttore:

```
while (1){
   <produce un dato D>
   while((inserisci + 1) % DIM_VETTORE == preleva)
   buffer[inserisci] = D;
   inserisci = (inserisci + 1) % DIM_VETTORE;
}
```

codice del processo consumatore:

```
while(1){
   <consuma il dato D>

   D = buffer[preleva];
   preleva = (preleva + 1) % DIM_VETTORE;
}
```

questa tecnica detta “**busy waiting**”, o **attesa attiva**, è poco efficiente

un altra tecnica è quella di utilizzare una variabile globale che memorizzi il numero degli elementi scritti del buffer

codice del processo produttore:

```
while (1){
   <produce un dato D>
   while(nelementi == DIM_VETTORE - 1)
   buffer[inserisci] = D;
   inserisci = (inserisci + 1) % DIM_VETTORE;
   nelementi = nelementi + 1;
}
```

codice del processo consumatore:

```
while(1){
   <consuma il dato D>

   D = buffer[preleva];
   preleva = (preleva + 1) % DIM_VETTORE;
   nelementi = nelementi - 1;
}
```

questa è teoricamente più efficiente ma non funziona perché non c'è nessuna architettura che esegue l'istruzione “nelementi = nelementi – 1” in un unica istruzione, quindi diventa in realtà una cosa di questo genere:

```
leggi nelementi in R1
R1 = R1 - 1;
scrivi R1 nelementi
```

questo provoca errate letture di dati (pensiamo a un sistema con scheduler time-sharing)

e quindi uno dei due processi potrebbe leggere dati inconsistenti

ad esempio:

- il consumatore legge 5
- l'esecuzione passa al produttore che aggiunge elementi
- l'esecuzione torna nuovamente al consumatore che decrementa 5 a 4, ma avrebbe dovuto decrementare 6 a 5, e quindi si è perso qualcosa per strada

MODELLO A SCAMBIO DI MESSAGGI

permette ai processi di **comunicare senza ricorrere a dati condivisi**; è il modello più diffuso fino all'utilizzo della memoria condivisa, e il più utilizzato nelle reti

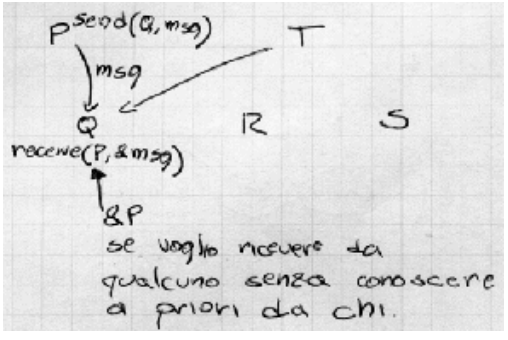
si basa su 2 primitive, che servono ad aprire un **canale di comunicazione**:

- **send**(P, msg) (ossia un processo e un messaggio)
- **receive**(Q, &msg)

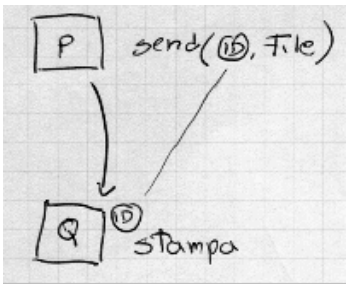
classificazione in base alla nominazione dei processi (modo con cui un processo si riferisce agli altri processi):

- diretta
- indiretta

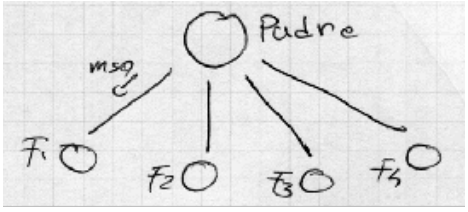
NOMINAZIONE (O NOMINA O COMUNICAZIONE) DIRETTA



il problema è che devo sapere l'ID:

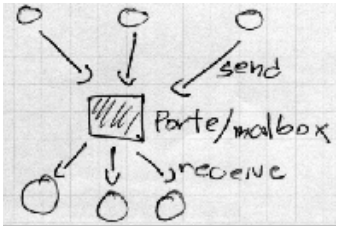


questo sistema va bene se ho un sistema del tipo qui sotto, di solito so l'ID se sono io il padre:



NOMINAZIONE INDIRETTA

si utilizza quando invece i processi sono meno correlati, come i processi di sistema, in cui non so l'ID; si basa su delle porte (o mailbox) su cui processi inviano messaggi e da cui i processi possono leggere



l'idea è quella della posta elettronica, dove noi per inviare un messaggio non andiamo a indicare l'indirizzo fisico della macchina

classificazione delle primitive in base alla sincronizzazione:

- **sincrono** (es. send sincrona blocca finché il messaggio non è stato ricevuto)
- **asincrono** (non si blocca)

la send e la receive possono essere sincrone e asincrone

generalmente in un buffer la send è asincrona mentre la receive è sincrona

generalmente in un buffer la send è asincrona mentre la receive è sincrona

PRODUTTORE/CONSUMATORE UTILIZZANDO LO SCAMBIO DI MESSAGGI

un processo produttore produce informazioni che sono consumate da un processo consumatore (es. un programma di stampa produce caratteri che sono consumati dal driver della stampante)

es.

codice produttore:

```
while(1){
<Produce D>
send(ID del Consumatore, D);
}
```

se il processo ricevente non è pronto:

- metto in attesa il processo mittente (quindi la send diventerebbe sincrona, i concetti in realtà non sono assoluti)
- ritorno un codice di errore

codice produttore migliorato:

```
while(1){
<Produce D>
if (send(ID del Consumatore, D) < 0){
<errore>
}
}
```

codice consumatore:

```
while(1){
receive(ID del produttore, &D);
<consuma D>
}
```

scritto così la receive è implicitamente sincrona

se il buffer è vuoto il sistema resta già in attesa (per come funziona la primitiva)

immaginiamo di gestire un word processor con questo sistema: diventa estremamente scomodo, per cui sono stati introdotti i thread

continua..

Ritorna sopra | Home page | Xelon