Passaggio di messaggi

Introduction

Fino ad adesso abbiamo visto che la comunicazione tra i processi veniva fatta attraverso l'utilizzo di variabili globali, adesso utilizzeremo la comunicazione diretta, tramite la quale un processo può comunicare attraverso un messaggio ad un altro processo

---

Interazione tra processi

Per l'interazione tra due processi devono essere soddisfatti due requisiti:

- sincronizzazione (mutua esclusione)
- comunicazione

Lo scambio di messaggi (*message passing*) è una soluzione al secondo requisito (funziona sia con memoria condivisa che distribuita)
Mentre per i semafori avevamo `wait` e `signal`, qui si hanno due istruzioni fondamentali `send(destination, message)` e `receive(source, message)` (`message` è un input per `send` mentre un output per `receive`) e spesso ci sta anche il test di recezione. Queste operazioni sono sempre **atomiche** (anche se si potrebbe bloccare in quando ad esempio non ha ricevuto nulla)

Sincronizzazione

La comunicazione richiede anche la sincronizzazione tra processi (il mittente deve inviare prima che il ricevente riceva).
Dunque ora capiremo quali operazioni devono essere bloccanti oppure no (il test di recezione non è mai bloccanti)

`send` e `receive` bloccanti

Se la `send` e la `receive` sono bloccanti vuol dire che un processo non può inviare un messaggio finché il precedente non è stato ricevuto e viceversa (chi prima fa l'operazione si blocca e chi la fa per seconda, oltre a non bloccarsi, sblocca anche la prima).
Tipicamente questo tipo di operazione viene chiamato *randevous* e richiede una sincronizzazione molto stretta

---

`send` non bloccante

Più tipicamente come approccio si preferisce usare la `send` non bloccante (la indicheremo con `nbsend`) e la `receive` bloccante.
In questo caso succede che il mittente continua (non importa se il messaggio è stato ricevuto o no) mentre il ricevente, se è stato eseguito per secondo non si blocca, altrimenti si blocca finché non riceve un messaggio dal mittente

---

`receive` non bloccante

Approccio piuttosto raro in cui accade che, **indipendentemente se il messaggio è stato ricevuto** o meno, l'**esecuzione continua**. Questa operazione viene indicata con `nbreceive` e può settare un bit nel messaggio per dire se la recezione è avvenuta oppure no. Se la recezione è non bloccante, allora tipicamente non lo è neanche l'invio

---

Indirizzamento

Il mittente deve poter dire a quale processo (o quali processi) vuole mandare il messaggio (lo stesso vale per il destinatario, anche se non sempre)
Si possono usare:

- **indirizzamento diretto**
- **indirizzamento indiretto**

Indirizzamento diretto

Con l'indirizzamento diretto la primitiva di `send` include uno specifico **identificatore di un processo** (o un gruppo di processi)
Per la `receive`, ci può essere oppure no. Viene infatti specificato un `sender` nel caso

in cui si vogliano ricevere messaggi solo da un determinato processo (utile per applicazioni fortemente cooperative), non viene specificato invece nel caso in cui si voglia ricevere da chiunque (dentro il messaggio ci sarà anche il mittente)

Ogni processo in questo senso ha una coda di messaggi; una volta piena, solitamente il messaggio si perde oppure viene ritrasmesso (es. syscall `listen`)

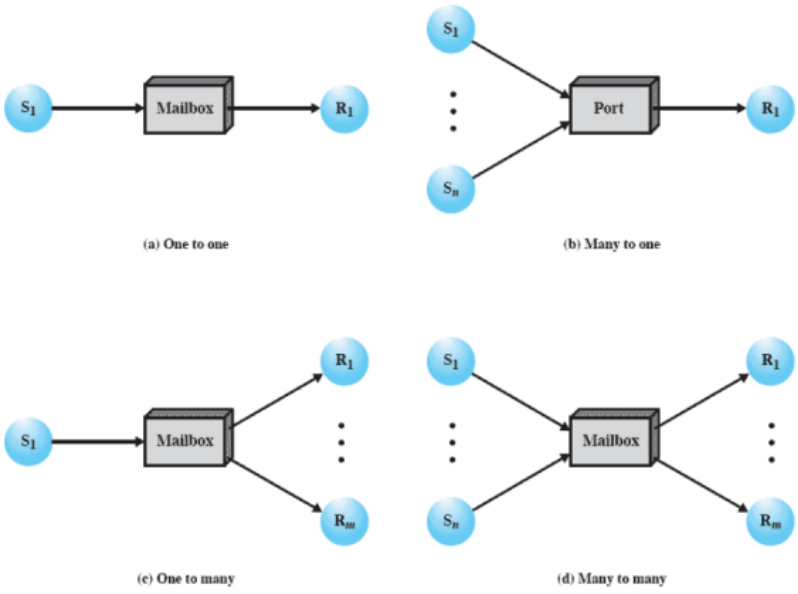
Indirizzamento indiretto

L'indirizzamento indiretto consiste nel cosiddetto sistema *mailbox* (creata esplicitamente da qualche processo).
Dunque quando un processo invia un messaggio lo mette in una determinata zona di memoria da cui poi il ricevente lo va a prendere (mittente/destinatario hanno come input la mailbox in cui si vuole mandare)

Se la ricezione è bloccante, e ci sono più processi in attesa su una ricezione, un solo processo viene svegliato (ci sono evidenti analogie con il produttore/consumatore)
Inoltre se la mailbox è piena anche `nbsend` si deve bloccare

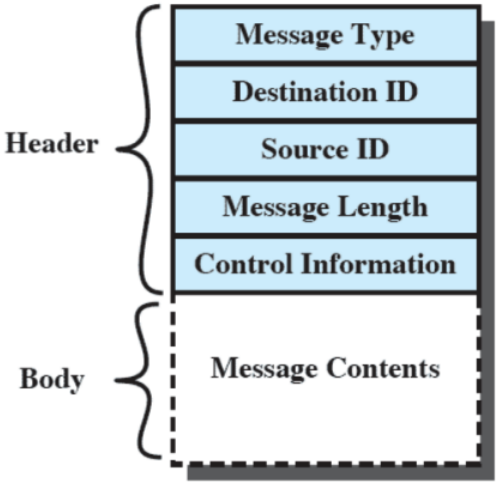
Comunicazione indiretta

Se serve solo per le versioni bloccanti e per comunicazioni x-to-one, la mailbox può avere dimensione 1



La prima riga è di fatto comunicazione diretta

Tipico formato dei messaggi



Mutua esclusione con i messaggi

```
const message null = /* null message */
mailbox box;

void P(int i) {
        message msg;
        while(true) {
                receive(box, msg);
                /* critical section */
                nbsend(box, msg);
                /* remainder */
        }
}

void main() {
        box = create_mailbox();
        // per prima cosa ci mando un messaggio, altrimenti non si
entrerebbe
        // mai nella critical section in quanto receive è bloccante e
nbsend no
        nbsend(box, null);
        parbegin (P(1),P(2),...,P(n));
}
```

Produttore/consumatore con i messaggi

Vediamo ora una soluzione per risolvere il problema di produttore/consumatore attraverso l'utilizzo dei messaggi (con qualsiasi numero di produttori e consumatori)

```
const int capacity = /* buffering capacity */;
mailbox mayproduce, mayconsume;
const message null = /* null message */;

void main() {
        mayproduce = crate_mailbox();
        mayconsume = create_mailbox();
        // facendo in questo modo mi permette di produrre tanto
quanto e la
        // capacità del buffer
        for(int i=1; i<=capacity; i++) {
                nbsend(mayproduce, null);
        }
        parbegin(producer, consumer);
}

void producer() {
        message pmsg;
        while(true) {
                receive(mayproduce, pmsg);
                pmsg = produce();
                // append....
                nbsend(mayconsume, pmsg);
        }
}

void consumer() {
        message cmsg;
        while(true) {
                receive(mayconsume, cmsg);
                consume(cmsg);
                nbsend(mayproduce, null);
        }
}
```

Mutua esclusione → ok
Deadlock → ok
Stravation → ok solo se le code di processi bloccati su una `receive` sono gestite in modo "forte" (i processi si sbloccano secondo un ordine FIFO)