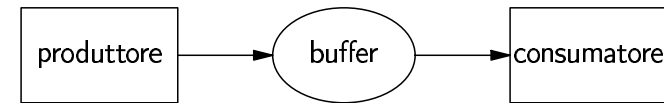


Ricordate!

Il diario delle lezioni e il forum sono accessibili da

<http://matteo.vaccari.name/so/>

Problema dei *Produttori e Consumatori*



buffer di dimensione limitata

il produttore deve attendere se il buffer è pieno

il consumatore deve attendere se il buffer è vuoto

Produttori e Consumatori, cont.

Soluzione con mutex, sleep e wakeup

```
void producer() {  
    while (1) {  
        item = produce_item();  
        insert_item(item);  
    }  
}
```

```
void consumer() {  
    while (1) {  
        item = remove_item();  
        consume_item(item);  
    }  
}
```

```
void producer() {  
    while (1) {  
        if (N == count) sleep();  
        item = produce_item();  
        lock(m);  
        insert_item(item);  
        count++;  
        if (1 == count) wakeup(consumer);  
        unlock(m);  
    }  
}
```

```
void consumer() {  
    while (1) {  
        if (0 == count) sleep();  
        lock(m);  
        item = remove_item();  
        count-;  
        if (N-1 == count) wakeup(producer);  
        unlock(m);  
        consume_item(item);  
    }  
}
```

Problema: *lost wakeup*

Se il wakeup viene eseguito *prima* che l'altro processo esegua *sleep*?

Soluzione di Java, e di Posix Threads

- sleep si può eseguire solo se ho il lock di un mutex
- eseguendo sleep perdo il lock

Portiamo la sleep dentro la CS (Java, Posix Threads)

```
void producer() {
    while (1) {
        item = produce_item();
        lock(m);
        if (N == count) sleep();
        insert_item(item);
        count++;
        if (1 == count) wakeup(consumer);
        unlock(m);
    }
}

void consumer() {
    while (1) {
        lock(m);
        if (0 == count) sleep();
        item = remove_item();
        count--;
        if (N-1 == count) wakeup(producer);
        unlock(m);
        consume_item(item);
    }
}
```

E se ci sono più di due threads?

```
void producer() {
    while (1) {
        item = produce_item();
        lock(m);
        while (N == count) sleep();
        insert_item(item);
        count++;
        if (1 == count) wakeup(consumer);
        unlock(m);
    }
}

void consumer() {
    while (1) {
        lock(m);
        while (0 == count) sleep();
        item = remove_item();
        count--;
        if (N-1 == count) wakeup(producer);
        unlock(m);
        consume_item(item);
    }
}
```

Semafori

- Dijkstra, 1965
- Due operazioni: down e up *atomiche per definizione*
- Down:
 - se sem > 0 allora
 - decrementa sem
 - altrimenti
 - sospendi il thread corrente
- Up:
 - se ci sono thread in attesa sul semaforo
 - svegliane uno
 - altrimenti
 - incrementa sem

Mutua esclusione con semafori

```
/* il semaforo è una variabile globale */
semaphore sem = 1;

/* codice eseguito da ciascun thread: */
while (1) {
    DoSomeWork();
    DOWN(sem);
    EnterCriticalSection();
    UP(sem);
}
```

Strutture dati thread-safe

Una struttura dati è thread safe solo se:

- l'accesso è consentito solo tramite apposite procedure
- l'accesso di più thread concorrenti non crea problemi

Meglio nascondere il codice di sincronizzazione nelle procedure di accesso ai dati, piuttosto che nel codice dei thread come fa Tanenbaum

Produttori-consumatori con semafori

```
semaphore mutex = 1;
semaphore free = N;
semaphore busy = 0;

void producer() {
    while (1) {
        item = produce_item();
        down(free);
        down(mutex);
        insert_item(item);
        up(mutex);
        up(busy);
    }
}

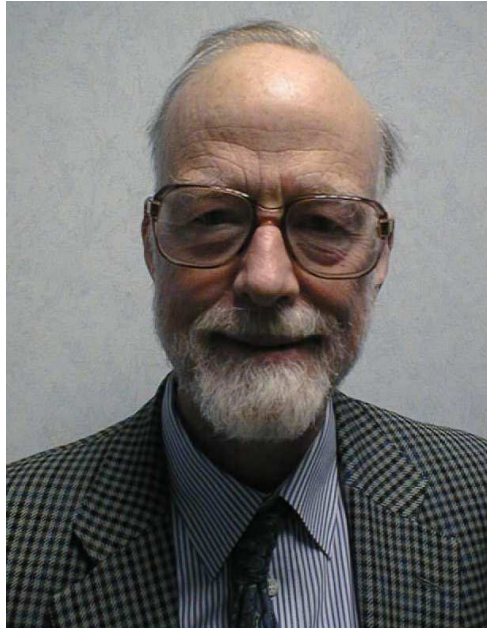
void consumer() {
    while (1) {
        down(busy);
        down(mutex);
        item = remove_item();
        up(mutex);
        up(free);
        consume_item(item);
    }
}
```

```
ITEM buf[N+1];
int head = 0;
int tail = 0;
semaphore mutex = 1;
semaphore free = N;
semaphore busy = 0;

void enter_item(ITEM it) {
    down(free);
    down(mutex);
    buf[tail] = it;
    tail = (tail+1) % (N+1);
    up(mutex);
    up(busy);
}

ITEM remove_item() {
    ITEM it;
    down(busy);
    down(mutex);
    it = buf[head];
    head = (head+1) % (N+1);
    up(mutex);
    up(free);
    return it;
}
```

Charles Anthony Richard Hoare, (1934–)



Un punto cruciale

La mutua esclusione non basta

Occorre poter bloccare i processi

```
condition c;  
...  
wait(c);  
...  
signal(c);
```

Monitor: Tony Hoare (1974), Per Brinch Hansen (1975)

Esempio di monitor:

```
monitor example;  
  integer i;  
  condition c;  
  
  procedure producer;  
  ...  
  end;  
  
  procedure consumer;  
  ...  
  end;  
end monitor;
```

le var del monitor sono *private*

la mutua esclusione è garantita

le var di tipo *condition* supportano
le op *wait* e *signal*

Cosa succede quando un thread esegue `signal(c)`?

Hoare: il processo che esegue `signal` viene bloccato

Hansen: la `signal` deve apparire come ultima istruzione

Gosling (Java): il processo svegliato è fuori dalla sezione critica

Producer-consumer con monitor

```
monitor ProducerConsumer
  condition c;
  integer count;
  procedure insert(item: integer)
  begin
    while count = N do wait(c);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(c);
  end;

  function remove: integer
  begin
    while count = 0 do wait(c);
    remove = remove_item;
    count := count - 1;
    if count = N-1 then signal(c);
  end;
end monitor;
```

```
procedure producer;
begin
  while true do
  begin
    item := produce_item;
    ProducerConsumer.insert(item);
  end
end;

procedure consumer;
begin
  while true do
  begin
    item := ProducerConsumer.remove;
    consume_item(item);
  end
end;
```

Equivalenza fra semafori e monitor

```
monitor semaphore
  condition c;
  integer n = 0;

  procedure up
  begin
    n := n + 1;
    signal(c);
  end

  procedure down
  begin
    while n = 0 do wait(c);
    n := n - 1;
  end
end monitor;
```

Posso implementare un semaforo come monitor

Java implementa il concetto di *monitor*

Mutua esclusione: occorre dichiarare *synchronized*

Condition variables: qualunque oggetto supporta *wait* e *notify*

Occorre essere *sincronizzati* sull'oggetto prima

Altrimenti: `IllegalMonitorStateException`

Posso implementare un monitor con i semafori

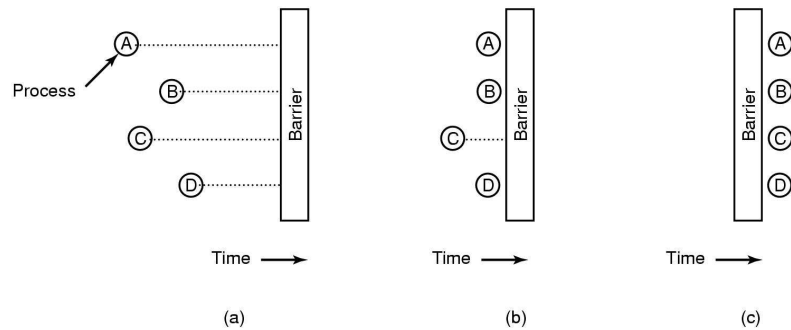
mutua esclusione: facile, uso un semaforo *mutex* per ogni monitor

- metto `down(mutex)` all'ingresso di ogni metodo
- e `up(mutex)` all'uscita

variabili condition: le implemento con un array di semafori

- un semaforo \forall processo \forall condition
- `wait: down(sem[pid])`
- `signal: foreach (pid in P) up(sem[pid])`

Barriere



Esercizio

Implementare le barriere per mezzo di:

- monitor (facile)
- semafori (leggermente meno facile)

message passing

Problema: sincronizzare processi **senza memoria condivisa**

Soluzione: *message passing*

send(destinazione, messaggio)

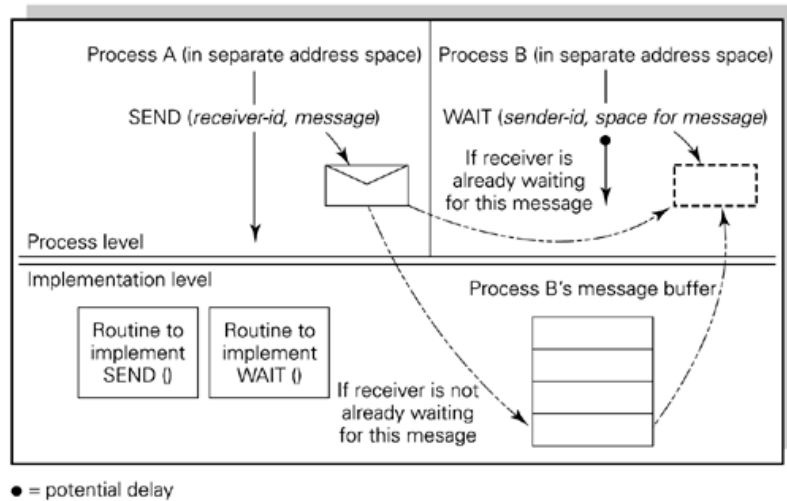
receive(destinazione, messaggio)

Message passing: sincrono o asincrono?

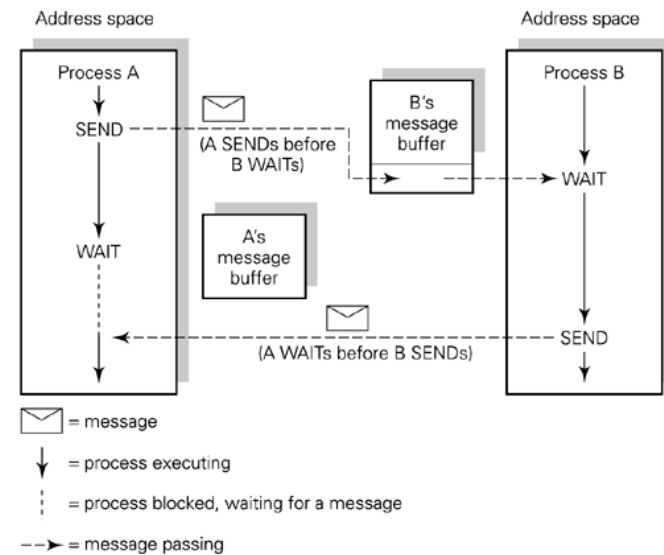
Asincrono:

- ▶ send e receive non bloccano ...
- ▶ (ho una *coda* di messaggi)
- ▶ ... a meno che la coda non sia piena (send) o vuota (receive)

Un sistema di scambio messaggi asincrono



Asynchronous message passing



Message passing: sincrono o asincrono?

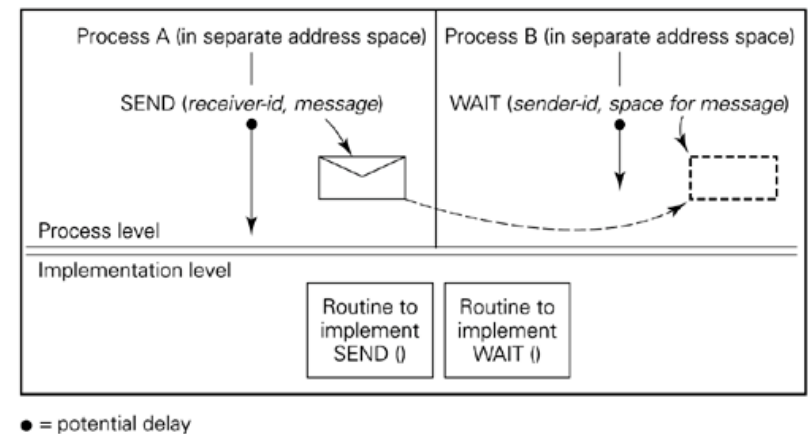
Un sistema di scambio messaggi sincrono

Sincrono:

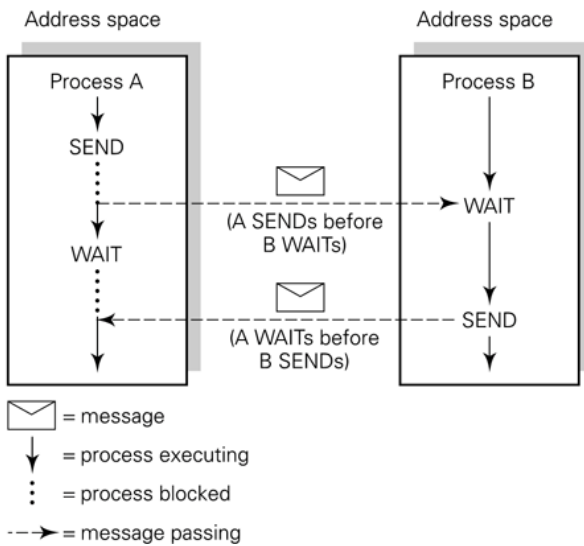
- ▶ se la send è eseguita prima ⇒ si blocca fino alla esecuzione di receive
- ▶ e viceversa

“Rendez-vous”

Esempio: Ada



Synchronous message passing



Canali inaffidabili

I messaggi possono andare perduti

⇒ acknowledge

Lost acknowledge

⇒ retransmit

Messaggi duplicati

⇒ numerazione

Problemi del message passing

- canali inaffidabili
- naming
- autenticazione
- dimensione dei messaggi

se i processi sono locali:

- performance

Naming

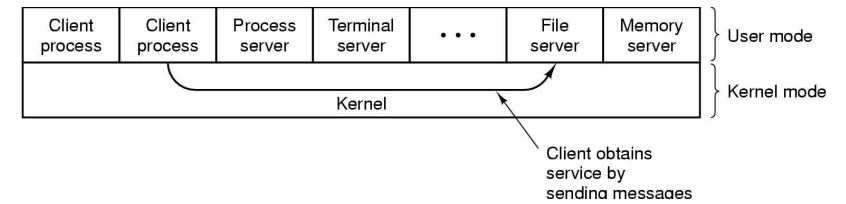
Mailboxes vs process addresses

(es. Unix pipes)

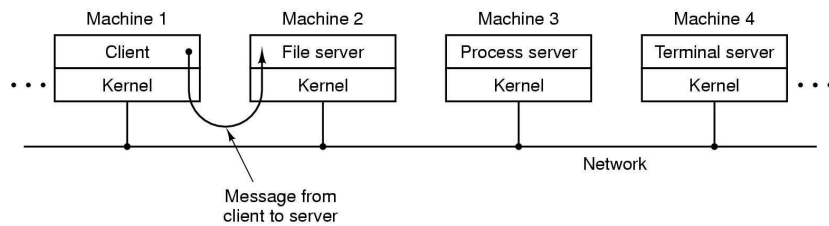
Message passing in Minix

- ▶ sincrono (rendez-vous)
- ▶ dimensione fissa
- ▶ destinatario: un processo. sorgente: un processo oppure ANY

Client-server (microkernel) OS architecture



Client-server (microkernel) OS architecture



Interazione client-server

```
// client
...
send(server, request);
receive(server, reply);
...

// server
while (true) {
    receive(ANY, request);
    do_request();
    send(client, reply);
}
```

Produttore consumatore con message-passing

```
#define N 100                /* number of slots in the buffer */

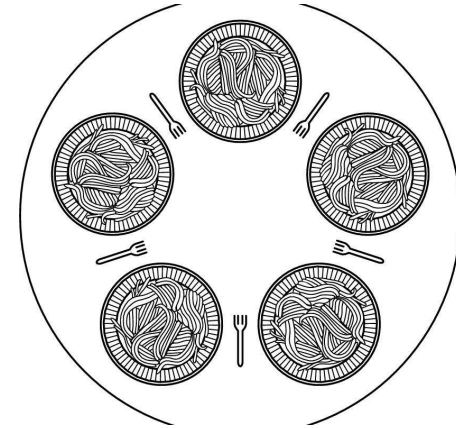
void producer(void)
{
    int item;
    message m;                /* message buffer */

    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        send(consumer, &m); /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

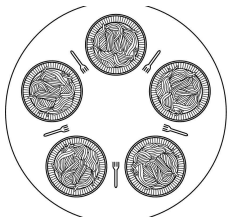
    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&m); /* extract item from message */
        send(producer, &m); /* send back empty reply */
        consume_item(item); /* do something with the item */
    }
}
```

Il problema dei filosofi a cena (Dijkstra, 1971)



Il problema dei filosofi a cena

- ▶ Per mangiare occorrono 2 forchette
- ▶ Le forchette vanno acquisite una per volta
- ▶ Come prevenire lo stallo?



Una non-soluzione

```
#define N 5
void philosopher(int n) {
    while (1) {
        think();
        take_fork(n);
        take_fork((n+1) % N);
        eat();
        put_fork(n);
        put_fork((n+1) % N);
    }
}
```

Un'altra non-soluzione:

```
void philosopher(int n) {
    while (1) {
        think();
        while (1) {
            take_fork(n);
            if (available((n+1) % N)) {
                take_fork((n+1) % N);
                break;
            } else {
                put_fork(n);
                put_fork((n+1) % N);
            }
        }
    };
    eat();
    put_fork(n);
    put_fork((n+1) % N);
}
}
```

La soluzione ovvia

```
#define N 5
semaphore mutex = 1;
void philosopher(int n) {
    while (1) {
        think();
        down(mutex);
        take_fork(n);
        take_fork((n+1) % N);
        eat();
        put_fork(n);
        put_fork((n+1) % N);
        up(mutex);
    }
}
```

Qual'è il problema di questa soluzione?

Si può migliorare...

inserendo un ritardo scelto in modo casuale

⇒ riduco la probabilità di *starvation*

algoritmo usato sulla rete *Ethernet*

... ma noi vogliamo una soluzione deterministica

...il problema è che...

Consente di mangiare ad un filosofo solo per volta

Proviamo ancora

```
#define N 5
semaphore mutex = 4;
void philosopher(int n) {
    while (1) {
        think();
        down(mutex);
        take_fork(n);
        take_fork((n+1) % N);
        eat();
        put_fork(n);
        put_fork((n+1) % N);
        up(mutex);
    }
}
```

Va meglio?

Altra soluzione: ordiniamo le risorse

Le forchette vanno prese in ordine crescente \Rightarrow fil. 0: prima forchetta 0, poi forchetta 1 \Rightarrow fil. 1: prima 1, poi 2
... \Rightarrow fil. 4: prima 0, poi 4

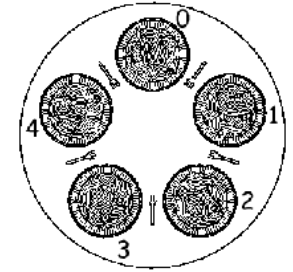
stallo: impossibile

parallelismo: non ottimale!

...sì ma se...

filosofo 0 sta mangiando;
filosofo 1 sta aspettando la forch. destra;
filosofo 2 sta aspettando la forch. destra;

fil. 2 non può mangiare



Soluzione ottimale (Dijkstra, 1971)

```
#define LEFT    ((i+N-1) % N)
#define RIGHT   ((i+1) % N)
#define THINKING 0
#define HUNGRY  1
#define EATING  2

int state[N];
semaphore mutex = 1;
semaphore s[N] = { 0, 0, ..., 0 };

void philosopher(i) {
    while (1) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

Soluzione di Dijkstra, cont.

```
void take_forks(int i) {
    down(mutex);
    state[i] = HUNGRY;
    test(i);
    up(mutex);
    down(s[i]);
}

void put_forks(int i) {
    down(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(mutex);
}

void test(int i) {
    if (HUNGRY == state[i] &&
        EATING != state[LEFT] && EATING != state[RIGHT])
    {
        state[i] = EATING;
        up(s[i]);
    }
}
```

Problema: *Readers and Writers*

Accesso condiviso a una risorsa (es. tabella di database)

Accesso consentito a:

- *un solo* scrittore, oppure
- un numero illimitato di lettori

Problemi della programmazione concorrente

- ▶ race conditions
- ▶ deadlock
- ▶ starvation
- ▶ livelock

Readers and writers, soluzione

```
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

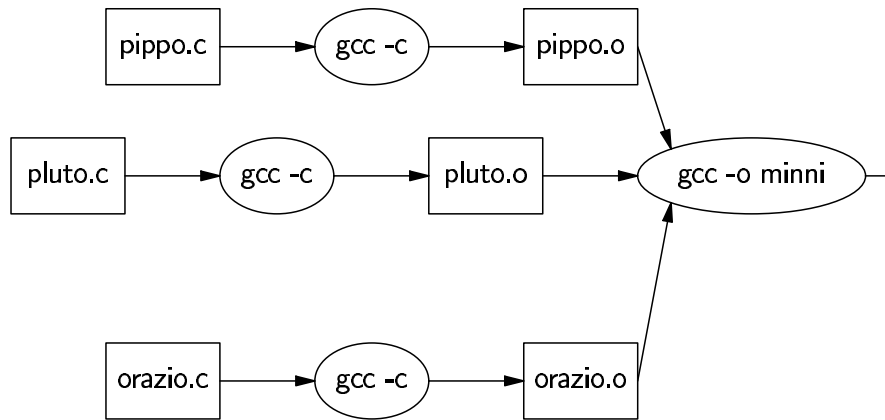
void reader() {
    while (1) {
        down(mutex);
        rc++;
        if (1 == rc) down(db);
        up(mutex);

        read_data_base();

        down(mutex);
        rc--;
        if (0 == rc) up(db);
        up(mutex);
    }
}

void writer() {
    while (1) {
        think_up_data();
        down(db);
        write_data_base();
        up(db);
    }
}
```

Il sistema di programmazione C



Un sistema per la manutenzione di programmi

Il comando "make(1)"

esempio di *makefile*

```
OBJ = pippo.o pluto.o orazio.o
minni: $(OBJ)
    gcc -o minni $(OBJ)
```

strumento fondamentale per compilare Linux

gcc: compiler driver

Fornisce un'interfaccia (command line) ai compilatori e al linker

Documentazione:

- man gcc
- info gcc (più estesa)

Alcune opzioni utili per gcc

- c: crea il .o; non linkare
- g: informazioni per il debug
- O: ottimizza
- Wall: massimo dei warning
- S: lascia i file assembler
- v: mostra i comandi generati per le varie fasi

Debugger

Il debugger GNU si chiama gdb

Interfaccia grafica: ddd