Sistemi Operativi
Esercizi Sincronizzazione

Docente: Claudio E. Palazzi
cpalazzi@math.unipd.it

Semafori (1)

- Semafori: variabili intere
  - contano il numero di richieste pendenti
  - il valore è 0 se non ci sono risorse disponibili a servire richieste (che quindi diventeranno pendenti) e > 0 altrimenti
- Due operazioni atomiche standard *Down* e *Up* (P e V)
  - down(S) … equivalente di P(S)
    - se S > 0 allora S= S - 1 ed il processo continua l'esecuzione
    - se S==0 ed il processo si blocca senza completare la primitiva
  - up(S) … equivalente di V(S)
    - se ci sono processi in attesa di completare la down su quel semaforo (e quindi necessariamente S == 0) uno di questi viene svegliato e S rimane a 0, altrimenti S viene incrementato;
    - in caso contrario (S > 0), allora S=S + 1

Semafori (2)

Soluzione del Produttore/ Consumatore con semafori

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Monitor (1)

- Oggetti (Strutture dati + procedure per accedervi)
- Mutua esclusione nell'esecuzione delle procedure
- Variabili di Condizione + wait() e signal()
- wait(X)
  - sospende sempre il processo che la invoca in attesa di una signal(X)
- signal(X)
  - sveglia uno dei processi in coda su X
  - se nessun processo è in attesa va persa
  - deve essere eseguita solo come ultima istruzione prima di uscire dal monitor (il processo svegliato ha l'uso esclusivo del monitor)

Monitor (2)

Esempio di monitor

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Monitor (3)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
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;
```

- Schema di soluzione Produttore/Consumatore
  - ad ogni istante solo una procedura del monitor è in esecuzione
  - il buffer ha *N* posizioni

Monitor (4)

- Caratteristiche principali dei monitor Java
  - ME nell'esecuzione dei metodi synchronized
  - non ci sono variabili di condizione
  - wait(), notify() simili a sleep() , wakeup()
  - è possibile svegliare tutti i processi in attesa

Monitor (5)

```
public class ProducerConsumer {
    static final int N = 100;                    // constant giving the buffer size
    static producer p = new producer( );         // instantiate a new producer thread
    static consumer c = new consumer( );// instantiate a new consumer thread
    static our_monitor mon = new our_monitor( );  // instantiate a new monitor
    public static void main(String args[ ]) {
       p.start( );                               // start the producer thread
       c.start( );                               // start the consumer thread
    }
    static class producer extends Thread {
       public void run( )  {                     // run method contains the thread code
          int item;
          while (true) {                         // producer loop
             item = produce_item( );
             mon.insert(item);
          }
       }
       private int produce_item( ) { ... }       // actually produce
    }
    static class consumer extends Thread {
       public void run( )  {                     run method contains the thread code
          int item;
          while (true) {                         // consumer loop
             item = mon.remove( );
             consume_item (item);
          }
       }
       private void consume_item(int item) { ... }     // actually consume
    }
```

Soluzione per Produttore/Consumatore in Java (parte 1)

Monitor (6)

```
static class our_monitor {                  // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );   // if the buffer is full, go to sleep
        buffer [hi] = val;                // insert an item into the buffer
        hi = (hi + 1) % N;                // slot to place next item in
        count = count + 1;                // one more item in the buffer now
        if (count == 1) notify( );        // if consumer was sleeping, wake it up
    }
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );   // if the buffer is empty, go to sleep
        val = buffer [lo];                // fetch an item from the buffer
        lo = (lo + 1) % N;                // slot to fetch next item from
        count = count – 1;                // one few items in the buffer
        if (count == N – 1) notify( );    // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
}
```

Soluzione per Produttore/Consumatore in Java (parte 2)

Scambio Messaggi (1)

- Non richedono accesso a supporti di memorizzazione comune
- primitive base
  - send(destination,&msg)
  - receive(source, &msg)
- decine di varianti, nel nostro caso :
  - la receive blocca automaticamente se non ci sono messaggi
  - i messaggi spediti ma non ancora ricevuti sono bufferizzati dal SO
  - Tipo mailbox

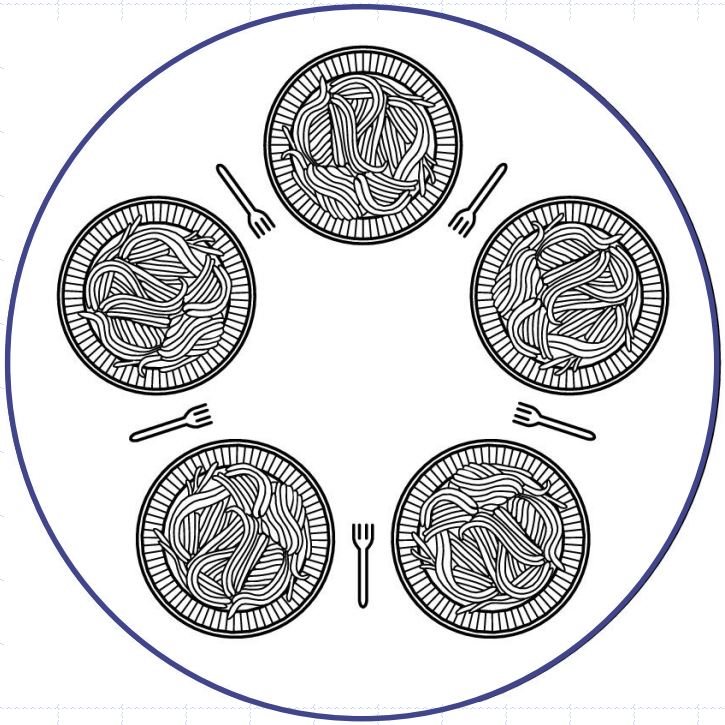
Scambio Messaggi (2)

```
#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 */
     }
}
```

Sol. Produttore/Consumatore con *N* messaggi

I filosofi a cena (1)

- I filosofi mangiano e pensano
- Per mangiare servono due forchette
- Ogni filosofo prende una forchetta per volta
- Come si può prevenire il *deadlock*

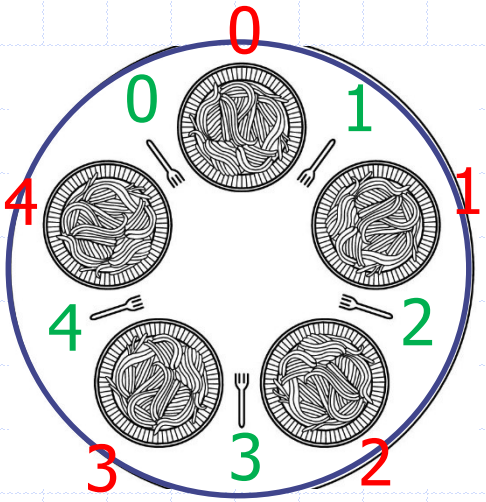


N Filosofi a Cena: Semafori Alt.

```
Filosofo(i) {
    while(1) {
        <pensa>
        if(i == X) {
            P(f [(i+1)%N]);
            P(f [i]);
        } else {
            P(f [i]);
            P(f [(i+1)%N]);
        }
        <mangia>
        V(f [i]);
        V(f [(i+1)%N]);
    }
}
```

Inizializzazione:
int semaforo **f[i]** = 1;

Per evitare deadlock inseriamo un filosofo "***mancino***": ad esempio, il filosofo ***X***



(4+1)%5 = 0

5 filosofi a cena coi monitor

```
Monitor Tavolo{
   boolean fork_used[5] = false; // forchette numerate da 0 a 4
   condition filosofo[5]; // se lo vogliamo fare in java, questa la dobbiamo
                                    togliere

   raccogli(int n){
      while(fork_used[n] || fork_used[(n+1)%5])
         filosofo[n].wait();
      fork_used[n] = true;
      fork_used[(n+1)%5] = true;
   }
   // in java dovevi aggiungere:
   // (synchronized)
   deposita(int n){
      fork_used[n] = false;
      fork_used[(n+1)%5] = false;
      filosofo[n].notify();   // se lo voglio fare in java devo togliere
                                                queste due "filosofo" e sostituire con
                                                notifyall()

      filosofo[(n+1)%5].notify();
   }
}
Filosofo(i){
   while (true){
      <pensa>
      Tavolo.raccogli(i);
      <mangia>
      Tavolo.deposita(i);
   }
}
```

Il problema dei lettori e scrittori (1)

- Un database molto esteso (db)
  - es. prenotazioni aeree ...
- Un insieme di processi che devono leggere o scrivere in db
- Più lettori possono accedere contemporaneamente a db
- Gli scrittori devono avere accesso esclusivo a db
- I lettori hanno precedenza sugli scrittori
  - se uno scrittore chiede di accedere mentre uno o più lettori stanno accedendo a db, lo scrittore deve attendere che i lettori abbiano finito

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

Con il semaforo mutex gestisco la mutua esclusione sulle variabili condivise

Con il semaforo db gestisco l'alternanza lettori-scrittori sul database; solo il primo lettore di una sequenza deve passare attraverso questo semaforo

Il barbiere sonnolento (1)

- Un barbiere dorme sulla poltrona di lavoro quando non ci sono clienti
- All'arrivo del primo cliente, il barbiere si sveglia, lo fa sedere sulla poltrona e lo serve
- Se arrivano altri clienti mentre il barbiere è al lavoro, si siedono su una sedia in attesa
- C'è un numero massimo di sedie per l'attesa (es. 5) oltre il quale ulteriori clienti se ne vanno dal negozio senza attendere di essere serviti
- Finito di servire un cliente, questi esce dal negozio e un altro cliente in attesa viene servito

Il barbiere sonnolento (2): soluz.

Il semaforo **customer** viene utilizzato per addormentare il barbiere in assenza di clienti e risvegliarlo quando arrivano

Il semaforo **barbers** viene utilizzato per occupare il barbiere e addormentare un cliente se il barbiere è già occupato a servire altri

Il semaforo **mutex** viene utilizzato per garantire la mutua esclusione sull'uso della variabile condivisa **waiting**

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```