SAPIENZA
UNIVERSITÀ DI ROMA

Corso di laurea in Ingegneria dell'Informazione
Indirizzo Informatica

Reti e sistemi operativi

Sincronizzazione dei processi

Motivazioni

- Processi (o thread) cooperanti devono poter accedere a risorse condivise (memoria, file, …)
- L'accesso concorrente non regolato a risorse condivise può facilmente portare ad **inconsistenze**, con conseguenze anche catastrofiche.
- Il caso più comune di possibili consistenze si verifica in caso di accessi non sincronizzati a memoria condivisa

Motivazioni (2/2)

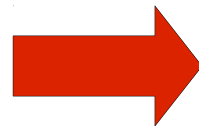
- Si prenda il caso di due processi P1 (bonifico) e P2 (prelievo) che accedano concorrentemente ad una memoria condivisa che rappresenta il saldo del nostro conto corrente (variabile **saldo)**

```
void bonifico( int importo )
{
  int nuovo_saldo = saldo + importo;
  saldo = nuovo_saldo;
}
```

```
void prelievo( int importo )
{
  int nuovo_saldo = saldo - importo;
  saldo = nuovo_saldo;
}
```

- Si supponga che inizialmente il nostro saldo sia 2000 €, e che stiamo prelevando 200 € nello stesso istante che sta arrivando il bonifico del nostro stipendio di 1000 €. Si supponga che lo scheduler che gestisce il conto utilizza un politica RR, una possibile schedulazione potrebbe essere la seguente:

```
...
int nuovo_saldo = saldo – importo; // P2
int nuovo_saldo = saldo + quota; // P1
saldo = nuovo_saldo; // P1
saldo = nuovo_saldo; // P2
...
```



saldo = 1800!!!

Race condition

Race condition e critical section

- Tale fenomeno di verifica quando il **risultato finale dell'esecuzione dei processi dipende dalla sequenza con cui vengono eseguiti**.
- Per il corretto funzionamento di un sistema di processi cooperanti, è necessario evitare qualsiasi condizione di race condition, ad esempio assicurandosi che quando un processo sta utilizzando una certa porzione di memoria condivisa, nessun altro processo potrà accedervi.
- **Critical section**: è una porzione di codice che accede a una risorsa condivisa con altri processi.

Critical section (1/2)

- Dato un insieme di n processi cooperanti $\{p_0, \dots p_{n-1}\}$
  - Ogni processo ha la sua critical section, all'interno della quale può modificare variabili comuni, accedere a file comuni, ecc..
  - Quando un processo entra nella sua sezione critica, nessun altro può entrare nella propria
- Ogni processo deve chiedere il permesso di entrare nella sezione critica attraverso una **entry section**, e uscirne attraverso una **exit section**.

Critical section (2/2)

- Le entry ed exit section sono anch'esse porzioni di codice, progettate per gestire l'accesso e l'uscita dalla critical section.

```
while(true)
{
    ENTRY SECTION

    Critical section

    EXIT SECTION

    Non critical section (resto del codice)
}
```

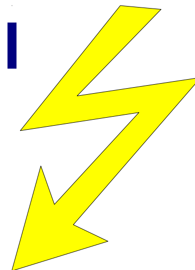
- In caso di **preemptive-kernel**, ovvero di un kernel che **accetta interruzioni anche quando sta servendo un'altra interruzione**, è necessario implementare delle critical section anche all'interno del sistema operativo.

Critical section: soluzione

- **Mutua esclusione**: se il processo Pi sta eseguendo la propria CS, nessun altro processo starà eseguendo la propria.
- **Progresso**: se nessun processo sta eseguendo la propria CS e ci sono processi in attesa di entrare nella propria, allora il prossimo ad entrare sarà scelto tra questi ultimi e **tale scelta non potrà essere rinviata all'infinito**. → “Prima o poi, una scelta viene effettuata”
- **Attesa limitata:** dopo che un processo ha richiesto di entrare nella propria CS, e prima che questa richiesta venga esaudita, ci deve essere **un limite di possibili accessi alle sezioni critiche per gli altri processi**. → “Prima o poi uno specifico processo in attesa viene scelto e può entrare nella propria CS”

Soluzione 1

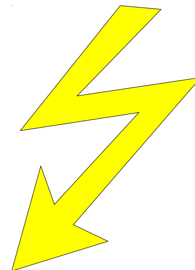
- Applicabile a due soli processi P_0 e P_1.
- Se turn = 1, allora P_1 può entrare nella propria CS.
- Funziona solo se i due processi sono strettamente alternati!
- Se turn = 0, allora P_1 non può entrare nella propria CS anche se P_0 non è all'interno della propria → **non soddisfa il requisito di progresso**



```
/* Processo i */
while(true)
{
    while (turn != i) { /* NO OP.*/ }
    Critical section
    tunr = j;
    Non critical section
}
```

Soluzione 2

- Applicabile a due soli processi P_0 e P_1.
- Se flag[1] è settato a *true*, significa che P_1 è **pronto per entrare** nella propria CS.
- In caso di race condition, potrebbe accadere che entrambi flag[0] e flag[1] siano impostati a *true*, con il risultato che nessuno potrà più entrare nella propria CS → **non soddisfa il requisito di progresso**



```
/* Processo i */
while(true)
{
    flag[i] = true
    while ( flag[j] ) { /* NO OP.*/ }

    Critical section

    flag[i] = false;

    Non critical section
}
```

Soluzione di Peterson (1/2)

- Applicabile a due processi P_0 e P_1, ma può essere estesa a più processi.
- Impiega entrambe le soluzioni introdotti in precedenza → La variabile turn definisce di chi è il turno di entrare, l'array di falg indica chi è pronto ad entrare.
- soddisfa tutte e tre i requisiti, assumendo che le istruzioni base load e store siano **atomiche**, ovvero **non interrompibili**.

```
/* Processo i */
while(true)
{
    flag[i] = true
    turn = j;
    while ( flag[j] && turn == j ) { }

    Critical section

    flag[i] = false;

    Non critical section
}
```

Soluzione di Peterson (2/2)

- Mutua esclusione: si prova per assurdo, ovvero assumendo entrambi i processi in CS → flag[0] == flag[1] == true → se uno dei due processi è **appena** entrato in CS, l'altro deve essere al più alle prese con l'ultimo controllo del ciclo while() → **non può entrare**.
- Progresso e attesa limitata si provano in maniera simile
  - Se nessuno dei due processi è in CS → flag[0] == flag[1] == false → l'entrata di uno dei due è diretta (progresso e attesa limitata).
  - Se entrami i processi sono in procinto di entrare in CS → flag[0] == flag[1] == true → turn è a 0 oppure a 1 → uno dei due processi è abilitato ad entrare in CS (progresso)
  - Se un processo è in CS mentre l'altro in attesa, quest'ultimo entrerà in CS non appena il primo ha terminato la sua CS, resettando il flag relativo (a.l.)

Sincronizzazione in hardware (1/2)

- La maggior parte delle architetture attuali fornisce un supporto hardware per le CS.
  - Possibilità di disabilitare le interruzioni (ad esempio, quel che spesso viene fatto per implementare un semplice kernel non-preemptive) → soluzione inefficiente in sistemi multiprocessore
  - Possibilità di chiamare delle **istruzioni atomiche di “lock” progettate ad-hoc**.
    - Testare e settare un valore nella stessa istruzione
    - Scambiare il contenuto di di due variabili nella stessa istruzione

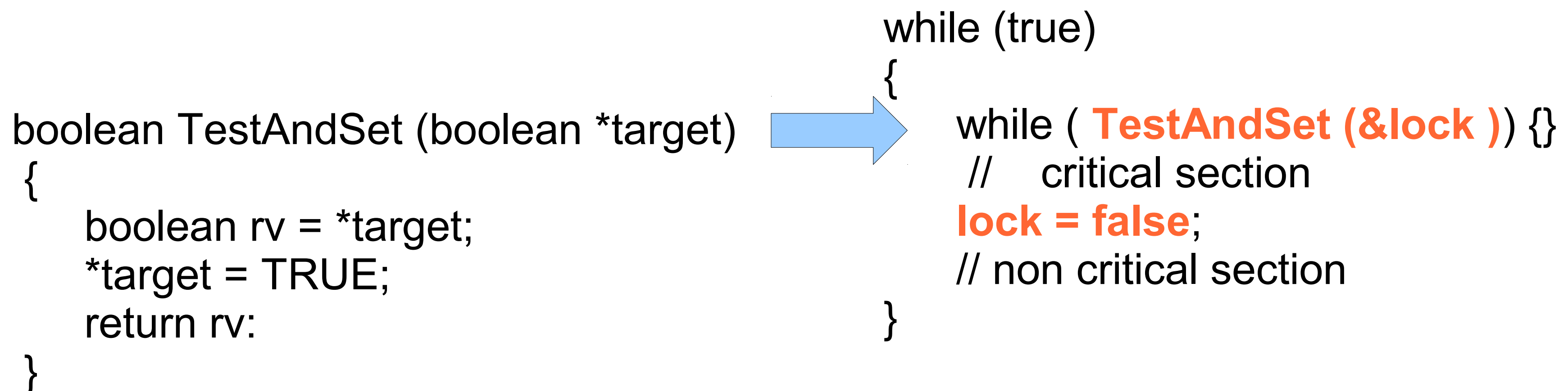
Sincronizzazione in hardware (2/2)

- Dal lato software, facilita di non poco le cose
- Non così dal lato hardware....

```
while(true)
{
    Acquisisci il lock (istruzione atomica)

    Critical section

    Rilascia il lock (istruzione atomica)

    Non critical section
}
```

Caso 1: istruzione TestAndSet

- Funzione atomica implementata in hardware → non può essere interrotta quindi non è divisibile in sotto-istruzioni



- Si utilizza una variabile booleana condivisa (**lock**) inizializzata a false:
  - TestAndSet() per richiedere (e in caso acquisire) il lock
  - Set a false per rilasciare il lock

Caso 2: istruzione Swap

- Funzione atomica implementata in hardware → non può essere interrotta quindi non è divisibile in sotto-istruzioni

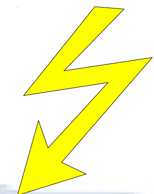
```
void Swap (boolean *a, boolean *b)
 {
       boolean temp = *a;
       *a = *b;
       *b = temp:
 }}
```

```
while (true)
{
      key = TRUE;
      while ( key == true)
          Swap (&lock, &key );
      //    critical section
      lock = false;
     // non critical section
}
```

- Si utilizza una variabile booleana condivisa (**lock**) inizializzata a false, ogni processo possiede una variabile **key** inizializzata a true prima di richiedere il lock.

Problema attesa non limitata

- **Problema**: Entrambi gli approcci presentati non soddisfano il requisito dell'attesa limitata
- Si pensi ad esempio a 2 processi P_1 e P_2 schedulati in round robin nei quali il primo passa ciclicamente molto tempo nella critical section **in fase con il quanto RR**:
  - Il processo entrerà sempre nella sua CS → requisito progresso OK
  - Il processo P_2 non entrerà mai nella sua CS → **NO requisito attesa limitata!!!**



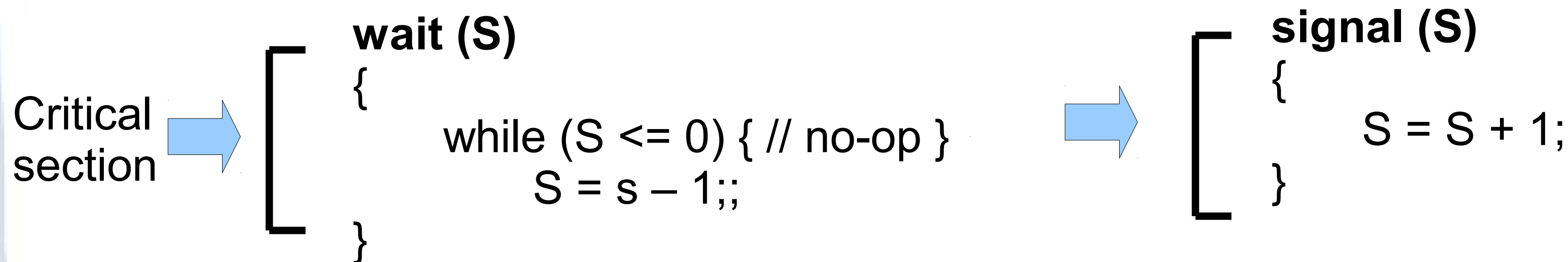
Una possibile soluzione

- Un array di booleani (**waiting**) in cui ogni elemento è associato ad un processo ed inizialmente settato a false.
- Se vi è più di un processo in attesa, la variabile lock sarà sempre settata a true → la mutua esclusione è garantita dal fatto che il processo attualmente in esecuzione setterà a false uno ed uno solo degli elementi di **waiting,** ovvero il primo con valore diverso da false.

```
while (true)
{
	waiting[i] = true;
	key = true;
	while (waiting[i] && key)
		key = TestAndSet(&lock);
	waiting[i] = false;

	// critical section

	j = (i + 1) % n;
	while ((j != i) && !waiting[j])
		j = (j + 1) % n;
	if (j == i) // Array waiting tutto a false
		lock = false;
	else
		waiting[j] = false;
	// non critical section
}
```

I semafori (1/2)

- Sono tool di sincronizzazione introdotti per semplificare il lavoro di programmazione
- Di base un semaforo è un **intero** (S nell'esempio), vi si accede con due operazioni base indivisibili:



- Semafori generici (max(S) > 1): usati per accessi multipli alle risorse
- Semafori binari (max(S) = 1, anche detti **mutex locks**) → usati ad esempio per garantire mutua esclusione.

I semafori (2/2)

- Semplice implementazione di mutua esclusione cn semaforo binario:

```
Semaphore mutex;    //  inizializzato a 1
do {
        wait (mutex);
        // Critical Section
        signal (mutex);
        // Non Critical Section
} while (TRUE);
```

- wait() e signal() devono essere eseguiti all'interno di una critical section ma...
- **Problema** di tutte le implementazione di sincronizzazione fin qui viste → **busy waiting** (ovvero while(condizione) { /*non fare nulla */ })

Implementazione dei semafori (1/2)

- Idea: usare lo scheduler. Si definiscono 2 operazioni:
  - **Block**: mette il processo (ovvero, il suo PCB) che invoca una certa operazione in una waiting queue asociata
  - **Wakeup**: rimuove un processo dalla waiting queue associata e lo sposta nella ready queue
- La **waiting queue sarà associata al semaforo stesso**, es.:

```
typedef struct{
        Int value;
        struct PCB *process_list;
} Semaphore;
```

- Intuitivamente tale lista dovrebbe essere implementata come un FIFO queue per garantire l'attesa limitata, ma non è così in generale

Implementazione dei semafori (2/2)

- Possibile implementazione wait:

```
wait(semaphore *S)
{
        S->value = S->value - 1;
        if (S->value < 0)
        {
                add this process to S->list;
                block();
        }
}
```

- Possibile implementazione signal:

```
signal(semaphore *S)
{
        S->value = S->value + 1;
        if (S->value <= 0)
        {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

- Attenzione! Il **busy waiting è stato solo limitato**, non è stato eliminato del tutto!

Deadlock and Starvation (1/2)

- **Deadlock**: due o più processi sono bloccati in attesa di un evento che può essere generato solamente da un altro processo bloccato → **tale evento non arriverà mai**
- **Starvation:** un processo potrebbe non essere mai rimosso dalla waiting queue associata ad un semaforo. Concetto già visto nel caso dello scheduling, e anche in questo caso lo scheduler ha grandi responsabilità...

Deadlock and Starvation (2/2)

- Esempio di deadlock: siano S e Q due semafori inizializzati a 1

Processo 0	Processo 1
wait (S);	wait (Q);
wait (Q);	wait (S);
...	...
...	...
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);

Inversione delle priorità (1/2)

- **Problema**: un processo a bassa priorità possiede un lock che è richiesto da processi a priorità più alta → **questi ultimi devono attendere la fine dei processi a priorità più bassa**
- Un esempio:
  - Il processo Ph (alta priorità) attende di accedere ad una risorsa a cui sta acceendo Pl (bassa priorità).
  - Nel mentre Pm (media priorità) entra nella ready queue e ovviamente viene schedulato al posto di Pl.
  - Ph dovrà attendere il completamento di Pm, **ovvero un processo un processo (Pm) a minor priorità rispetto a Ph ne ha influenzato la schedulazione**

Inversione delle priorità (2/2)

- **Possibile soluzione**: se Ph desidera accedere alla risorsa correntemente occupata da Pl, quest'ultimo eredita temporaneamente la priorità di Ph, impedendo a Pm di venire eseguito.
- Quando Pl rilascia la risorsa
  - La sua priorità originaria viene ristabilita.
  - Ph accede subito alla risorsa

Problema del Bounded-Buffer (1/2)

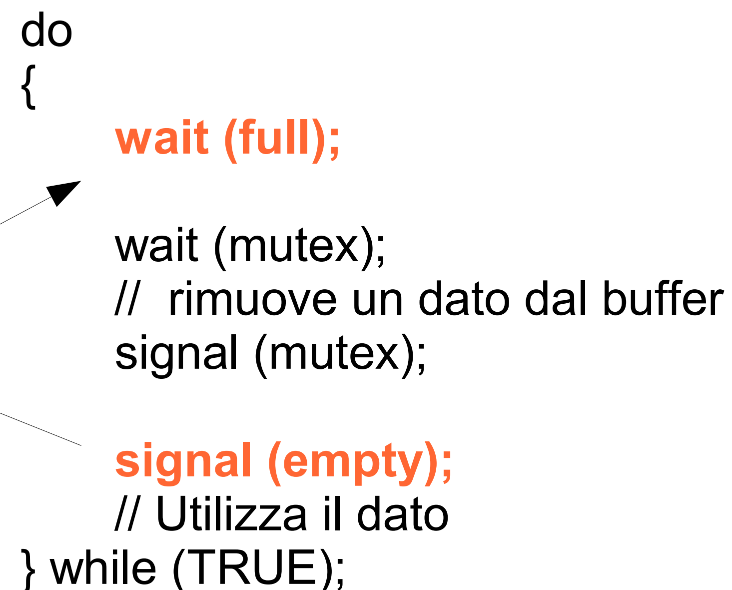
- Ritornando al problema del produttore-consumatore, ci troviamo di fronte ad una situazione in cui i due proecessi comunicano attraverso un buffer (circolare) **limitato**.
- Attraverso l'uso dei semafori è possibile ottenere un'implementazione molto elegante:
  - Un semaforo binario (mutex lock) per accedere in maniera concorrente al buffer.
  - 2 semafori che contano i buffer liberi/occupati usati per **sincronizzare** i due processi.

Problema del Bounded-Buffer (2/2)

- Produttore (mutex inizializzato a 1, empty inizializzato a n, ove n numero di locazioni del buffer):

```
do
{
        //   produce un dato
        wait (empty);

        wait (mutex);
        //  aggiunge il dato al buffer
        signal (mutex);

        signal (full);
} while (TRUE);
```

- Consumatore (full inizializzato a 0):



```
do
{
        wait (full);

        wait (mutex);
        //  rimuove un dato dal buffer
        signal (mutex);

        signal (empty);
        // Utilizza il dato
} while (TRUE);
```

Problema dei lettori e degli scrittori (1/2)

- Un area di memoria è condivisa da una serie di processi, alcuni dei quali possono solo leggere i dati (*lettori*), quindi non possono fare danni, altri possono anche scrivere (*scrittori*)m quindi potenzialmente si possono creare condizioni critiche. Ovviamente bisogna **garantire accesso esclusivo quando uno scrittore sta modificando i dati**.
- Ci sono molte variazioni al problema, ad esempio:
  - Prima variante: Nessun lettore deve attendere di accedere ai dati se vi sono altri lettori in accesso e scrittori in attesa
  - **Problema** prima variante: starvation → gli scrittori potrebbere non accedono mai alla memoria

Problema dei lettori e degli scrittori (2/2)

- Possibile implementazione variante 1
- Scrittore (wrt inizializzato a 1)

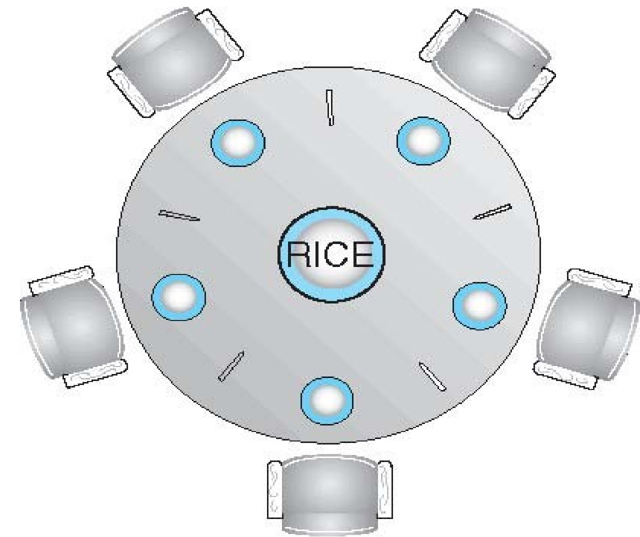
```
do
{
        wait (wrt) ;
        //    scrittura
        signal (wrt) ;
} while (TRUE);
```

- Lettore (mutex inizializzato a 1, readcount a 0)

```
do
{
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)

        // Lettura ...

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```

Problema dei filosofi a cena (1/3)

- Uno dei più classici problemi di sincronizzazione
- 5 filosofi (=**processi**) seduti ad una tavola rotonda, trascorrono la loro vita pensando e mangiando
- Per mangiare han bisogno di 2 bacchette (=**risorse**) ma nel tavolo ce ne sono solo 5
- Possono prendere solo le bacchetta di fronte a loro (sx e dx)
- Non è dato sapersi a priori quando hanno fame
- Possono prendere una bacchetta alla volta (ovvero, è necessaria un'operazione a bacchetta)



- Nel problema base, un altro requisito è che i filosofi non possano dialogare gli uni con gli altri.

Problema dei filosofi a cena (2/3)

- Una prima soluzione prevede l'uso di 5 semafori binari:

```
Semaphore chopstick[5] = {1};

do
{
        wait ( chopstick[i] ); // prova a prendere bacchetta di sx
        wait ( chopstick[ (i + 1) % 5] ); // .. bacchetta di dx

        //  mangia

        signal ( chopstick[i] ); // Rilascia bacchetta di sx
        signal ( chopstick[ (i + 1) % 5] ); // Rilascia bacchetta di dx

        //  pensa

} while (TRUE);
```

- Se sono affamati tutti nello stesso momento, potrebbero riuscire a prendere un bastoncino ciascuno (primo wait()), bloccandosi a vicenda → **deadlock**

Problema dei filosofi a cena (3/3)

- Per evitare il deadlock:
  - I filosofi possono iniziare a prendere le proprie bacchette se sono entrambe disponibili → è necessario inserire una sezione crititca
  - Definisco un ordine fisso: i filosofi con ID dispari prendono prima la forchetta di sinistra e poi quella di destra, viceversa i filosofi con ID pari

Monitor (1/2)

- Astrazione di alto livello introdotta per semplificare ulteriormente la vita dei programmatori.
- Evitano errori del tipo:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - wait (mutex) e/o signal (mutex) mancante
  - → **deadlock** and **starvation**
- I monitor garantiscono mutua esclusione in maniera semplificata

Monitor (2/2)

- Informalmente, un monitor è una sorta di “oggetto” le cui variabili interne sono “private”, ovvero accessibili attraverso i metodi (procedure) messi a disposizione dall'oggetto
- Solo un processo alla volta può essere attivo all'interno del monitor, ovvero solo un processo alla volta può eseguire una delle procedure del monitor

```
Monitor monitor_name
{
        // shared data declarations
        Type1 name1;
        Type2 name2;
        // …

        procedure P1 (…) { …. }
        // ...
        procedure Pn (…) {……}

        Initialization code (…) { … }
}
```

Variabili condizionali (1/3)

- Per implementare costrutti di sincronizzazione più complessi, non basta la semplice mutua esclusione.
- Un monitor deve avere la possibilità di attendere (**wait()**) fino al realizzarsi di una determinata condizione.
- Ovviamente il monitor deve dare la possibilità, una volta realizzata tale condizione, di sbloccare (signal()) altri processi in attesa di una determinata condizione.
- Una variabile condizionata in pratica è una coda di processi in attesa di che qualche condizione si verifichi

Variabili condizionali (2/3)

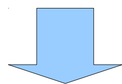
- Alla variabile condizionale x sono associate 2 operazioni:
  - **x.wait()**: un processo che invoca questa operazione viene sospeso finché non viene invocata x.signal() da un altro processo
  - **x.signal()**: riattiva un processo in attesa su x.wait().
    Se nessun processo è in attesa su x, **x.signal() non ha nessun effetto su x** (a differenza dei semafori, ove un signal significava un incremento del valore del semaforo)

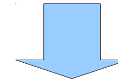
Variabili condizionali (3/3)

- Se il processo P invoca **x.signal ()**, con il processo Q in attesa su x (ovvero, ha precedentemente invocato x.wait ()), cosa succede?
  - **Signal and wait**: viene eseguito subito Q, e P attende che Q esca dal monitor o si blocchi in attesa di un'altra condizione
  - **Signal and continue**: Q attende che P esca dal monitor o si blocchi in attesa di un'altra condizione
- Pro e contro per entrambe le opzioni, dipende dall'implementazione

Problema dei filosofi con un monitor

```
monitor DiningPhilosophers
{// Begin monitor

// Variabili condivise
int  chopsticks[5] = { 2, 2, 2, 2, 2 };
Condition  c_available[5];

void  startEating(int i)
{
      // Controlla che entrambe le bacchette
      // siano disponibili
      if(chopsticks[i] != 2)
            c_available[i].wait();
      chopsticks[(i - 1) mod 5] -= 1;
      chopsticks[(i + 1) mod 5] -= 1;
}
```





```
void stopEating(int i)
{
      chopsticks[(i - 1) mod 5] += 1;
      chopsticks[(i + 1) mod 5] += 1;
      // Sblocca filosofo di sinistra
      if(chopsticks[(i - 1) mod 5] == 2)
            c_available[(i - 1) mod 5].signal();
      // Sblocca filosofo di destra
      If(chopsticks[(i + 1) mod 5] == 2)
            c_available[(i + 1) mod 5].signal();
}

}// End monitor
```

- Deadlock risolto, starvation ancora possibile