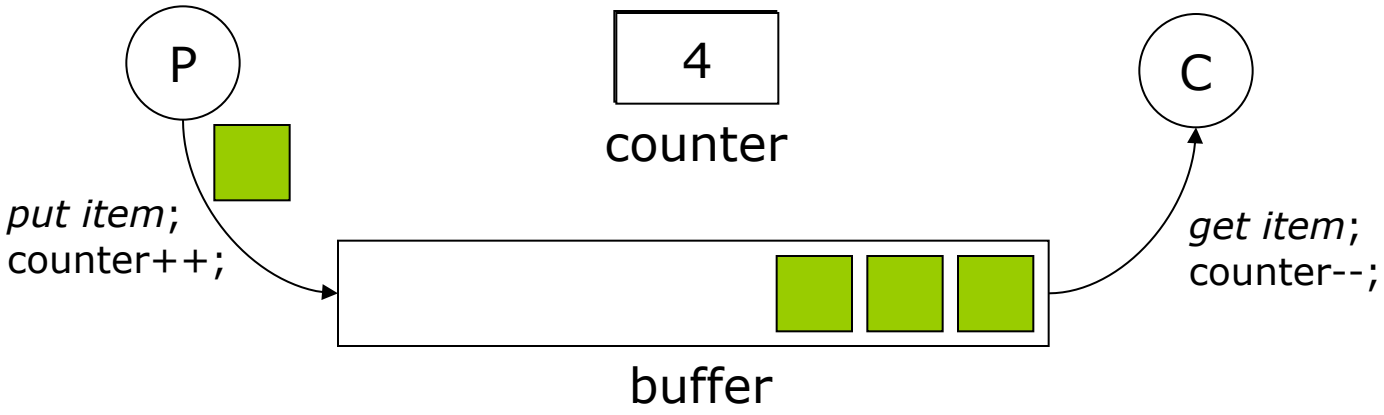
Sincronizzazione dei processi
(prima parte)

Accesso concorrente a dati condivisi

- **L'accesso concorrente da parte di più processi a dati condivisi può portare alla loro inconsistenza.**
- **Per garantire la consistenza dei dati sono necessari meccanismi che assicurino l'esecuzione ordinata dei processi concorrenti.**
- Esempio:
  - Due processi: *produttore* (P) e *consumatore* (C)
  - Due variabili condivise: *buffer* e *counter*



Produttore

```
while (true) {

     /* produce un elemento e lo inserisce in nextProduced */

     while (counter == BUFFER_SIZE) ; // non fa nulla

     buffer [in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

Consumatore

```
while (true)  {

     while (counter == 0) ; // non fa nulla

     nextConsumed =  buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /*  consuma l'item in nextConsumed */
}
```

L'aggiornamento del contatore è un'operazione critica!

Produttore e Consumatore

- L'istruzione "**counter++**" può essere implementata in linguaggio macchina come:

  ***register*$_1$ = counter**
  ***register*$_1$ = *register*$_1$ + 1**
  counter = *register*$_1$

- L'istruzione "**counter--**" può essere implementata come:

  ***register*$_2$ = counter**
  ***register*$_2$ = *register*$_2$ – 1**
  counter = *register*$_2$

Interleaving delle istruzioni (1/2)

- Nel caso in cui il produttore ed il consumatore cercassero di aggiornare la variabile **counter** contemporaneamente, le istruzioni in linguaggio macchina potrebbero risultare interfogliate (*interleaved*).

- L'*interleaving* dipende dal modo in cui i due processi produttore e consumatore sono schedulati.

- Qualsiasi interleaving è possibile, ma in ogni caso l'ordine interno di ogni singola istruzione di alto livello deve essere conservato.

Interleaving delle istruzioni (2/2)

- Si assuma che **counter** sia inizialmente **5**. Un possibile interleaving delle istruzioni è il seguente:

T

Prod: **register$_1$ = counter** (*register1 = 5*)
Prod: **register$_1$ = register$_1$ + 1** (*register1 = 6*)
Cons: **register$_2$ = counter** (*register2 = 5*)
Cons: **register$_2$ = register$_2$ – 1** (*register2 = 4*)
Prod: **counter = register$_1$** (*counter = 6*)
Cons: **counter = register$_2$** (*counter = 4*)

- Il valore finale di **counter** dovrebbe essere 5, mentre in in questo caso è erroneamente pari a ***4*** (invertendo le ultime due istruzioni si otterrebbe erroneamente **6**).

Race condition

- ***Race condition***: Situazione in cui più processi accedono e manipolano dati condivisi in modo concorrente e il valore finale dei dati condivisi dipende dall'ordine con cui sono avvenuti gli accessi.

- Per eliminare le ***race condition***, i processi concorrenti devono essere **sincronizzati** in modo che la **manipolazione delle variabili condivise sia consentita a un solo processo per volta**!

Il problema della sezione critica (1/2)

- ***N*** processi $\{P_0,\ldots,P_{N-1}\}$ che competono per usare dati condivisi.
- Ogni processo ha un segmento di codice, chiamato ***sezione critica***, nel quale i dati condivisi vengono acceduti e modificati.
- ***Problema***: garantire che quando un processo è in esecuzione nella sua sezione critica, nessun altro processo possa eseguire la propria sezione critica.
- ***L'esecuzione delle sezioni critiche*** da parte dei processi ***deve essere mutuamente esclusiva*** nel tempo.

Il problema della sezione critica (1/2)

Una **soluzione al problema della sezione critica** deve soddisfare i seguenti requisiti:

1) **Mutua esclusione (ME)**. Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2) **Progresso (P)**. Se nessun processo è in esecuzione nella propria sezione critica, allora soltanto i processi che cercano di entrare nella sezione critica possono partecipare alla decisione di chi entrerà nella propria sezione critica, e questa decisione deve avvenire in un tempo finito.
3) **Attesa limitata (AL)**. Deve esistere un limite nel numero di volte che altri processi sono autorizzati ad entrare nelle rispettive sezioni critiche dopo che un processo P_i ha fatto richiesta di entrare nella propria sezione critica e prima che quella richiesta sia soddisfatta.

- Si assume che ogni processo esegua a velocità non nulla.

- Non è possibile fare assunzioni sulla velocità relativa degli N processi.

Sezione critica: soluzioni per due processi

- Soltanto due processi: P_0 e P_1
- Struttura generale del processo P_i ::

```
while (true)
{
   entry section
   sezione critica
   exit section
   sezione non critica
}
```

- I processi possono far uso di variabili condivise per sincronizzare le loro azioni.

Algoritmo 1 [Dijkstra]

- Variabili condivise tra i due processi $\mathbf{P_0}$ e $\mathbf{P_1}$:
  - **int turn**;
  - **Inizialmente turn = 1**
  - **Quando turn = i $\Rightarrow$ P_i può entrare in esecuzione nella propria sezione critica**
- Processo P_i ::

```
while (true) {
    while (turn != i) ;  // do no-op
sezione critica
turn = j;
sezione non critica
}
```

- Non soddisfa il requisito del **Progresso** (perchè ***richiede una stretta alternanza dei processi***) infatti il **Progresso** implica che se nessun processo è nella sezione critica l'accesso alla sezione critica deve essere consentito.

Algoritmo 1 [Dijkstra]

P0

```
turno = 1;

while(true)
{
   while (turno != 0);
    ...
    <sezione critica>
     turno = 1;
     ...
}
```

P1

```
turno = 1;
Operazione di input bloccante
while(true)
{
   while (turno != 1);
    ...
    <sezione critica>
     turno = 0;
     ...
}
```

Algoritmo 2 [Dijkstra]

- Variabili condivise:
  - **boolean flag[] = new boolean[2];**
  - **Inizialmente flag[0] e flag[1] sono posti a false.**
  - **Quando flag[i] = true ⇒ P_i è pronto ad entrare nella sua sezione critica**
- Processo P_i ::

```
while (true) {
    flag[i] = true;
    while (flag[j]) ;
      sezione critica
    flag[i] = false;
    sezione non critica
}
```

I processi possono entrare in un loop infinito quando ai due flag viene assegnato il valore **true**.

Algoritmo 2 [Dijkstra]

P0

```
while(true)
{
  pronto[0] = true; // ←
  while(pronto[1]);

 <sezione critica>

  pronto[0] = false;
}
```

P1

```
while(true)
{
  pronto[1] = true; // ←
  while(pronto[0]);

 <sezione critica>

  pronto[1] = false;
}
```

Algoritmo 3 [Peterson]

- Usa le variabili condivise degli algoritmi 1 e 2.
- Processo P_i ::

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j) ;

    sezione critica

    flag[i] = false;

    sezione non critica
}
```

- Soddisfa tutti e tre i requisiti (ME, P, AL); tuttavia vale soltanto per due processi.

Algoritmo 2 [Peterson]

P0

```
while(true)
{
  pronto[0] = true;
  turno = 1;
  while(pronto[1] && turno == 1);

 <sezione critica>

  pronto[0] = false;
}
```

P1

```
while(true)
{
  pronto[1] = true;
  turno = 0;
  while(pronto[0] && turno == 0);

 <sezione critica>

  pronto[1] = false;
}
```

- non accade mai che un processo si blocchi se l'altro non è nella sezione critica
- Un processo in attesa, prima o poi si sblocca.

Algoritmo del fornaio [Lamport] (1/3)

Soluzione al problema della sezione critica per N processi:

- Ogni processo è identificato da un numero intero ($P_1, P_2, \ldots, P_n$).
- Prima di entrare nella sezione critica, il processo "riceve" un numero. Il possessore del numero più piccolo entra nella sezione critica.

- **NOTA BENE**: Lo schema di numerazione non assicura che ogni processo riceva un numero diverso, ma garantisce che genererà sempre numeri in ordine crescente; ad esempio: 1,2,3,3,3,3,4,5...

- Se i processi P_i e P_j "ricevono" lo stesso numero, si verifica il valore del loro indice, se $i < j$, allora P_i è servito prima; altrimenti sarà P_j ad essere servito per primo.

Algoritmo del fornaio [Lamport] (2/3)

- Dati condivisi

  boolean choosing[] = new boolean[n]; *Il stato di scelta del numero*

  int number[] = new int [n]; *// numero di prenotazione*

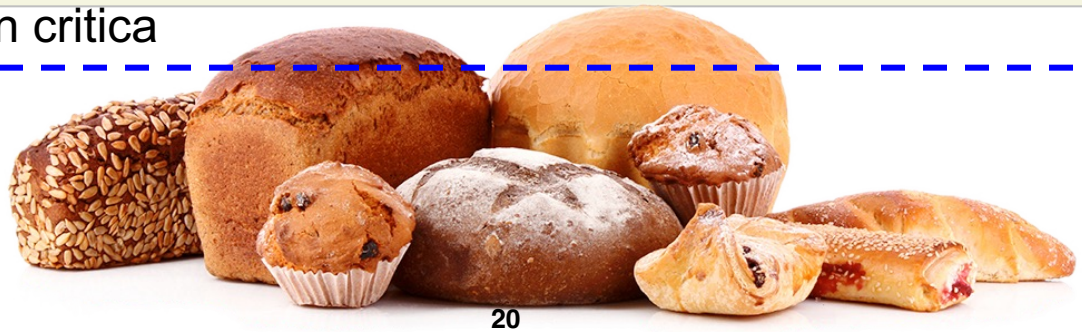
- Le strutture dati sono inizializzate a **false** ed a **0** rispettivamente.

- La notazione **<** indica l'ordinamento lessicografico tra coppie del tipo: (ticket #, process id #)
  - **(*a*,*b*) < (*c*,*d*) se *a* < *c*** oppure ***a* == *c* && *b* < *d***
  - **max (a_0,…, a_{n-1}) è un numero *k*, tale che $k \geq a_i$ per *i* : 0,…, *n* – 1**

Algoritmo del fornaio [Lamport] (3/3)

P_i

```
while (true) {
   choosing[i] = true; //sta prelevando il numero
   number[i] = max(number[0], number[1],…, number [n-1]) + 1;
   choosing[i] = false;
   for (j = 0; j < n; j++) {
     while (choosing[j]) ; //attende se qualche Pj sta prelevando il numero
     while ((number[j] != 0) && ( (number[j],j) <
   (number[i],i)));
   }
   sezione critica
   number[i] = 0;
   sezione non critica
}
```



Istruzioni atomiche

- Definizione: un'istruzione è detta *atomica* se completa la propria esecuzione senza interruzioni.

- Se le istruzioni:

  counter++;

  counter--;

  nel caso del *produttore/consumatore* fossero state atomiche non si sarebbe potuto verificare un interleaving delle relative istruzioni di basso livello.

- L'atomicità di una istruzione può essere garantita disabilitando gli interrupt durante la sua esecuzione.

Semafori

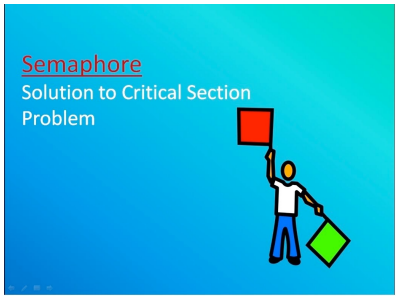
- Strumento di sincronizzazione che, *in determinate implementazioni*, non richiede busy waiting (come negli algoritmi descritti).
- **Semaforo *S***: variabile intera (**da inizializzare!**).
- Può essere acceduta esclusivamente attraverso due operazioni ***atomiche***:

→ ***wait* (*S*):**

```
while (S <= 0) ; // do no-op
S--;
```

→ ***signal* (*S*):**

```
S++;
```



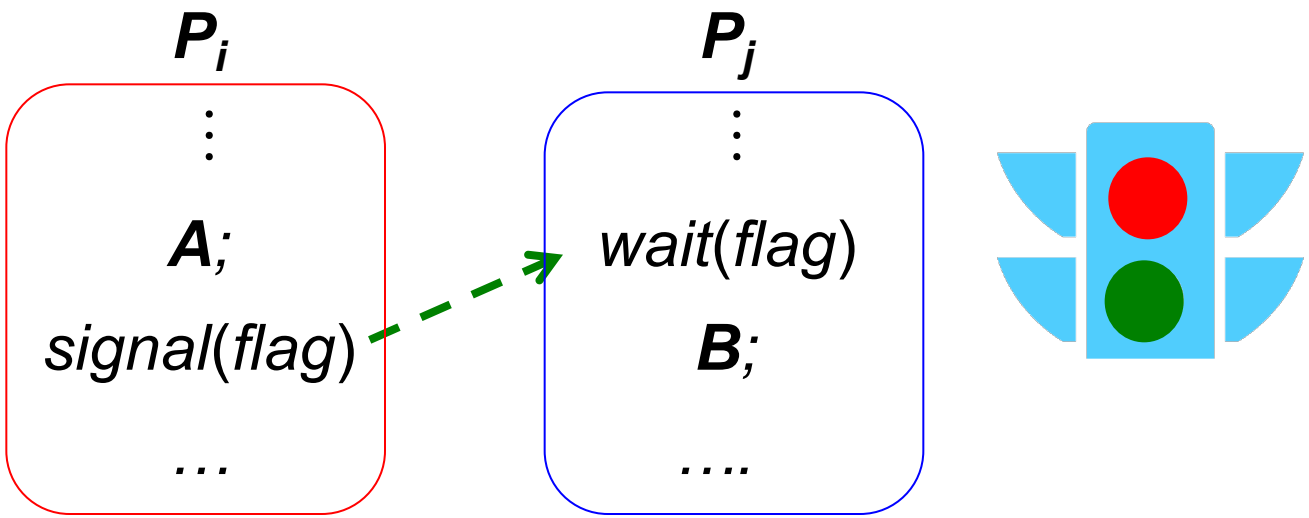
Semafori: altri schemi di sincronizzazione

- Sequenzializzazione di due operazioni A e B:

  Esegue *B* in P_j solo dopo che *A* esegue in P_i

- Usa il semaforo *flag* inizializzato a 0.

- Schema:



Implementazione dei semafori

- Un semaforo può essere concettualmente implementato come segue:

```
class Semaforo {
  int value;
  LinkedList<Process> listaProcessi =
      new LinkedList<Process> ();
}
```

- Per evitare l'attesa attiva (**busy waiting**) assumiamo di poter usare le seguenti operazioni:
  - ***block*()** sospende il processo che la invoca (running→ ready).
  - ***wakeup*(*P*)** riprende l'esecuzione del processo bloccato ***P*** (ready→ running).

Operazioni associate ai semafori

- Supponiamo che sia dichiarato il Semaforo **S** con valore iniziale **1:**

***wait(S)*:**

```
S.value--;
 if (S.value < 0) {
     aggiunge questo processo a S.listaProcessi;
     block();
 }
```

***signal*(*S*):**

```
S.value++;
 if (S.value <= 0) {
     rimuove un processo P da S.listaProcessi;
     wakeup(P);
 }
```

- Il semaforo può assumere valori negativi; in questo caso il valore assoluto indica il numero di processi in attesa su quel semaforo.

Semafori: sezione critica di *N* processi

- Per gestire la mutua esclusione tra N processi si può usare un semaforo

- Dati condivisi:

  semaforo mutex; // inizialmente *mutex* = 1

- Processo P_i:

```
while (true) {
    wait(mutex);
        sezione critica

    signal(mutex);
        sezione non critica;
}
```