

Swap

- Alternativa a TestAndSet è **Swap**
- Swap scambia in modo atomico i valori dei suoi due parametri

Swap

- Implementazione:

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

- Anche in questo caso l'unica particolarità sta nel tipo di esecuzione della routine, che è atomica

Uso di swap

- Usiamo Swap per realizzare l'accesso in mutua esclusione a una sezione critica
- Oltre a lock (variabile condivisa) utilizzeremo anche una variabile booleana (locale) "chiuso":

Uso di swap

- Implementazione:

```
chiuso = true;  
while (chiuso) Swap(&lock, &chiuso);  
  
<sezione critica>  
  
lock = false;
```

- si esce dal ciclo while solo quando lock era false
- All'uscita da Swap lock risulta automaticamente impostato a true

Critica

- Entrambi i metodi visti garantiscono la mutua esclusione ...

Critica

- Entrambi i metodi visti garantiscono la mutua esclusione ...
- ... ma non l'attesa limitata!
- non c'è garanzia che un processo che vorrebbe eseguire TestAndSet oppure Swap non venga sempre prevaricato da altri
- è possibile usare TestandSet per realizzare una soluzione che non presenta il problema dell'attesa illimitata?

Attesa limitata

- Sì, l'algoritmo è complicatissimo!!! (non da studiare)

sezione di ingresso

```
attesa[i] = true;
chiave = true;
while (attesa[i] && chiave)
    chiave = TestAndSet(&lock);
attesa[i] = false
```

sezione di uscita

```
j = (i+1) % n;
while ((j != i) && !attesa[j])
    j = (j+1) % n;
if (j == i) lock = false;
else attesa[j] = false
```

- in ogni caso persiste il problema del **busy waiting** ...

Semafori

Semafori

- Strumenti di sincronizzazione introdotti da Dijkstra nel 1965 per minimizzare il busy-waiting e per semplificare la vita ai programmatori
- **semaforo** = variabile a cui (dopo l'inizializzazione) si può accedere solo tramite due operazioni atomiche:
 - **P** (*proberen*, verificare in olandese) e
 - **V** (*verhogen*, incrementare),
- Sono anche note come wait e signal, down e up

Semafori

- **semaforo** = variabile a cui si può accedere (dopo l'inizializzazione) solo tramite due operazioni atomiche note come P (*proberen*, verificare in olandese) e V (*verhogen*, incrementare), anche note come wait e signal
- Possiamo immaginare lo stato del semaforo come un campo di tipo intero
- in pseudocodice:

```
P (S) {  
    while (S <=0) no_op;  
    S--;  
}
```

definizione
di P e V

```
V (S) {  
    S++;  
}
```

La P non realizza
un'attesa attiva?

mutex inizializzato
al valore 1

```
P(mutex);  
  
<sezione critica>  
  
V(mutex);
```

utilizzo di un semaforo mutex,
P e V per controllare una
sezione critica

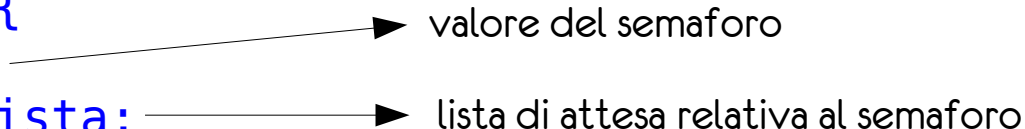
Spinlock e sospensione

- Il tipo di attesa comportato dai semafori implementati come visto è detto **spinlock**: lo spinlock fa un'attesa attiva
- sono possibili altre implementazioni che evitano lo spinlock. Richiedono strutture di appoggio
- ogni semaforo ha associata una **lista di PCB**: quelli dei processi sospesi su quel semaforo
- quando un processo si sospende su di un semaforo, lo scheduler della CPU la assegna a un altro processo
- quando il valore del semaforo viene alzato uno dei processi nella sua coda di attesa viene scelto e fatto passare dallo stato waiting allo stato ready

Semafori con code 1/2

- Vediamo una possibile implementazione del tipo di dato semaforo, nel caso questo includa una coda di attesa, e delle procedure P e V

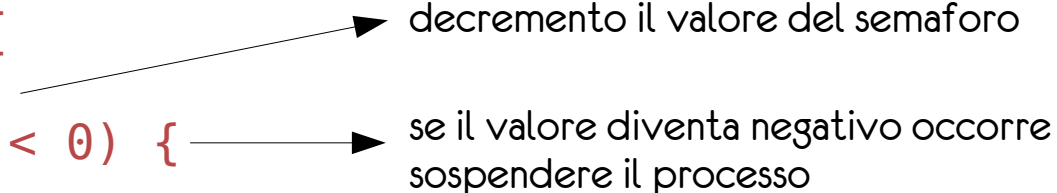
```
typedef struct {  
    int valore;  
    processo *lista;  
} semaforo;
```



valore del semaforo

lista di attesa relativa al semaforo

```
P (semaforo *s) {  
    S->valore - -;  
    if (S->valore < 0) {
```

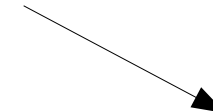


decremento il valore del semaforo

se il valore diventa negativo occorre sospendere il processo

```
        <aggiungi il PCB di questo processo a S->lista>  
        block();  
    }
```

```
}
```



block è una system call che richiama lo scheduler della CPU, che deve riassegnare la medesima a un altro processo, e il dispatcher, che deve effettuare il context switch

Semafori con code 2/2

- Vediamo una possibile implementazione del tipo di dato semaforo, nel caso questo includa una coda di attesa, e delle procedure P e V

```
V (semaforo *S) {  
    S->valore++;  
    if (S->valore < 0) {  
        <scegli un PCB P da S->lista>  
        wakeup(P);  
    }  
}
```

incremento il valore del semaforo

se il valore è negativo, allora ci sono processi da risvegliare

wakeup è una system call che rende il processo P ready

NB: poiché come prima operazione P decrementa sempre il valore del semaforo, quando questo ha valore negativo, il suo valore assoluto indica il numero dei processi in attesa

Inizializzazione e uso

- I semafori di mutua esclusione sono inizializzati a 1 e possono valere al più 1:
 - 1 = risorsa disponibile,
 - 0 (o valore negativo) = risorsa occupata
- I semafori che possono assumere valori > 1 sono detti semafori contatori. Il numero N, valore del semaforo, indica una quantità di risorse disponibili
 - in certe implementazioni un processo può richiedere un numero $M \geq 1$ di risorse
 - il codice di P e V che abbiamo visto non consente di realizzare questo tipo di richieste
 - l'implementazione Unix (che studieremo per la parte di lab) invece lo consente

Tipi di sincronizzazione

- I semafori sono strumenti che consentono di realizzare molti tipi di sincronizzazione diversa: dipende da come si usano
- **Mutua esclusione**: tutti i processi coinvolti parentesizzano le loro sezioni critiche con $P(\text{mutex}) \dots V(\text{mutex})$, dove mutex è un semaforo di mutua esclusione
- **Accesso limitato**: tutti i processi coinvolti parentesizzano le loro sezioni critiche con $P(\text{nr}) \dots V(\text{nr})$, dove nr è un semaforo contatore. Un numero limitato di processi potrà eseguire in parallelo una certa sezione di codice
- **Ordinamento**: si può controllare l'ordine di esecuzione di due processi, ad esempio sia sem un semaforo inizializzato a 0:

P1:	P2:
$P(\text{sem})$ codice	codice $V(\text{sem})$

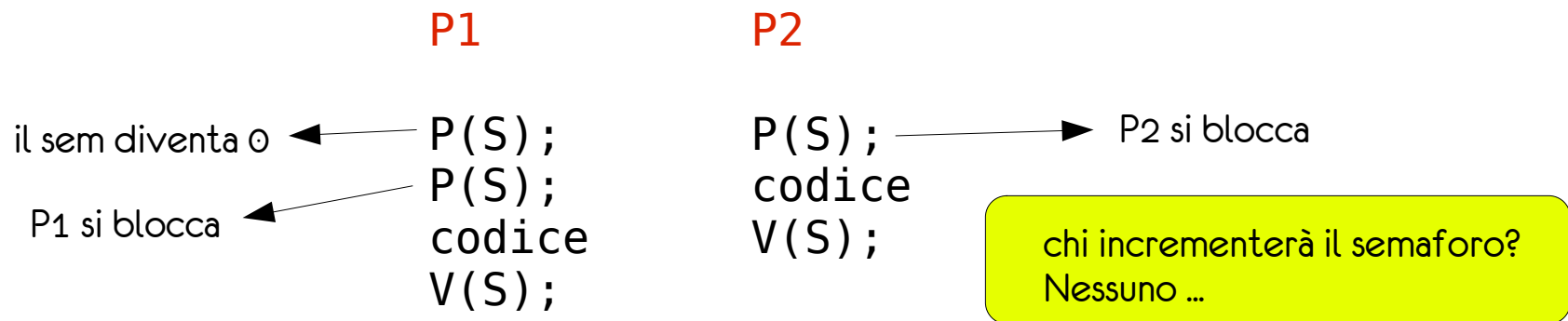
in quale ordine vengono eseguiti P1 e P2?

Operazioni atomiche

- Le operazioni sui semafori devono essere eseguite in modo atomico: sono sezioni critiche perché i semafori sono variabili condivise
- su sistemi monoprocesso, atomicità ottenuta **disabilitando gli interrupt**: non è un problema perché i tempi di esecuzione di P e V sono molto brevi
- su sistemi multiprocesso, invece, si utilizzano prevalentemente spinlock perché disabilitare le interruzioni di tutti i processori crea cali di prestazione
- => busy-waiting ridotta ma non eliminata

Attenzione

- Il cattivo uso dei semafori da parte di un programmatore può creare **deadlock** (stallo), es. sia S un semaforo che inizialmente vale 1:



Attenzione

- Situazioni più subdole in cui si può generare deadlock sono causate dall'utilizzo di più semafori

Attenzione

- Situazioni più subdole in cui si può generare deadlock sono causate dall'utilizzo di più semafori
- Esempio: siano S e T due semafori inizializzati ad 1 e siano P1 e P2 due processi che eseguono le seguenti operazioni:

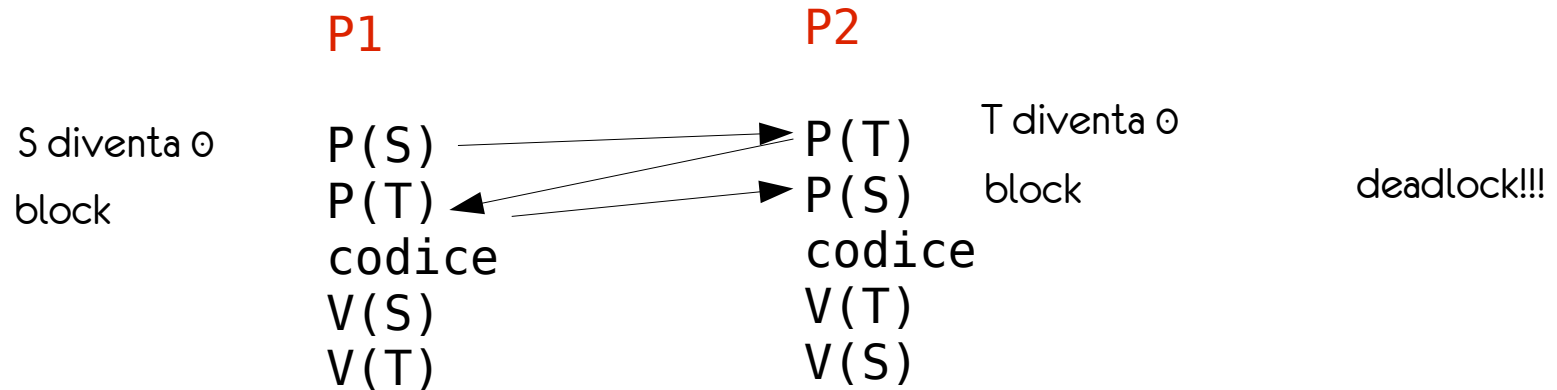
P1

P(S)
P(T)
codice
V(S)
V(T)

P2

P(T)
P(S)
codice
V(T)
V(S)

Attenzione



- Se P1 esegue P(S) poi P2 esegue P(T), quindi P1 esegue P(T), bloccandosi, e P2 esegue P(S), bloccandosi: si ha un deadlock
- NB: il deadlock non si ha ad ogni esecuzione, **dipende dall'interleaving delle istruzioni!!!** Es. se P1 esegue P(S) e subito dopo P(T) entra in sezione critica e poi alza il valore dei due semafori, lasciando entrare P2

Problemi classici di sincronizzazione

- Produttori – consumatori con buffer limitato (in parte già visto)
- Lettori – scrittori
- Cinque filosofi

Problemi classici di sincronizzazione

- **Produttori – consumatori con buffer limitato** (in parte già visto): si usano 3 semafori:
 - **mutex**: mutua esclusione
 - **libere**: numero di caselle libere del buffer
 - **occupate**: numero di caselle occupate del buffer
- Lettori – scrittori
- Cinque filosofi

produttori - consumatori

Produttore

```
forever {  
    <produci un nuovo elemento>  
  
    P(libere);  
    P(mutex);  
  
    <inserisci nuovo elemento>  
  
    V(mutex);  
    V(occupate);  
}
```

Consumatore

```
forever {  
  
    P(occupate);  
    P(mutex);  
  
    <estrai elemento>  
  
    V(mutex);  
    V(libere);  
  
    <consuma elemento>  
}
```

Inizialmente **mutex** vale 1, **libere** vale N (capienza del buffer) e **occupate** vale 0

produttore

Produttore

```
forever {
```

```
    <produci un nuovo elemento>
```

```
    P(libere);
```

```
    P(mutex);
```

```
    <inserisci nuovo elemento>
```

```
    V(mutex);
```

```
    V(occupate);
```

```
}
```

se il buffer è pieno libere vale 0 quindi il produttore si sospende e rimane sospeso finché non c'è qualche cella a disposizione
NB: la P decrementa libere

l'inserzione vera e propria va effettuata in mutua esclusione per mantenere la consistenza dei dati nel buffer, che è un'area di memoria condivisa

alla fine incremento il numero delle celle occupate, attivando eventualmente un consumatore

consumatore

Consumatore

```
forever {
```

```
    P(occupate);
```

```
    P(mutex);
```

```
    <estrai elemento>
```

```
    V(mutex);
```

```
    V(libere);
```

```
    <consuma elemento>
```

```
}
```

rimane fermo finché non c'è nulla da consumare, quando occupate viene incrementato da un produttore lo decrementa, indicando che sta per consumare un oggetto, quindi procede

come prima l'accesso al buffer va fatto in mutua esclusione per evitare che si produca un'inconsistenza nei dati

quando si estrae un elemento si libera una cella. Il processo segnala questo evento incrementando il semaforo "libere". Se il buffer era pieno e un produttore era in attesa, ritorna ready

l'elaborazione dell'oggetto estratto, la sua consumazione, non viene fatta in ME, per minimizzare l'esecuzione in sezione critica. Infatti non richiede ulteriori accessi al buffer

Problemi classici di sincronizzazione

- Produttori - consumatori con buffer limitato
- **Lettori - scrittori**: dei processi lettori e scrittori usano una stessa area di memoria. Gli scrittori modificano la risorsa, quindi accedono in ME con tutti. I lettori non modificano la risorsa, quindi possono accedere alla memoria condivisa in parallelo.
 - Si utilizzano:
 - **scrittura**: semaforo di ME
 - una variabile intera condivisa **numlettori** che conta quanti processi stanno leggendo in questo momento
 - **mutex**: semaforo di ME per l'accesso a **numlettori**
- Cinque filosofi