- Lo standard POSIX definisce i semafori named e unnamed: vediamo solo quelli unnamed.
- Un semaforo unnamed si dichiara così:

```
sem_t s; // tipo sem_t definito in semaphore.h
```

si deve includere l'header e compilare con `-pthread`

- Si inizializza tamite la funzione `sem_init`:

```
int sem_init(sem_t *sem, int pshared, unsigned
int value)
```

dove:

- `sem` è il puntatore alla variabile dichiarata
- `pshared` indica se il semaforo è condiviso tra thread di un processo o tra processi diversi
- `value` è il valore a cui viene inizializzato il semaforo

- Perciò per inizializzare un semaforo `s` condiviso tra thread:

```
sem_init(&s,0,val);
```

  - dove `val` è il valore a cui viene inizializzato
  - il secondo parametro `0` serve ad indicare che il semaforo può essere usato solo dai thread del processo che l'ha creato

- Mentre per inizializzare un semaforo `s` condiviso tra processi:

```
sem_init(&s,1,val);
```

dove `s` deve essere una variabile allocata in un segmento di memoria condivisa

Semafori Posix: Up e Down

- L'equivalente della funzione "up" è
  - sem_post(&s);
- e l'equivalente della "down" è
  - sem_wait(&s);

- Esistono inoltre i mutex: una versione semplificata di semafori binari (hanno solo due stati) su cui sono ammesse solo le operazioni lock e unlock

- Dichiarazione e inizializzazione di un mutex:
  - pthread_mutex_t m;
- Inizializzazione del mutex nella condizione unlocked:
  - pthread_mutex_init(&m,NULL); // default attributes
- Lock del mutex:
  - pthread_mutex_lock(&m);
- Unlock del mutex:
  - pthread_mutex_unlock(&m);

- Considerare il programma race.c con thread (in appunti6b) e risolvere il problema di corsa critica tramite semafori Posix

- Considerare il programma pc_sem_thr.c (in appunti6b) che risolve il problema di N produttori e un consumatore
  - Aggiungete due variabili che contino il numero di item inseriti e consumati e verificate se tutti gli item prodotti sono consumati.

Mutex e condizioni (Pthreads)

In UNIX i Phtreads si possono sincronizzare con altri paradigmi fra cui uno ispirato a quello dei monitor:

usando esplicitamente dei mutex per la mutua esclusione (non c'è gestione automatica come per le procedure dei monitor o i metodi synchronized)

Variabili condizione con operazioni wait, signal e broadcast (segnala una condizione a tutti i thread in attesa su quella)

In particolare le variabili condizione (tipo `pthread_cond_t`) si inizializzano con:

```
pthread_cond_init(&cond, NULL);
```

dove `cond` è l'identificativo della variabile e con `NULL` si usano gli attributi di default

Un thread, dopo avere conquistato l'accesso esclusivo a variabili condivise con

```
pthread_mutex_lock(&m);
```

può sospendersi su una variabile condizione cond (di tipo `pthread_cond_t`) con:

```
pthread_cond_wait(&cond,&m);
```

questo comporta implicitamente anche l'unlock di `m`, cioè il rilascio della mutua esclusione, e il nuovo lock di `m` quando il thread verrà sbloccato dall'attesa della condizione.

Mutex e condizioni (Pthreads)

Un thread può "segnalare" una condizione con:

```
pthread_cond_signal(&cond);
```

che "risveglia" uno dei threads bloccati su `cond` (se non ce ne sono, non ha effetto), oppure con:

```
pthread_cond_broadcast(&cond);
```

che “risveglia” tutti i threads bloccati sulla condizione.

Attenzione al noto problema: per riprendere l’esecuzione, i thread che avevano fatto wait devono riprendere l’accesso mutuamente esclusivo sul mutex che avevano passato come secondo argomento a wait.

Tipicamente, la segnalazione viene fatta da un thread che opera sulle stesse variabili condivise e quindi ha il lock sullo stesso mutex (anzi è raccomandato che questo avvenga sempre, per evitare il problema della "perdita della segnalazione" come per sleep e wakeup).

Il thread sbloccato, o i thread sbloccati in caso di broadcast, potrà (potranno) riprendere il lock (uno per volta) quando il thread segnalante lo rilascerà.

Mutex e condizioni (Pthreads)

Questo è un tipico schema di programma per un thread t1 che opera su variabili condivise: attende, anche usando una var. condizione, che una condizione booleana test sulle variabili sia falsa, poi modifica le variabili condivise ed eventualmente segnala (a uno/tutti) una condizione:

```
pthread_mutex_lock(&m);
 while (test) pthread_cond_wait(&cond1,&m);
 /* modifica variabili condivise */
 /* eventuale pthread_cond_signal(&cond2) */
 pthread_mutex_unlock(&m);
```

Qui è veramente opportuno un `while` e non un `if` (all'uscita della `wait` viene di nuovo eseguito il test) perché:

chi segnala `cond1` non necessariamente conosce quale `test` t1 attende che diventi falso

se anche fosse così, non è detto che t1 conquisti la mutua esclusione immediatamente dopo la segnalazione della condizione; un altro thread t2 che esegue ad es. lo stesso codice di t1 può superare `lock` e modificare le var. condivise rendendo nuovamente vero `test`

Ad esempio per un “produttore” :

```
pthread_mutex_lock(&m);
while (buf.count==N)pthread_cond_wait(&empty,&m);
      /* deposita elemento in buf */
if (buf.count==1) pthread_cond_signal(&full);
pthread_mutex_unlock(&m);
```

E per un “consumatore”:

```
pthread_mutex_lock(&m);
while (buf.count==0)pthread_cond_wait(&full,&m);
      /* preleva elemento da buf */
if (buf.count==N-1) pthread_cond_signal(&empty);
pthread_mutex_unlock(&m);
```

Data la soluzione del problema Produttori-Consumatore risolto con semafori (in appunti 6b), implementare una soluzione basata su mutex e variabili condizione