SC che operano su thread

… **`pthread_create()`** etc …

Thread POSIX

- Obiettivi
  - Fornire una breve introduzione
  - descrivere alcune delle numerose SC relative ai thread POSIX
  - fare qualche esempio concreto
- Finora
  - tutti i nostri processi avevano un singolo flusso di controllo (o *thread*)
  - adesso è possibile far condividere dati, file aperti etc a più flussi di controllo (o *thread*) ognuno con il proprio PC e il proprio stack

Creare un thread: pthread_create

```
#include <pthread.h>

int pthread_create(
 pthread_t *thread_id,  /*ID del nuovo thread*/
 const pthread_attr_t *attr, /*attributi*/
 void* (*start_fcn) (void *), /* funz inizio*/
 void* arg /* argomenti */
);
/* (0) success (error number) failure */
```

Creare un thread: pthread_create (2)

- Notare il tipo della funzione di inizio:

```
void* start_fcn (
 void* arg          /* argomenti */
);
/* Returns exit status */
```

Creare un thread: pthread_create (3)

- Semantica :

  `pthread_create(&tid, attr, &myfun, arg)`

  - cerca di creare un nuovo thread,
  - se ci riesce in **`tid`** ritorna l'identificatore del nuovo thread creato
  - il thread inizia l'esecuzione con **`myfun(arg)`**
  - **`attr`** struttura che specifica gli attributi del nuovo thread e deve essere settata con chiamate apposite (vedi man **`pthread_attr_init`**)
  - noi useremo solo gli attributi di default **`attr==NULL`**
  - ritorna il codice di errore invece di usare errno
  - NOTA: il thread creato ha una sua copia di errno (!)

pthread_create : un esempio

- Accessi incontrollati ad una variabile condivisa ...

```
#include <pthread.h>

static int x; /* la var condivisa */

static void* myfunz (void* arg){
  while (true) {
    printf("Secondo thread: x=%d\n",++x);
    sleep(1);
   }
}
```

pthread_create : un esempio (2)

- Accessi incontrollati ad una variabile condivisa ...

```
int main (void) {
  pthread_t tid;
  int err;

  if ( err=pthread_create(&tid, NULL, &myfun, \
          NULL) ) != 0 ) { /* gest errore */ }
  else { /* secondo thread creato */
    while (true) {
      printf("Primo thread: x=%d\n",++x);
      sleep(1);}
}
```

pthread_create : un esempio (3)

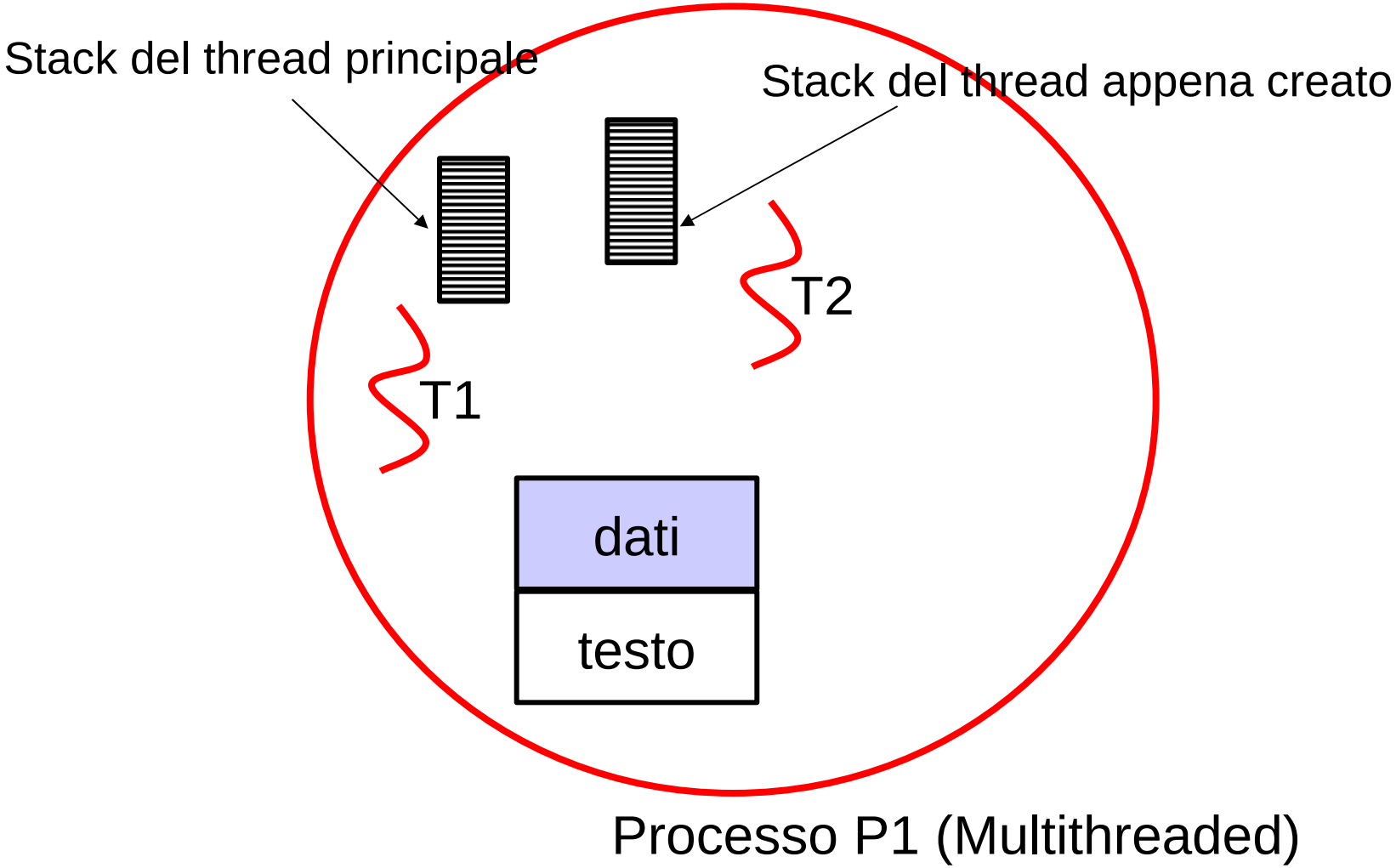
- Compiliamo:

```
bash:~$ gcc main.c -o threadtest -lpthread
```

- Eseguiamo:

```
bash:~$ ./threadtest
Primo thread: x=1
Secondo thread: x=2
Primo thread: x=3
Primo thread: x=4
CTRL-C
bash:~$
```

Cosa è accaduto ...



Attendere un thread: pthread_join

```
#include <pthread.h>

int pthread_join(
 pthread_t thread_id,  /*ID del nuovo thread*/
 void** status_ptr /* valore ritornato */
);
/* (0) success (error number) failure */
```

Attendere un thread: pthread_join (3)

- Semantica :

  `pthread_join(tid, &status)`

  - sospende il processo che lo invoca finchè il thread identificato da **`tid`** termina
  - se **`&status!=NULL`** salva in **`status`** lo stato della terminazione
  - normalmente quando un thread termina, la memoria occupata dal suo stack privato e la posizione nella tabella dei thread non vengono rilasciate finche qualcuno non chiama la **`pthread_join()`** su quel thread
  - quindi non chiamare la **`pthread_join()`** causa memory leak
  - **`pthread_detach()`** permette di liberare le risorse senza attendere il join (attenzione, dopo aver fatto la detach la join su quel thread non si può più invocare)

Terminare un thread: pthread_exit

- Un thread può terminare normalmente:
  - Chiamando **`pthread_exit()`**
  - invocando **`return`** dalla funzione con cui è stato attivato

- può anche essere terminato da un altro thread
  - vedi cancellazione ….

Terminare un thread: pthread_exit()

```
#include <pthread.h>

void pthread_exit(
 void* retval      /* valore di ritorno */
);
/* NEVER RETURNS */
```

- chiama le routine di cleanup registrate per il thread (con **`pthread_cleanup_push()`** o similari, vedi poi)
- termina l'esecuzione salvando lo stato in modo che possa essere recuperato con una join.

Terminazione: un esempio...

- Usiamo la lista di argomenti per passare il numero di iterazioni da fare … e forniamo l'*exit status*

```
#include <pthread.h>

static int x;
static void* myfunz (void* arg){
  while (x < (int) arg) {
    printf("Secondo thread: x=%d\n",++x);
    sleep(1); }
  pthread_exit((void *) 17);
  /*equivalente a  'return (void *) 17;'*/
}
```

Terminazione: un esempio… (2)

- Usiamo la lista di argomanti per passare il numero di iterazioni da fare … e forniamo l'exit status

```
#include <pthread.h>


static int x;
static void* myfunz (void* arg){
  while (x < (int) arg) {
    printf("Secondo thread: x=%d\n",++x);
    sleep(1); }
  pthread_exit((void *) 17);
  /*equivalente a  'return (void *) 17;'*/
}
```

È necessario un cast esplicito

Terminazione : un esempio (3)

```
int main (void) {
  pthread_t tid;
  int err, status;  /* per l'exit status */
  if ( err=pthread_create(&tid, NULL, &myfun, \
          (void*)4) ) != 0 ) { /*gest err */ }
  else { /* secondo thread creato */
    while (x<4) {
      printf("Primo thread: x=%d\n",++x);
      sleep(1);}
  pthread_join(tid,(void*) &status);
  printf("Thread 2 ends: %d status",status);
}
```

Terminazione : un esempio (4)

```
int main (void) {
  pthread_t tid;
  int err, status;  /* per l'exit status */
  if ( err=pthread_create(&tid, NULL, &myfun, \
          (void*)4) ) != 0 ) { /*gest err */ }
  else { /* secondo thread creato */
    while (x<4) {
      printf("Primo thread: x=%d\n",++x);
      sleep(1);}
  pthread_join(tid,(void*) &status);
  printf("Thread 2 ends: %d status",status);
return 0;
}
```

È necessario un cast esplicito

Terminazione : un esempio (5)

```
int main (void) {
  pthread_t tid;
  int err, status;  /* per l'exit status */
/* controllo prima che lo spazio sia
  sufficiente per rappresentare un intero */
  assert(sizeof(int)<= sizeof(void*);

  if ( err=pthread_create(&tid, NULL, &myfun, \
          (void*)4) ) != 0 ) { /*gest err */ }
);
…….
```

Terminazione : un esempio (6)

- Compiliamo:

```
bash:~$ gcc main.c -o threadtest -lpthread
```

- Eseguiamo:

```
bash:~$ ./threadtest
Primo thread: x=1
Secondo thread: x=2
Primo thread: x=3
Primo thread: x=4
Thread 2 ends: 17 status
bash:~$
```

Le interferenze (*race condition*)

- I due esempi visto non sono corretti!
  - L'accesso e l'aggiornamento della variabile **`x`** non sono indivisibili!
  - Non è possibile assumere niente sull'ordine di schedulazione dei thread
  - Entrambi eseguono:

```
printf("...thread: x=%d\n", ++x );
```

  - … ma sull'aggiornamento ci possono essere interferenze….

Le interferenze (*race condition*) (2)

- Potrebbe accadere che:
  - **`x=5`**
  - il thread T1 legge 5, ma prima di incrementarla viene deschedulato
  - il thread T2 va in esecuzione, anche lui legge 5
  - T2 scrive 6, poi si sospende
  - T1 viene rieseguito e scrive anche lui 6 in **`x`**
- un incremento andato perduto!
  - Il problema è più serio per strutture dati condivise più grandi, *possono andare distrutti o persi dati importanti*

Le corse critiche (*race condition*) (3)

- Qual è il problema?
  - Garantire che un thread abbia accesso esclusivo a dati condivisi
- Nel nostro esempio
  - la lettura e l'aggiornamento di x con x+1 da parte del thread T1 devono avvenire senza che T2 acceda ad x a sua volta (*regione critica*)
  - dobbiamo garantire che i thread abbiano SEMPRE accesso *mutuamente esclusivo* alle proprie regioni critiche

Mutua esclusione : i `mutex`

- Semafori binari che permettono di bloccare (*lock*) i dati condivisi in modo che il thread possa accederli in ME

```
#include <pthread.h>
int pthread_mutex_lock(
   pthread_mutex_t *mutex,  /* mutex to lock */
)
/* (0) success (error numb) on error */

int pthread_mutex_unlock(
   pthread_mutex_t *mutex, /*mutex to unlock*/
)
/* (0) success (error numb) on error */
```

Mutua esclusione : i `mutex` (2)

- Come si usano:
  - Per ogni insieme di dati da accedere in mutua esclusione si definisce un mutex
  - Prima di accedere ai dati si chiama

```
pthread_mutex_lock(&mutex_data)
/* elaborazione in ME */
pthread_mutex_unlock(&mutex_data)
/* elaborazione NON in ME */
```

  - cosa succede ? ….

Mutua esclusione : i `mutex` (3)

```
pthread_mutex_lock(&mutex_data)
/* elaborazione in ME */
pthread_mutex_unlock(&mutex_data)
/* elaborazione NON in ME */
```

- se nessuno ha già settato (lock) il mutex **`mutex_data`** allora viene settato ed il thread prosegue
- altrimenti, il thread si blocca finchè chi ha bloccato il mutex non lo rilascia

Mutua esclusione : i `mutex` (4)

```
pthread_mutex_lock(&mutex_data)
/* elaborazione in ME */
pthread_mutex_unlock(&mutex_data)
/* elaborazione NON in ME */
```

- se nessuno è in attesa su una (lock) il mutex **`mutex_data`** allora viene sbloccato ed il thread prosegue
- altrimenti, **`mutex_data`** rimane bloccato ed uno dei thread in attesa di entrare nella regione critica viene svegliato

Mutua esclusione : i `mutex` (5)

Inizializzazione:

- se il mutex è globale è possibuile inizializzarlo usando la macro **`PTHREAD_MUTEX_INITIALIZER`**
- se il mutex non è globale è necessario invocare

```
int pthread_mutex_init (
  pthread_mutex_t * mtx,     /* mutex */
  const pthread_mutexattr_t * attr
  /*attributi  (NULL attr default)*/
)
/* Returns: always 0 */
```

Mutua esclusione : un esempio

```
void Pthread_mutex_lock(pthread_mutex_t *mtx)
    {  int err;
    if ( ( err=pthread_mutex_lock(mtx)) != 0 ) {
       errno=err;
       perror("lock");
       pthread_exit(errno); /* maybe */
    }
    else printf("locked ");
}
int Pthread_mutex_unlock(pthread_mutex_t *mtx) {
   /* as above */
    else printf("unlocked ");
}
```

Mutua esclusione : un esempio (2)

- Accessi controllati ad una variabile condivisa ...

```
#include <pthread.h>
static pthread_mutex_t mtx = \
  PTHREAD_MUTEX_INITIALIZER; /*only for extr */
static int x;
static void* myfunz (void* arg){
  while (true) {
    Pthread_mutex_lock(&mtx);
    printf("Secondo thread: x=%d\n",++x);
    Pthread_mutex_unlock(&mtx);
    sleep(1);
   }
}
```

Mutua esclusione : un esempio (3)

- Accessi controllati ad una variabile condivisa ...

```
int main (void) {
  pthread_t tid;
  int err;
  if ( err=pthread_create(&tid, NULL, &myfun, \
          NULL) ) != 0 ) { /* gest errore */ }
  else { /* secondo thread creato */
    while (true) {
      Pthread_mutex_lock(&mtx);
      printf("Primo thread: x=%d\n",++x);
      Pthread_mutex_unlock(&mtx);
      sleep(1); }
}
```

Mutua esclusione : un esempio (4)

- Compiliamo:

```
bash:~$ gcc main.c -o mutextest -lpthread
```

- Eseguiamo:

```
bash:~$ ./mutextest
locked Primo thread: x=1
unlocked locked Secondo thread: x=2
unlocked locked Primo thread: x=3
unlocked locked Secondo thread: x=4
CTRL-C
bash:~$
```

ME : alcuni commenti

- Per rendere corretta la seconda versione senza rendere il programma sequenziale dobbiamo togliere l'accesso a **x** dalla condizione del while es:

```
while (true) {
      Pthread_mutex_lock(&mtx);
      done = (x >= 10);
      if (!done)
         printf("Primo thread: x=%d\n",++x);
      Pthread_mutex_unlock(&mtx);
      if (done) break; /* not in ME */
      sleep(1); }
```

ME : alcuni commenti (2)

- Il controllo della ME è tutto a discrezione del programmatore: se non invoco il lock nessuno mi impedisce di accedere a **x**!

```
while (true) {

      done = (x >= 10);
      if (!done)
         printf("Primo thread: x=%d\n",++x);

      if (done) break;
      sleep(1); }
```

ME : alcuni commenti (3)

- È estremamente facile sbagliarsi e creare situazioni di *deadlock*

```
while (true) {
    Pthread_mutex_lock(&mtx);
    done = (x >= 10);
    if (!done)
       printf("Primo thread: x=%d\n",++x);
    Pthread_mutex_lock(&mtx); /* double lock */
    if (done) break;
    sleep(1); }
```

ME : alcuni commenti (4)

- Se una call alla unlock fallisce senza che il fallimento sia gestito accuratamente può bloccare tutti irreversibilmente!
  - Testare la unlock e riportare l'errore è il massimo che si può fare
  - si può riprovare la unlock fallita, ma non è detto che abbia successo!

```
Pthread_mutex_lock(&mtx);
done = (x >= 10);
  … … … …
/* err: la unlock fallisce a run time */
Pthread_mutex_unlock(&mtx);
```

ME : alcuni commenti (5)

- Un approccio più adeguato consiste nell'incapsulare l'accesso ai dati condivisi in una collezione di funzioni (in Java in una classe opportunamente syncronized) ed evitare di sparpagliare gli accessi ai mutex come abbiamo fatto nell'esempio!
- Per programmi più complessi diventa un incubo!

- Vediamo l'esempio rivisitato ….

Mutua esclusione : esempio 2

- Definisco una funzione che incapsula i dati condivisi ed il codice della sezione critica ...

```
static int get_and_incr_x (int incr) {
  static int x=0;  /* var condivisa */
  /* mutex per x */
  static pthread_mutex_t mtx;
  int rtn;
  inizializza();
  Pthread_mutex_lock(&mtx);
  rtn = (x += incr);
  Pthread_mutex_unlock(&mtx);
  return rtn;
}
```

Mutua esclusione : esempio2 (2)

- Accessi controllati ad una variabile condivisa ...

```
#include <pthread.h>
static void* myfunz (void* arg){

  while (true) {
     printf("T2:x=%d\n", get_and_incr_x(1));
     sleep(1);
   }
  return (void*)0;
}
```

Mutua Esclusione : esempio2 (3)

```
int main (void) {
  pthread_t tid;
  int err;
  if ( err=pthread_create(&tid, NULL, &myfun, \
          NULL ) != 0 ) { /*gest err */ }
  else { /* secondo thread creato */
    while (true) {
      printf("T1: x=%d\n", get_and_incr_x(1));
      sleep(1);
  }
  return 0;
}
```

Mutua esclusione : esempio2 (4)

- Più simile alla soluzione iniziale e più leggibile
  - generalmente significa anche più manutenibile
- Per il secondo esempio (con il test nel while) :
  - dobbiamo effettuare sia il test che l'incremento nella funzione che incapsula x
  - per esercizio …
- ci sono altri tipi di lock
  - *read-write locks, spin locks,barriers*
  - non li vedremo!

Condition variables

- I mutex non permettono di programmare in modo molto efficiente situazioni in cui i thread devono essere avvisati tempestivamente del verificarsi di certi eventi!
- Es: produttore-consumatore

Produttore

```
while(true){
    produce x
    lock(coda)
    inserisci x
    unlock(coda)
}
```

Consumatore

```
while(true) {
    lock(coda)
    if(!empty) estrai y
    unlock(coda)
    elabora y
    }
```

Condition variables (2)

- Se il produttore è lento il consumatore trova spesso la coda vuota ed esegue una serie di lock,test,unlock completamente inutili

```
while(true){
    produce x
    lock(coda)
    inserisci x
    unlock(coda)
}
```

```
while(true) {
    lock(coda)
    if(!empty) estrai y
    unlock(coda)
    elabora y
    }
```

Condition variables (3)

- Con le VC si può avvertire un thread che un evento atteso si è verificato senza sprecare lavoro.
- Idea di base:
  - una variabile di condizione C rappresenta un evento (su un dato in mutua esclusione, cui è associato un mutex) es: *coda vuota*
- Due operazioni possibili:
  - **`pthread_cond_signal(&C)`** l'evento si è verificato
  - **`pthread_cond_wait( &C, &mutex )`** si mette in attesa dell'evento legato a **`C`** e rilascia il mutex **`mutex`**

Condition variables (4)

```
#include <pthread.h>

int pthread_cond_signal(
 pthread_cond_t * cond /* cond variable*/
 );
/* (0) success (err value) error*/
```

- segnala che l'evento indicato da **`cond`** si è verificato, svegliando uno dei thread bloccati su una wait sulla stessa variabile
- *se nessuno è in attesa viene persa!*

Condition variables (5)

```
#include <pthread.h>
int pthread_cond_wait(
  pthread_cond_t * cond, /* cond variable*/
  pthread_mutex_t * mtx /* mutex*/
 );
/* (0) success (err value) error*/
```

- si blocca in attesa che qualcuno chiami signal su **`cond`**,
- va chiamata dopo aver acquisito il lock sul mutex **`mtx`**
- durante l'attesa il mutex viene rilasciato (**`unlock`**)
- quando il thread viene svegliato dalla signal anche **`mtx`** viene di nuovo acquisito (**`lock`**) in maniera atomica prima di riprendere l'esecuzione

Condition variables (6)

Inizializzazione:

- se il CV è globale è possibuile inizializzarlo usando la macro **PTHREAD_COND_INITIALIZER**
- se il mutex non è globale è necessario invocare

```
int pthread_cond_init (
  pthread_cond_t * cnd,    /* cond var*/
  const pthread_condattr_t * attr
  /*attributi  (NULL attr default) Linux lo
   ignora*/
)
/* Returns: 0 success 1 error*/
```

Produttore-Consumatore rivisto

- Il problema del produttore consumatore può essere risolto con le variabili di condizione:

Produttore

```
while(true){
    produce x
    lock(mtx)

    inserisci x
    signal(C)
    unlock(mtx)
}
```

Consumatore

```
while(true) {
    lock(mtx)
    while (empty) {
        wait(C,mtx) }
    estrai y
    unlock(mtx)
    elabora y
}
```

Produttore-Consumatore rivisto (2)

- NOTA (1): è importante osservare che la wait va invocata quando il mutex **`mtx`** è locked: la wait fa automaticamente la unlock per permettere ad altri di procedere nella sezione critica. Quando il consumatore si risveglia dalla wait mtx è di nuovo locked in modo da non avere interferenze

Produttore

```
while(true){
    produce x
    lock(mtx)
     inserisci x
    signal(C)
unlock(mtx) }
```

Consumatore

```
while(true) {
  lock(mtx)
  while (empty) {
       wait(C,mtx) }
   estrai y
unlock(mtx); elab y }
```

Produttore-Consumatore rivisto (3)

- NOTA (2): *perché la wait sta in un ciclo while(empty)?*
  - Perché dobbiamo assicurarci che la coda sia davvero vuota (se la signal è già stata fatta nessuno la ripete)
  - Perché la wait (some SC bloccante) può essere interrotta da un segnale e (se lo è) deve rimettersi in attesa se le condizioni non sono mutate

Produttore

```
while(true){
    produce x
    lock(mtx)
     inserisci x
    signal(C)
unlock(mtx) }
```

Consumatore

```
while(true) {
  lock(mtx)
  while (empty) {
       wait(C,mtx) }
   estrai y
unlock(mtx); elab y }
```

Produttore-consumatore: cenni codice

```
/* lo implementeremo integralmente alle
   esercitazioni */ /* … coda come lista …*/
struct node {
   int info;
   struct node * next;
};
/* testa lista inizialmente vuota*/
 struct node * head=NULL;
/* mutex e cond var*/
pthread_mutex_t mtx=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

Produttore-consumatore: … (2)

```
/* consumatore */
static void* consumer (void*arg) {
  struct node * p;
  while(true) {
    Pthread_mutex_lock(&mtx);
    while (head == NULL){
      Pthread_cond_wait(&cond, &mtx);
      printf("Waken up!\n"); fflush(stdout);
      }
    p=estrai();
    Pthread_mutex_unlock(&mtx);
    /* elaborazione p … … */ }
  return (void*)0;}
```

Produttore-consumatore: … (3)

```
/* produttore */
static void* producer (void*arg) {
  struct node * p;
   for (i=0; i<N; i++) {
     p=produci();
     Pthread_mutex_lock(&mtx);
     inserisci(p);
     Pthread_cond_signal(&cond);
     Pthread_mutex_unlock(&mtx);
  }
  return (void*)0;
}
```

Cancellazione

```
#include <pthread.h>
int pthread_cancel(
 pthread_t * tid /* thread id*/
 );
/* (0) success (err value) error*/
```

- fa terminare il thread indicato appena raggiunge un *punto di cancellazione*:
  - di default un PC è una chiamata di funzione ‘safe’, ovvero, non le chiamate su mutex, non malloc, realloc etc
  - spesso ambiguo
  - *si possono settare espiciti punti di cancellazione quando lo stato è consistente con* `pthread_testcancel()`

Prod-cons con cancellazione

```
int main (void){
  int t1,t2,status1,status2;
  struct node * p;
  Pthread_create(&t1,NULL,&producer,NULL);
  Pthread_create(&t2,NULL,&consumer,NULL);
  sleep(30);
  pthread_join(t1,(void*) &status1);
  pthread_cancel(t2);
  pthread_join(t2,(void*) &status2);
  /* elab return status */
  return 0;
}
```

Prod-cons con cancellazione (2)

- Alcuni commenti sul consumer:
  - **`pthread_cond_wait()`** è un cancellation point, e se il consumatore viene cancellato in questo punto si risveglia con il mutex locked e POI esegue la cancellazione: mutex può restare locked
  - se l'elaborazione del dato estratto contiene chiamate a funzioni di libreria (es. **`printf`** …), consumer può essere cancellato in quel punto, magari prima di aver fatto la **`free()`** della memoria usata dall'elemento **`p`**
  - in generale: si possono registrate delle funzioni che vengono chiamate al momento della cancellazione (e della terminazione in genere) per "mettere la cose a posto" (*cleanup handlers*)

Cleanup handlers

```
#include <pthread.h>
void pthread_cleanup_push(
 void (*handler) (void*), /*funz di cleanup*/
 void *arg; /*dati*/
 );
 void pthread_cleanup_pop(
   int execute; /* si esegue anche?*/
);
```

- vengono eseguiti in caso di una **`pthread_exit`** o di una cancellazione in ordine inverso di registrazione
- servono tipicamente per sbloccare mutex, liberare memoria, chiudere descrittori etc

Prod-cons: con cleanup handler

- Prima di fare la lock sul mutex:
  - Si registra un handler che fa la unlock
- vediamo un esempio di cleanup handler per il nostro problema produttore consumatore

```
static void cleanup_handler(void* arg) {
  free(arg);
  Pthread_mutex_unlock(&mtx);
 }
```

Prod-cons: con cleanup handler (2)

```
static void* consumer (void*arg) {
  struct node * p;
  pthread_cleanup_push(cleanup_handler,p);
  while(true) {
    Pthread_mutex_lock(&mtx);
    while (head == NULL){
      Pthread_cond_wait(&cond, &mtx);
      printf(”Waken up!\n”); fflush(stdout);}
    p=estrai();
    Pthread_mutex_unlock(&mtx);
    /* elaborazione p … … */ }
  pthread_cleanup_pop(false);
  return (void*)0;}
```

Cancellazione: alcune considerazioni

- La cancellazione di un thread è un evento brutale e difficile da programmare bene (considerando tutti i casi …)
- se è possibile meglio usare altri metodi per terminare un thread
  - nel nostro esempio: accordarsi su un valore che termini l'elaborazione del consumatore, passare al consumatore il numero di item da elaborare etc…
- in alcuni casi questo non è possibile (es. terminare un thread bloccato su una read() …) quindi è ragionevole conoscere ed usare la cancellazione.