Laboratorio di sistemi operativi
A.A. 2010/2011
Gruppo 2
Gennaro Oliva
23
I thread POSIX

I lucidi di seguito riportati sono distribuiti nei termini della licenza Creative Commons “Attribuzione/Condividi allo stesso modo 2.5” il cui testo integrale è consultabile all'indirizzo:
http://creativecommons.org/licenses/by-sa/2.5/it/legalcode

thread

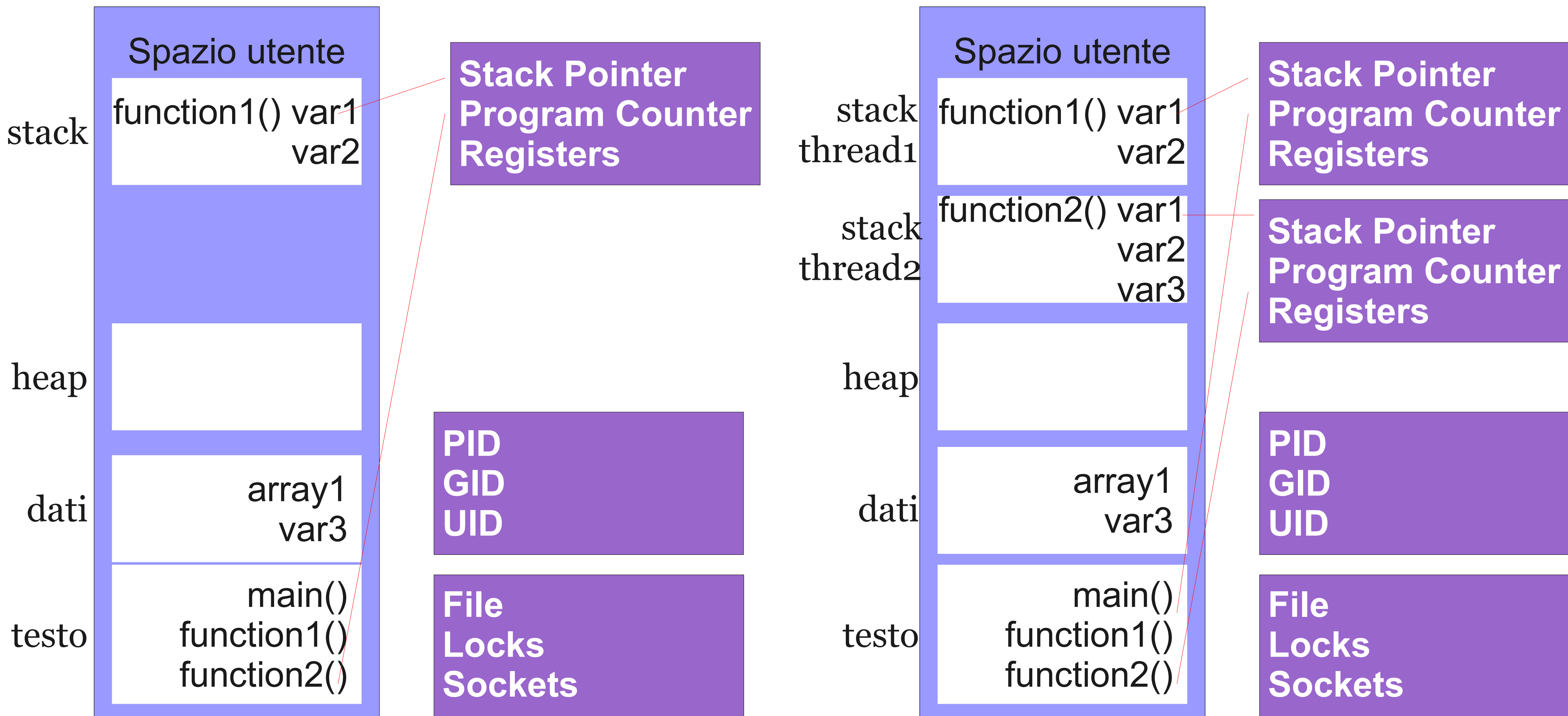
- Un thread è un flusso indipendente di istruzioni che può essere schedulato per l'esecuzione dal sistema operativo
- Dal punto di vista del programmatore un thread è una funzione che può essere eseguita indipendentemente dal programma principale (la procedura main)
- Un programma multi-thread è un programma che contiene funzioni che possono essere eseguite indipendentemente e parallelamente dal sistema operativo

Proprietà fondamentali di un thread

- Un thread esiste all'interno di un processo e ne utilizza parte delle risorse (heap, testo, dati) condividendole con gli altri thread
- Ha una serie di risorse private (stack, program counter e registri) che gli consentono di eseguire un flusso di istruzioni indipendente
- Termina se durante l'esecuzione il processo termina
- È più leggero rispetto ad un processo perché necessita di molte meno risorse

Spazio utente

- Confronto dello spazio utente tra processi single e multi-thread



La condivisione di risorse

- La condivisione delle risorse del processo da parte di tutti i suoi thread comporta che
  - i cambiamenti fatti da un thread su una risorsa condivisa (come la chiusura di un file o di un socket) si ripercuotono su tutti gli altri thread
  - i thread di uno stesso processo possono leggere e scrivere nelle stesse locazioni di memoria, pertanto sono necessari opportuni meccanismi di sincronizzazione

pthreads

- Per superare le differenze tra le librerie di thread proprietarie fornite dalle varie implementazioni di UNIX e facilitare la realizzazione di applicazioni multi-thread portabili, nel 1995 l'IEEE rilasciò lo standard POSIX Threads o più semplicemente pthreads
- Lo standard ha continuato ad evolversi negli anni: l'ultima versione pubblicata risale al 1994
- Le specifiche ufficiali dello standard sono disponibili all'indirizzo: www.unix.org/version3/ieee_std.html

Motivazioni

- La motivazione principale all'introduzione dei thread è la possibilità di realizzare flussi indipendenti di istruzioni all'interno di una stessa applicazione senza creare nuovi processi
- Rispetto ai processi i thread vengono generati con molto meno overhead da parte del sistema operativo e vengono gestiti con molte meno risorse
- Cosi come accade per i processi, nelle moderne architetture multiprocessore e multicore ogni thread può essere schedulato su una diversa CPU o su un diverso core

Motivazioni

- Rispetto alla programmazione classica le applicazioni multi-thread:
  - semplificano la progettazione del codice in applicazioni che gestiscono eventi asincroni in cui si possono assegnare thread specifici a determinati eventi che ogni thread gestisce in modo sincrono
  - semplificano la condivisione delle informazioni in quanto condividono la memoria di uno stesso processo e pertanto lo scambio di informazioni non necessita meccanismi di IPC quali pipe, fifo o socket che possono pregiudicare le prestazioni
  - consentono di sovrapporre fasi di I/O con fasi di processing attribuendo ad ognuna di esse un thread dedicato
  - sui sistemi multiprocessore o multicore consentono di assegnare task di lavoro indipendenti a diverse unità di calcolo (core o CPU) per l'esecuzione parallela

Identificazione di un thread

- Ad ogni thread viene assegnato un thread_id (o tid) che lo indentifica univocamente all'interno di un singolo processo
- Thread appartenenti a processi diversi possono avere lo stesso tid
- I tid vengono rappresentati mediante il tipo pthread_t la cui natura dipende dall'implementazione (struct, intero, ...)
- La funzione:

  pthread_t pthread_self(void);

- restituisce il thread id del thread chiamante
- Dal momento che la natura del tipo pthread_t dipende dall'implementazione, per verificare se due tid sono uguali, è necessario utilizzare la funzione:

  int pthread_equal(pthread_t t1, pthread_t t2);

- Questa funzione restituisce un valore diverso da zero se t1 e t2 sono uguali oppure zero altrimenti

Creazione di un thread

- La creazione di un nuovo thread durante l'esecuzione di un programma viene richiesta invocando la funzione:

  ```
  int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
      void *(* start)(void *), void *arg);
  ```

- In caso di successo viene generato un novo thread che eseguirà la funzione start passandole l'argomento arg
- La funzione start accetta come unico parametro un puntatore generico (void *) e restituisce un puntatore generico
- Il tid del nuovo thread verrà memorizzato nell'area di memoria puntata da tid (allocata dal chiamante), mentre l'argomento attr consentirà di specificare gli attributi del thread (è possibile usare NULL per gli attributi di default)
- La funzione pthread_create restituisce 0 in caso di successo oppure un codice d'errore in caso contrario

Gestione degli errori

- Si tenga ben presente che le funzioni della libreria pthread solitamente restituiscono `0` in caso di successo ed un codice d'errore altrimenti
- La variabile errno non viene modificata dalle funzioni della libreria
- Nella progettazione dello standard si è preferito segnalare anomalie utilizzando il codice di ritnorno dalla funzione piuttosto che utilizzare una variabile
- L'uso di una variabile globale sarebbe stato impossibile dal momento che questa è condivisa da tutti i thread

strerror_r

- Per l'interpretazione degli errori e la conversione del valore numerico in una stringa leggibile è possibile utilizzare la funzione

  int strerror_r(int errnum, char *buf, size_t len);

- che memorizza la stringa d'errore corrispondente al codice d'errore errnnum all'interno di buf: un array di caratteri la cui dimensione è specificata da len
- La funzione restituisce 0 in caso di successo e -1 in caso di errore settando opportunamente la variabile errno (strerror_r non è una funzione della libreria pthread, si può usare sempre!)

Esempio d'uso di pthread_create

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
void *dummy_thread(void *arg) { pthread_exit(NULL); }
void *dummy_fork(void *arg) { exit(0); }
void pthread_perror(char *string, int errcode) {
  char errmsg[256];
  strerror_r(errcode,errmsg,256);
  printf("%s:%s\n",string,errmsg);
}
int main (int argc, char *argv[]) {
   pthread_t *threads;
   int rc,nt,*pids;
   long threads_microsec,fork_microsec,t;
   struct timeval ora,dopo;
   float perc;
   if ( argc != 2 )
     printf("Numero di argomenti %d non valido\n",argc),exit(1);
   nt=strtol(argv[1],NULL,10);
   if (nt < 0)
     printf("Numero di thread %d non valido\n",nt),exit(1);
   threads = (pthread_t *) malloc ( nt * sizeof(pthread_t) );
```

Esempio d'uso di pthread_create

```
    pids = (int *) malloc ( nt * sizeof(int) );
    if (pids == NULL)
      perror("malloc"),exit(1);
    gettimeofday(&ora,NULL);
    for(t=0; t<nt; t++)
       if ( (rc = pthread_create(&threads[t], NULL, dummy_thread, NULL)) )
         pthread_perror("pthread_create",rc),exit(1);
    gettimeofday(&dopo,NULL);
    threads_microsec=dopo.tv_usec-ora.tv_usec;
    threads_microsec+=1000000*(dopo.tv_sec-ora.tv_sec);
    printf("Tempo per la creazione dei threads %ld nsec\n",threads_microsec);
    gettimeofday(&ora,NULL);
    for(t=0; t<nt; t++) {
       if ( (pids[t] = fork()) < 0 )
         perror("fork"),exit(1);
       if ( pids[t] == 0 )
         dummy_fork(NULL);
    }
    gettimeofday(&dopo,NULL);
    fork_microsec=dopo.tv_usec-ora.tv_usec;
    fork_microsec+=1000000*(dopo.tv_sec-ora.tv_sec);
    printf("Tempo per la creazione dei processi %ld nsec",fork_microsec);
    perc = 100*(((float)fork_microsec-threads_microsec)/(float)fork_microsec);
    printf(" (%.2f%%)\n",perc);
    pthread_exit(NULL);
}
```

Condivisione di memoria nei thread

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
int glob = 1;
void * incrementa (void *arg) {
  int *p;
  p = arg;
  *p = (*p) + 1;
  glob++;
  return NULL;
}
void pthread_perror(char *string, int errcode) {
  char errmsg[256];
  strerror_r(errcode,errmsg,256);
  printf("%s:%s\n",string,errmsg);
}
int main (int argc, char *argv[]) {
  int local = 1, rc;
  int *dyn;
  pthread_t tid;
  dyn = (int *) malloc (sizeof (int));
  *dyn = 1;
  if( (rc = pthread_create (&tid, NULL, incrementa, (void *) dyn)) )
    pthread_perror("pthread_create",rc),exit(1);
  if( (rc = pthread_create (&tid, NULL, incrementa, (void *) &local)))
    pthread_perror("pthread_create",rc),exit(1);
  sleep (1);
  printf ("glob=%d local=%d *dyn=%d\n", glob, local, *dyn);
  pthread_exit (0);
}
```

Passaggio di parametri

- Il quarto parametro passato alla funzione pthread_create è l'unico parametro che viene passato alla funzione start ed è un puntatore generico (void *)
- Con tale parametro è possibile specificare un indirizzo di memoria relativo a qualsiasi tipo di variabile effettuando un opportuno cast
- In tal caso è il thread creatore ed il nuovo thread condividono l’area di memoria indirizzata e pertanto una qualsiasi modifica fatta su tale area da parte del thread creatore si ripercuote immediatamente sul valore letto dal thread creato
- Se si desidera passare alla funzione start più parametri è possibile utilizzare una struct e passarne il puntatore
- Così come accade per i processi non è possibile sapere in quale ordine verranno eseguiti i due thread, quindi è opportuno evitare modifiche dei parametri da parte del thread creatore

Esempio sbagliato di passaggio

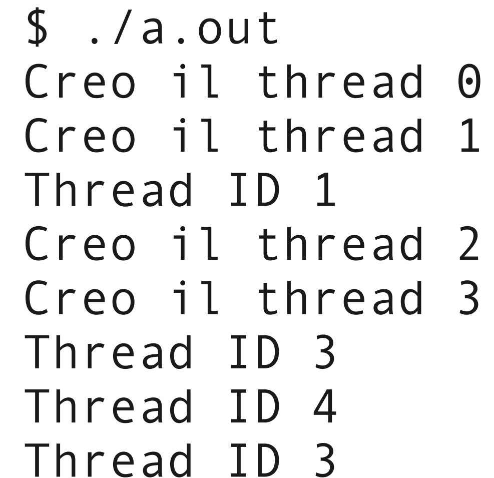
- Supponiamo di voler passare ad ogni thread un intero identificativo crescente come argomento

```
#include <pthread.h>
#include <stdio.h>
void *printid(void *id) {
  int *myid;
  myid=id;
  printf("Thread ID %d\n",*myid);
  return NULL;
}
int main() {
int t;
pthread_t threads[4];
for(t=0; t<4; t++) {
   printf("Creo il thread %d\n", t);
   pthread_create(&threads[t], NULL,printid, &t);
}
pthread_exit(0);
}
```

Esempio sbagliato di passaggio

- La variabile contatore t viene modificata prima che i thread vi accedano in lettura provocando l'output errato:

```
#include <pthread.h>
#include <stdio.h>
void *printid(void *id) {
  int *myid;
  myid=id;
  printf("Thread ID %d\n",*myid);
  return NULL;
}
int main() {
int t;
pthread_t threads[4];
for(t=0; t<4; t++) {
   printf("Creo il thread %d\n", t);
   pthread_create(&threads[t], NULL,printid, &t);
}
pthread_exit(0);
}
```



Esempio corretto di passaggio

- Per risolvere questo problema è possibile definire un array con gli identificativi e passare ad ogni thread un diverso elemento dell'array

```
#include <pthread.h>
#include <stdio.h>
void *printid(void *id) {
   int *myid;
   myid=id;
   printf("Thread ID %d\n",*myid);
   return NULL;
}
int main() {
int t,id[4];
pthread_t threads[4];
for(t=0; t<4; t++) {
    id[t]=t;
    printf("Creo il thread %d\n", t);
    pthread_create(&threads[t], NULL,printid, id+t);
}
pthread_exit(0);
}
```

```
$ ./a.out
Creo il thread 0
Creo il thread 1
Thread ID 0
Creo il thread 2
Creo il thread 3
Thread ID 2
Thread ID 3
Thread ID 1
```

Esempio corretto di passaggio

- In alternativa è possibile allocare dinamicamente, per ciascun thread un'area di memoria e passarne l' indirizzo alla funzione start

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
void *printid(void *id) {
  int *myid;
  myid=id;
  printf("Thread ID %d\n",*myid);
  free(myid);
  return NULL;
}
int main() {
int t,*id;
pthread_t threads[4];
for(t=0; t<4; t++) {
   id=(int *) malloc (sizeof(int));
   *id=t;
   printf("Creo il thread %d\n", t);
   pthread_create(&threads[t], NULL,printid, id);
}
pthread_exit(0);
}
```

```
$ ./a.out
Creo il thread 0
Creo il thread 1
Thread ID 0
Creo il thread 2
Creo il thread 3
Thread ID 2
Thread ID 3
Thread ID 1
```

Esempio corretto di passaggio

- In alternativa, ricordando che il passaggio dei parametri in C viene effettuato per valore, è possibile passare l'indice come valore del puntatore

```
#include <pthread.h>
#include <stdio.h>
void *printid(void *id) {
  long myid;
  myid=(long) id;
  printf("Thread ID %ld\n",myid);
  return NULL;
}
int main() {
long t;
pthread_t threads[4];
for(t=0; t<4; t++) {
   printf("Creo il thread %ld\n", t);
   pthread_create(&threads[t], NULL,printid, (void *) t);
}
pthread_exit(0);
}
```

```
$ ./a.out
Creo il thread 0
Creo il thread 1
Thread ID 0
Creo il thread 2
Creo il thread 3
Thread ID 1
Thread ID 2
Thread ID 3
```

Terminazione di un thread

- L'invocazione di exit, _exit o _Exit provoca la terminazione di un processo e con esso di tutti i suoi thread

- Per terminare un singolo thread è possibile:

1) invocare
   return val
   dalla funzione start

2) utilizzare la funzione:
   void pthread_exit(void *val)

3) invocare da un altro thread la funzione:
   void pthread_cancel(pthread_t tid);
   che richiede la terminazione del thread tid e restituisce 0 in caso di successo oppure un codice d'errore

Valore d'uscita di un thread

- Il valore d'uscita del thread può essere specificato dopo return o come parametro di pthread_exit
- Se il valore passato è un indirizzo di memoria è necessario che i dati memorizzati alla locazione corrispondente sopravvivano alla terminazione
- Il valore di ritorno quindi non può puntare ad una variabile automatica (si possono usare variabili globali o allocate dinamicamente)

pthread_cancel

- Per terminare un altro thread in esecuzione è possibile utilizzare la funzione:

  int pthread_cancel (pthread_t tid);

- che invia al thread tid una richiesta di cancellazione
- Di default la chiamata a pthread_cancel equivale alla situazione in cui il thread tid invoca pthread_exit con argomento PTHREAD_CANCELED
- Un thread può ignorare le richieste di cancellazione, se vengono impostati opportunamente i suoi attributi come verrà illustrato successivamente

pthread_join

- Ogni thread può recuperare il valore d'uscita di un altro thread utilizzando la funzione:

  int pthread_join(pthread_t tid, void **ret);

- La funzione sospende l'esecuzione del thread che la invoca in attesa che il thread con identificativo tid termini
- Nel caso in cui il thread sia già terminato la funzione ritorna immediatamente restituendo `0` in caso di successo oppure un codice d'errore altrimenti
- ret è un parametro di output usato per copiare il valore d'uscita del thread terminato nella locazione puntata da *ret
- Si presti attenzione al fatto che ret è un puntatore a puntatore essendo il valore d'uscita un puntatore
- Se si desidera ignorare valore d'uscita è possibile passare NULL
- Nel caso che tid è un thread cancellato ret varrà la costante PTHREAD_CANCELED

Esempio d'uso di pthread_join

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
void *dummy_thread(void *arg) { pthread_exit(NULL); }
void *dummy_fork(void *arg) { exit(0); }
void pthread_perror(char *string, int errcode) {
  char errmsg[256];
  strerror_r(errcode,errmsg,256);
  printf("%s:%s\n",string,errmsg);
}
int main (int argc, char *argv[]) {
   pthread_t *threads;
   int rc,nt,*pids;
   long threads_microsec,fork_microsec,t;
   struct timeval ora,dopo;
   float perc;
   if ( argc != 2 )
     printf("Numero di argomenti %d non valido\n",argc),exit(1);
   nt=strtol(argv[1],NULL,10);
   if (nt < 0)
     printf("Numero di thread %d non valido\n",nt),exit(1);
   threads = (pthread_t *) malloc ( nt * sizeof(pthread_t) );
   pids = (int *) malloc ( nt * sizeof(int) );
   if (pids == NULL)
     perror("malloc"),exit(1);
```

Esempio d'uso di pthread_join

```
   gettimeofday(&ora,NULL);
   for(t=0; t<nt; t++)
      if ( (rc = pthread_create(&threads[t], NULL, dummy_thread, NULL)) )
        pthread_perror("pthread_create",rc),exit(1);
   for(t=0; t<nt; t++)
     if ( (rc = pthread_join(threads[t],NULL) ) )
       pthread_perror("pthread_join",rc),exit(1);
   gettimeofday(&dopo,NULL);
   threads_microsec=dopo.tv_usec-ora.tv_usec;
   threads_microsec+=1000000*(dopo.tv_sec-ora.tv_sec);
   printf("Tempo per la creazione dei threads %ld nsec\n",threads_microsec);
   gettimeofday(&ora,NULL);
   for(t=0; t<nt; t++) {
      if ( (pids[t] = fork()) < 0 )
        perror("fork"),exit(1);
      if ( pids[t] == 0 )
        dummy_fork(NULL);
   }
   for(t=0; t<nt; t++)
     if ( waitpid(pids[t],NULL,0) < 0 )
       perror("waitpid");
   gettimeofday(&dopo,NULL);
   fork_microsec=dopo.tv_usec-ora.tv_usec;
   fork_microsec+=1000000*(dopo.tv_sec-ora.tv_sec);
   printf("Tempo per la creazione dei processi %ld nsec",fork_microsec);
   perc = 100*(((float)fork_microsec-threads_microsec)/(float)fork_microsec);
   printf(" (%.2f%%)\n",perc);
   pthread_exit(NULL);
}
```

Attributi di un thread

- Gli attributi di un thread possono essere utilizzati per impostare alcune proprietà tra cui:
  - la possibilità di ignorare le richieste di cancellazione
  - la possibilità di essere eseguito in modalità detached
- Per impostare gli attributi di un thread in fase di creazione è necessario utilizzare una variabile di tipo pthread_attr_t che va inizializzata con:

```
int pthread_attr_init(pthread_attr_t *attr);
```

- e finalizzata con:

```
int pthread_attr_destroy(pthread_attr_t
*attr);
```

Detached State

- Se non interessa il valore di uscita di un thread e non si intende aspettarne la terminazione utilizzando pthread_join è possibile creare il thread in detached state
- A tal fine è possibile utilizzare la funzione
- int pthread_attr_setdetachstate(pthread_attr_t *attr, int state);
- Specificando come state PTHREAD_CREATE_DETACHED
- e successivamente invocare pthread_create specificando attr come secondo parametro

Cancellabilità di un thread

- Durante l'esecuzione un thread può essere cancellabile o non cancellabile e cambiare questo attributo nel corso del programma
- All'atto della creazione tutti i thread sono cancellabili in quanto non è possibile attribuire la non cancellabilità mediante una variabile pthread_attr_t
- Quando un altro thread chiama pthread_cancel
  - se il thread è cancellabile, viene terminato
  - se non è cancellabile, la richiesta di cancellazione viene memorizzata, in attesa che il thread diventi cancellabile

Cancellabilità di un thread

- int pthread_setcancelstate(int state, int *oldstate);
- imposta la cancellabilità a state e restituisce la vecchia cancellabilità in oldstate
- state e oldstate possono assumere i valori:
  - PTHREAD_CANCEL_ENABLE
  - PTHREAD_CANCEL_DISABLE

Thread fork ed exec e segnali

- Se un thread invoca fork durante l'esecuzione verrà generato un processo figlio costituito dal solo thread che ha invocato la fork
- Quando un thread invoca una funzione della famiglia exec l'intero processo diventa un nuovo programma e tutti i suoi thread vengono terminati
- Le chiamate a signal influenzano tutti i thread:
  - un segnale inviato ad un processo multi-thread che ha impostato un handler per quel dato segnale, viene recapitato ad un thread a caso che interrompe momentaneamente l'esecuzione ed esegue l'handler
  - quando la reazione al segnale prevede la terminazione del processo, tutti i suoi thread vengono terminati

pthread_kill

- Per inviare un segnale ad un determinato thread è possibile utilizzare la funzione

  int pthread_kill(pthread_t tid, int sig);

- che invia il segnale sig al thread tid
- Il comportamento è identico al caso precedente fatta eccezione che in questo caso si conosce il thread che esegue l'handler

Analogie tra processi e thread

- In tabella una comparazione tra le principali funzioni per la gestione di thread e processi

Pocessi	Thread
fork	pthread_create
exit	pthread_exit
waitpid	pthread_join
getpid	pthread_self
kill	pthread_kill

Funzioni non thread-safe

- Una funzione che può essere invocata senza rischi da più thread contemporaneamente viene detta thread-safe
- Le funzioni che restituiscono indirizzi di aree di memoria opache (non allocate dal programma) sono spesso non thread-safe
- Esempi di funzioni non thread-safe illustrate o utilizzate durante questo corso sono:

  gethostbyname, rand, readdir, strtok
- L'utlizzo simultaneo di tali funzioni da parte di thread concorrenti potrebbe generare risultati inaspettati

Funzioni reentrant

- Per alcune funzioni non thread-safe sono disponibili delle versioni alternative dette reentrant che godono di questa proprietà
- Le funzioni reentrant per quelle citate precedentemente sono:

  gethostbyname_r, rand_r, readdir_r, strtok_r
- La pagina di manuale di una funzione solitamente indica se questa non è thread-safe e nel caso esista fornisce informazioni sulla sua versione reentrant

Laboratorio di sistemi operativi
A.A. 2010/2011
Gruppo 2
Gennaro Oliva
23
Sincronizzazione di thread POSIX

Thread e sincronizzazione

- I thread di uno stesso processo condividono lo stesso spazio di indirizzamento, pertanto, eventuali modifiche concorrenti effettuate da più thread su strutture dati condivise, possono causare problemi di inconsistenza
- Per scongiurare tali spiacevoli situazioni sono necessari meccanismi di sincronizzazione
- La sincronizzazione è necessaria quando due o più thread accedono ad una struttura dati condivisa ed almeno uno di essi vi accede in scrittura

La sincronizzazione è necessaria quando due o più thread accedono ad una struttura dati condivisa ed almeno uno di essi vi accede in scrittura

Thread non sicronizzati

- La sincronizzazione non è necessaria quando:
  - tutti i thread accedono ad una struttura dati in sola lettura
  - la struttura dati viene letta e modificata sempre e solo dallo stesso thread

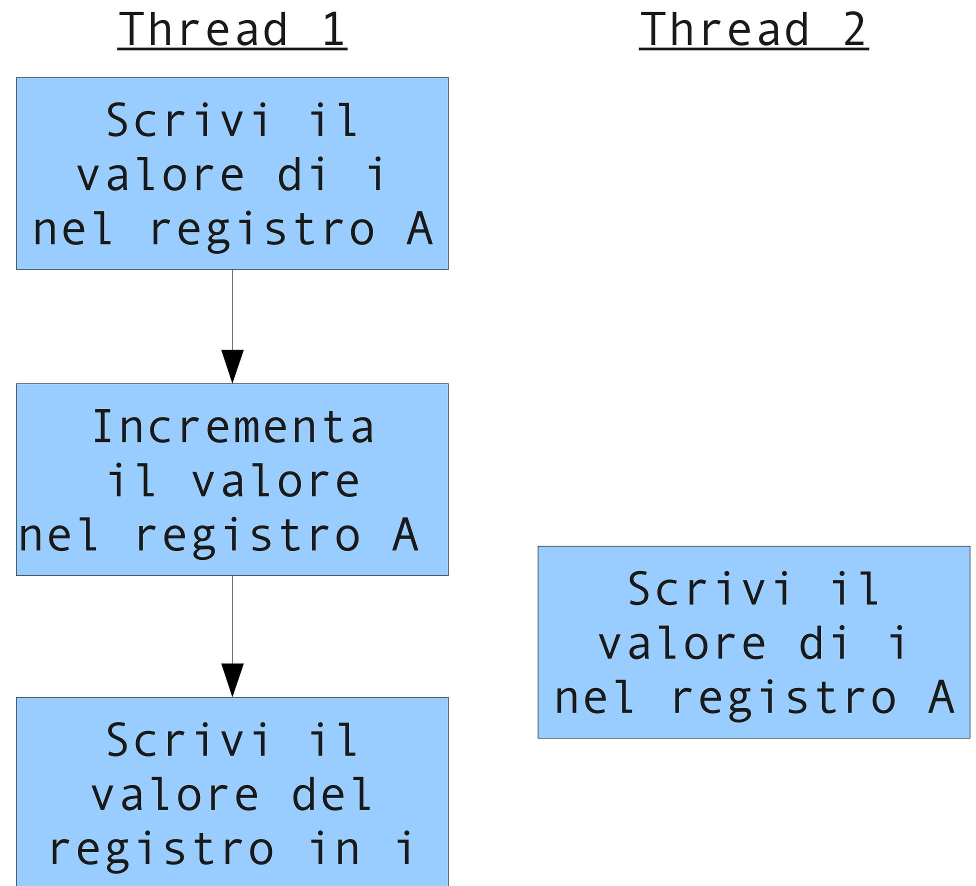
La sincronizzazione è necessaria quando due o più thread accedono ad una struttura dati condivisa ed almeno uno di essi vi accede in scrittura

Modifica di una variabile

- Su alcune architetture la modifica del valore di una variabile non è atomica
- Supponiamo di voler incrementare la variabile i di una unità (eseguiamo l'istruzione i++)
- La modifica viene effettuata secondo la seguente sequenza di istruzioni atomiche:
  - carica la variabile i in un registro
  - incrementa il registro
  - memorizza il valore del registro in i

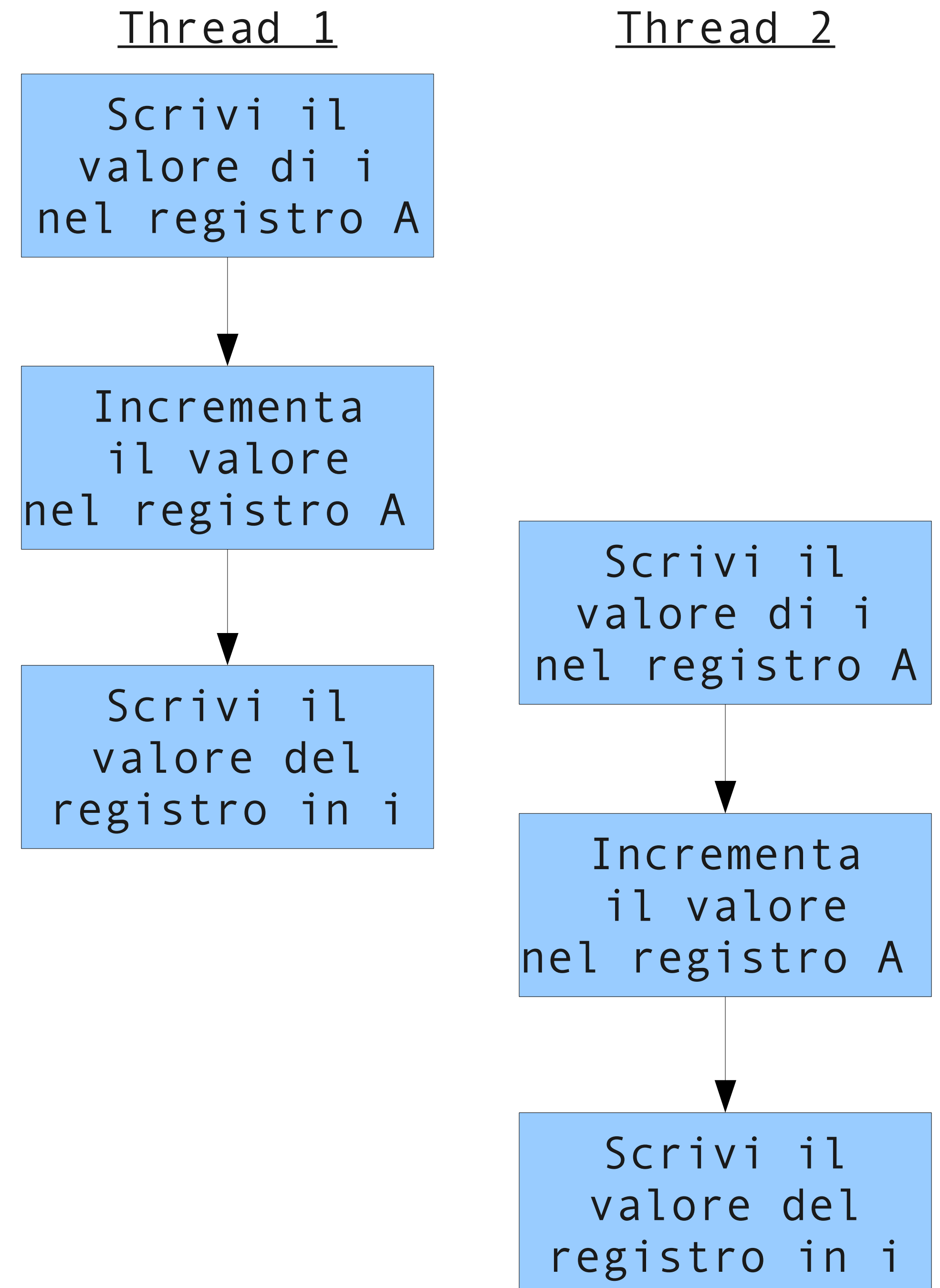
Modifica di una variabile

- In questa situazione il thread 2 non legge il valore incrementato, quindi non ha una visione aggiornata dello stato del programma



Modifica di una variabile

- Se anche il thread 2 incrementa la variabile, al termine delle operazioni, il valore della variabile risulta incrementato di un unità invece che di due



Race Conditions

- In generale quando quando due o più thread o processi accedono ad una risorsa condivisa e il risultato finale dipende dall'ordine in cui i processi vengono schedulati parliamo di race condition
- Per evitare il verificarsi di tali condizioni è possibile utilizzare un sistema di sincronizzazione chiamato mutex (mutua esclusione) che impedisce l'accesso ad una risorsa condivisa da parte di altri thread fintanto che è questa occupata da un thread

mutex

- Un thread che voglia accedere ad una risorsa condivisa protetta da mutex dovrà:
  - acquisire il lock sul mutex
  - eseguire l'accesso
  - rilasciare il lock sul mutex
- Un thread che tenta di acquisire il lock di un mutex già acquisito viene sospeso
- Nel momento in cui il lock viene rilasciato, i thread sospesi tornano ad essere eseguibili
- Il primo thread schedulato, acquisirà il mutex, forzando gli altri thread a ritornare in attesa

pthread_mutex_t

- Il tipo pthread_mutex_t consente di dichiarare una variabile di tipo mutex che va sempre comunque inizializzata prima dell'utilizzo
- Se la variabile è statica, è sufficiente assegnarle la costante PTHREAD_MUTEX_INITIALIZER
- Se la variabile è allocata dinamicamente, è necessario utilizzare la funzione di inizializzazione:

  int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
- dove mutex è il puntatore alla variabile mutex ed attr è un puntatore ai suoi attributi (NULL per attributi standard)
- Se un mutex è stato allocato dinamicamente è necessario invocare la funzione di finalizzazione:

  int pthread_mutex_destroy(pthread_mutex_t *mutex);
- prima di deallocare la variabile

Lock e Unlock

- Un thread tenta di acquisire un lock su un mutex utilizzando:

  int pthread_mutex_lock(pthread_mutex_t *mutex);

- Se il lock del mutex non è stato già acquisito da un altro thread la funzione acquisisce il lock e ritorna; in caso contrario la funzione sospende l'esecuzione del thread in attesa che il lock venga rilasciato
- Un thread rilascia un lock su un mutex utilizzando la funzione:

  int pthread_mutex_unlock(pthread_mutex_t *mutex);

- Un thread può rilasciare esclusivamente lock da esso acquisiti e non quelli acquisiti da altri thread

Trylock

- In alternativa un thread può provare ad acquisire il lock di un mutex senza eventualmente sospendere l'esecuzione utilizzando:
- int pthread_mutex_trylock(pthread_mutex_t *mutex);
- Se il lock del mutex non è stato già acquisito da un altro thread la funzione acquisisce il lock e ritorna; in caso contrario la funzione ritorna immediatamente restituendo il codice d'errore EBUSY
- In questo modo è possibile continuare l'esecuzione del thread e tentare periodicamente di acquisire il lock fino a quando l'operazione non ha successo

Atomicità dei mutex

- La mutua esclusione non inibisce in modo automatico l'accesso ad una variabile o ad una risorsa condivisa
- L'applicazione deve essere progettata in modo tale che tutti i thread che condividono la risorsa acquisiscano il lock sul mutex prima dell'accesso
- Il sistema operativo garantisce esclusivamente l'atomicità delle primitive di gestione dei mutex

Esempio d'uso di mutex

```
#include <pthread.h>
pthread_mutex_t count_mutex=PTHREAD_MUTEX_INITIALIZER
long long count;

void incrementa_contatore() {
  pthread_mutex_lock(&count_mutex);
  count = count + 1;
  pthread_mutex_unlock(&count_mutex);
}

long long leggi_contatore() {
  long long c;
  pthread_mutex_lock(&count_mutex);
  c = count;
  pthread_mutex_unlock(&count_mutex);
  return (c);
}
```

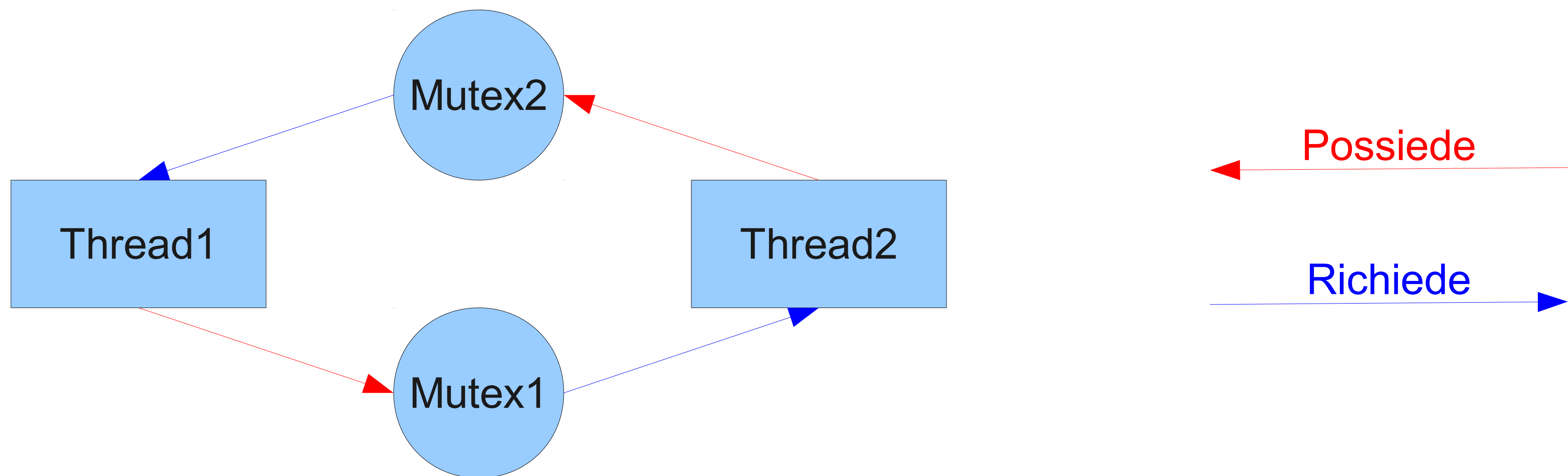
- La funzione incrementa_contatore utilizza il mutex per assicurarsi che l'incremento di count avvenga atomicamente
- La funzione leggi_contatore utilizza il mutex per assicurarsi che la lettura del contatore sia atomica

Deadlock

- I mutex possono generare condizioni di stallo chiamate deadlock durante le quali ogni thread aspetta un evento che solo un altro thread (anch'esso in stallo) può provocare
- Un esempio di deadlock si verifica in presenza di due mutex e di due thread che avendone acquisito uno tentano di acquisire l'altro
- Un modo semplice per evitare questa situazione è acquisire i lock sempre nello stesso ordine, ma questo non è sempre possibile
- Una alternativa può essere quella di utilizzare trylock

Deadlock

- Nella rappresentazione grafica Thread1 ha acquisito il lock del Mutex1 ed è bloccato nell'attesa che si liberi Mutex2
- Viceversa il Thread2 ha acquisito il lock del Mutex2 ed è blocatto nell'attesa che si liberi il Mutex1



Condition Variable

- Le condition variable sono un'ulteriore meccanismo di sincronizzazione per i thread che possono essere utilizzate come punto di rendevouz in programmi multi-thread
- Le condition variable consentono ad un thread di attendere che una determinata condizioni si verifichi evitando race condition
- Ogni condition variable è protetta da un mutex che preserva la struttura dati utilizzata per verificare la condizione
- I thread che vogliono verificare o modificare la condizione devono acquisire il lock sul mutex

Condition Variable

- Ogni thread che vuole verificare la condizione acquisisce il lock sul mutex ad essa associata e successivamente effettua la verifica
- Se la condizione non è verificata il thread atomicamente rilascia il mutex e si aggiunge all'elenco dei thread in attesa che la condizione cambi stato, sospendendo l'esecuzione
- Un thread che vuole modificare la condizione acquisisce il lock ed effettua la modifica; quindi rilascia il lock e segnala ai thread in attesa che la modifica è stata effettuata
- Il primo thread schedulato dopo la modifica acquisisce atomicamente il lock sul mutex e procede l'esecuzione

Condition Variable

- Quando un thread modifica la condizione e segnala a quelli in coda il cambiamento tutti i thread in attesa diventano eseguibili
- Ogni thread che ritorna dall'attesa non può essere sicuro che la condizione sia vera ma deve verificarne nuovamente lo stato
- Un altro thread può essere stato schedulato dopo il segnale e prima di lui e aver modificato nuovamente lo stato della condizione
- Per questo motivo le funzioni di attesa vengono inserite all’interno di un ciclo per la verifica della condizione

pthread_cond_t

- Il tipo pthread_cond_t consente di dichiarare una condition variabile che analogamente a quanto accade con i mutex va sempre comunque inizializzata prima dell'utilizzo
- Se la variabile è statica, è sufficiente assegnarle la costante PTHREAD_COND_INITIALIZER
- Se la variabile è allocata dinamicamente, è necessario utilizzare la funzione di inizializzazione:

  int pthread_cond_init(pthread_cond_t *cond, const const pthread_condattr_t *attr);
- dove cond è il puntatore alla condition variabile ed attr è un puntatore ai suoi attributi (NULL per attributi standard)
- Se una condition variable è stata allocata dinamicamente è necessario invocare la funzione di finalizzazione:

  int pthread_cond_destroy(pthread_cond_t *cond);
- prima di deallocare la variabile

Esempio d'uso di Condition Variable

- Un esempio d'uso di condition variable è rappresentato da una coda di lavoro in un'applicazione multi-thread in cui ci sono thread che inseriscono messaggi nella coda (produttori) e thread che leggono e rimuovono i messaggi per il processing (consumatori)
- L'accesso alla coda è protetto da un mutex: è necessario acquisire il lock per inserire o rimuovere messaggi dalla coda
- La presenza di messaggi all'interno della coda è gestita mediante una condition variable: la condizione è falsa se la coda è vuota o vera in presenza di messaggi
- I thread consumatori acquisiscono il lock e verificano la condizione e nel caso questa sia falsa mediante una singola operazione atomica rilasciano il lock e sospendono l'esecuzione
- I thread produttori acquisiscono il lock, inseriscono il messaggio, rilasciano il lock e successivamente inviano un segnale ai thread consumatori in attesa

Esempio d'uso di Condition Variable

```
#include <pthread.h>
struct msg {
    struct msg *m_next;
    /* ... dati ... */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;
void process_msg(void){ /* Verifica */
    struct msg *mp;
    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* processa la richiesta mp */
    }
}
void enqueue_msg(struct msg mp){ /*Modifica*/
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

1) Acquisisce il lock sul mutex associato

2) Verifica la condizione

3) Se la condizione non è verificata il lock viene rilasciato ed il thread sospeso

Esempio d'uso di Condition Variable

```
#include <pthread.h>
struct msg {
    struct msg *m_next;
    /* ... dati ... */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;
void process_msg(void){ /* Verifica */
    struct msg *mp;
    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* processa la richiesta mp */
    }
}
void enqueue_msg(struct msg mp){ /*Modifica*/
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

1) Acquisisce il lock sul mutex associato

2) Verifica la condizione

3) Se la condizione non è verificata il lock viene rilasciato ed il thread sospeso

4) Acquisisce il lock sul mutex associato

5) Modifica la condizione

6) Rilascia il lock

7) Segnala ai thread in attesa che la modifica è stata effettuata

Esempio d'uso di Condition Variable

```
#include <pthread.h>
struct msg {
    struct msg *m_next;
    /* ... dati ... */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;
void process_msg(void){ /* Verifica */
    struct msg *mp;
    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* processa la richiesta mp */
    }
}
void enqueue_msg(struct msg mp){ /*Modifica*/
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

8) Il thread schedulato riprende l'esecuzione e acquisisce atomicamente il lock del mutex

9) Il messaggio viene rimosso dalla coda

10) Il lock viene rilasciato

4) Acquisisce il lock sul mutex associato
5) Modifica la condizione
6) Rilascia il lock
7) Segnala ai thread in attesa che la modifica è stata effettuata

Esempio d'uso di Condition Variable

- Il segnale può essere inviato al di fuori del lock in quanto i thread consumatori verificano la condizione all'interno del ciclo while
- Un consumatore che si sveglia e trova la condizione falsa perché un altro consumatore ha prelevato e cancellato il messaggio si pone nuovamente in attesa

Thread specific data

- I Thread-specific data (anche detti thread-private data) consentono di allocare aree di memoria per informazioni associate ai singoli thread
- Tali aree sono dette specifiche o private poiché ogni thread ne possiede una propria copia a cui può accedere senza sincronizzarsi con gli altri
- Le aree di memoria thread-specific vengono utilizzate in alternativa agli array globali indicizzati in cui ogni elemento (o insieme di elementi) è associato ad uno specifico thread poiché questi possono risultare complicati da gestire e da proteggere

errno

- Le aree di memoria thread-specific consentono inoltre di adattare le interfacce utilizzate normalmente nella programmazione single thread in ambiente multi-thread
- Un esempio di tale applicazione è l'implementazione dei servizi forniti nella applicazioni single thread dalla variabile errno in applicazioni multithread
- Nelle applicazioni single thread errno è una variabile globale che system call e funzioni di libreria standard modificano in caso di errore
- In un programma multi-thread, per utilizzare lo stesso meccanismo e conoscere gli errori verificatisi in base al valore di errno, la variabile viene memorizzata in un'area di memoria thread-specific
- In questo modo un thread che invoca una funzione che a modifica il valore di errno, non altera il valore della variabile per gli altri thread

Creazione di una chiave

- Prima di allocare un'area di memoria thread-specific è necessario creare una chiave da associare ai dati utilizzando la funzione:

  int pthread_key_create(pthread_key_t *keyp,
  void (*destructor)(void *))

- La chiave viene memorizzata in keyp e verrà utilizzata da tutti i thread per accedere al proprio spazio thrad-specific
- Subito dopo la creazione l'indirizzo dell'area di memoria thread-specific per ogni thread è NULL

Distruttore

- Ad ogni chiave è possibile associare una funzione distruttore che viene eseguita alla terminazione regolare del thread (return o pthread_exit) solo nel caso in cui il puntatore all'area di memoria thread-specific non è NULL
- Al distruttore viene passato come unico argomento l'indirizzo dell'area di memoria thread-specific
- Il distruttore viene solitamente utilizzato per liberare la memoria thread-specific associata alla chiave e allocata dinamicamente per evitare che venga persa (memory leak)
- Passano NULL come secondo argomento a pthread_key_create non verrà invocato nessun distruttore

pthread_{set,get}specific

- Una volta creata una chiave, per associarle un area di memoria thread-specific si può utilizzare la funzione:

  int pthread_setspecific(pthread_key_t *key, const void* val);

- che associa l'indirizzo val alla chiave key
- Per conoscere l'indirizzo dell'area di memoria thread-specific associato ad una chiave è possibile utilizzare:

  void* pthread_getspecific(pthread_key_t *key);

- che restituisce l'indirizzo di memoria associato alla chiave key
- Tutti i thread utilizzano la stessa variabile key per accedere alla propria area di memoria thread-specific ottenendo un indirizzo diverso

Esempio d'uso thread-specific data

```
#include <string.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define ERRSTRINGSIZE 256
pthread_key_t key;
pthread_mutex_t protect=PTHREAD_MUTEX_INITIALIZER;
char *strerror_reentrant(int errnum) {
    char *errstring;
    int len,rc;
    errstring = (char *) pthread_getspecific(key);
    if (errstring == NULL) {
        if ( (errstring = (char *) malloc(ERRSTRINGSIZE*sizeof(char))) == NULL )
            return(NULL);
        if ( (rc = pthread_setspecific(key, errstring)) != 0 )
          fprintf(stderr,"pthread_setspecific error %d\n",rc),exit(1);
    }
    pthread_mutex_lock(&protect);
    len = 1+strlen(strerror(errnum));
    if ( len > ERRSTRINGSIZE )
      return NULL;
    strncpy ( errstring, strerror(errnum), len);
    pthread_mutex_unlock(&protect);
    return errstring ;
}

int main(int argc, char *argv[]) {
...
pthread_key_create(&key, free);
...
pthread_create ( &tid, NULL, start, (void *) i );
...
```

L'area di memoria thread-specific viene allocata solo la prima volta: getspecific restituisce NULL se non è stata ancora eseguita una setspecific

La chiave va creata prima che i thread utilizzino la funzione