

Pthreads

Corso di Sistemi Distribuiti e Cloud Computing A.A. 2016/17

Valeria Cardellini

Materiale didattico su thread POSIX

- POSIX Threads Programming
computing.llnl.gov/tutorials/pthreads/
- Esempi di codice ed articoli sul sito del corso
- Per approfondimenti
 - K.A. Robbins, S. Robbins, “UNIX Systems Programming: Communication, Concurrency and Threads”, Prentice Hall, 2003.
 - D. Butenhof, “Programming with POSIX Threads”, Addison Wesley, 1997. Chapter 3 su sincronizzazione:
[ptgmedia.pearsoncmg.com/images/9780201633924/
samplepages/0201633922.pdf](http://ptgmedia.pearsoncmg.com/images/9780201633924/samplepages/0201633922.pdf)
 - W.R. Stevens, S.A. Rago, “Advanced Programming in the UNIX(R) Environment, 3rd Edition”, Addison-Wesley, 2013.
www.informit.com/articles/article.aspx?p=2085690
 - POSIX Standard: www.unix.org/version3/ieee_std.html

Un esempio per iniziare

- Architettura sw di un Web server
- Server iterativo

```
while (1) {
    accept_new_connection();
    read_command_from_connection();
    handle_connection(); /* gestione richieste
sulla connessione*/
    close_connection();
}
```

- Bassa QoS
- Attacco DoS facile: client invia comandi lentamente o si ferma a metà

- Server multi-process

```
while (1) {
    accept_new_connection();
    read_command_from_connection();
    if (fork() == 0) {
        handle_connection();
        close_connection();
        exit(0);
    }
}
```

- Crash del sistema per numero eccessivo di processi
- Degrado delle prestazioni a causa del numero elevato di processi
- Processi isolati (occorre usare IPC)

Avremmo bisogno di...

- Generare “processi” più velocemente e con un minor overhead sulle risorse del sistema
- Usare meno memoria
- Condividere dati tra flussi di esecuzione



Quali soluzioni?

- Migliorare l’architettura sw del server (preforking, event-driven, ...)
 - Soluzioni già esaminate
- Usare i thread
 - Come? Obiettivo delle prossime lezioni

Processi in UNIX

- Processo in UNIX: è creato dal SO (con un certo overhead) e contiene informazioni sulle sue risorse e sul suo stato di esecuzione
 - Process ID, user ID, group ID
 - Variabili d'ambiente
 - Working directory
 - Codice del programma
 - Registri
 - Stack
 - Heap
 - Descrittori di file
 - Maschera dei segnali e disposizioni per i segnali
 - Librerie condivise
 - Meccanismi per la comunicazione tra processi
- Un processo in UNIX ha un unico thread di controllo

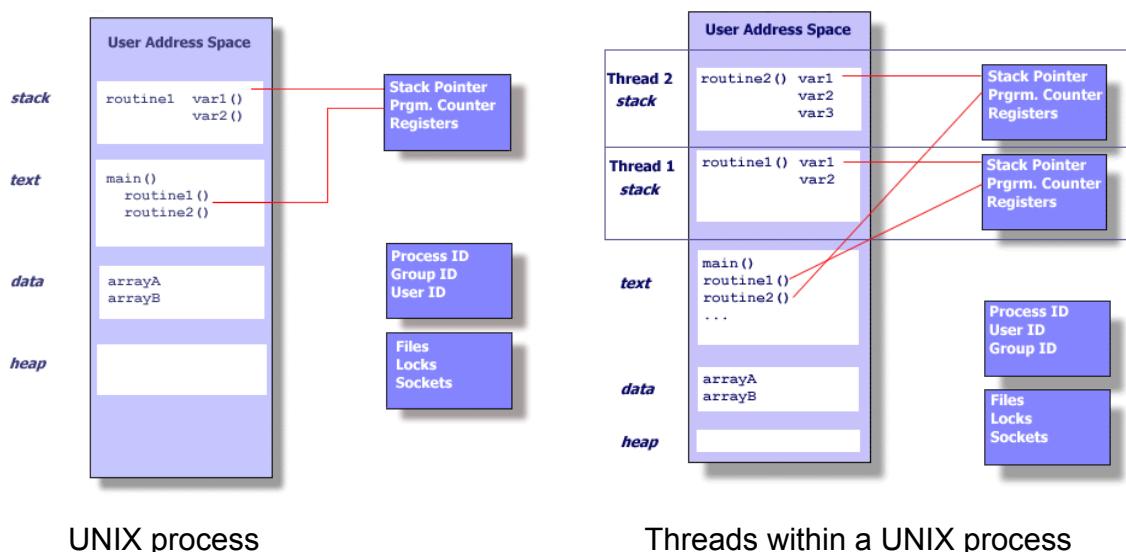
Relazione tra thread e processi

- Thread: **flusso indipendente di istruzioni** che viene schedulato ed eseguito come tale dal SO
 - Stato e risorse di un thread:
 - Thread ID
 - Stack
 - Registri (incluso program counter e stack pointer)
 - Proprietà di scheduling (priorità e politica)
 - Maschera dei segnali
 - Dati specifici del thread
 - Variabile `errno`
 - Un thread condivide con gli altri thread nello stesso processo:
 - Memoria globale
 - Codice del programma
 - Descrittori aperti
 - Working directory
 - Group ID e user ID
 - Gestori dei segnali e disposizioni per i segnali

Relazione tra thread e processi (2)

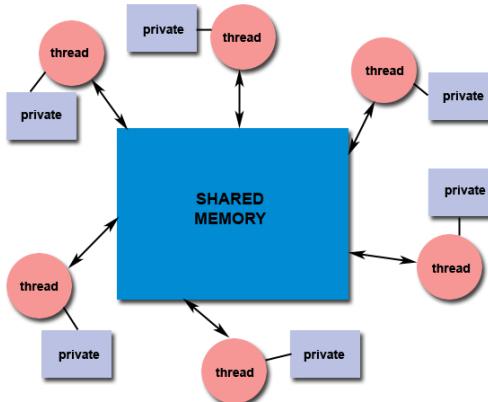
- In ambiente UNIX un thread:
 - Esiste all'interno di un processo ed usa le sue risorse
 - Vive come flusso indipendente finché non muore il suo processo
 - Condivide risorse con altri thread dello stesso processo
 - E' leggero rispetto ad un processo
- Vivendo nello stesso processo
 - I thread condividono lo spazio di indirizzamento
 - Cambiamenti fatti da un thread sono visibili agli altri thread
 - Letture e scritture nella stessa area di memoria richiedono sincronizzazione

Process and threads in UNIX



Modello a memoria condivisa

- Tutti i thread dello stesso processo accedono alla stessa **memoria globale condivisa**
- I thread hanno anche i loro **dati privati**
- Occorre sincronizzare l'accesso ai dati globali condivisi

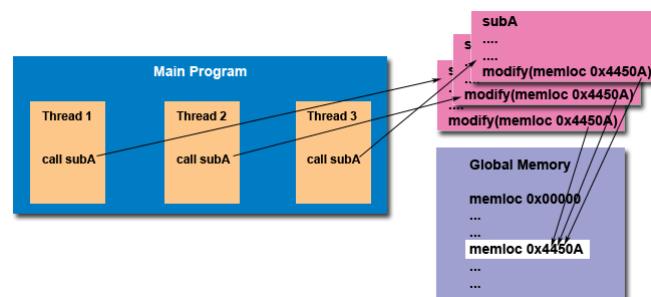


Valeria Cardellini - SDCC 2016/17

8

Thread safeness

- Capacità di eseguire diversi thread in modo sicuro, senza sporcare i dati condivisi
- Esempio: 3 thread chiamano una stessa funzione di libreria
 - Questa funzione accede/modifica una struttura globale in memoria
 - Può accadere che i thread cerchino di modificare allo stesso tempo l'area condivisa
 - Una libreria *thread-safe* sincronizza l'accesso di ciascun thread alla struttura condivisa
- Esempio: `getenv()` non è *thread-safe*



Valeria Cardellini - SDCC 2016/17

9

Thread design patterns

- Three common design patterns to use threads within a program:
 - Manager/worker
 - One manager (aka boss or master) thread dispatches other threads to do useful work, which are usually part of a *worker thread pool*
 - Thread pool can be either static or dynamic
 - Peer
 - Similar to manager/worker, except the manager also does some useful work (i.e., is a peer to the other threads)
 - Pipeline
 - Similar to how pipelining works in a processor: the work is divided into a chain of tasks; each task is concurrently managed by a different thread

Valeria Cardellini - SDCC 2016/17

10

Issues in thread programming

- Thread programs are more difficult to develop
- They often contain bugs that are notoriously difficult to find and fix
- Recall: design first, not just code!

If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions.

— Albert Einstein

Valeria Cardellini - SDCC 2016/17

11

POSIX threads

- L'interfaccia di programmazione dei thread in UNIX è stata standardizzata da IEEE nel 1995, con lo standard POSIX 1003.1
 - Tutte le implementazioni fedeli a questo standard si chiamano **POSIX threads** o **pthreads**
 - Lo standard POSIX è definito solo per il linguaggio C
- pthreads definisce un insieme di tipi C e funzioni, esportate dall'header file **pthread.h** e implementate tramite la libreria **libpthread**
 - Convenzione per i nomi: tutte le funzioni della libreria pthread iniziano con **pthread_**
 - Esempio di compilazione in Linux

```
gcc -pthread -o main main.c
```
- In Linux esiste la chiamata di sistema **clone()**, che offre molti dei benefici dei thread
 - Attenzione: non è portabile! Meglio usare pthreads

Valeria Cardellini - SDCC 2016/17

12

L'insieme di funzioni di pthreads

- Tre classi di funzioni
 - Più di 60 funzioni, analizzeremo le principali
 - La maggior parte delle funzioni restituisce 0 in caso positivo, altrimenti un valore diverso da 0 che rappresenta il codice di errore
- **Gestione dei thread**: funzioni per creare, distruggere, aspettare un thread
- **Mutex**: costrutti e funzioni di mutua esclusione (*mutex=mutual exclusion*) per garantire che un solo thread possa eseguire ad un certo istante un blocco di codice
- **Condition variable**: costrutti e funzioni per la comunicazione tra thread che condividono un mutex, per aspettare o segnalare il verificarsi di condizioni

Valeria Cardellini - SDCC 2016/17

13

Identificazione di un thread

- Come un processo, anche un thread ha il proprio ID (detto **tid**)
 - A differenza dell'ID di un processo (pid), il tid ha senso solo all'interno del processo a cui il thread appartiene
 - Tipo di dato `pthread_t` è **opaco** (ad es., a seconda dell'implementazione può essere una struttura)
- `pthread_self` per conoscere il proprio tid

```
pthread_t pthread_self(void);
```
- `pthread_equal` per confrontare due tid

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Creazione di un thread

```
int pthread_create(pthread_t *tidp,  
                  pthread_attr_t *attr, void *(*start_rtn)(void *),  
                  void *arg);
```

- `pthread_create()` crea un nuovo thread
- Parametri della funzione
 - `tidp`: tid del thread creato
 - `attr`: attributi del thread da creare (NULL per valori di default, alcuni attributi possono essere modificati)
 - `start_rtn`: funzione eseguita dal thread creato
 - `arg`: argomento di ingresso per `start_rtn`
- Una volta creato, un thread può creare altri thread
 - No gerarchia o dipendenza implicita tra thread
- Dopo aver creato il thread, il programmatore sa quando il SO schedulerà quel thread per l'esecuzione?

Esempio: stampare l'ID del thread

```
#include ...
#include <pthread.h>

pthread_t ntid;

void printids(const char *s) {
    pid_t pid;
    pthread_t tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
           (unsigned int)tid, (unsigned int)tid);
}

void *thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}
```

vedere threadid.c

Valeria Cardellini - SDCC 2016/17

16

Esempio: stampare l'ID del thread (2)

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

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0) {
        printf("Error; can't create thread: %s\n", strerror(err));
        exit(1);
    }
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

Output su Linux

```
main thread: pid 32579 tid 3086538432 (0xb7f8d6c0)
new thread:  pid 32579 tid 3086535600 (0xb7f8cbb0)
```

- **Osservazioni:**

- `sleep()` in `main()` per evitare che il thread principale termini prima del nuovo thread
- `pthread_self()` nel nuovo thread anziché uso della variabile globale `ntid`
- Perché e quali sono le soluzioni alternative?

Valeria Cardellini - SDCC 2016/17

17

Terminazione di un thread

- Un thread può terminare perché:
 1. L'intero processo viene terminato con una chiamata a `exit()` o `exec()`
 2. Il thread termina l'esecuzione della sua routine
 3. Il thread chiama `pthread_exit()`

```
int pthread_exit(void *rval_ptr);
```

- Usata per far terminare esplicitamente un thread
- Non causa la chiusura di file aperti dal thread che la invoca
- Attenzione alla terminazione di `main()`: usare `pthread_exit()` in `main()` anziché `exit()` per evitare che il processo (e tutti i thread) siano terminati

4. Il thread viene cancellato da un altro thread tramite `pthread_cancel()`

```
int pthread_cancel(pthread_t tid);
```

Esempio: creazione e terminazione di thread

```
#include ...
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}
```

[vedere hello.c](#)

Esempio: creazione e terminazione di thread (2)

```
int main(int argc, char *argv[ ])
{
    pthread_t threads[NUM_THREADS];
    int err;
    long t;

    for (t=0; t<NUM_THREADS; t++) {
        printf("In main: creating thread %d\n", t);
        err = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (err != 0) {
            printf("Error; can't create thread %d: %s\n", t, strerror(err));
            exit(1);
        }
    }
    pthread_exit(NULL);
}
```

Un possibile output su Linux

```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
```

20

Valeria Cardellini - SDCC 2016/17

Ritorno di un valore alla terminazione

- Il valore di ritorno posto in `rval_ptr` di `pthread_exit()` è visibile ad un altro thread nel processo che invoca `pthread_join()`

```
int pthread_join(pthread_t tid, void **rval_ptr);
```

- Il thread che invoca `pthread_join()` si blocca finché il thread identificato da `tid` non termina (a meno che non sia già terminato)
- Il valore di ritorno del thread che termina con `pthread_exit()` è salvato nella locazione di memoria puntata da `*rval_ptr`
 - Tale area di memoria deve essere **persistente** all'uscita del thread: allocata nello heap tramite `malloc` oppure una variabile globale
- Vedere come esempio **exitstatus.c**
- `pthread_join()` può essere invocata solo su un thread joinable
- Errore logico: cercare di effettuare più join sullo stesso thread

Valeria Cardellini - SDCC 2016/17

21

Passaggio di argomenti ai thread

- Come passare **più di un argomento in input** ad un thread creato con `pthread_create()` oppure far ritornare **più di un argomento in output** da un thread che termina con `pthread_exit()`?
 - Passando un puntatore ad una struttura contenente tutti gli argomenti di input o output
 - Attenzione: l'area di memoria puntata deve essere ancora valida quando il thread chiamante termina
 - Ad esempio, in caso di output non deve essere allocata nello stack del thread che termina
 - Vedere **badexit.c** come esempio di codice errato: per passare l'output viene usata una variabile automatica allocata nello stack del thread che termina

Esempio: passaggio di parametri

- L'esempio seguente è sbagliato: perché?

Per il codice completo vedere **hello_arg4.c**

```
int err;
long t;

for (t=0; t<NUM_THREADS; t++) {
    printf("Creating thread %ld\n", t);
    err = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
    ...
}
```

Motivo dell'errore: viene passato l'indirizzo di `t` (allocato nello stack) e durante il ciclo viene modificato il contenuto dell'indirizzo passato come parametro

Esempio: passaggio di parametri

- Come passare in modo corretto un valore intero ad ogni thread creato
 - Usando una struttura dati univoca per ogni thread

```
...
int *taskids[NUM_THREADS];
int t;

for (t=0; t<NUM_THREADS; t++) {
    taskids[t] = (int *)malloc(sizeof(int));
    *taskids[t] = t;
    printf("Creating thread %d\n", t);
    err = pthread_create(&threads[t], NULL, PrintHello, (void *)taskids[t]);
    ...
}
...
```

Per il codice completo vedere [hello_arg1.c](#)

Esempio: passaggio di parametri (2)

- Come passare ad ogni thread creato più argomenti di input tramite una struttura

```
...
struct thread_data {
    int thread_id;
    int sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg) {
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *)threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}
```

Esempio: passaggio di parametri (3)

- Segue da lucido precedente

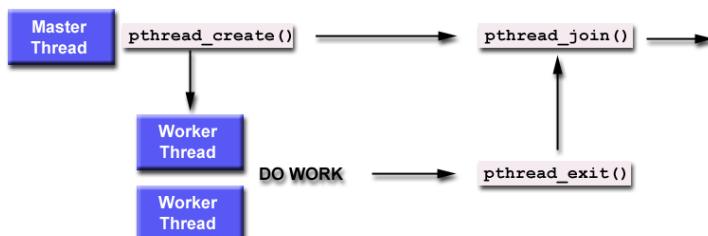
```
int main (int argc, char *argv[ ]) {  
    ...  
    thread_data_array[t].thread_id = t;  
    thread_data_array[t].sum = sum;  
    thread_data_array[t].message = messages[t];  
    err = pthread_create(&threads[t], NULL, PrintHello,  
                        (void *)&thread_data_array[t]);  
    ...  
}
```

- L'esempio è corretto?

Per il codice completo vedere [hello_arg2.c](#)

Thread joinable o detached

- Già analizzata `pthread_join()`
 - Fornisce un primo meccanismo per sincronizzare i thread



- Se un thread è nello stato *detached*, appena esso termina il SO può reclamare le sue risorse
- Se è invece nello stato *joinable*, il suo stato di terminazione è mantenuto finché non viene invocata `pthread_join()` su di esso
- Per distaccare un thread

```
int pthread_detach(pthread_t tid);
```

Thread joinable o detached (2)

- Quando si crea un thread, uno dei suoi attributi definisce se il thread è joinable o detached
 - Attenzione: alcune implementazioni di pthreads non creano per default un thread nello stato joinable
 - No `pthread_join()` su un thread detached
- Come creare un thread joinable (o detached)?
 - Dichiарare una variabile attributo di tipo `pthread_attr_t`
 - Inizializzare la variabile attributo usando la funzione `pthread_attr_init()`
 - Impostare lo stato joinable o detached usando la funzione `pthread_attr_setdetachstate()`
 - Liberare le risorse usate per l'attributo usando la funzione `pthread_attr_destroy()`

Esempio: creazione di thread joinable

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NUM_THREADS 4
void *BusyWork(void *t)
{
    int i;
    long tid;
    double result=0.0;
    tid = (long)t;
    printf("Thread %ld starting...\n",tid);
    for (i=0; i<1000000; i++)
    {
        result = result + sin(i) * tan(i);
    }
    printf("Thread %ld done. Result = %e\n", tid, result);
    pthread_exit((void*) t);
}
```

vedere joinable.c

Esempio: creazione di thread joinable (2)

```
int main (int argc, char *argv[])
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int err;
    long t;
    void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(t=0; t<NUM_THREADS; t++) {
        printf("Main: creating thread %ld\n", t);
        err = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
        if (err != 0) {
            printf("Error: can't create thread %ld: %s\n", t, strerror(err));
            exit(1);
        }
    }
}
```

Esempio: creazione di thread joinable (3)

```
/* Free attribute and wait for the other threads */
pthread_attr_destroy(&attr);
for (t=0; t<NUM_THREADS; t++) {
    err = pthread_join(thread[t], &status);
    if (err != 0) {
        printf("Error: can't join with thread %ld: %s\n", t, strerror(err));
        exit(-1);
    }
    printf("Main: completed join with thread %ld having a status of
           %ld\n", t,(long)status);
}

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Gestione dello stack

- Lo standard POSIX non definisce la dimensione dello stack di un thread
 - Terminazione del programma o corruzione di dati se si supera la dimensione (di default) dello stack
 - Funzioni `pthread_attr_setstacksize()` per impostare la dimensione dello stack e
`pthread_attr_getstacksize()` per conoscerla
 - La dimensione dello stack deve essere un multiplo della dimensione della pagina
 - Funzioni `pthread_attr_setstackaddr()` per impostare l'indirizzo di memoria dello stack e
`pthread_attr_getstackaddr()` per conoscerlo

Esempio: gestione dello stack

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
    double A[N][N];
    int i,j;
    long tid;
    size_t mystacksize;
    tid = (long)threadid;
    pthread_attr_getstacksize (&attr, &mystacksize);
    printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            A[i][j] = ((i*j)/3.452) + (N-i);
    pthread_exit(NULL);
}
```

vedere stacksize.c

Esempio: gestione dello stack (2)

```
int main(int argc, char *argv[])
{
    pthread_t threads[NTHREADS];
    size_t stacksize;
    int err;
    long t, pagesize;

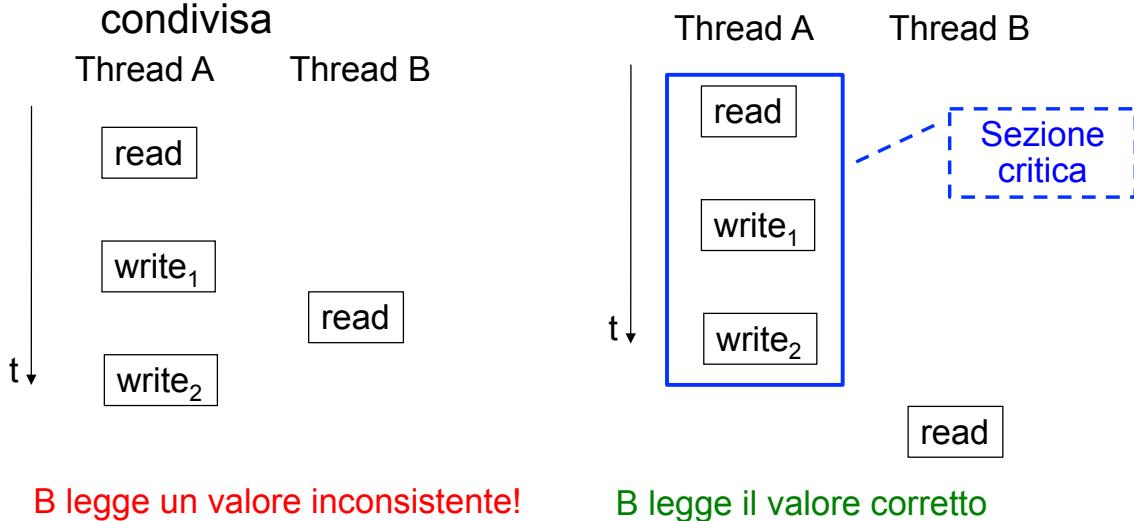
    pagesize = sysconf(_SC_PAGESIZE);
    printf("Page size = %ld bytes\n", pagesize);
    pthread_attr_init(&attr);
    pthread_attr_getstacksize(&attr, &stacksize);
    printf("Default stack size = %li bytes\n", stacksize);
    stacksize = sizeof(double)*N*N+MEGEXTRA;
    printf("Amount of stack needed per thread = %li bytes\n", stacksize);
    stacksize = (stacksize/pagesize)*pagesize;
    pthread_attr_setstacksize(&attr, (size_t)stacksize);
    printf("Creating threads with stack size = %li bytes\n", stacksize);
    for (t=0; t<NTHREADS; t++) {
        err = pthread_create(&threads[t], &attr, dowork, (void *)t);
        ...
    }
}
```

Sincronizzazione tra thread

- `pthread_join()` è una forma elementare di sincronizzazione
 - Ma non basta
- Sincronizzazione attraverso **struttura dati condivisa** (variabili globali, statiche e dinamiche)
 - Condivise tra thread, di cui almeno uno può modificarla
 - Necessari meccanismi di protezione dei dati condivisi
- Meccanismi forniti da `pthreads`
 - Semafori di mutua esclusione (mutex)
 - Lock lettore/scrittore
 - Variabili condition
- Compito del programmatore
 - **Corretto utilizzo dei meccanismi di sincronizzazione**

Evitare interferenze

- Come prevenire il verificarsi di **race condition** (**interferenza**)?
- Esempio: due thread accedono alla stessa variabile condivisa



Evitare interferenze (2)

- Un altro esempio: due thread non sincronizzati incrementano la stessa variabile globale

Thread A	Thread B	Saldo
Legge saldo: € 1000		€ 1000
	Legge saldo: € 1000	€ 1000
	Deposita € 200	€ 1000
Deposita € 200		€ 1000
Aggiorna saldo € 1000 + € 200		€ 1200
	Aggiorna saldo € 1000 + € 200	€ 1200

Mutex: MUTual EXclusion

- Protezione della sezione/regione critica
 - Ad es. variabile condivisa modificata da più thread
 - Obiettivo: fare in modo che le sezioni critiche di due o più thread non vengano mai eseguite contemporaneamente (mutua esclusione)
 - Solo un thread alla volta può accedere ad una risorsa condivisa protetta da un **mutex**
 - Il mutex è un **semaforo binario**
 - Due soli stati del mutex: aperto (*unlocked*) o chiuso (*locked*)
 - Solo un thread alla volta può possedere il mutex (ottenerne il lock)
- Interfaccia
 - **Lock** per bloccare una risorsa condivisa
 - **Unlock** per liberare una risorsa condivisa

Condizioni per mutua esclusione

4 condizioni per assicurare la mutua esclusione:

- a) Un solo thread per volta esegue la sezione critica
- b) Non viene fatta nessuna assunzione sulla velocità relativa dei thread
- c) Nessun thread che sta eseguendo codice esterno alla sezione critica può bloccare un altro thread
- d) Nessun thread attende indefinitamente di entrare nella sezione critica

Uso di mutex

- Sequenza tipica:
 - Il mutex viene creato ed inizializzato
 - Più thread tentano di accedere alla risorsa condivisa invocando l'operazione di lock del mutex
 - Un solo thread riesce ad acquisire il mutex, mentre gli altri si bloccano
 - Variante di lock non bloccante: **trylock**
 - Il thread che ha acquisito il mutex usa la risorsa condivisa
 - Lo stesso thread la rilascia invocando l'operazione di unlock del mutex
 - Attenzione: un thread può sbloccare solo i mutex su cui ha acquisito il lock
 - Un altro thread tra quelli bloccati acquisisce il mutex

Mutex: tipo e inizializzazione

- In pthreads un mutex è una variabile di tipo `pthread_mutex_t`
- Prima di usare il mutex, occorre inizializzarlo
- Inizializzazione
 - **Statica**: contestuale alla dichiarazione (più efficiente)

```
pthread_mutex_t mymutex =
PTHREAD_MUTEX_INITIALIZER;
```
 - **Dinamica**: attraverso la funzione `pthread_mutex_init()`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      pthread_mutexattr_t *attr);
```
- Parametri della funzione `pthread_mutex_init()`
 - `mutex`: puntatore al mutex da inizializzare
 - `attr`: puntatore agli attributi del mutex
 - Estensioni per sistemi real time; se NULL usa valori di default

Mutex: operazioni lock e trylock

- Due varianti per il lock di un mutex
 - bloccante (standard)
 - non bloccante (utile per evitare deadlock)

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- **Parametro**
 - `mutex`: puntatore al mutex da bloccare
- **Valore di ritorno**
 - 0 in caso di successo, diverso da 0 altrimenti
 - `trylock()` restituisce `EBUSY` se il mutex è occupato

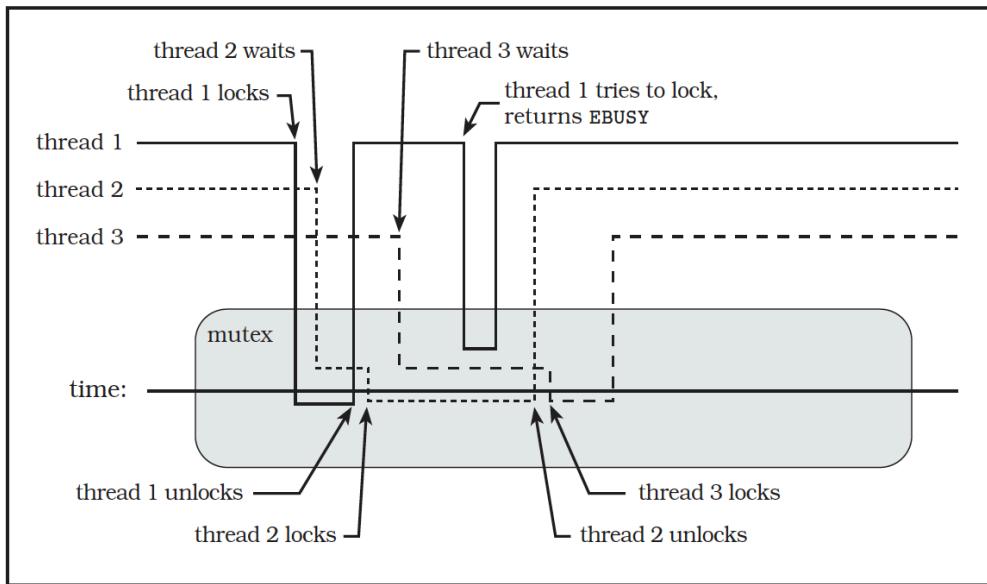
Mutex: unlock e distruzione

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);  
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- **Parametro**
 - `mutex`: puntatore al mutex da sbloccare/distruggere
- **Valore di ritorno**
 - 0 in caso di successo, diverso da 0 altrimenti
- **Se il mutex è stato allocato dinamicamente,**
`pthread_mutex_destroy()` per liberare la memoria occupata
- Cosa succede se un thread fa riferimento ad un mutex dopo che questo è stato distrutto?
 - Non definito

Mutex: timing diagram

- Example: 3 threads sharing a mutex



Esempio: protezione di struttura dati

```
#include <stdlib.h>
#include <pthread.h>

struct foo {
    int f_count;
    pthread_mutex_t f_lock;
    /* ... more stuff here ... */
};

struct foo *foo_alloc(void)           /* allocate the object */
{
    struct foo *fp;
    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}
```

Conta il numero di thread che stanno usando la struttura dati condivisa: la struttura può essere deallocated solo quando non vi sono thread che la stanno usando

Esempio: protezione di struttura dati (2)

```
void foo_hold(struct foo *fp)           /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;                      incremento del contatore
    pthread_mutex_unlock(&fp->f_lock);
}

void foo_rele(struct foo *fp)           /* release a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    }
    else
        pthread_mutex_unlock(&fp->f_lock);
}
```

decremento del contatore
protetto da mutex: se nessun
thread sta usando la
struttura, viene deallocated

Esempio: protezione di funzione di libreria thread-unsafe

- Generatore di numeri casuali protetto da mutex
 - Restituisce un numero tra 0 e 1

```
#include <pthread.h>
#include <stdlib.h>
int randsafe(double *ramp)
{
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int error;
    if (error = pthread_mutex_lock(&lock))
        return error;
    *ramp = (rand() + 0.5)/(RAND_MAX + 1.0); rand(): int tra 0 e RAND_MAX
    return pthread_mutex_unlock(&lock);
}
```

- In alternativa, se la funzione deve restituire un numero tra 0 e RAND_MAX, basta sostituire l'assegnamento in verde con
`*ramp = rand();`

Esempio: flag di sincronizzazione

```
#include <pthread.h>
static int doneflag = 0;
static pthread_mutex_t donelock = PTHREAD_MUTEX_INITIALIZER;

int getdone(int *flag) {      /* get the flag */
    int error;
    if (error = pthread_mutex_lock(&donelock))
        return error;
    *flag = doneflag;
    return pthread_mutex_unlock(&donelock);
}

int setdone(void) {          /* set the flag */
    int error;
    if (error = pthread_mutex_lock(&donelock))
        return error;
    doneflag = 1;
    return pthread_mutex_unlock(&donelock);
}
```

Flag di sincronizzazione
pari ad 1 se setdone è stata
invocata almeno una volta,
0 altrimenti.
setdone imposta il flag ad
1, getdone ritorna il valore
del flag

Attenzione: un errore comune è NON
proteggere gli accessi in lettura a
variabili condivise

Valeria Cardellini - SDCC 2016/17

48

Esempio: flag di sincronizzazione (2)

- Come esempio, usiamo il flag di sincronizzazione per decidere se eseguire un altro comando in un'applicazione multithreaded

```
...
void docommand(void);
...
int error = 0;
int done = 0;

while (!done && !error) {
    docommand();
    error = getdone(&done);
}
```

Valeria Cardellini - SDCC 2016/17

49

Issues with mutex management

- Mutex management introduces potential overheads, delays and risks of deadlock:
 - **Lock overhead**: the more locks the program uses, the more overhead associated with the lock usage
 - **Lock contention**: the more fine-grained the available locks, the less likely one thread will request a lock held by the others
 - E.g., locking a row rather than locking the entire table
 - **Deadlock**: two or more competing threads are each waiting for a lock that the other holds
- Tradeoff between lock overhead and lock contention

Mutex granularity

- **Granularity**: measure of the amount of code/data the mutex is protecting
- Coarse granularity (i.e., small number of mutexes, each protecting relatively large amount of code/data)
 - reduces lock overhead
 - but increases lock contention
- Fine granularity (i.e., large number of mutexes, each protecting relatively small amount of code/data)
 - reduces lock contention
 - but increases lock overhead
 - can be more deadlock-prone because of mutex dependencies

Granularità della sincronizzazione

- La sincronizzazione può essere:
 - Per sezione critica
 - Quando una struttura condivisa viene modificata in un unico punto nel codice
 - Basta associare un mutex alla sezione critica, soluzione più intuitiva
 - Per struttura
 - Quando la struttura può essere modificata in più punti nel codice
 - Utile se più strutture devono essere condivise contemporaneamente
 - È necessario associare un mutex alla struttura
- Inoltre può essere necessario definire più mutex per proteggere diverse strutture dati

Gerarchia di mutex e deadlock

- Può essere necessario acquisire più mutex per accedere a diverse risorse condivise
- Esempio:

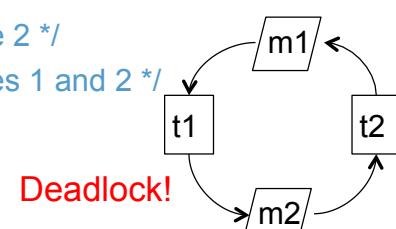
Thread 1

```
pthread_mutex_lock(&m1);      /* use resource 1 */  
pthread_mutex_lock(&m2);      /* use resources 1 and 2 */  
pthread_mutex_unlock(&m2);  
pthread_mutex_unlock(&m1);
```

Thread 2

```
pthread_mutex_lock(&m2);      /* use resource 2 */  
pthread_mutex_lock(&m1);      /* use resources 1 and 2 */  
pthread_mutex_unlock(&m1);  
pthread_mutex_unlock(&m2);
```

E' corretto?



Deadlock!

Grafo di allocazione delle risorse

Gerarchia di mutex e deadlock

- Attenzione: possibile situazione di deadlock!
 - Nota: anche un thread “da solo” può creare un deadlock eseguendo un lock su un mutex che ha già acquisito se il mutex è di tipo standard (PTHREAD_MUTEX_NORMAL); un mutex ricorsivo può invece essere bloccato più volte
- Come evitare il deadlock?
 - Acquisire i mutex sempre nello stesso “ordine”: **fixed locking strategy**
 - Ma non è sempre possibile! Può essere necessario utilizzare algoritmi specifici e pthread_mutex_trylock(): **try and back off**

Thread 1

```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);
    /* processing */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
```

Thread 2

```
for (; ; ) {
    pthread_mutex_lock(&m2);
    if (pthread_mutex_trylock(&m1)==0)
        break;           /* got it */
    pthread_mutex_unlock(&m2);
    /* didn't get it */
}
```

Valeria Cardellini - SDCC 2016/17

54

Lock lettore/scrittore

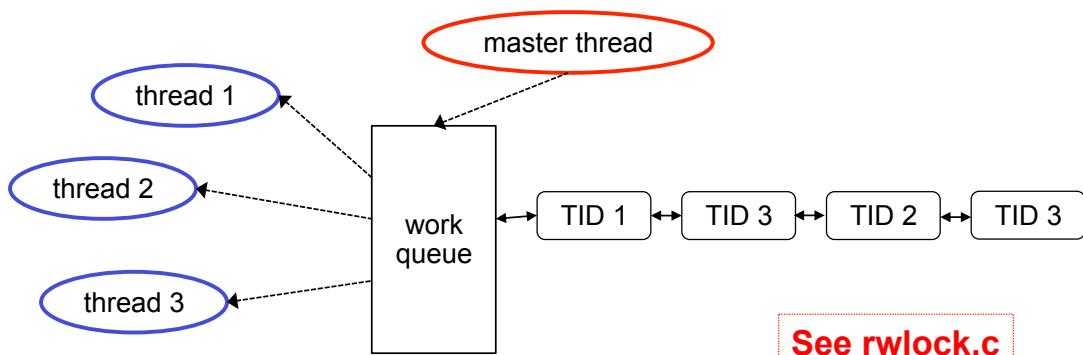
- Simile al mutex, ma consente un *maggior grado di parallelismo*
 - Operazioni non in conflitto eseguite in modo concorrente
- Mutex
 - 2 stati: locked, unlocked
 - Solo un thread alla volta può acquisire il mutex
- Lock lettore/scrittore
 - 3 stati: **read-locked**, **write-locked**, **unlocked**
 - **Solo un thread** alla volta può acquisire il lock lettore/scrittore in modalità di **scrittura**
 - **Molteplici thread** possono acquisire **contemporaneamente** il lock lettore/scrittore in modalità di **lettura**
 - Se un lock lettore/scrittore è write-locked, i thread che cercano di acquisire il lock si bloccano
 - Se un lock lettore/scrittore è read-locked, i thread che cercano di acquisire il lock in lettura riescono, mentre i thread che cercano di acquisire il lock in scrittura si bloccano
 - **Problema lettori/scrittori con preferenza ai lettori**
- Adatti per gestire strutture dati prevalentemente lette piuttosto che modificate

Lock lettore/scrittore (2)

- E' una variabile di tipo `pthread_rwlock_t`
- Per inizializzare il lock lettore/scrittore
 - Staticamente: `pthread_rwlock_t mymutex = PTHREAD_RWLOCK_INITIALIZER;`
 - Dinamicamente: funzione `pthread_rwlock_init()`
- Per bloccare il lock lettore/scrittore in modalità di lettura
 - Funzione `pthread_rwlock_rdlock()`
- Per bloccare il lock lettore/scrittore in modalità di scrittura
 - Funzione `pthread_rwlock_wrlock()`
- Per sbloccare il lock lettore/scrittore
 - Funzione `pthread_rwlock_unlock()`
- Per distruggere il lock lettore/scrittore
 - Funzione `pthread_rwlock_destroy()`

Example: reader–writer lock

- A queue of job requests is protected by a single reader–writer lock
 - Multiple worker threads obtain jobs assigned to them by a single master thread
 - Master thread adds a job to the queue
 - Worker threads search the queue concurrently for jobs assigned to them (through thread ID) and take only those jobs that match their thread ID off the queue



Example: reader–writer lock

- See the code:
 - Lock the queue’s reader-writer lock in write mode to add a job to the queue or remove a job from the queue
 - Lock the queue’s reader-writer lock in read mode to search the queue concurrently
- Reader–writer lock improves performance if threads search the queue much more frequently than they add or remove jobs

Variabile condition

- Meccanismo di sincronizzazione su cui un thread si può bloccare in attesa
 - Associata ad una condizione logica arbitraria (predicato)
 - Evita il polling ([busy-waiting](#)) in attesa del verificarsi di una condizione: invece di while ($x \neq y$)
 1. Lock su un mutex
 2. Valuta la condizione ($x \neq y$)
 3. Se falsa, l’attesa termina ed unlock (esplicito) sul mutex
 4. Se vera, thread sospeso in attesa del verificarsi della condizione ed unlock (implicito) sul mutex
- Tipo di dato della variabile condition:
`pthread_cond_t`
- Tipo di dato degli attributi della variabile condition:
`pthread_condattr_t`

Variabile condition: inizializzazione

- Per inizializzare/distruggere una variabile di condizione
- Inizializzazione
 - **Statica**: contestuale alla dichiarazione

```
pthread_cond_t mycond = PTHREAD_COND_INITIALIZER;
```
 - **Dinamica**: attraverso la funzione `pthread_cond_init()`

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_condattr_t *attr);  
  
int pthread_cond_destroy(pthread_cond_t *cond);
```

Variabile condition: sincronizzazione

- Alla variabile condition è **sempre associato un mutex che la protegge**
 - Per valutare il predicato, il thread deve prima bloccare il **mutex associato**
 - Se il predicato è verificato: il thread esegue le sue operazioni e **rilascia esplicitamente** il mutex associato
 - Se il predicato non è verificato: **in modo automatico e atomico**
 - il mutex associato viene **rilasciato implicitamente** e
 - il thread si blocca sulla variabile condition
 - Un thread bloccato **riacquisisce in modo automatico e atomico** il mutex associato nel momento in cui viene **svegliato** da un altro thread; il mutex deve essere esplicitamente rilasciato dal thread che sveglia
 - Un thread **deve bloccare il mutex associato prima** di **cambiare** lo stato della condizione; il mutex deve essere esplicitamente rilasciato **dopo** aver risvegliato uno o più thread

Variabile condition: wait

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

- Parametri della funzione
 - cond: puntatore all'oggetto condizione su cui bloccarsi
 - mutex: puntatore al mutex che protegge la condizione
- Valore di ritorno
 - 0 in caso di successo, diverso da 0 altrimenti
- Prima di invocare la funzione, il mutex associato alla condizione deve essere acquisito dal thread
- Se la condizione è falsa, la funzione
 - aggiunge il thread corrente all'elenco dei thread in attesa di quella condizione, sospende il thread e rilascia il mutex
- Ritorna quando la condizione è vera
 - Quando ritorna, il mutex è nuovamente bloccato

Variabile condition: wait (2)

while (x != y); 

```
pthread_mutex_lock(&m);  
while (x != y)  
    pthread_cond_wait(&cv, &m);  
/* modify x or y if necessary */  
pthread_mutex_unlock(&m);
```

- Quando la wait ritorna, il thread **deve verificare nuovamente la condizione**
 - Per diversi motivi: wakeup intercettate, uso di predicati laschi (anziché forti), wakeup spurie
 - Conseguenza: inserire sempre la wait **all'interno di un ciclo while** per verificare la condizione di attesa al ritorno della wait

Variabile condition: wait (3)

- Anche versione temporizzata di wait per attendere il verificarsi della condizione
 - Da specificare tempo assoluto ($t_{now} + x$) anziché relativo (x)
 - Trascorso il timeout senza che si sia verificata la condizione, la funzione ritorna con il codice d'errore ETIMEDOUT

```
int pthread_cond_timedwait(pthread_cond_t *cond,  
                           pthread_mutex_t *mutex, struct timespec *timeout);
```

Variabile condition: signal e broadcast

- Due varianti per notificare la modifica dello stato della condizione
 - **signal**: risveglia un solo thread in attesa sulla variabile condition
 - Non viene specificato quale thread viene risvegliato se vi sono più thread in attesa
 - Se non vi sono thread sospesi sulla condition, la signal non ha effetto
 - **broadcast**: risveglia tutti i thread in attesa sulla variabile condition, che contendono sul mutex associato alla variabile condition

```
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Parametro di ingresso
 - cond: puntatore alla variabile condition
- Valore di ritorno
 - 0 in caso di successo, diverso da 0 altrimenti

Variabile condition: signal e broadcast (2)

- Attenzione: occorre chiamare signal solo dopo aver modificato la condizione
- Occorre bloccare il mutex associato prima di modificare la condizione
- Esempio (sbagliato!)

Thread A

```
pthread_mutex_lock(&mutex);
while (condition == FALSE)
    pthread_cond_wait(&cond, &mutex);
pthread_mutex_unlock(&mutex);
```

Thread B

```
condition = TRUE;           Perché è sbagliato?
pthread_cond_signal(&cond);
```

Variabile condition: signal e broadcast (3)

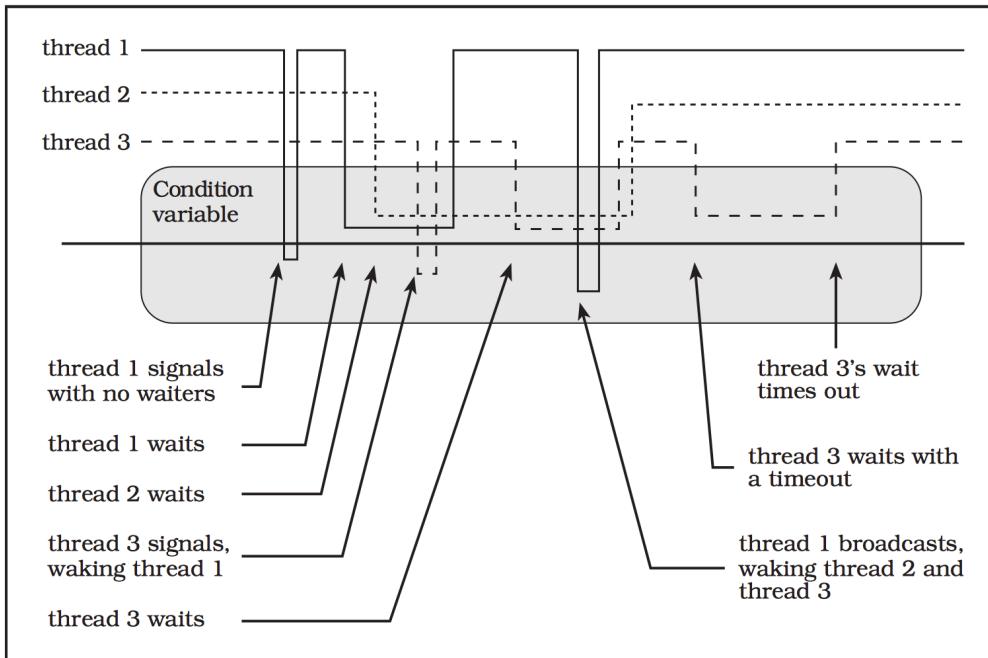
- Per non far perdere al thread A la notifica, la soluzione corretta è:

Thread B

```
pthread_mutex_lock(&mutex);
condition = TRUE;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

Variable condition: timing diagram

- Example: 3 threads interacting with a condition variable



Common *pitfalls* with condition variables

- Fail to lock mutex before calling `pthread_cond_wait`
- Wait on a mutex you didn't lock
- Fail to lock mutex before changing the predicate
- Assume the predicate is always true on wakeup and thus test the predicate only with an if statement

```
if (! condition) pthread_cond_wait()  
rather than  
while (! condition) pthread_cond_wait()
```
- When in doubt between signal and broadcast, use signal
- Signal before wait
- Forget to unlock the mutex

Esempio: produttore-consumatore

```
#include <pthread.h>
struct msg {
    struct msg *m_next;
    /* ... more stuff here ... */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void process_msg(void)
{
    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);
        /* now process the message mp */
    }
}
```

Valeria Cardellini - SDCC 2016/17

[vedere prodcons_apue2.c](#)

70

Esempio: produttore-consumatore (2)

```
void enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_cond_signal(&qready);
    pthread_mutex_unlock(&qlock);
}
```

Possiamo invertire l'ordine di signal e unlock?

Se sì, con quale vantaggio?

Con broadcast quale è l'ordine corretto rispetto ad unlock?

Valeria Cardellini - SDCC 2016/17

71

Signal/broadcast and unlock order

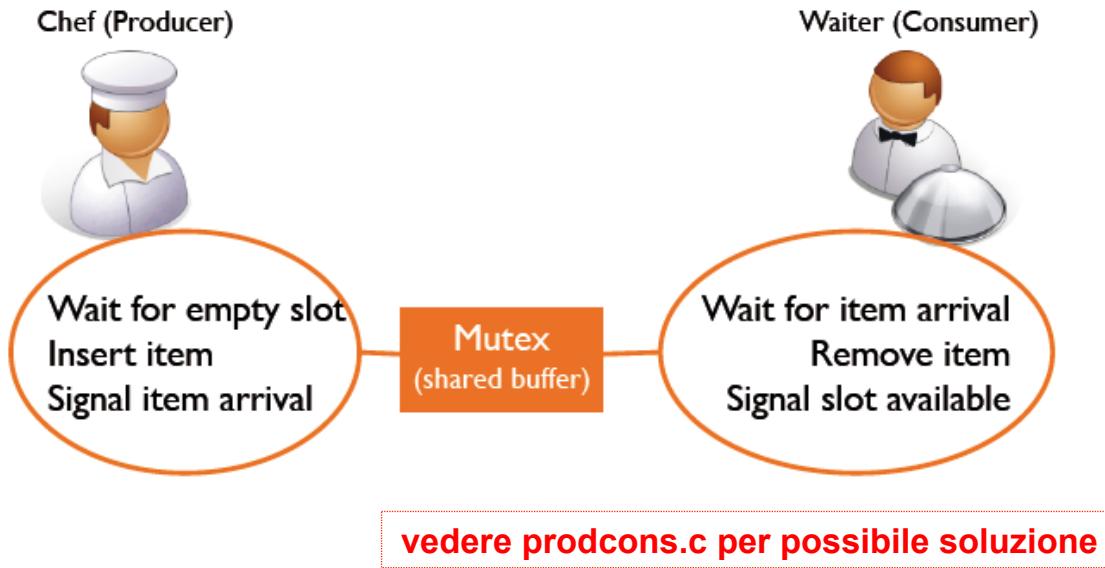
- For simplicity, hold the lock when calling signal or broadcast
 - In some cases (see previous slide) it is ok if you don't hold the lock when signaling
 - Consider to move the unlock before the signal/broadcast *if and only if* there is substantial performance improvement

Esercizio: produttore-consumatore con buffer limitato

- Considerare due tipi di thread (produttori e consumatori) che producono o consumano elementi (di tipo intero) posti in un buffer circolare di dimensioni finite
 - Un consumatore può prelevare dal buffer solo se il buffer non è vuoto
 - Un produttore può inserire un elemento nel buffer solo se il buffer non è pieno
- Scrivere il codice per la gestione del buffer usando pthreads e verificarne il corretto funzionamento

Esercizio: produttore-consumatore con buffer limitato

- Di cosa abbiamo bisogno?



Valeria Cardellini - SDCC 2016/17

74

Signal or broadcast?

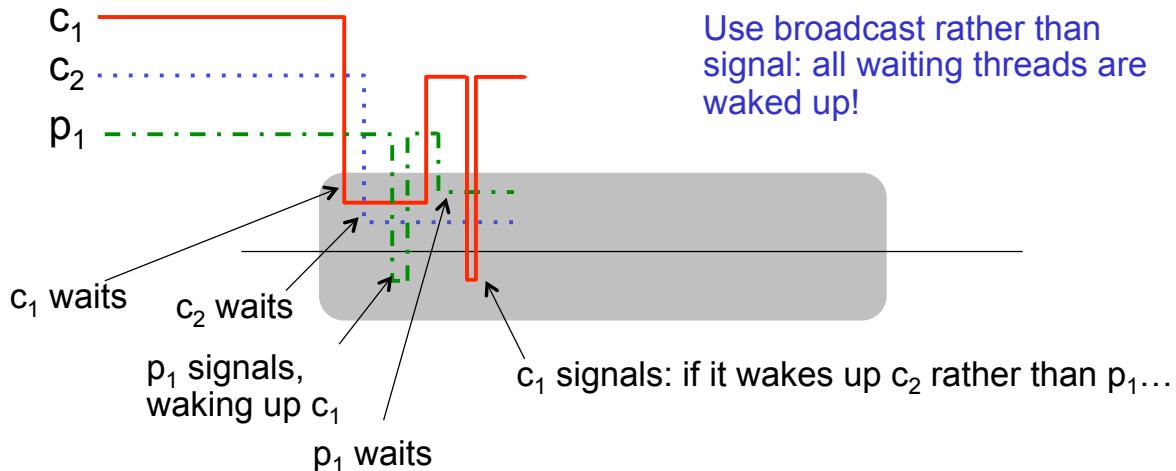
- The predicate condition is **one-to-one** with the condition variable: use **signal**
 - Signal may be more efficient than broadcast
 - Example: see prodcos.c
- **More than one predicate condition** associated with the same condition variable: use **broadcast** instead of signal
 - With signal a thread waiting for the wrong condition might be woken up and that thread would go back to sleep without a thread of the correct type getting signaled
 - Example: producer-consumer problem with bounded buffer where the same condition variable is used for the buffer state (which could be either not empty or not full): see next slide
- When in **doubt** between signal and broadcast, use **broadcast**

Valeria Cardellini - SDCC 2016/17

75

Signal or broadcast? (2)

- One condition variable for the buffer state and signal
- Bounded buffer with single element and initially empty
- 2 consumers, 1 producer: c_1, c_2, p_1



Safety and liveness

- Desirable properties for concurrent systems
 - **Safety**: bad things don't happen
 - **Liveness**: good things (eventually) happen
- **Mutual exclusion** is primarily about safety
 - Want to ensure two threads don't “collide” in terms of accessing shared data
- ...but may have consequences for liveness too!
 - i.e., must ensure our program doesn't get stuck

Liveness properties

- From a theoretical viewpoint, they must ensure that we eventually make progress, i.e., want to avoid
 - **Deadlock**: threads sleep waiting for each other
 - **Livelock**: threads execute but make no progress, i.e. do nothing useful
- Practically speaking, we also want good performance
 - **No starvation**: single thread must make progress
 - More generally, we may aim for **fairness**
 - **Minimality**: no unnecessary waiting or signalling
- The properties are often at odds with safety

Esempi di server TCP multi-threaded

- Analizziamo 3 versioni di un server TCP multi-threaded
 - A. Un thread per client, accept nel main thread
 - B. Pre-threading (pool statico), accept nei worker thread con mutex
 - C. Pre-threading (pool statico), accept nel main thread con condition variable e relativo mutex
- Per il codice completo, vedere il sito del corso
 - Applicazione client multiprocess (codice in client.c) per generare richieste di dimensione specificata in input
 - Applicazione server con tempi di esecuzione
 - Gestione del segnale SIGINT per catturare la terminazione del server e stampare il tempo di CPU (system e user) del server

Esempio server A

```
#include <signal.h>
#include <pthread.h>
#include "basic.h"

int main(int argc, char **argv)
{
    int                listensd, connsd, n;
    void               sig_int(int);
    void               *doit(void *);
    pthread_t          tid;
    struct sockaddr_in servaddr;

    if (argc != 1) {
        fprintf(stderr, "usage: serverthr_basic\n");
        exit(1); }

    if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "socket error");
        exit(1); }

    memset((void *)&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);
```

Valeria Cardellini - SDCC 2016/17

80

Esempio server A (2)

```
if ((bind(listensd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0) {
    fprintf(stderr, "bind error");
    exit(1); }

if (listen(listensd, BACKLOG) < 0 ) {
    fprintf(stderr, "listen error");
    exit(1); }

if (signal(SIGINT, sig_int) == SIG_ERR) {
    fprintf(stderr, "signal error");
    exit(1); }

for ( ; ; ) {
    if ((connsd = accept(listensd, NULL, NULL)) < 0) {
        fprintf(stderr, "accept error");
        exit(1); }

    if ((n = pthread_create(&tid, NULL, &doit, (void *)connsd)) != 0) {
        errno = n;
        fprintf(stderr, "pthread_create error");
        exit(1); }
}

}
```

Valeria Cardellini - SDCC 2016/17

81

Esempio server A (3)

```
void *doit(void *arg)
{
    void web_child(int);
    int n;

    if ((n = pthread_detach(pthread_self())) != 0) {
        errno = 0;
        fprintf(stderr, "pthread_detach error");
        exit(1);
    }
    web_child((int) arg); /* process client request */
    if (close((int) arg) == -1) {
        fprintf(stderr, "close error");
        exit(1);
    }
    return(NULL);
}
void sig_int(int signo)
{
    void pr_cpu_time(void);
    pr_cpu_time();
    exit(0);
}
```

Esempio server B

```
typedef struct {
    pthread_t      thread_tid;      /* thread ID */
    long           thread_count;    /* # connections handled */
} Thread;
Thread *tptr; /* array of Thread structures; calloc'ed */
int      listensd, nthreads;
socklen_t addrlen;
pthread_mutex_t mlock;
```

Esempio server B (2)

```
#include <signal.h>
#include <pthread.h>
#include "basic.h"
#include "serverthr_pre_1.h"

pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char **argv)
{
    int i;
    void sig_int(int), thread_make(int);
    struct sockaddr_in servaddr;

    if (argc != 2) {
        fprintf(stderr, "usage: serverthr_pre_1 <#threads>\n");
        exit(1); }
    nthreads = atoi(argv[1]);
    if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "socket error");
        exit(1); }
```

Esempio server B (3)

```
memset((void *)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

if ((bind(listensd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0) {
    fprintf(stderr, "bind error");
    exit(1); }
if (listen(listensd, BACKLOG) < 0 ) {
    fprintf(stderr, "listen error");
    exit(1); }
tptr = (Thread *)calloc(nthreads, sizeof(Thread));
if (tptr == NULL) {
    fprintf(stderr, "calloc error");
    exit(1); }
for (i = 0; i < nthreads; i++)
    thread_make(i); /* only main thread returns */
if (signal(SIGINT, sig_int) == SIG_ERR) {
    fprintf(stderr, "signal error");
    exit(1); }

for ( ; ; )
    pause(); /* everything done by threads */
}
```

Esempio server B (4)

```
void sig_int(int signo)
{
    int i;
    void pr_cpu_time(void);

    pr_cpu_time();

    for (i = 0; i < nthreads; i++)
        printf("thread %d, %ld connections\n", i, tptr[i].thread_count);
    exit(0);
}
```

Esempio server B (5)

```
#include <pthread.h>
#include "basic.h"
#include "serverthr_pre_1.h"
void thread_make(int i)
{
    void *thread_main(void *);
    int n;
    if ((n = pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i)) != 0)
    {
        errno = n;
        fprintf(stderr, "pthread_create error");
        exit(1);
    }
    return; /* main thread returns */
}
```

Esempio server B (6)

```
void *thread_main(void *arg)
{
    int    connsd, n;
    void  web_child(int);

    printf("thread %d starting\n", (int) arg);
    for ( ; ; ) {
        if ((n = pthread_mutex_lock(&mlock)) != 0) {
            fprintf(stderr, "pthread_mutex_lock error");
            exit(1); }
        if ((connsd = accept(listensd, NULL, NULL)) < 0 ) {
            fprintf(stderr, "accept error");
            exit(1); }
        if ((n = pthread_mutex_unlock(&mlock)) != 0) {
            fprintf(stderr, "pthread_mutex_unlock error");
            exit(1); }
        tptr[(int) arg].thread_count++;
        web_child(connsd);           /* process client request */
        if (close(connsd) == -1) {
            fprintf(stderr, "close error");
            exit(1); }
    }
}
```

Valeria Cardellini - SDCC 2016/17

88

Esempio server C

```
typedef struct {
    pthread_t      thread_tid;      /* thread ID */
    long          thread_count;    /* # connections handled */
} Thread;
Thread  *tptr;                  /* array of Thread structures; calloc'ed */
#define MAXNCLI      32
int       clisd[MAXNCLI], igt, igt;
pthread_mutex_t clisd_mutex;
pthread_cond_t  clisd_cond;
```

Valeria Cardellini - SDCC 2016/17

89

Esempio server C (2)

```
#include <pthread.h>
#include <signal.h>
#include "basic.h"
#include "serverthr_pre_2.h"
static int nthreads;
pthread_mutex_t clsd_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t clsd_cond = PTHREAD_COND_INITIALIZER;
int main(int argc, char **argv)
{
    int i, listensd, connsd;
    void sig_int(int), thread_make(int);
    socklen_t addrlen, clilen;
    struct sockaddr *cliaddr;
    struct sockaddr_in servaddr;
    if (argc != 2) {
        fprintf(stderr, "usage: serverthr_pre_2 <#threads>\n");
        exit(1);
    }
    nthreads = atoi(argv[1]);
    if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "socket error");
        exit(1);
    }
    Valeria Cardellini - SDCC 2016/17 }
```

90

Esempio server C (3)

```
memset((void *)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
if ((bind(listensd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0) {
    fprintf(stderr, "bind error");
    exit(1);
}
if (listen(listensd, BACKLOG) < 0 ) {
    fprintf(stderr, "listen error");
    exit(1);
}
cliaddr = malloc(addrlen);
ptr = (Thread *)calloc(nthreads, sizeof(Thread));
if (ptr == NULL) {
    fprintf(stderr, "calloc error");
    exit(1);
}
iget = iput = 0;           /* create all the threads */
for (i = 0; i < nthreads; i++)
    thread_make(i);          /* only main thread returns */
if (signal(SIGINT, sig_int) == SIG_ERR) {
    fprintf(stderr, "signal error");
    exit(1);
}
```

Valeria Cardellini - SDCC 2016/17

91

Esempio server C (4)

```
for ( ; ; ) {
    clilen = addrlen;
    connsd = accept(listensd, cliaddr, &clilen);
    pthread_mutex_lock(&clisd_mutex);
    clisd[iput] = connsd;
    if (++iput == MAXNCLI)
        iput = 0;
    if (iput == igit) {
        printf("iinput = igit = %d", iput);
        exit(1);
    }
    pthread_cond_signal(&clisd_cond);
    pthread_mutex_unlock(&clisd_mutex);
}
void sig_int(int signo)
{
    int i;
    void pr_cpu_time(void);
    pr_cpu_time();
    for (i = 0; i < nthreads; i++)
        printf("thread %d, %ld connections\n", i, tptr[i].thread_count);
    exit(0);
}
```

Esempio server C (5)

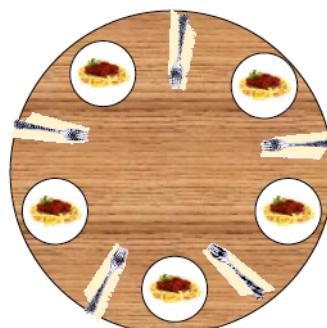
```
#include <pthread.h>
#include "basic.h"
#include "serverthr_pre_2.h"
void thread_make(int i)
{
    int n;
    void *thread_main(void *);
    if ((n = pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i)) != 0) {
        errno = n;
        fprintf(stderr, "pthread_create error");
        exit(1);
    }
    return; /* main thread returns */
}
void *thread_main(void *arg){
    int connsd, n;
    void web_child(int);
    printf("thread %d starting\n", (int) arg);
```

Esempio server C (6)

```
for ( ; ; ) {
    if ((n = pthread_mutex_lock(&clisd_mutex)) != 0) {
        fprintf(stderr, "pthread_mutex_lock error");
        exit(1);
    }
    while (iget == input)
        pthread_cond_wait(&clisd_cond, &clisd_mutex);
    connsd = clisd[iget];           /* connected socket to service */
    if (++iget == MAXNCLI)
        iget = 0;
    if ((n = pthread_mutex_unlock(&clisd_mutex)) != 0) {
        fprintf(stderr, "pthread_mutex_unlock error");
        exit(1);
    }
    tptr[(int) arg].thread_count++;
    web_child(connsd);            /* process client request */
    if (close(connsd) == -1) {
        fprintf(stderr, "close error");
        exit(1);
    }
}
```

Esercizio: filosofi a cena

- Un classico problema di sincronizzazione, formulato da Dijkstra nel 1965
- 5 filosofi sono seduti attorno a un tavolo circolare; ogni filosofo ha un piatto di spaghetti che necessitano di 2 forchette per poter essere mangiati; sul tavolo vi sono in totale 5 forchette
- Ogni filosofo ha un comportamento ripetitivo, che alterna due fasi:
 - una fase in cui pensa
 - l'altra in cui mangia



Esercizio: filosofi a cena (2)

- Osservazioni
 - I filosofi non possono mangiare tutti insieme: ci sono solo 5 forchette
 - 2 filosofi vicini non possono mangiare contemporaneamente perché condividono una forchetta
- Rappresentando ogni filosofo con un thread, realizzare una politica di sincronizzazione che eviti situazioni di deadlock e starvation
 - Ogni filosofo può mangiare quando ha fame e nessuno muore di fame

Filosofi a cena: una soluzione inefficiente

- Un solo filosofo alla volta può mangiare
 - A turno un solo filosofo alla volta è abilitato a prendere entrambe le forchette per mangiare
- Problema: inefficienza
 - Solo un filosofo alla volta può mangiare

Filosofi a cena: una soluzione sbagliata

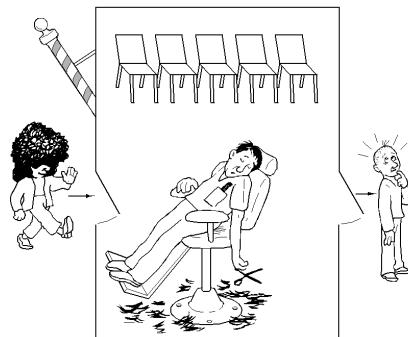
- Quando un filosofo ha fame:
 1. prende la forchetta a sinistra del piatto
 2. prende quella che a destra del suo piatto
 3. mangia
 4. mette sul tavolo le due forchette
- Problema: possibilità di deadlock
 - Se tutti i filosofi afferrano contemporaneamente la forchetta di sinistra, rimangono tutti in attesa di prendere la forchetta di destra (un evento che non si potrà mai verificare!)
 - È verificata la condizione di *hold-and-wait* del deadlock

Filosofi a cena: un'altra soluzione sbagliata

- Variante della soluzione precedente
- Quando un filosofo ha fame:
 1. prende la forchetta a sinistra del piatto
 2. verifica se quella a destra del suo piatto è disponibile
 3. se lo è, la prende e mangia
 4. altrimenti, posa la forchetta a sinistra, aspetta un certo tempo e poi ripete dal passo 1
- Problema: possibilità di starvation
 - Se tutti i filosofi afferrano contemporaneamente la forchetta di sinistra, si accorgono che la forchetta di destra non è disponibile, posano la forchetta di sinistra, aspettano, ...
- Come migliorare le soluzioni analizzate?

Esercizio: barbiere sonnolento

- In un negozio di barbiere ci sono
 - una sedia per lavorare
 - n sedie per far attendere i clienti
- Il barbiere di solito dorme
- Quando arriva un cliente, questi sveglia il barbiere e si fa servire
- Se nel frattempo arrivano altri clienti, si accomodano sulle sedie oppure vanno via se sono tutte occupate



Valeria Cardellini - SDCC 2016/17

100

Esercizio: barbiere sonnolento (2)

- Rappresentando il barbiere ed i clienti con i thread, realizzare una politica di sincronizzazione che eviti situazioni di deadlock e starvation

Valeria Cardellini - SDCC 2016/17

101

Exercise: reader-writer problem

- There are many readers and some occasional writers
- We need to allow only one writer at any time
- For performance however we want many readers performing the read at once, without waiting for another reader
- We want to avoid starvation of the writers if there is a constant stream of readers: in the solution we give writers preferential access to the lock
- In the code we shorten the names avoiding pthread_
 - E.g., mutex_lock for pthread_mutex_lock and cond_wait for pthread_cond_wait

Exercise: reader pseudo-code

```
reader() {
    mutex_lock(&m);
    while (writers) // readers that arrive after the writers arrived will have to wait
        cond_wait(&turn, &m);
    while (writing) // readers that arrive while there is an active writer will also wait
        cond_wait(&turn, &m);
    reading++;      // increase the number of active readers
    mutex_unlock(&m);
    // Read here
    mutex_lock(&m);
    reading--;
    cond_broadcast(&turn); // so that all threads wake up and check their condition
    mutex_unlock(&m);
}
```

Exercise: writer pseudo-code

```
writer() {  
    mutex_lock(&m);  
    writers++;      // increase the number of arrived writers  
    while (reading || writing) // writers must wait for everyone  
        cond_wait(&turn, &m);  
    writing++;      // active writer  
    mutex_unlock(&m);  
    // Write here  
    mutex_lock(&m);  
    writing--;  
    writers--;  
    cond_broadcast(&turn);  
    mutex_unlock(&m);  
}  
Can you improve the code?  
How can you only wake up readers or one writer?
```