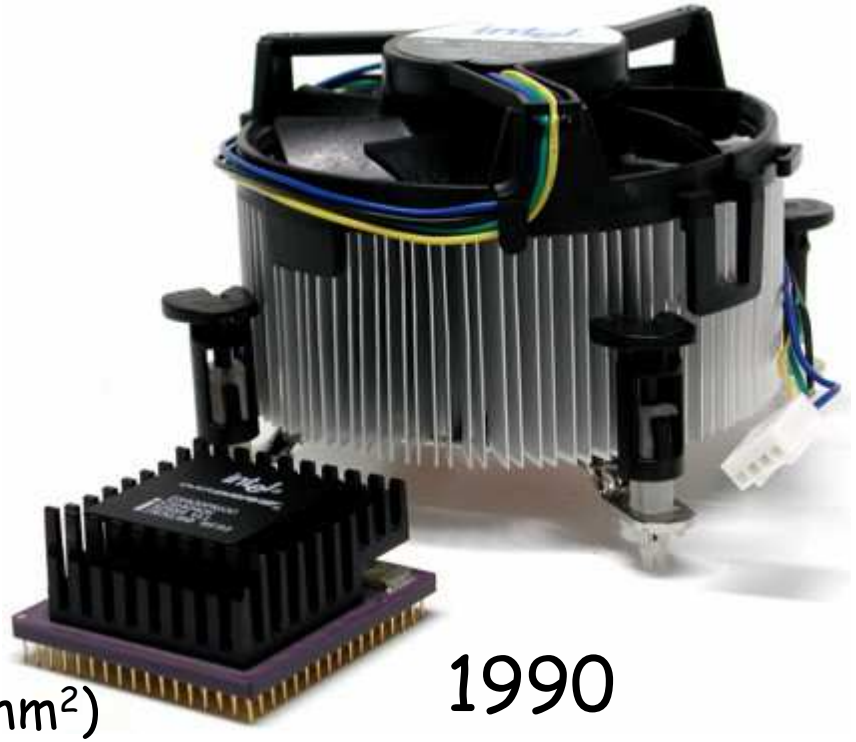
Perche' le CPU multicore

I processi industriali producono chip sempre piu' densi in termini di transistor

Inoltre:
Watt ~ Volt^2 x frequenza
Frequenza ~ Volt

Watt ~ frequenza^3



2000

1990

Il rapporto Watt/mm^2 (calore per mm^2) diventa troppo grande per raffreddare i chip economicamente

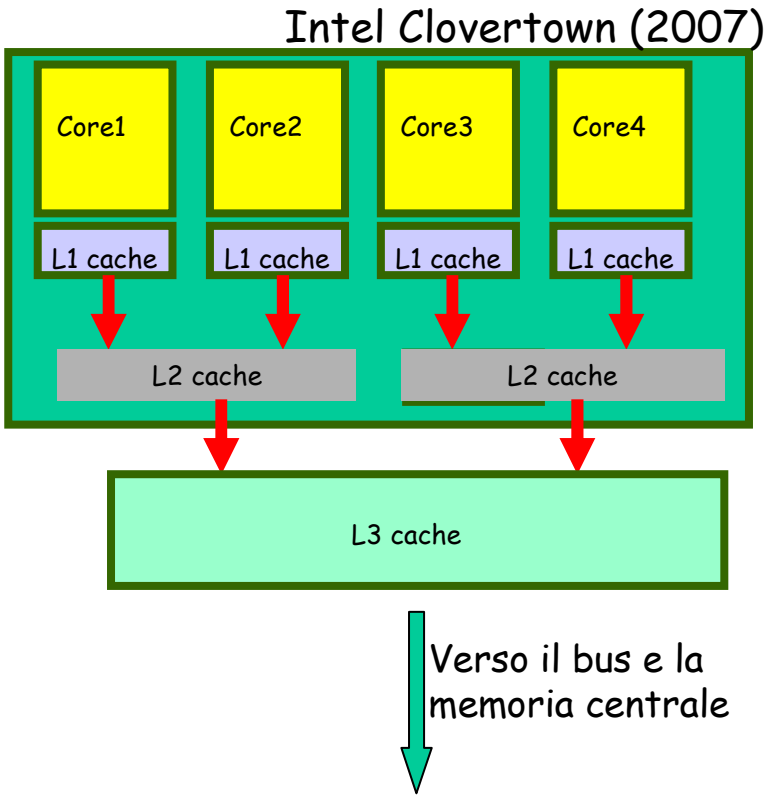
Perche' le CPU multicore

	core	freq.	perf.	potenza
CPU1	1	1	1	1
CPU2	1	1.5	1.5	3.3
CPU3	2	0.75	1.5	0.8

CPU3 con 2 core e una frequenza ridotta del 25%, ha le stesse prestazioni di CPU2 con 1/3 della potenza (e del calore dissipato)

Una CPU multicore

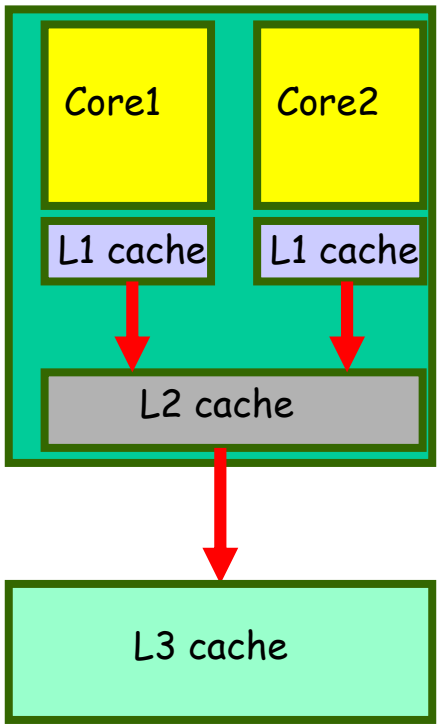
- Una CPU multicore e' un insieme di unita' processanti autonome in un unico chip, che condividono risorse (cache, bus, e memoria centrale
- I core possono condividere la cache o avere cache proprie
- Oggi si hanno al piu' 6-8 core



E' un esempio di calcolatore parallelo a memoria condivisa

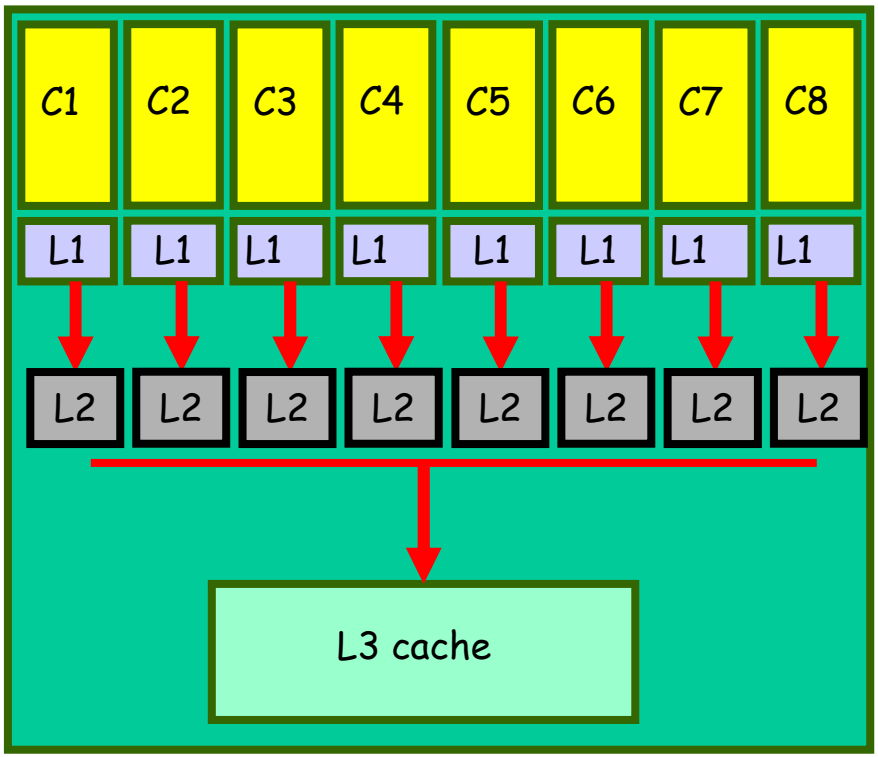
Architetture multicore: IBM

IBM Power 5 (2004)



Core = 2
Cache L1 = 2x32KB
Cache L2 = 1875 MB
Cache L3 = 36 MB
PP ~ 15.2 Gflops (1.9 Ghz)

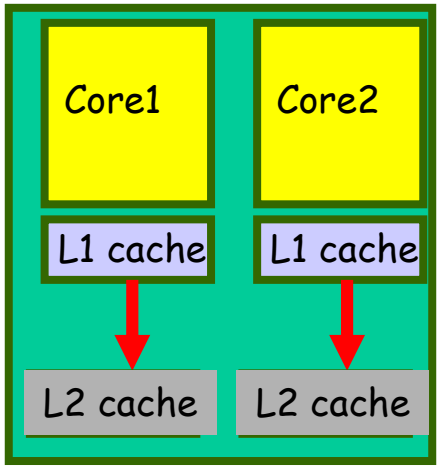
IBM Power 7 (2010)



Core = 4, 6, 8
Cache L1 = 8x32KB
Cache L2 = 8x256 MB
Cache L3 = 32 MB
PP ~ 512 GFlops (4 Ghz)

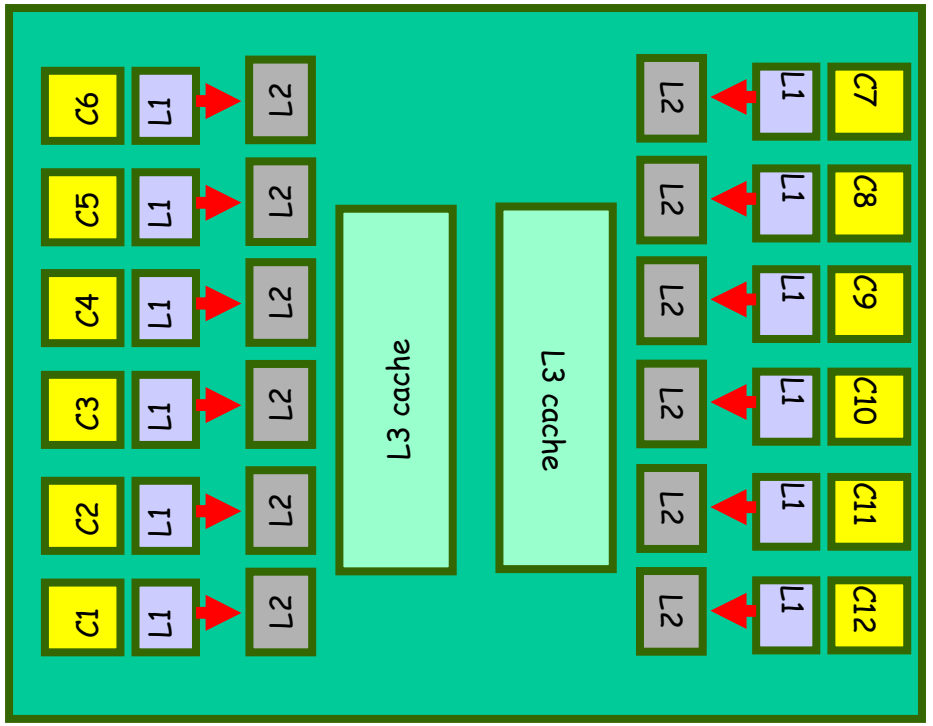
Architetture multicore: AMD

Opteron (2005)



Core = 2
Cache L1 = 2x32KB
Cache L2 = 2x1 MB
PP ~ 22 Gflops (2.8 Ghz)

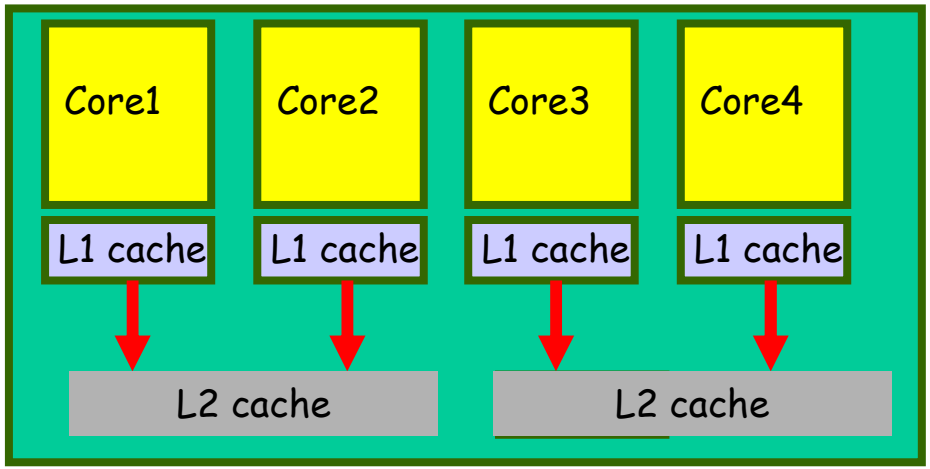
Opteron (2010)



Core = 12
Cache L1 = 12x32KB
Cache L2 = 12x1 MB
Cahce L3 = 2x6 MB
PP ~ 192 Gflops (2.6 GHz)

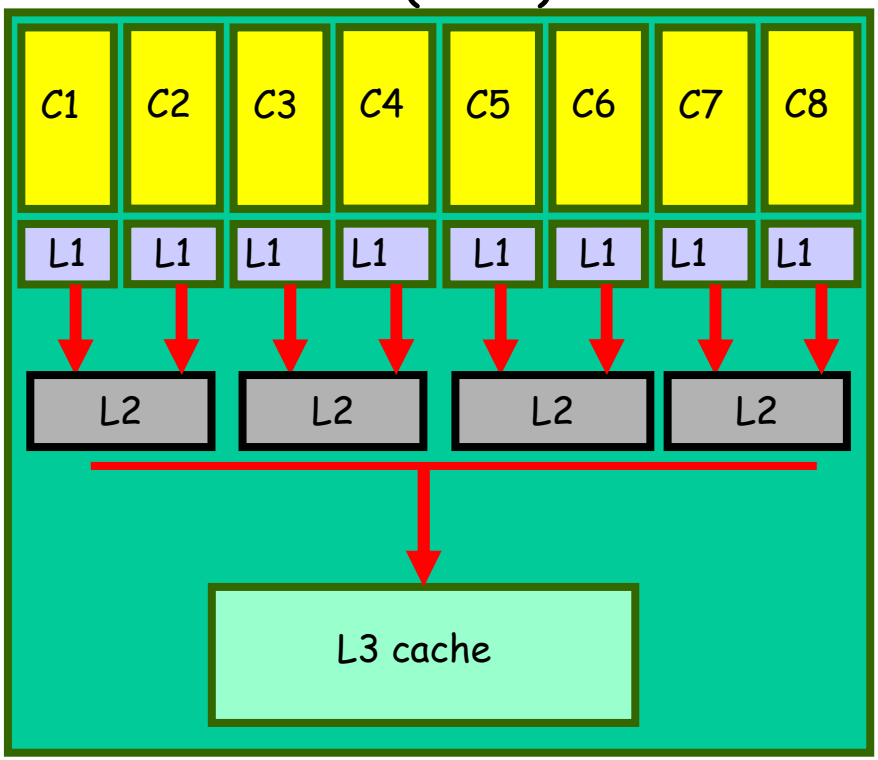
Architetture multicore: Intel

Clovertown (2007)



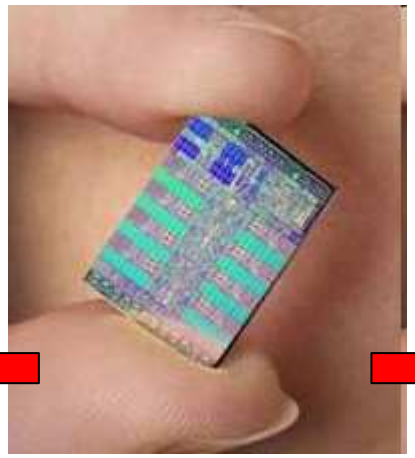
Core = 4
Cache L1 = 4x364KB
Cache L2 = 2x4 MB
PP ~ 48 GFlops (3 GHz)

Beckton (2011)



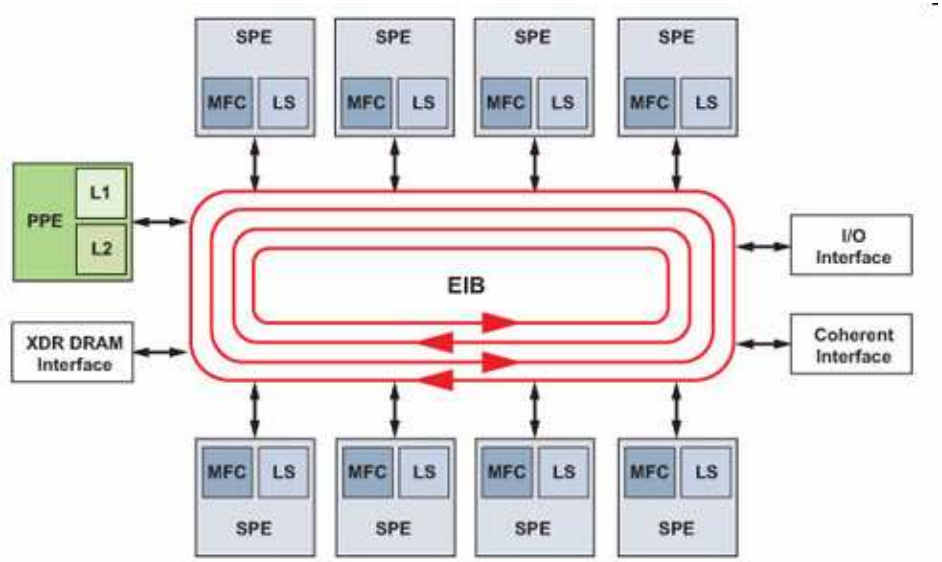
Core = 8
Cache L1 = 8x644KB
Cache L2 = 4x3 MB
Cache L3 = 1MB
PP ~ 304 Gflops (3 GHz)

Architettura multicore: Cell



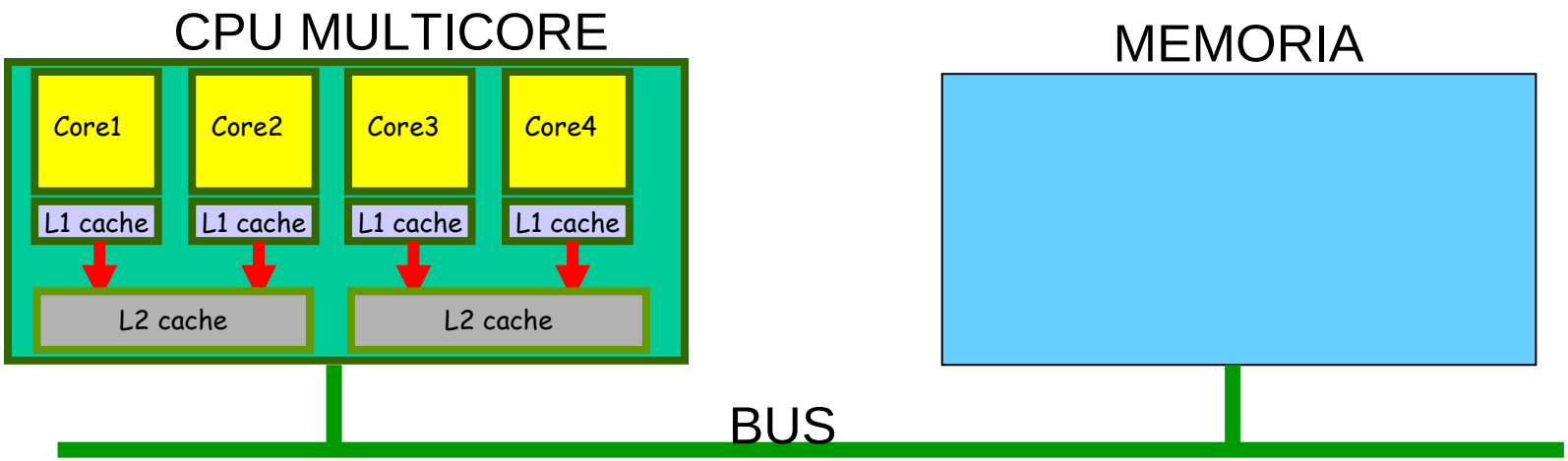
PS3 @ myhome

Roadrunner @ LosAlamos



Core = 8
PP ~ 25 Gflops (DP) –
256 Gflops (SP)

Sviluppo di applicazioni per CPU multicore



Tutti i core accedono alla stessa memoria principale (spesso alla stessa cache)

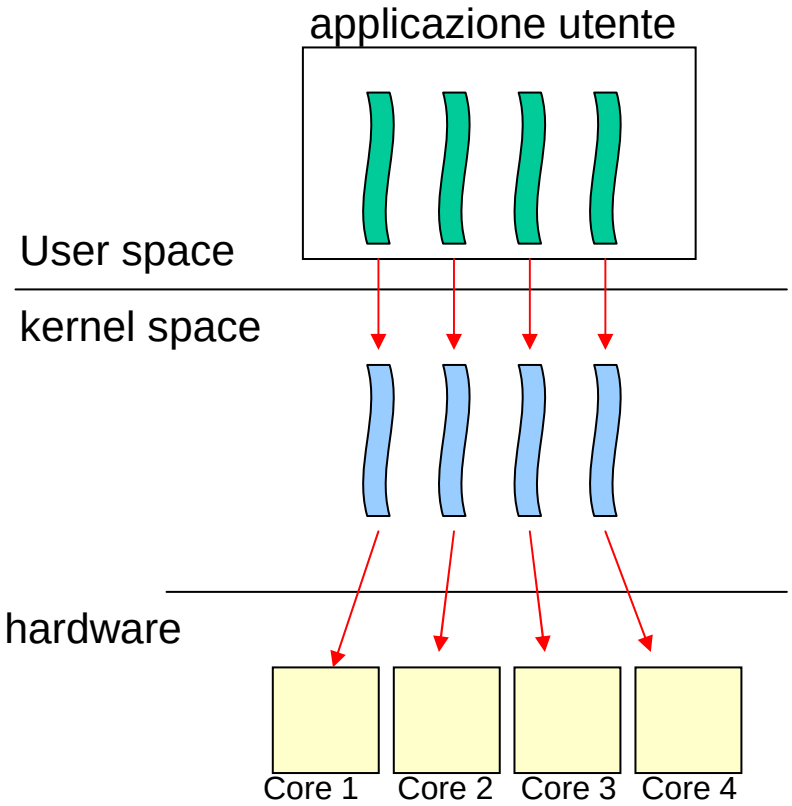
modello a **memoria condivisa**

Lo strumento naturale per lo sviluppo di applicazioni
per CPU multicore sono
i threads

Definizione di Threads

- Nei moderni S.O. un ***thread*** (o *lightweight process*, **LWP**) è spesso l'unità di base di utilizzo della CPU e consiste di:
  - Program counter
  - Insieme dei registri
  - Spazio dello stack
- Un thread condivide con i thread ad esso associati:
  - Spazio di indirizzamento (non c'e' protezione !!)
  - Dati globali
  - File aperti
  - Gestori dei segnali

Implementazione dei thread



Nei moderni S.O. i threads sono implementati secondo il modello 1-1 (Kernel level threads)

Il S.O. schedula threads indipendenti su differenti core della CPU

Threads POSIX (Pthreads)

- Uno standard POSIX (IEEE 1003.1c) per la creazione e sincronizzazione dei threads
- Definizione delle API
- Threads conformi allo standard POSIX sono chiamati Pthreads
- Lo standard POSIX stabilisce che registri dei processori, stack e signal mask sono individuali per ogni thread
- Lo standard specifica come il sistema operativo dovrebbe gestire i segnali ai Pthtrads i specifica differenti metodi di cancellazione (asincrona, ritardata, …)
- Permette di definire politiche dischedulìng e priorita'
- Alla base di numerose librerie di supporto per vari sistemi

Creazione di un thread

```
#include <pthread.h>
int pthread_create( pthread_t  *tid,
                    const pthread_attr_t  *attr,
                    void *(*func)(void*),
                    void *arg );
```

tid: puntatore all'identificativo del thread ritornato dalla funzione

attr: attributi del thread (priorita', dimensione stack, …) . A meno di esigenze particolari si usa NULL

func: funzione di tipo **void *** che costituisce il corpo del thread

arg : unico argomento di tipo **void *** passato a **func**

La funzione ritorna subito dopo creato il thread

Attesa per la fine di un thread

```
#include <pthread.h>

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

tid: identificativo del thread di cui attendere la fine

status: puntatore al valore di ritorno del thread. Di solito NULL

Processi vs Pthread

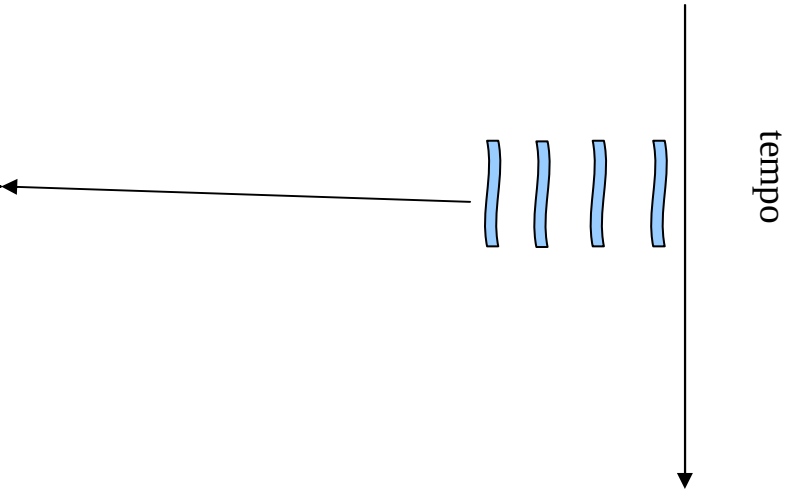
	Processi	Threads
creazione	**fork+exec**	**pthread_create**
attesa	**wait**	**pthread_join**

Altre funzioni

```
int    pthread_detach(pthread_t   thread_id);
int    pthread_exit(void   *value_ptr);
pthread_t   pthread_self();
```

Esempio 4 thread su CPU 4core

```
#include "c_timer.h"
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
void *thread(void *) ;

int main( ){                              // inizio main
      pthread_t    tid1, tid2, tid3, tid4;

      pthread_create(&tid1, NULL, thread, NULL);
      pthread_create(&tid2, NULL, thread, NULL);
      pthread_create(&tid3, NULL, thread, NULL);
      pthread_create(&tid4, NULL, thread, NULL);

      pthread_join(tid1, NULL);
      pthread_join(tid2, NULL);
      pthread_join(tid3, NULL);
      pthread_join(tid4, NULL);

}  //fine main

void *thread (void *arg){                 // inizio thread function
int   i;   float sum;   double btime, etime;

      btime = get_cur_time();
      sum=0;
      for (i=0;  i<10000000  ; i++){
            sum=sum+a;
            a = -a;
      }

      etime = get_cur_time();
      printf("ELAPSED = %f\n", etime-btime);
}                                         // fine thread function
```



Ogni thread somma 10000000 numeri

N.B. compilare il tutto linkando la libreria libpthread.a

Esecuzione su CPU 4 core

1 thread

ELAPSED = 0.069968
tempo totale = 0.070168

2 threads

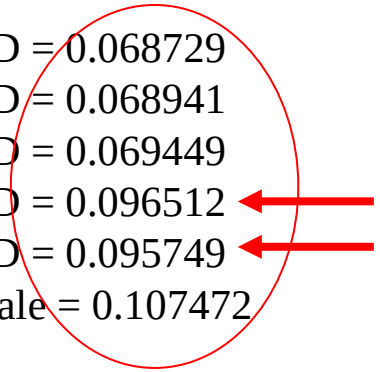
ELAPSED = 0.067141
ELAPSED = 0.069108
tempo totale = 0.069311

4 threads

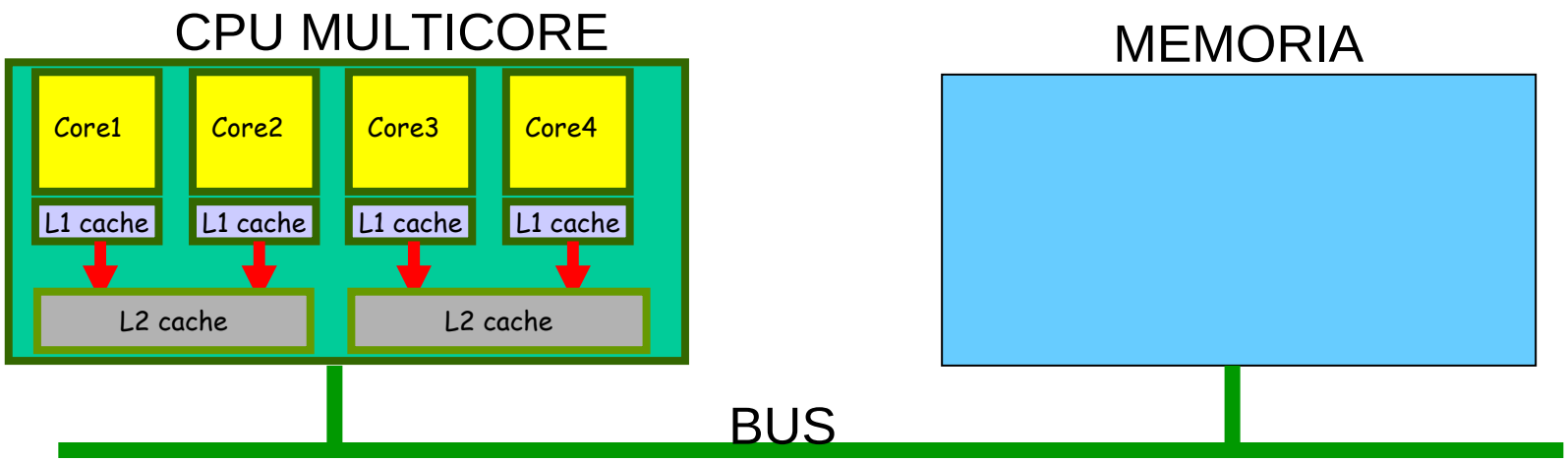
ELAPSED = 0.066929
ELAPSED = 0.066822
ELAPSED = 0.068158
ELAPSED = 0.068875
tempo totale = 0.069170

5 threads

ELAPSED = 0.068729
ELAPSED = 0.068941
ELAPSED = 0.069449
ELAPSED = 0.096512
ELAPSED = 0.095749
tempo totale = 0.107472



Sincronizzazione dei threads



- Il principale problema nell'uso dei threads e' il rischio di corsa critica sui dati condivisi

sincronizzare gli accessi alla memoria condivisa

Uso di semafori e/o altri strumenti

Semafori

- I semafori sono strumenti di sincronizzazione che superano il problema dell'attesa attiva
- Il semaforo contiene una variabile intera S che serve per proteggere l'accesso alle sezioni critiche
- Si può accedere al semaforo (alla variabile) solo attraverso due operazioni atomiche
  - *wait(S):*
  - *signal(S):*

definizione di wait e signal

wait (S)

- se S>0 allora S=S-1 e il processo continua l'esecuzione
- altrimenti (S=0) il processo si blocca

signal(S)

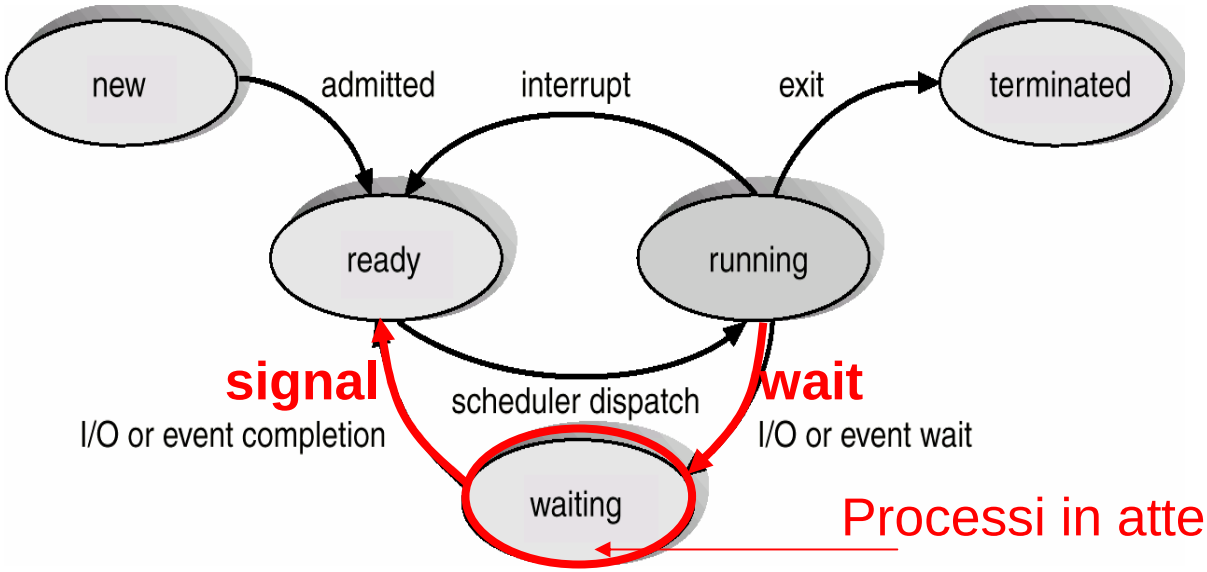
- esegue S=S+1

altri nomi di wait e signal

- wait ( ) == down ( ) == P ( )
- signal ( )== up ( ) == V ( )

Edsger W. Dijkstra

Implementazione dei semafori



Un semaforo permette al S.O. di rimuovere un processo dalla ready-list, e quindi dalla competizione per l'uso della CPU, risolvendo il problema della attesa attiva

tipi di semafori

- Semaforo ***contatore*** — intero che può assumere valori in un dominio non limitato.
- Semaforo ***binario*** (mutex) — intero che può essere posto solo a 0 o 1; può essere implementato più semplicemente.

- Molto dipende dal S.O. (windows ha 5 tipi di semafori)

Uso dei semafori (1)

- Sezione critica con n processi
- Variabili condivise tra gli n processi:
  semaphore mutex; // inizialmente mutex = 1

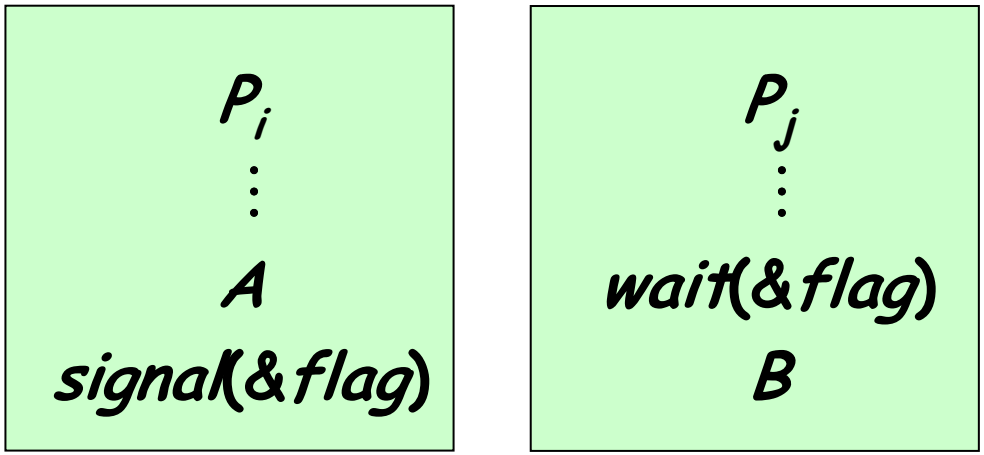
- Processo P_i:

```
do {
        sezione non critica
        wait( & mutex);
                sezione critica
        signal(& mutex);
        sezione non critica
} while (1);
```

Uso dei semafori (2)

- I semafori possono anche essere usati per sincronizzare le operazioni di due o piu' processi

- Ad esempio: si vuole eseguire *B* in P_j solo dopo che *A* è stato eseguito in P_i
- Si impiega un semaforo *flag* inizializzato 0
- Codice:



Thread Posix e semafori

I principali meccanismi di sincronizzazione previsti dallo standard POSIX 1003.1c sono:

- **I mutex (per le sezioni critiche)**
- **le variabili condizione (per la sincronizzazione)**

Mediante tali strumenti è possibile realizzare meccanismi di accesso alle risorse

I mutex

Un mutex è un semaforo binario (0 oppure 1)

Un mutex è mantenuto in una struttura
pthread_mutex_t

Tale struttura va allocata e inizializzata

Per inizializzare:
- se la struttura è allocata staticamente:
pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER

- se la struttura è allocata dinamicamente (e.g. se si usa malloc): chiamare pthread_mutex_init

I mutex

```
#include <pthread.h>

int pthread_mutex_lock( pthread_mutex_t   mutex)
```

mutex: semaforo binario

Implementazione della funzione wait

I mutex

```
#include <pthread.h>

int pthread_mutex_unlock( pthread_mutex_t   mutex)
```

mutex: semaforo binario

Implementazione della funzione signal

Esempio : somma con 4 core

```
.......
pthread_mutex_t m1=PTHREAD_MUTEX_INITIALIZER;
void *thread(void *) ;
int somma = 0;

int main( ){                          // inizio main
      pthread_t    tid1, tid2, tid3, tid4;
      pthread_create(&tid1, NULL, thread, NULL);
      pthread_create(&tid2, NULL, thread, NULL);
      pthread_create(&tid3, NULL, thread, NULL);
      pthread_create(&tid4, NULL, thread, NULL);
      pthread_join(tid1, NULL);
      pthread_join(tid2, NULL);
      pthread_join(tid3, NULL);
      pthread_join(tid4, NULL);
}  //fine main

void *thread (void *arg){              // inizio thread function
int   i;   float sum;   double btime, etime, a=10;

      btime = get_cur_time();
      sum=0;
      for (i=0;  i<10000000  ; i++){
            sum=sum+a;
            a = -a;
      }
      pthread_mutex_lock(&m1);
      somma=somma+sum;
      pthread_mutex_unlock(&m1);

      etime = get_cur_time();
      printf("ELAPSED = %f\n", etime-btime);
}                                      // fine thread function
```

m1 e' un semaforo binario (mutex)

somma e' una variabile condivisa

Sezione critica protetta da semafori

N.B. compilare il tutto linkando la libreria libpthread.a

Variabili di condizione

Le variabili condizione (condition) sono uno strumento di sincronizzazione che premette ai threads di sospendere la propria esecuzione in attesa che siano soddisfatte alcune condizioni su dati condivisi.

ad ogni condition viene associata una coda nella quale i threads possono sospendersi (tipicamente, se la condizione non e` verificata).

Definizione di variabili condizione:

pthread_cond_t: è il tipo predefinito per le variabili condizione

Operazioni fondamentali:

- **inizializzazione: pthread_cond_init**
- **sospensione: pthread_cond_wait**
- **risveglio: pthread_cond_signal**

Var.Cond. : inizializzazione

L'inizializzazione di una v.c. si puo` realizzare con:

```
int pthread_cond_init(pthread_cond_t* cond,
                      pthread_cond_attr_t* cond_attr)
```

dove

- **cond** : individua la condizione da inizializzare
- **attr** : punta a una struttura che contiene gli attributi della condizione; se NULL, viene inizializzata a default.

in alternativa, una variabile condizione può essere inizializzata staticamente con la costante:
PTHREAD_COND_INIZIALIZER

esempio: pthread_cond_t C= PTHREAD_COND_INIZIALIZER ;

Var.Cond. : Wait

La **sospensione** su una condizione si ottiene mediante:

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

dove:

cond: e` la variabile condizione

mux: e` il mutex associato ad essa (la verifica della condizione e' una s.c.)

Effetto:

- il thread chiamante si sospende sulla coda associata a cond, e il mutex mux viene liberato

Al successivo risveglio (provocato da una signal), il thread rioccupera` il mutex automaticamente.

Var.Cond. : Signal

Il **risveglio** di un thread sospeso su una variabile condizione puo` essere ottenuto mediante la funzione:

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

dove **cond** e` la variabile condizione.

Effetto:

- se esistono thread sospesi nella coda associata a cond, ne viene risvegliato uno (non viene specificato quale).
- se non vi sono thread sospesi sulla condizione, la signal non ha effetto.

- Per risvegliare tutti i thread sospesi su una variabile condizione(v. signal_all):

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

Esempio: produttore – consumatore

buffer circolare di interi, di dimensione data (**BUFFER_SIZE**) il cui stato e` dato da:

- numero degli elementi contenuti: **cont**
- puntatore alla prima posizione libera: **writepos**
- puntatore al primo elemento occupato : **readpos**

- il buffer e` una risorsa da accedere in modo mutuamente esclusivo: predispongo un mutex per il controllo della mutua esclusione nell'accesso al buffer: **lock**
- i thread produttori e consumatori necessitano di sincronizzazione in caso di :
  - buffer pieno: definisco una condition per la sospensione dei produttori se il buffer e` pieno (**notfull**)
  - buffer vuoto: definisco una condition per la sospensione dei produttori se il buffer e` vuoto (**notempty**)

Descrizione del buffer (tipo prodcons)

```
#include <stdio.h>
#include <pthread.h>
#define BUFFER_SIZE 16

typedef struct  {
    int buffer[BUFFER_SIZE];
    pthread_mutex_t lock;
    int cont, readpos, writepos;
    pthread_cond_t notempty;
    pthread_cond_t notfull;
}prodcons;
```

Inizializzazione del buffer

```
void init (prodcons *b)  {
    pthread_mutex_init (&b->lock, NULL);
    pthread_cond_init (&b->notempty, NULL);
    pthread_cond_init (&b->notfull, NULL);
    b->cont=0;
    b->readpos = 0;
    b->writepos = 0;
}
```

Funzione richiamata dal main per inizializzare il buffer

Funzione inserimento

```
void inserisci (prodcons *b, int data) {
    pthread_mutex_lock (&b->lock);
    /* controlla che il buffer non sia pieno:*/
    while ( b->cont==BUFFER_SIZE)
            pthread_cond_wait (&b->notfull, &b->lock);
    /* scrivi data e aggiorna lo stato del buffer */
    b->buffer[b->writepos] = data;
    b->cont++;
    b->writepos++;
    if (b->writepos >= BUFFER_SIZE)
    b->writepos = 0;
    /* risveglia eventuali thread (consumatori) sospesi: */
    pthread_cond_signal (&b->notempty);
    pthread_mutex_unlock (&b->lock);
}
```

Funzione richiamata dal produttore

Funzione estrai

```
int estrai (prodcons *b) {
    int data;
    pthread_mutex_lock (&b->lock);
    while (b->cont==0) /* il buffer e` vuoto? */
    pthread_cond_wait (&b->notempty, &b->lock);
    /* Leggi l'elemento e aggiorna lo stato del buffer*/
    data = b->buffer[b->readpos];
    b->cont--;
    b->readpos++;
    f (b->readpos >= BUFFER_SIZE)
    b->readpos = 0;
    /* Risveglia eventuali threads (produttori):*/
    pthread_cond_signal (&b->notfull);
    pthread_mutex_unlock (&b->lock);
    return data;
}
```

Funzione richiamata dal consumatore

Programma test (1)

```
/* Programma di test: 2 thread
- un thread inserisce sequenzialmente max interi,
- l'altro thread li estrae sequenzialmente per stamparli */
#define OVER (-1)
#define max 20
prodcons buffer;
void *producer (void *data) {
    int n;
    printf("sono il thread produttore\n\n");
    for (n = 0; n < max; n++) {
        printf ("Thread produttore %d --->\n", n);
        Inserisci (&buffer, n);
    }
    inserisci (&buffer, OVER);
    return NULL;
}
```

Programma test (2)

```
void *consumer (void *data) {
    int d;
    printf("sono il thread consumatore \n\n");
    while (1) {
    d = estrai (&buffer);
    If (d == OVER)
            break;
    printf("Thread consumatore: --> %d\n", d);
    }
    return NULL;
}
```

Programma test (3)

```
main () {
    pthread_t  th_a,   th_b;
    void *retval;
    init (&buffer);
    /* Creazione threads: */
    pthread_create (&th_a, NULL, producer, 0);
    pthread_create (&th_b, NULL, consumer, 0);
    /* Attesa teminazione threads creati: */
    pthread_join (th_a, &retval);
    pthread_join (th_b, &retval);
    return 0;
}
```