Pthread (**man pthreads**)

Abbiamo detto che lo switching tra threads e' molto piu' veloce che non quello tra processi. Vediamo ad esempio la differenza di tempo per la creazione di 50000 processi e altrettanti threads:

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Pthread

Nel caso di architetture SMP l'uso dei threads e' vantaggioso rispetto ad esempio all'uso di MPI. Le librerie MPI generalmente implementano la comunicazione tra processi nello stesso nodo via “shared memory”. Usando i Pthreads non c'e' una copia intermedia di memoria, infatti i threads condividono aree di memoria.

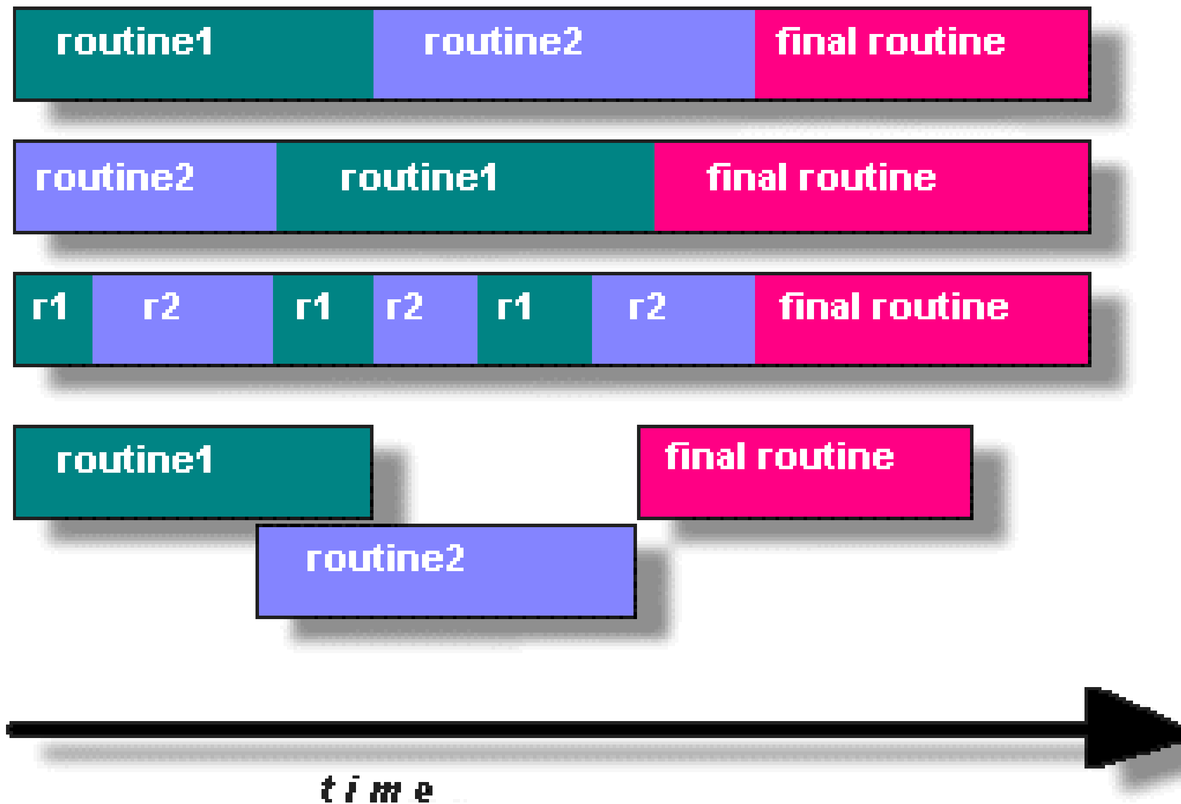
Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
IBM 375 MHz POWER3	0.5	16
IBM 1.5 GHz POWER4	2.1	11
Intel 1.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Pthread

Tuttavia: Several systems today (QNX and Plan 9, for instance) take the stance that threads `fix the symptom, but not the problem'. Rather than using threads because the OS context switch time is too slow, a better approach, according to the architects of these systems, is to fix the OS. It's ironic, now that even PC-hosted desktop OSes provide MMU-protected multitasking, the fashionable programming model has become multiple threads running in a common address space, making debugging difficult, and also making it more difficult to generate reliable code. With fast context switching, existing OS services like explicitly allocated shared memory between a team of cooperating processes can create a `threaded' environment, without opening the Pandora's box of problems that a fully shared memory space entails.

Pthreads

In generale un programma potra' essere parallelizzato usando i pthread se puo' essere suddiviso in parti indipendenti. Ad esempio la routine1 e routine2 possono essere “interchanged”, “interleaved” e/o “overlapped”:



Pthreads

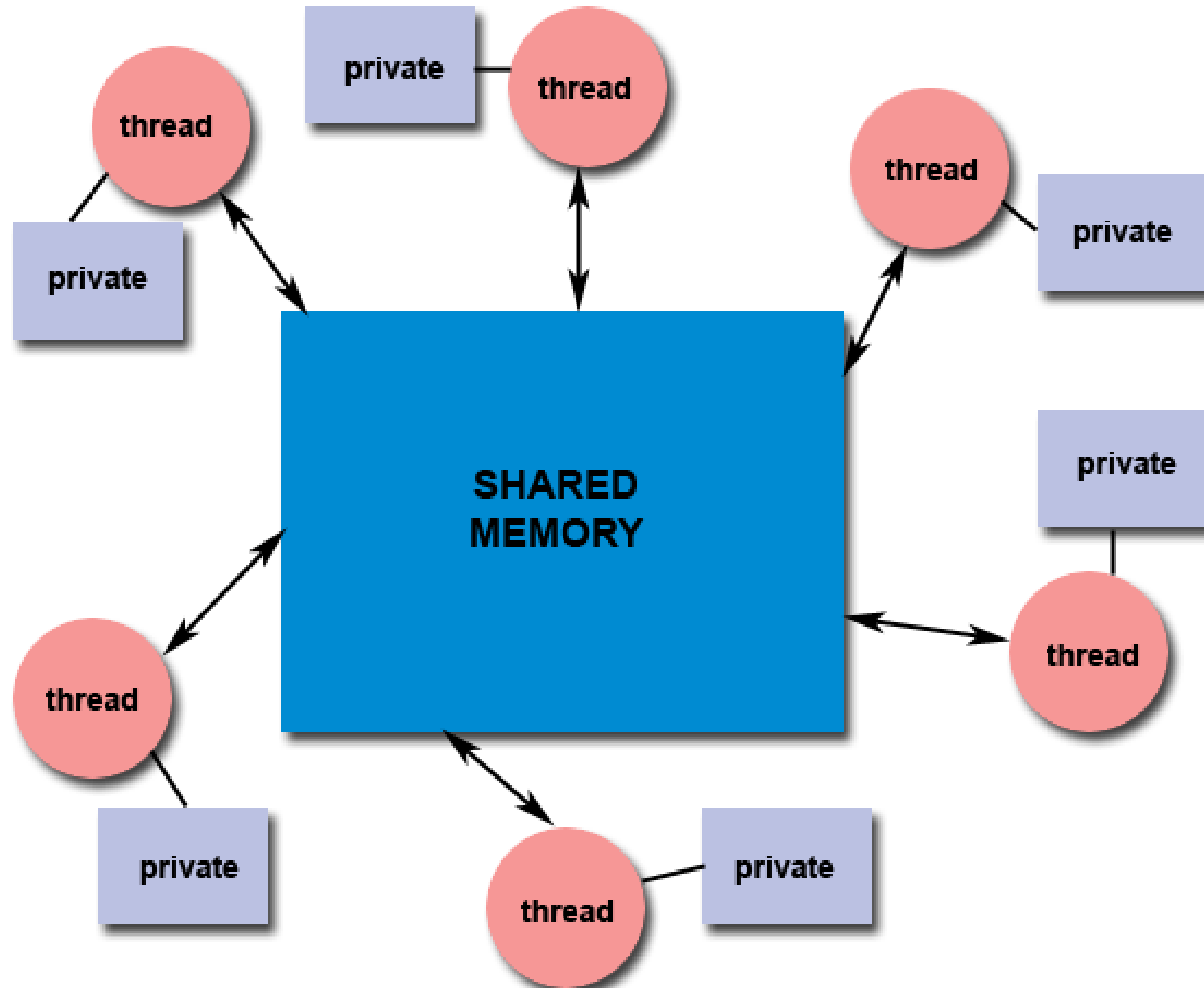
Esistono diversi modelli per programmi “threaded” (ad esempio):

Manager/Worker (Master/Slave): un singolo thread, il manager assegna lavoro ai vari workers. Il manager si occupa tipicamente di tutto l'I/O e della suddivisione del lavoro. Sono possibili almeno du modelli, quello statico e quello dinamico.

Pipeline: Il lavoro e' suddiviso in una serie successiva di sottooperazioni che possono essere eseguite in serie (in parallelo) da thread differenti.

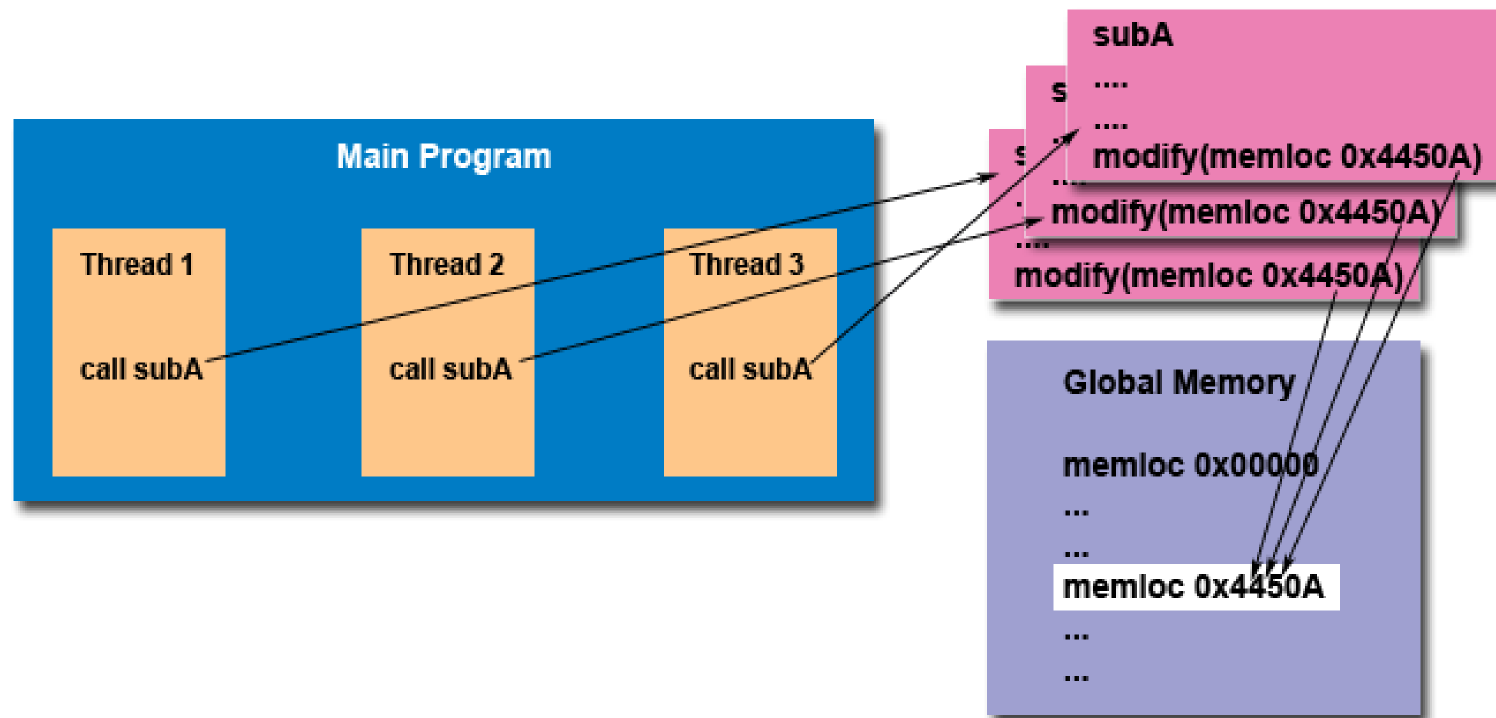
Peer: Simile al Manager/Worker ma il manager partecipa al lavoro.

Pthreads (Shared Memory Model)



Thread-safeness

In generale si dice che una librearia e' thread-safe se le sue chiamate possono essere eseguite su piu' threads senza incorrere in problemi di sovrascrittura di aree di memoria condivista o incorrere in “race conditions” (per le applicazioni e' la stessa cosa):



Pthreads

Per compilare (24_pthreads):

```
$ gcc -o main main.c -lpthread
```

Per vedere i threads:

```
$ ps -lm
...
0 - 500 16214 3934 99   -   -   - 6598 - pts/8
     00:02:42 main
0 S 500     -    -  0  80   0 -      - futex_ -
      00:00:00 -
1 R 500     -    - 98  80   0 -     - -       -
     00:01:21 -
1 R 500     -    - 98  80   0 -     - -       -
     00:01:21 -
```

pthread_create

```
int pthread_create (pthread_t * thread, const
pthread_attr_t * attr, void *(*start_routine)(void*), void
* arg);
```

thread ritorna il thread id (unsigned long int definito in bits/pthreadtypes.h)

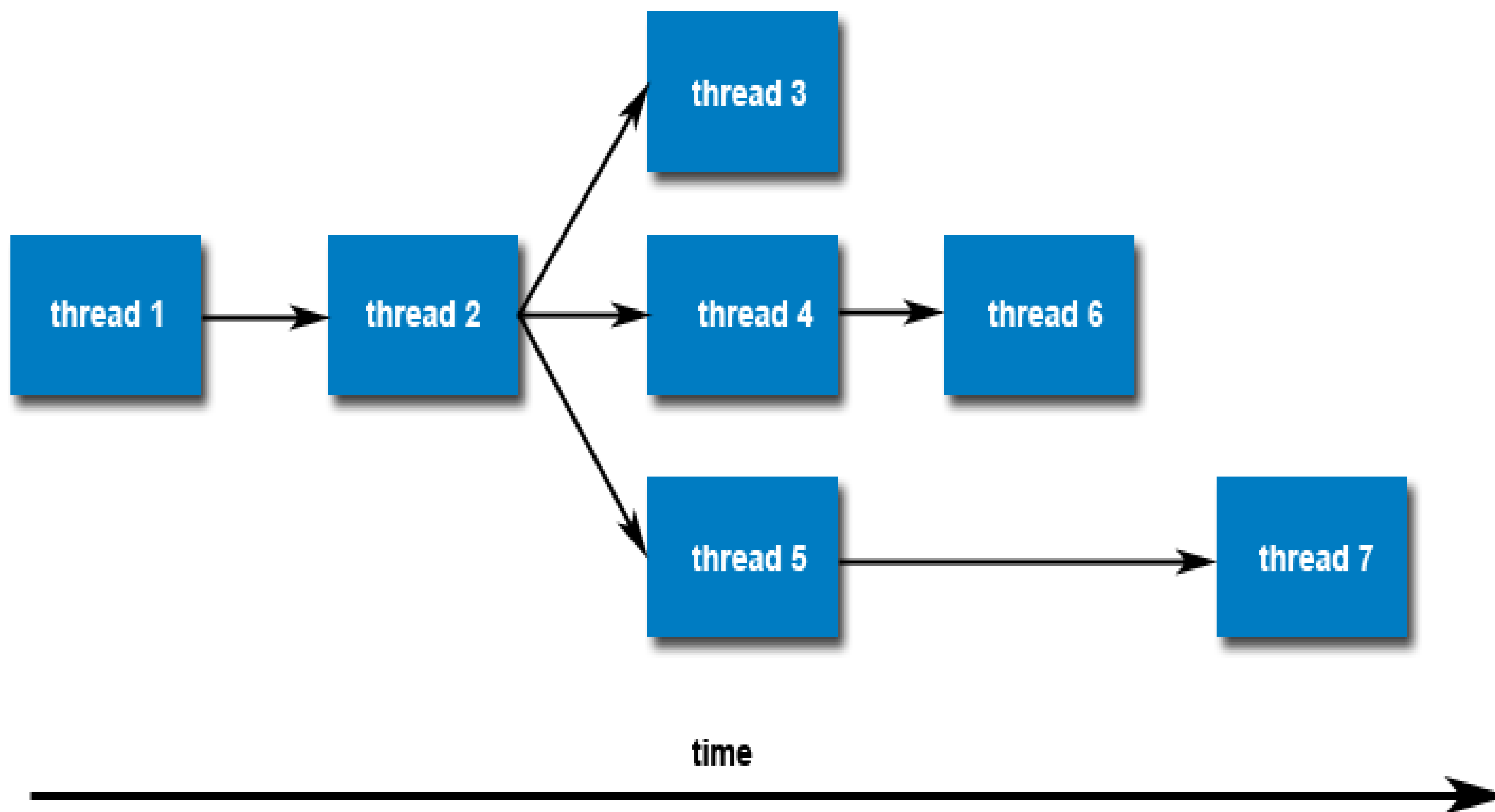
attr se e' NULL verranno usati gli attributi di default. Altimrnti e' possibile definire ad esempio la politica di scheduling, la “stack size” etc. etc.

start_routine e' un pinter alla routine che il thread deve eseguire una volta che e' stato creato

arg e' il pointer all'unico argomento che puo' essere passato alla funzione (come passare piu' di un argomento ?)

Pthreads

Il main contiene un thread unico di default. Ogni thread puo' crearne a sua volta un altro:



Threads terminazione

Un thread puo' essere terminato in diversi modi:

- Il thread fa un return dalla sua routine di partenza
- il thread fa una chiamata a pthread_exit
- il thread e' cancellato da un altro thread mediante una pthread_cancel
- il processo nel suo insieme e' termiato a causa di una chiamata ad exec o exit (due parole su exit _exit ...)

pthread_exit

```
void pthread_exit(void *value_ptr);
```

`value_ptr` valore di ritorno del thread (N.B. Il valore ritornato non deve essere “contenuto” nello scope locale altrimenti andra' perso una volta che il thread sara' terminato)

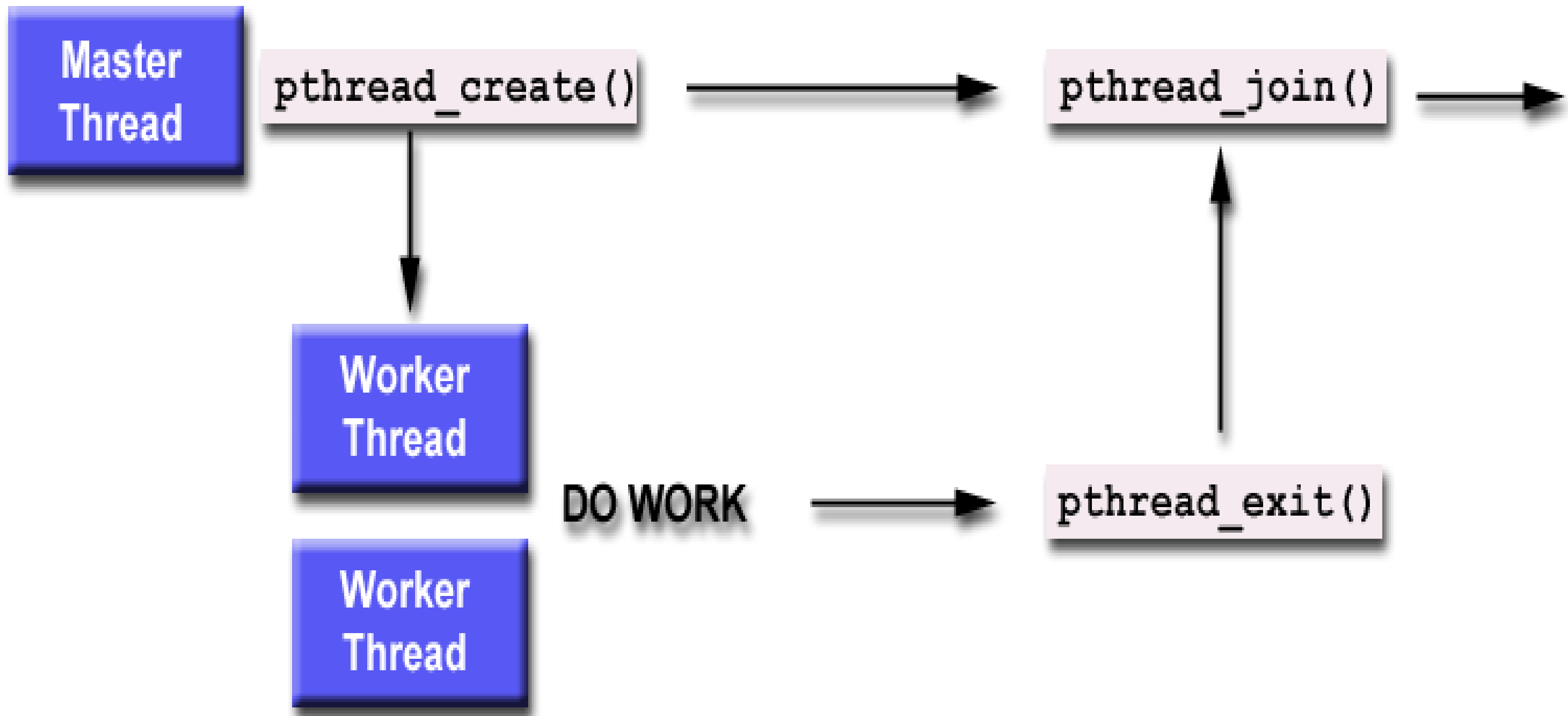
Passaggio degli argomenti ai thread

25_pthread: metodo “semanticamente errato” per passare valori ai thread.

26_pthread: metodo “semanticamente corretto” per passare valori ai thread

Joining (sincronizzazione)

Quando un thread e' creato uno dei suoi attributi dice se e' “joinable” o meno



pthread_join

```
int pthread_join(pthread_t threadid, void **value_ptr);
```

La chiamata blocca il thread fino che il threaid specificato non e' terminato.

value_ptr contiene il riferimento al valore eventualmente ritornato dal thread mediante thread_exit

pthread_attr_

Se un thread deve essere “joined” e' meglio crearlo con gli attributi giusti per favorire la portabilita' del codice:

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

Inizializza un oggetto attributo del thread attr

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

Distrugge l'oggetto attr

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Serve a fare il setting degli attributi.

pthread_detach

```
int pthread_detach(pthread_t threadid);
```

La chiamata mette il threaid nello stato datached. Se non e' necessario creare un thread come joinable e' cosigliabile crearlo come detached, questo potrebbe liberare risorse di sistema.

Quando il thread detached ha finito il suo lavoro le risorse ad esso associate vengono liberate.

pthread_attr_

```
pthread_attr_getstacksize (const pthread_attr_t *attr,
size_t stacksize);
pthread_attr_setstacksize (pthread_attr_t *attr, size_t
stacksize);
pthread_attr_getstackaddr (const pthread_attr_t *attr,
size_t stackaddr);
pthread_attr_setstackaddr (pthread_attr_t *attr, size_t
stackaddr);
```

Le varie implementazioni POSIX thread prevedono dei valori di default per la dimensione dello stack dei vari thread. In generale, onde favorire la portabilita' del codice e' buona abitudine non “affidarsi” alle dimensioni di default.

Esercizi

Prima 2 esempi **27_pthread 28_pthread**

Riscrivere il calcolo di PI con il metodo monte carlo usando i pthread (il numero di thread sara' argomento da linea di comando) **28_ptpimc**

Mutex

Abbreaviazione per “mutual exclusion”. Un mutex fa si che ad ogni istante di tempo solo un thread possa avere acesso ad una certa risorsa condivisa di dati. Tipicamente:

- viene creata ed inizializzata una variabile mutex
- piu' thread cercano di eseguire un “lock” sulla variabile mutex
- solo un thread riesce, quindi solo il dato thread possiede la variabile mutex
- il thread libera la variabile mutex
- un altro thread puo' eseguire un lock sulla variabile mutex
- alla fine la variabile mutex e' “liberata”

pthread_mutex_

Le variabili mutex deveno essere dichiarate con il tipo pthread_mutex_t e devono essere inizializzate prima di essere usate. Inizializzate staticamente:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

oppure dinamicamente con la chiamata:

```
int pthread_mutex_init(pthread_mutex_t * mutex, const
                  pthread_mutexattr_t * attr);
```

in questo modo e' possibile anche mettere degli attributti alla variabile. La funzione:

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

deve essere usata per “distruggere” mutex non piu' utili.

pthread_mutex_

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

Una variabile mutex e' sempre creata come “unlocked”. Su di essa un thread puo' eseguire un lock. Se un secondo thread esegue a sua volta un lock, la chiamata non ritorna fino che il mutex non e' liberato. Trylock invece e' non bloccante, se il mutex e' “locked” la funzione ritorna un EBUSY

Esercizi

Esempio di mutex gia' visto **29_pthread.**

Provate a riscrivere il calcolo di PI con metodo monte carlo usando i mutex **29_pimcm**

Provate ad implementare un prodotto scalare vettore vettore.