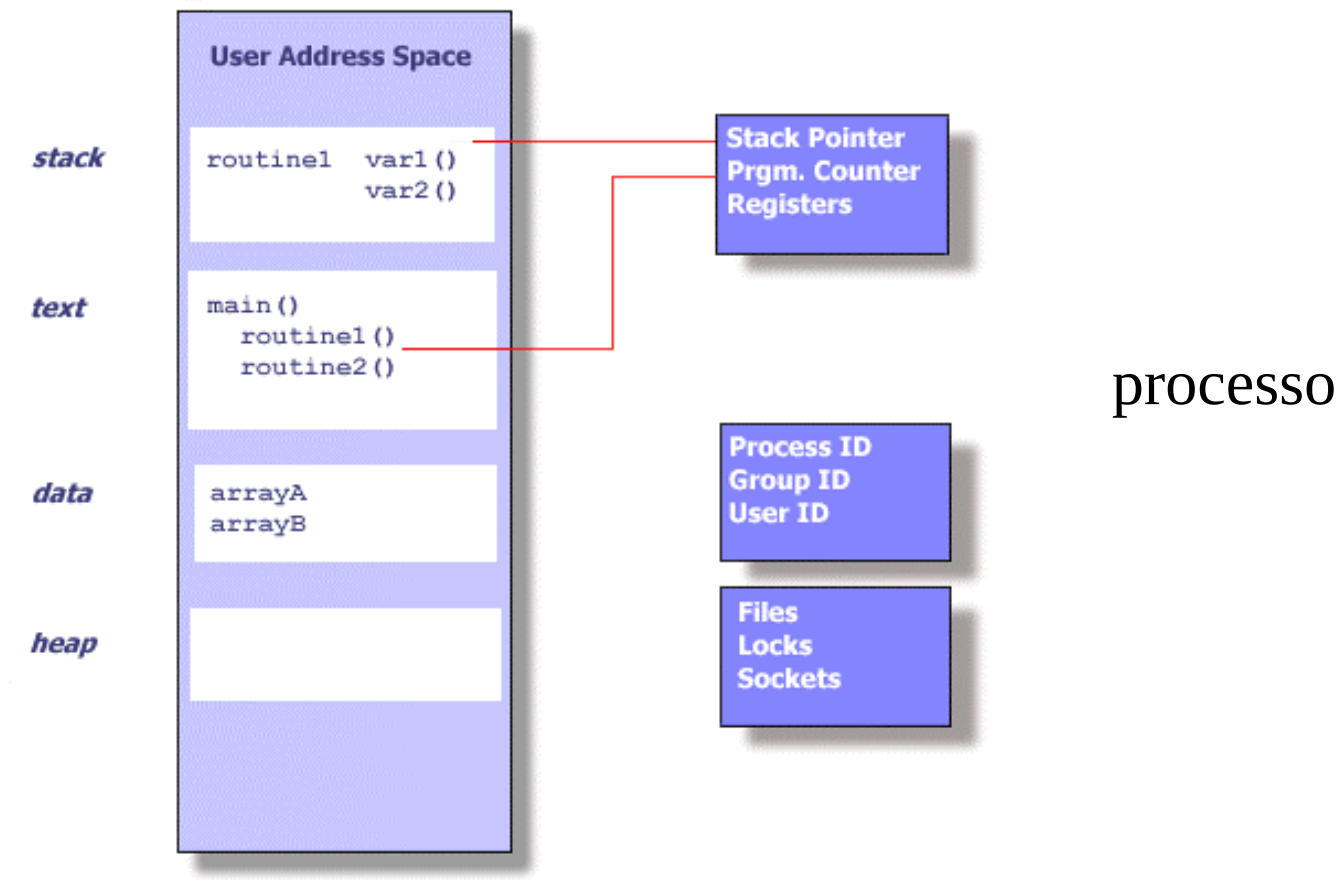
I THREAD

Un thread è un singolo flusso di istruzioni, all'interno di un processo, che lo scheduler può fare eseguire separatamente e concorrentemente con il resto del processo. Per fare questo uno thread deve possedere strutture dati per realizzare un proprio flusso di controllo.
Uno thread può essere pensato come una procedura che lavora in parallelo con altre procedure.

Vediamo le interazioni tra processi e thread.



processo

Il processo ha il proprio contesto, ovvero il proprio process ID, Program Counter, Stato dei Registri, Stack, Codice, Dati, File descriptor, Entità IPC, Azioni dei segnali.
Il Codice è pensato per eseguire procedure sequenzialmente.

L'astrazione dei thread vuole consentire di eseguire procedure concorrentemente (in parallelo), ovviamente scrivendo tali procedure in modo opportuno. Ciascuna procedura da eseguire in parallelo sarà un thread.

I THREAD

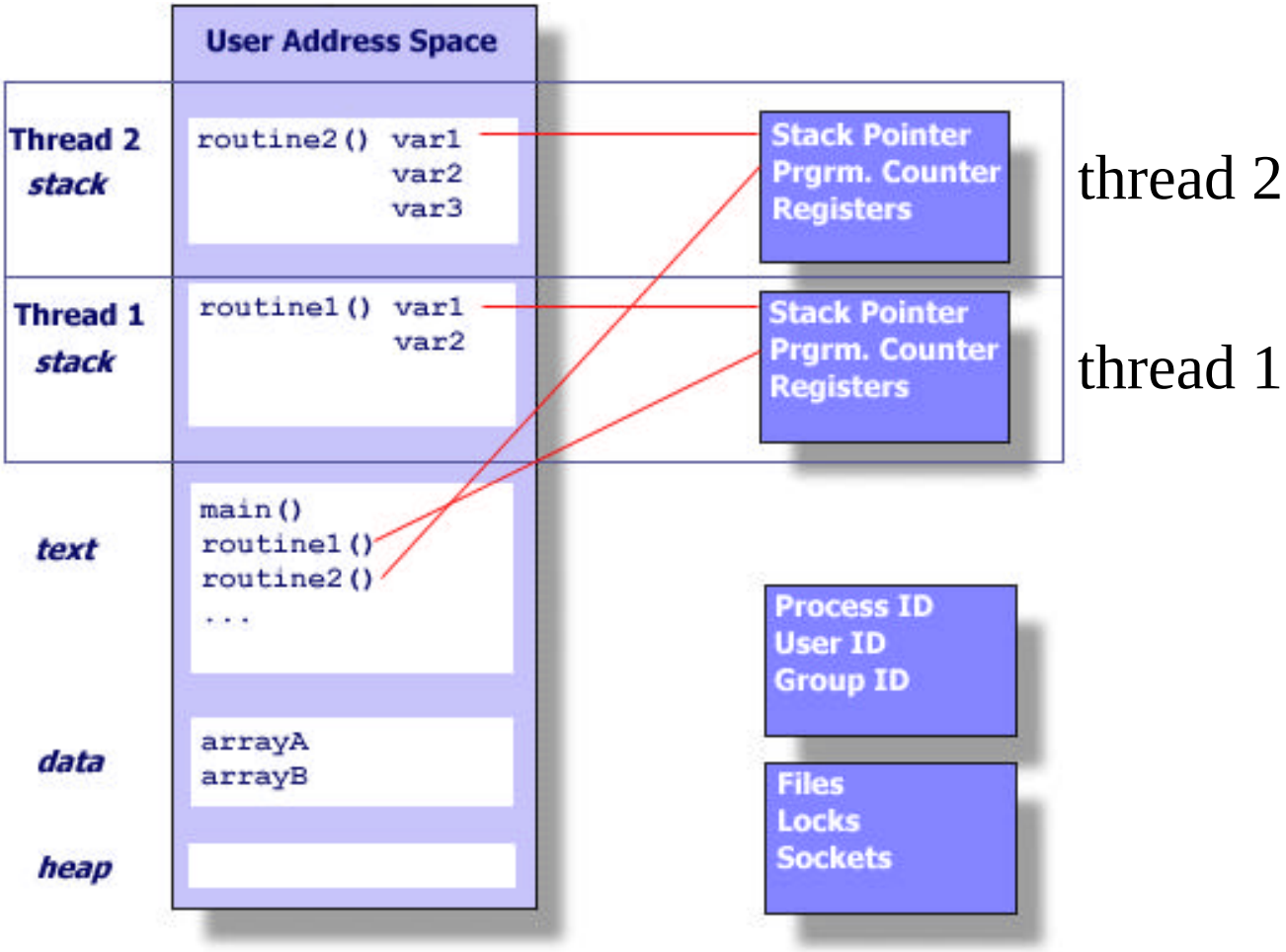
Un thread è un singolo flusso di istruzioni, all'interno di un processo, che lo scheduler può fare eseguire separatamente e concorrentemente con il resto del processo. Per fare questo uno thread deve possedere un proprio

Un Processo può avere più thread, tutti questi condividono le risorse del processo (dati e CPU) ed eseguono nello stesso spazio utente.

Ciascun Thread può usare tutte le variabili globali del processo, e condivide la tabella dei descrittori di file del processo.

Ciascun thread in più avrà potrà avere anche dei propri dati, e sicuramente avrà un proprio Stack, un proprio Program Counter ed un proprio Stato dei Registri.

Dati globali ed entità del Thread (Dati,Stack,Codice,Program Counter, Stato dei Registi) rappresentano lo stato di esecuzione del singolo thread.



THREAD

Vantaggi:
•Visibilità dei dati globali: condivisione di oggetti semplificata.
•Più flussi di esecuzione.
•gestione semplice di eventi asincroni (I/O per esempio)
•Comunicazioni veloci. Tutti i thread di un processo condividono lo stesso spazio di indirizzamento, quindi le comunicazioni tra thread sono più semplici delle comunicazioni tra processi.
•Context switch veloce. Nel passaggio da un thread ad un altro di uno stesso processo viene mantenuto buona parte dell'ambiente.

Svantaggi:
•Concorrenza invece di parallelismo: gestire la mutua esclusione
•Routine di libreria devono essere rientranti (thread safe call): i thread di un programma usano il s.o. mediante system call che usano dati e tabelle di sistema dedicate al processo. Le syscall devono essere costruite in modo da poter essere utilizzate da più thread contemporaneamente. Es: la funzione char *inet_ntoa() scrive il proprio risultato in una variabile di sistema (del processo) e restituisce al chiamante un puntatore a tale variabile. Se due thread di uno stesso processo eseguono "nello stesso istante" la chiamata a due inet_ntoa() ognuno setta la variabile con un valore. Cosa leggono i due chiamanti dopo che le chiamate sono terminate?

Thead Safe Call
Le implementazioni dei thread che soddisfano gli standard POSIX devono mettere a disposizione funzioni thread safe, ovvero che non causano problemi nella scrittura/lettura di strutture dati interne al s.o. In particolare tutte le funzioni dello standard ANSI C e le funzioni degli standard POSIX.1 (alcune funzionalità per comunicazioni via rete) sono thread safe, ad eccezione di: inet_ntoa, asctime, ctime, getlogin, rand, readdir, strtok, ttyname, gethostXXX, getprotoXXX, getservXXX (∃ implementazioni rientranti, il nome finisce con _r).

PTHREAD

I Thread sono stati standardizzati. IEEE POSIX 1003.1c (1995) specifica l'interfaccia di programmazione (Application Program Interface - API) dei thread. I thread POSIX sono noti come Pthread.

Le API per Pthread distinguono le funzioni in 3 gruppi:

Thread management: funzioni per creare, eliminare, attendere la fine dei pthread

Mutexes: funzioni per supportare un tipo di sincronizzazione semplice chiamata "mutex" (abbreviazione di mutua esclusione). Comprende funzioni per creare e eliminare la struttura per mutua esclusione di una risorsa, **acquisire** e **rilasciare** tale risorsa.

Condition variables: funzioni a supporto di una ***sincronizzazione*** più complessa, **dipendente** dal **valore** di **variabili**, secondo i modi definite dal programmatore. Comprende funzioni per creare e eliminare la struttura per la sincronizzazione, per **attendere** e **segnalare** le modifiche delle variabili.

Convenzione sui nomi delle funzioni:
Gli identificatori della libreria dei Pthread iniziano con **pthread_**
In particolare:

pthread_	indica gestione dei thread in generale
pthread_ attr_	funzioni per gestione proprietà dei thread
pthread_mutex_	gestione mutua esclusione
pthread_mutexattr_	proprietà delle strutture per mutua esclusione
pthread_cond_	gestione delle variabili di condizione
pthread_condattr_	proprietà delle variabili di condizione
pthread_key_	dati speciali dei thead

il file **phtreads.h** contiene le definizioni dei pthread

In compilazione (linking) usando *gcc* aggiungere il flag **-lpthread** per usare la libreria dei pthread. Ad es: gcc -o thr1 thr1.c -lpthread

Creazione ed esecuzione di un pthread

int **pthread_create** (pthread_t * **thread**, pthread_attr_t *attr,
void* (***start_routine**)(void *), void * arg);
crea una thread e lo rende eseguibile, cioè lo mette a disposizione dello scheduler che prima o poi lo farà partire.
Il primo parametro **thread** è un puntatore ad un identificatore di thread in cui verrà scritto l'identificatore del thread creato.
Il terzo parametro **start_routine** è il nome (indirizzo) della procedura da fare eseguire dal thread. Deve avere come unico argomento un puntatore.
Il secondo parametro seleziona caratteristiche particolari: può essere posto a NULL per ottenere il comportamento di default.
Il quarto parametro è un puntatore che viene passato come argomento a start_routine.

void **pthread_exit** (void *retval);
termina l'esecuzione del thread da cui viene chiamata, immagazzina il valore puntato da retval, restituendolo ad un altro thread che attende la sua fine.
Il sistema libera le risorse allocate al thread.
Il caso del programma principale (main) è particolare.
Se il main termina prima che i thread da lui creati siano terminati e non chiama la funzione pthread_exit, allora tutti i thread sono terminati. se invece il main chiama pthread_exit allora i thead possono continuare a vivere fino alla loro teminazione.

Esempio banale

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

int N=-1;

void *PrintHello(void *threadid)
{
   N++;
   printf("\n%d: Hello World!  N=%d\n", threadid, N);
   pthread_exit (NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t < NUM_THREADS;t++){
      printf("Creating thread %d\n", t);
      rc = pthread_create (&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n",rc);
         exit(-1);
      }
   }
   pthread_exit (NULL);
}
```

Passaggio degli Argomenti al pthread

La **pthread_create** prevede un puntatore per il passaggio dei parametri al thread nel momento in cui comincia l'esecuzione.
Si ponga attenzione nel caso il thread debba modificare i parametri, oppure il chiamante debba modificare i parametri, potrebbero insorgere problemi, meglio dedicare una struttura dati ad ok, per il passaggio.

CORRETTO

```
int *taskids[NUM_THREADS];

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

SBAGLIATO

```
int rc, t;

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

Gli identificatori dei pthread

pthread_t **pthread_self** (void);
restituisce l’identificatore del thread che la chiama.

pthread_t **pthread_equal** (pthread_t pthread1, pthread_t pthread2);
restituisce 1 se i due identificatori sono uguali

Attesa per la Terminazione di un Thead

```
void main()
{
   pthread_t thread[NUM_THREADS];
   // dichiaro l'attributo
   pthread_attr_t attr;
   int rc, t, status;

   // inizializzo l'attributo
   pthread_attr_init(&attr);
   // setto l'attributo specificando che il thread verrà creato in modo
   // da poterne attendere la fine (operazione join)  undetached
   pthread_attr_setdetachstate (&attr,
                                PTHREAD_CREATE_UNDETACHED);

   for(t=0;t < NUM_THREADS;t++)
   {
      printf("Creating thread %d\n", t);
      rc = pthread_create (&thread[t], &attr, funzione, NULL);
   }

   // libero la memoria usata per l'attributo
   pthread_attr_destroy(&attr);

   for(t=0;t < NUM_THREADS;t++)
   {
       rc = pthread_join (thread[t], (void **)&status);
       printf("Completed join with thread %d status= %d\n",t, status);
   }
pthread_exit(NULL);
}
```

Garantire la Mutua Esclusione

due thread devono decrementare il valore di una variabile globale data, se questa è maggiore di zero
Inizialmente data vale 1.

```
THEAD1                      THEAD2

if(data>0)                  if(data>0)
        data --;                    data --;
```

A seconda del tempo di esecuzione dei due thread,la variabile data assume valori diversi.

---

data	THEAD1	THEAD2
1	if(data>0)	
1	data --;	
0		if(data>0)
0		data --;

0 = valore finale di data

---

data	THEAD1	THEAD2
1	if(data>0)	
1		if(data>0)
1	data --;	
0		data --;

-1 = valore finale di data

Mutex Variables

Mutex à l'abbreviazione di "mutua esclusione".
Una variabile mutex (più formalmente, di tipo pthread_mutex_t) è una variabile che serve per regolare l'accesso a dei dati che debbono essere protetti dall'accesso contemporaneo da parte di più thread.

Ogni thread, prima di accedere a tali dati, deve effettuare una operazione di lock su una stessa variabile mutex. L'operazione detta "lock" di una variabile mutex blocca l'accesso da parte di altri thread.

Infatti, se più thread eseguono l'operazione di lock su una stessa variabile mutex, solo uno dei thread termina la lock() e prosegue l'esecuzione, gli altri rimangono bloccati nella lock. In tal modo, il processo che continua l'esecuzione può accedere ai dati (protetti mediante la mutex).

Finito l'accesso, il thread effettua un'operazione detta "unlock" che libera la variabile mutex. Un'altro thread che ha precedentemente eseguito eseguito la lock della mutex potrà allora terminare la lock ed accedere a sua volta ai dati.

La tipica sequenza d'uso diuna mutex è quindi:

- creare ed inizializzare la mutex
- più thread cercano di accedere alla mutex chiamando la lock
- un solo thread termina la lock e diviene proprietario della mutex, gli altri sono bloccati.
- il thread proprietario accede ai dati, o esegue funzioni.
- il thread proprietario libera la mutex eseguendo la unlock
- un'altro thread termina la lock e diventa proprietario di mutex
- .... e cosi via
- al termine la mutex è distrutta.

Esempio di Protezione con Mutex Variables

```
#define NUMTHRDS 10
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexdata;    int data;

void *decrementa(void *arg)
{
   pthread_mutex_lock (&mutexdata);
   if(data>0)
      data--;
   pthread_mutex_unlock (&mutexdata);
   pthread_exit((void*) 0);
}

void main (int argc, char *argv[])
{
  int i;    int status;    pthread_attr_t attr;
  data=4;
  pthread_mutex_init (&mutexdata, NULL);
  for(i=0;i < NUMTHRDS;i++)        {
        // per default i thread consentono il join,
        pthread_create ( &callThd[i], NULL, decrementa, (void *)i);
   }
   for(i=0;i < NUMTHRDS;i++)        { // aspetto la fine dei thread
        pthread_join ( callThd[i], (void **)&status);
   }

   printf ("data =  %d \n", data);
   pthread_mutex_destroy (&mutexdata);
   pthread_exit(NULL);
}
```

Evitare il blocco con Mutex Variables

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

è come la lock(), ma se si accorge che la mutex è già in possesso di altro thread (e quindi si rimarrebbe bloccati) restituisce immediatamente il controllo al chiamante con risultato EBUSY
In caso la chiamata vada a buon fine e si ottenga la proprietà della mutex, restituisce 0.