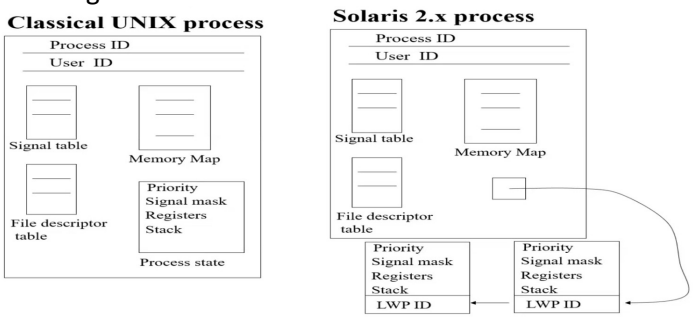
Processi e thread 8 Thread Unix

sabato 18 ottobre 2025 12:54

Vediamo ora come gestire i thread in unix :



Nella seconda versione di hanno i thread control block , mentre nel primo si avevano i process block. inoltre avendo unico address space non c'è bisogno di ritirare su tutto. Vediamo in dettaglio come farlo :

int pthread_create(pthread_t *tid, pthread_attr_t *attr, void *(*funct)(void*), void *arg)	
Descrizione	invoca l'attivazione di un thread
Parametri	1) *tid: buffer di informazioni sul thread 2) *attr: buffer di attributi (NULL identifica il default) 3) (*funct): puntatore a funzione per il thread 4) *arg: puntatore al buffer degli argomenti
Restituzione	un valore diverso da zero in caso di fallimento

pthread_t e' un **unsigned int**

void pthread_exit(void *status)	
Descrizione	invoca la terminazione del thread chiamante
Parametri	*status: puntatore che definisce il codice di uscita
Restituzione	non ritorna in caso di successo

Dove il terzo parametro è puntatore a funzione (funzione all'interno della quale deve vivere il thread); p_thread_attributes è una struttura che contiene dati per fare startup a grana fine. Per quanto rigurda l'exit , il thread comunica l'indirizzo di memoria (unsigned long).

int pthread_join(pthread_t tid, void **status)	
Descrizione	invoca l'attesa di terminazione di un thread
Parametri	1) tid: identificatore del thread (indicativo) 2) **status: puntatore al puntatore al buffer contenente il codice di uscita
Restituzione	-1 in caso di fallimento, altrimenti l'identificatore del thread terminato

pthread_t pthread_self(void)	
Descrizione	chiede l'identificatore del thread chiamante
Restituzione	-1 in caso di fallimento, altrimenti l'identificatore del thread

Il join permette di riunire la mia esecuzione con altro thread -> attendo che finisca : t_id è il thread che voglio attendere. In UNIX è possibile rilasciare tutte le risorse di un thread , appena quest'ultimo finisce :

SYNOPSIS top

```
#include <pthread.h>

int pthread_detach(pthread_t thread);

Compile and link with -pthread.
```

DESCRIPTION top

The **pthread_detach**() function marks the thread identified by *thread* as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Attempting to detach an already detached thread results in unspecified behavior.

RETURN VALUE top

On success, **pthread_detach**() returns 0; on error, it returns an error number.

Vediamo degli esempi :

1. Thread sort string

a.

```c
#include <string.h>
#include <stdlib.h>

#define MAX_CHAR        1000
#define MAX_STRING      2000

int string_number=0;
char * string_array[MAX_STRING];
void *status;
void * OrderThread(void * new_string) {
int i,j;
  //printf("Ordering thread active\n");
  for (i=0 ; i < string_number; i++) {
    if (strcmp(new_string,string_array[i]) <= 0) {
      for (j=0; j<(string_number-i); j++) {
        string_array[string_number - j] = string_array[string_number - j - 1];
      }
      break;
    }
  }
  string_array[i] = new_string;
  string_number++;
  pthread_exit((void *)NULL);
}
int main () {
  int i;
  pthread_t tid = 0;
  char * old_buffer;
  void * return_code;
  char buffer[MAX_CHAR];
  int notfirst = 0;
  while(1) {
    printf("Insert string: ");
    scanf("%s", buffer);
    if (strcmp(buffer, "quit") == 0 && notfirst == 0) return 0;
    if (strcmp(buffer, "quit") == 0) break;
    old_buffer = strdup(buffer);
    if (notfirst) pthread_join(tid, &status);
    else notfirst = 1;
    i=pthread_create(&tid, NULL, OrderThread, (void *)old_buffer);
    if (i) {
      printf("cannot create thread for error %d\n", i);
      exit(EXIT_FAILURE);
    }
  }

  pthread_join(tid, &return_code);
  for (i=0; i< string_number; i++) printf("String %d: %s\n", i,string_array[i]);
  return 0;
}
```

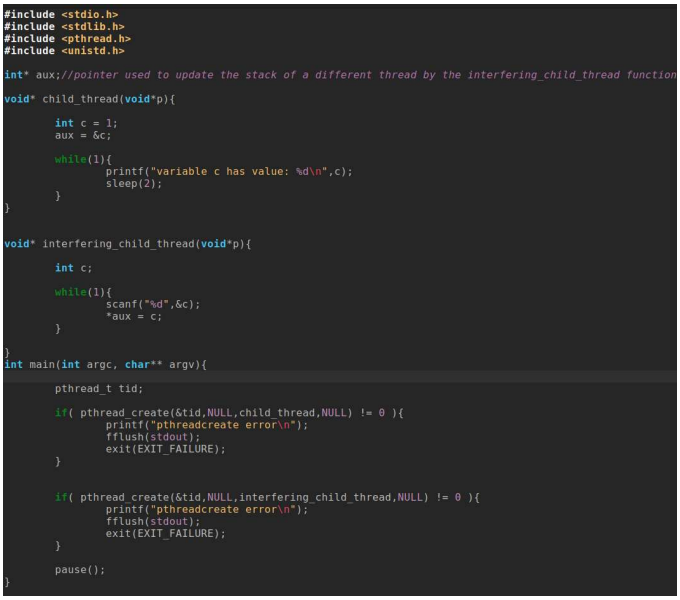
b. Per compilarlo dobbiamo usare –**lpthread**

c. Il suo output ordina le stringhe in modo alfabetico che passo

d. In questo esempio spostiamo solo i puntatori

2. Thread interference

a.



```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int* aux;//pointer used to update the stack of a different thread by the interfering_child_thread function

void* child_thread(void*p){

        int c = 1;
        aux = &c;

        while(1){
                printf("variable c has value: %d\n",c);
                sleep(2);
        }
}


void* interfering_child_thread(void*p){

        int c;

        while(1){
                scanf("%d",&c);
                *aux = c;
        }

}
int main(int argc, char** argv){

        pthread_t tid;

        if( pthread_create(&tid,NULL,child_thread,NULL) != 0 ){
                printf("pthreadcreate error\n");
                fflush(stdout);
                exit(EXIT_FAILURE);
        }


        if( pthread_create(&tid,NULL,interfering_child_thread,NULL) != 0 ){
                printf("pthreadcreate error\n");
                fflush(stdout);
                exit(EXIT_FAILURE);
        }

        pause();
}
```

b. Anche qui per compilare **–lpthread**

c. L'output è il seguente : ogni 2 secondi stampa il valore 1 e se digitiamo altro numero lo stamnperà, sovrascrivendo il vecchio valore

3. Thread performance

a. Questo esempio fa vedere il tempo per generare un processo vs quello per tirare su altri thread

b.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

#define SPAWNS 50000

void* child_thread(void*p){
        return NULL;
}

int main(int argc, char** argv){

   int i;
   int status;
   void *thread_status;
   pthread_t tid;

   if (argv[1]==NULL){
        printf("missing arg[1]\n");
        exit(EXIT_FAILURE);
   }

   if (strcmp("processes", argv[1]) ==0) {
        for (i=0;i<SPAWNS;i++){
          if(fork()) wait(&status);
          else exit(0);
        }
   }

   if (strcmp("threads", argv[1]) ==0) {
        for (i=0;i<SPAWNS;i++){
          pthread_create(&tid,NULL,child_thread,NULL);
          pthread_join(tid,&thread_status);
        }

   }
}
```

c. Anche qui per compilare **–lpthread**
d. E per vedere le prestazione : **time** ./a.out processes/threads
e. Si nota che con i threads è molto più veloce

Vediamo ora come terminare i thread :

Ambienti a processi

`void exit(int)` → Terminazione dell' unico thread attivo, e quindi dell'intero processo

Ambienti a thread

`void exit(int)` System-call → Terminazione del thread corente, non necessariamente dell'intero processo

`void exit_group(int)` System-call → Terminazione dell'intero processo

exit() e' mappata su exit_group() in stdlib
per conformita' a sistemi legacy

Sorge un problema: è possibile chiamare exit senza chiamare exit_group? Si e dobbiamo usare la seguente segnatura : **syscall(par1,par2,...) .** Vediamolo in esempio:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include<unistd.h>


void* child_thread(void*p){

        again:
        sleep(1);
        printf("I'm alive\n");
        goto again;
}

int main(int argc, char** argv){

   pthread_t tid;

   if(pthread_create(&tid,NULL,child_thread,NULL) != 0){
        printf("pthread_create error\n");
        fflush(stdout);
        exit(EXIT_FAILURE);
   }
        //servizio exit all'interno del sistema
   syscall(60,0);

}
```

Il quale se scritto cosi , faccio uscire il thread main , lasciando in esecuzione il thread figlio. Vediamo ora ultima cosa : **libreria rientrante vs libreria non rientrante :** la prima si ha quando la libreria offre servizi thread-safe (hanno meccanismi per sincronizzazione tra thread : scanf,printf,malloc) ; come verificarlo? Hanno il suffisso __r