Lezione 8 - 3/5/21 (Queues, Thread)

Queues

Code di messaggi

Una coda di messaggi è una lista concatenata memorizzata all'interno del kernel ed identificata con una chiave all'interno dei programmi, chiamata **"queue identifier".**

La chiave viene successivamente condivisa tra i processi interessati ed essi genereranno degli ulteriori identificativi da usare durante l'interazione con la coda

Ogni messaggio inserito nella coda ha tre campi:

- Un tipo (intero "long")
- Una lunghezza non negativa
- Un insieme di dati (bytes) di lunghezza corretta

```
struct msg_buffer{
  long mtype;
  char mtext[100];
}msg_buffer;
```

Al contrario delle FIFO, i messaggi in una coda possono essere recuperati anche sulla base del tipo e non solo del loro ordine "assoluto" di arrivo.

Creazione coda

```
int msgget(key_t key, int msgflg)
```

Restituisce l'identificativo di una coda basandosi sulla chiave "key" e sui flags:

- **IPC_CREAT:** crea una coda se non esiste già, altrimenti restituisce l'identificativo di quella già esistente
- **IPC_EXCL:** (da usare assieme al precedente) fallisce se coda già esistente
- **0xxx:** numero ottale di permessi, analogo a quello che si può usare nel file system

```
//msgget.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
key_t queueKey = 56; //Unique key (Supposed)
int queueId = msgget(queueKey, 0777 | IPC_CREAT | IPC_EXCL);
```

Ottenere una chiave univoca

```
key_t ftok(const char *path, int id)
```

Restituisce una chiave basandosi sul path di una cartella o file esistente ed accessibile, la chiave dovrebbe essere univoca e sempre la stessa per ogni coppia <path, id> in ogni istante sullo stesso sistema

Tip: Per avere un id unico potrei creare un path o un file temporaneo creare una chiave univoca usarla ed eliminare il path o file per non rischiare un duplicato.

```
//ftok.c
#include <sys/ipc.h>
key_t queue1Key = ftok("/tmp/unique",1);
key_t queue2Key = ftok("/tmp/unique",2);
```

Esempio:

```
//ipcCreation.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
void main(){
    remove("/tmp/unique"); //Remove file
    key_t queue1Key = ftok("/tmp/unique", 1); //Get unique key → fail
    creat("/tmp/unique", 0777); //Create file
    queue1Key = ftok("/tmp/unique", 1); //Get unique key → ok
    int queueId = msgget(queue1Key ,0777 | IPC_CREAT); //Create queue → ok
    queueId = msgget(queue1Key , 0777); //Get queue → ok
    msgctl(queue1Key,IPC_RMID,NULL); //Remove non existing queue → fail
    msgctl(queueId,IPC_RMID,NULL); //Remove queue → ok
    queueId = msgget(queue1Key , 0777); //Get non existing queue → fail
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Create queue → ok
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Get queue → ok
    queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL); /* Create
    already existing queue -> fail */
}
```

Le queue sono persistenti

```
//persistent.c
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>
void main(){
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL);
    perror("Error:");
}
```

Se eseguiamo questo programma **dopo** aver eseguito il precedente “*ipcCreation.c*” verrà generato un **errore** dato che la **coda esiste già** ed abbiamo **usato** il **flag IPC_EXCL**!

Inviare messaggi

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Aggiungere un messaggio di dimensione ***msgsz*** alla coda identificata da ***msquid***, puntata dal buffer ***msgp***. Il messaggio viene inserito immediatamente se c'è abbastanza spazio disponibile, altrimenti la chiamata si blocca fino a che abbastanza spazio diventa disponibile. Se ***msgflg*** è IPC_NOWAIT allora la chiamata fallisce in assenza di spazio.

Ricevere messaggi

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Rimuove un messaggio dalla coda ***msqid*** e lo salva nel buffer ***msgp***. ***msgsz*** specifica la lunghezza massima del testo del massaggio (***mtext*** della struttura ***msgp***). Se il messaggio ha una lunghezza maggiore e ***msgflg*** è MSG_NOERROR allora il messaggio viene troncato (viene persa la parte in eccesso), se MSG_NOERROR non è specificato allora il messaggio non viene chiamato e la chiamata fallisce. A seconda di ***msgtyp*** viene recuperato il messaggio:

- **msgtyp = 0:** primo messaggio della coda
- **msgtyp > 0:** primo messaggio di tipo **msgtyp,** o primo messaggio di tipo diverso da **msgtyp** se MSG_EXCEPT è impostato come flag.
- **msgtyp <0:** primo messaggio il cui tipo T è min(T≤|msgtyp|)

In fine, il flag **IPC_NOWAIT** fa **fallire** la **syscall** se **non** sono **presenti messaggi** (altrimenti hang)

Modificare la coda

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Modifica la coda identificata da ***msqid*** secondo i comandi **cmd**, riempiendo buf con informazioni sulla coda (ad esempio tempo di ultima scrittura, di ultima lettura, numero messaggi nella coda, etc…). Valori possibili per **cmd**:

- **IPC_STAT:** recupera informazioni da kernel
- **IPC_SET:** imposta alcuni parametri a seconda di *buf*
- **IPC_RMID:** rimuove immediatamente la coda
- **IPC_INFO:** recupera informazioni generali sui limiti delle code nel sistema
- **MSG_INFO:** come IPC_INFO ma con informazioni differenti
- **MSG_STAT:** come IPC_STAT ma con informazioni differenti

msqid_ds structure

```
struct msqid_ds {
  struct ipc_perm msg_perm; /* Ownership and permissions */
  time_t msg_stime; /* Time of last msgsnd(2) */
  time_t msg_rtime; /* Time of last msgrcv(2) */
  time_t msg_ctime; //Time of creation or last modification by msgctl
  unsigned long msg_cbytes; /* # of bytes in queue */
  msgqnum_t msg_qnum; /* # of messages in queue */
  msglen_t msg_qbytes; /* Maximum # of bytes in queue */
  pid_t msg_lspid; /* PID of last msgsnd(2) */
  pid_t msg_lrpid; /* PID of last msgrcv(2) */
};
```

ipc_perm structure

```
struct ipc_perm {
  key_t __key; /* Key supplied to msgget(2) */
  uid_t uid; /* Effective UID of owner */
  gid_t gid; /* Effective GID of owner */
  uid_t cuid; /* Effective UID of creator */
  gid_t cgid; /* Effective GID of creator */
  unsigned short mode; /* Permissions */
  unsigned short __seq; /* Sequence number */
};
```

Esempio:

```
//modifica.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

struct msg_buffer{
    long mtype;
    char mtext[100];
} msgpSND,msgpRCV;
void main(){
    struct msqid_ds mod;
    msgpSND.mtype = 1;
    strcpy(msgpSND.mtext,"This is a message from sender");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
    msgctl(queueId,IPC_RMID,NULL); //Remove queue if exists
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Create queue
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
    msgctl(queueId,IPC_STAT,&mod); //Modify queue
    printf("Msg in queue: %ld\nCurrent max bytes in queue: %ld\n\n",mod.msg_qnum, mod.msg_qbytes);
    mod.msg_qbytes = 200; //Change buf to modify queue bytes
    msgctl(queueId,IPC_SET,&mod); //Apply modification
    printf("Msg in queue: %ld --> same number\nCurrent max bytes in queue: %ld\n\n",mod.msg_qnum, mod.msg_qbytes);
    if(fork()!=0)//Parent keep on writing on the queue
```

```
    {
        printf("[SND] Sending third message with a full queue...\n");
        msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
        printf("[SND] msg sent\n");
        printf("[SND] Sending fourth message again with IPC_NOWAIT\n");
        if(msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),IPC_NOWAIT)==-1)//Send msg
        {
            perror("Queue is full --> Error");
        }
    }
    else // Child keeps reading the queue every 3 seconds
    {
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 1 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 2 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 3 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 4 with msg '%s'\n",msgpRCV.mtext);
        printf("[Reader] Received msg 5 with msg '%s'\n",msgpRCV.mtext);
        sleep(3);
        if(msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,IPC_NOWAIT)==-1)
        {
            perror("Queue is empty --> Error");
        }
        else
        {
            printf("[Reader] Received msg 6 with msg '%s'\n",
            msgpRCV.mtext);
        }
    }
    while(wait(NULL)>0);
}
```

Threads

Threads

I thread sono **singole sequenze di esecuzione** all'interno **di un processo**, aventi alcune delle proprietà dei processi. I threads **non sono indipendenti** tra loro e **condividono** il **codice**, i **dati** e le **risorse** del sistema **assegnate al processo di appartenenza**. Come ogni singolo processo, i threads hanno alcuni **elementi indipendenti**, come lo **stack**, il **PC** ed i **registri del sistema**.

La creazione di threads consente un **parallelismo delle operazioni** in maniera **rapida** e **semplificata**. Context switch tra threads è rapido, così come la loro creazione e terminazione. Inoltre, la **comunicazione** tra threads è **molto veloce**.

ATTENZIONE!!!

Per la **compilazione** è necessario aggiungere il flag **-pthread**, ad esempio:

```
gcc -o program main.c -pthread
```

Creazione

In C i thread **corrispondono** a delle **funzioni** eseguite in parallelo al codice principale. Ogni thread è **identificato** da un **ID** e può essere **gestito come** un processo **figlio**, con **funzioni** che **attendono la sua terminazione**.

```
int pthread_create(
  pthread_t *restrict thread, /* Thread ID */
  const pthread_attr_t *restrict attr, /* Attributes */
  void *(*start_routine)(void *), /* Function to be executed */
  void *restrict arg /* Parameters to above function */
);
```

Esempio:

```
//threadCreate.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void*)3;
}
void main(){
    pthread_t t_id;
    int arg=10;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
}
```

Terminazione

Un nuovo thread termina in uno dei seguenti modi:

- Chiamando la funzione `noreturn void phtread_exit(void *retval);` specificando un puntatore di ritorno.
- Ritorna dalla funzione associata al thread specificando un valore di ritorno.
- Viene cancellato
- Qualche thread chiama `exit()` , o il thread che eseguue `main()` ritorna dallo stesso, terminando così tutti i suoi threads.

Cancellazione di un thread

```
int pthread_cancel(pthread_t thread);
```

Invia una **richiesta** di cancellazione al thread specificato, il quale reagirà (come e quando) a seconda i due suoi attributi: **state** e **type**. **State** può essere ***enabled*** (default) o ***disabled***: se ***disabled*** la richiesta rimarrà in attesa fino a che state diventa ***enabled***, se ***enabled*** la cancellazione avverrà a seconda di **type**. **Type**

può essere ***deferred*** (default) o ***asynchronous***: attende la chiamata di un ***cancellation point*** o termina in qualsiasi momento, rispettivamente. Cancellation points sono funzioni definite nella libreria pthread.h (**lista**).
State e **type** possono essere modificati:

int pthread_setcancelstate(int state, int *oldstate);
con state **= PTHREAD_CANCEL_DISABLE o PTHREAD_CANCEL_ENABLE**
int pthread_setcanceltype(int type, int *oldtype);
Con type **= PTHREAD_CANCEL_DEFERRED o PTHREAD_CANCEL_ASYNCHRONOUS**

Aspettare un thread

Un processo (thread) che avvia un nuovo thread può aspettare la sua terminazione mediante la funzione:

```
int pthread_join(pthread_t thread, void **retval);
```

Se il valore di ritorno del thread non è nullo (parametro di **pthread_exit** o di **return**), esso viene **salvato** nella **variabile puntata** da **retval**. Se il **thread** era stato **cancellato**, **retval** è riempito con **PTHREAD_CANCELED**.

N.B.

Un thread può essere aspettato solo se è joinable e un thread può essere aspettato da al massimo un thread.

Esempio:

```
//thJoin.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void * my_fun(void * param){
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    char * str = "Returned string";
    pthread_exit((void *)str); //or ‘return (void *) str;’
}
void main(){
    pthread_t t_id;
    void * retFromThread; //This must be a pointer to void!
    pthread_create(&t_id, NULL, my_fun, (void *)&t_id); //Create
    pthread_join(t_id,&retFromThread); // wait thread
    // We must cast the returned value!
    printf("Thread %ld returned '%s'\n",t_id,(char *)retFromThread);
}
```

```
//threadJoin.c (v. esempio threadCreate.c)
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void*)3;
}
void main(){
    pthread_t t_id;
```

```
    int arg=10, retval;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
    pthread_join(t_id, (void **)&retval);
    printf("retval=%d\n", retval);
}
```

```
//thJoin1.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void * my_fun(void * param){
    sleep(2);
}
void * my_fun2(void * param){
    if(pthread_join( *(pthread_t *) param,NULL) != 0)
        printf("Error\n");
}
void main(){
    pthread_t t_id,t_id2;
    pthread_create(&t_id, NULL, my_fun, NULL); //Create
    pthread_create(&t_id2, NULL, my_fun, (void *)&t_id); //Create
    pthread_join(t_id,NULL); // wait thread
    sleep(1);
    perror();
}
```

Attributi di un thread

Ogni thread viene creato con degli attributi specificati nella struttura ***pthread_attr_t***. Tutto ciò funziona come con le maschere dei segnali che vengono prima settate e poi sovrascritte (la vecchia con la nuova).

La struttura va inizializzata con:

```
//imposta tutti gli attributi al valore di default
int pthread_attr_init(pthread_attr_t *attr);
```

La si distrugge con:

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

I vari attributi della struct possono, e devono, essere modificati singolarmente con le seguenti funzioni:

```
int pthread_attr_setxxxx(pthread_attr_t *attr, params);
int pthread_attr_getxxxx(const pthread_attr_t *attr, params);
```

Elenco attributi:

- `...detachstate(pthread_attr_t *attr, int detachstate)`

    - ***PTHREAD_CREATE_DETACHED*** → non può essere aspettato
    - ***PTHREAD_CREATE_JOINABLE*** → default, può essere aspettato
    - Può essere cambiato durante l'esecuzione con: `int pthread_detach(pthread_t thread);` Può però solo diventare da joinable a detached e non videversa.
- `...sigmask_np(pthread_attr_t *attr,const sigset_t *sigmask);`

Non utilizzate:

- ...affinity_np(...)
- ...setguardsize(...)
- ...inheritsched(...)
- ...schedparam(...)
- ...schedpolicy(...)

Esempio:

```
//threadAttr.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void*)3;
}
void main(){
    pthread_t t_id;
    pthread_attr_t attr;
    int arg=10, detachSTate;
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED); //Set detached
    pthread_attr_getdetachstate(&attr,&detachSTate); //Get detach state
    if(detachSTate == PTHREAD_CREATE_DETACHED)
        printf("Detached\n");
    else
        printf("Not Detached\n");
    pthread_create(&t_id, &attr, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE); //Inneffective
    printf("Tried to set joinable\n");
    pthread_attr_destroy(&attr);
    sleep(3);
    int esito = pthread_join(t_id, (void **)&detachSTate);
    printf("Esito '%d' is different 0\n", esito);
}
```

Note:

Normalmente un thread è **joinable** ma si può **cambiare** anche in corso di esecuzione in **modalità detached**. I **joinable** liberano le risorse **quando viene fatto il join** invece i **detached** quando **terminano** (entrambi le **rilasciano** ad una **terminazione** del **processo principale**).

Conclusioni

QUEUES: Le code sono un metodo di comunicazione comodo per inviare e ricevere informazioni anche "complesse" tra processi generici.

THREADS: I "**thread**" sono una sorta di "**processi leggeri**" che permettono di **eseguire funzioni** "**in concorrenza**" in modo più semplice rispetto alla **generazioni di processi** veri e propri (**forking**).