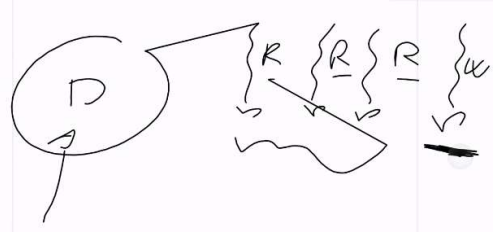
Sincronizzazione 10 Readers e writers

martedì 25 novembre 2025 16:27

Potrebbe sorgere un problema ora : data una struttura dati condivisa è possibile avere più thread che ci scrivono dati? Possibile avere più thread che vi scrivono ? La risposta è dipende : se in presenza di più thread che scrivono va bene (garantita la consistenza), ma basta che c'è almeno un thread che voglia leggere (writer) che si ha il problema della sezione critica.



Quindi per sincronizzare Readers e Writers : utilizziamo 2 distributori di gettoni , gestendo in modo opportuno le attività di ognuno (mantenendo accesso sempre possibile ad entrambi): si arriva alla sequenzializzazione dei writer (porta il token a 0 : i lettori non possono accedere alla struttura condivisa); per i lettori si ha che : se solo lettori ok , altrimenti permetto ai lettori di passare prima dei scrittori avendo al massimo n-1 lettori (lo scrittore non entra mai in sezione critica). Vediamo quindi un esempio :

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <unistd.h>

#define SIZE (100)
#define END (100000)
#define NUM_READERS 2
#define AUDIT if(0)

long *v;

int sem_write, sem_read;

void * producer(void){

        struct sembuf oper;

        long data = 0;
        long my_index = 0;
        pid_t pid;

        pid = getpid();
        printf("Process %d ready to produce\n",pid);

retry:
        oper.sem_num = 0;
        oper.sem_op = -1;
        oper.sem_flg = 0;
        semop(sem_write,&oper,1);

#ifndef SINGLE_LOCK
        oper.sem_op = -NUM_READERS;
        semop(sem_read,&oper,1);
#endif

        for(my_index=0;my_index<SIZE;my_index++){
                v[my_index] = data;
        }

        oper.sem_num = 0;
        oper.sem_flg = 0;

#ifndef SINGLE_LOCK
        oper.sem_op = NUM_READERS;
        semop(sem_read,&oper,1);
#endif
        oper.sem_op = 1;
        semop(sem_write,&oper,1);
        my_index = (my_index+1)%SIZE;
        data++;
        goto retry;
}
```

```c
void * consumer(void){

        long data = 0;
        long my_index = 0;
        long value;
        struct sembuf oper;
        pid_t pid;
        int rounds = 0;


        pid = getpid();
        printf("process %d ready to consume\n",pid);

retry:
        rounds++;
        oper.sem_num = 0;
        oper.sem_op = -1;
        oper.sem_flg = SEM_UNDO;
        semop(sem_write,&oper,1);

#ifndef SINGLE_LOCK
        semop(sem_read,&oper,1);

        oper.sem_op = 1;
        semop(sem_write,&oper,1);
#endif

        value = 0;
        for(my_index = 0; my_index<SIZE;my_index++){
                value += v[my_index];
        }

        AUDIT
        printf("consumer %d got value %ld\n",pid,value);

        if ((rounds) >= END){
                printf("ending condition met - last read summ is %ld - round is %d\n",value,rounds);
                exit(0);
        }

        oper.sem_num = 0;
        oper.sem_op = 1;
        oper.sem_flg = SEM_UNDO;

#ifndef SINGLE_LOCK
        semop(sem_read,&oper,1);
#else
        semop(sem_write,&oper,1);
#endif

        goto retry;

}
```

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

        int prod, cons;
        int i;
        key_t key = IPC_PRIVATE;

        sem_write = semget(key,1,IPC_CREAT|0666);
        if(sem_write == -1){
                printf("semget error\n");
                exit(-1);
        }
        semctl(sem_write,0,SETVAL,1);
        sem_read = semget(key,1,IPC_CREAT|0666);
        if(sem_read == -1){
                printf("semget error\n");
                exit(-1);
        }
        semctl(sem_read,0,SETVAL,NUM_READERS);
        v = (long*)mmap(NULL,SIZE*sizeof(long),PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_SHARED,0,0);
        if (v == NULL){
                printf("mmap error\n");
                exit(-1);
        }
        prod = fork();
        if (prod == -1){
                printf("fork on producer error\n");
                exit(-1);
        }
        if (prod == 0){
                producer();
        }
        i = 0;
respawn:
        cons = fork();
        if (cons == -1){
                printf("fork on consumer error\n");
                exit(-1);
        }
        if (cons == 0){
                consumer();
        }
        if(++i < NUM_READERS) goto respawn;

        for (i=0;i<NUM_READERS;i++){
                wait(NULL);
        }
        semctl(sem_write,IPC_RMID,0);
        semctl(sem_read,IPC_RMID,0);
        pause();
}
```