Sincronizzazione tra processi e shared memory

Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma
Luca Iezzi Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Reti di Calcolatori A.A. 2022/23

Prof.ssa Chiara Petrioli Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma
Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma

1. Sincronizzazione tra processi

Sincronizzazione tra processi

Finora i processi, una volta avviati, hanno eseguito il programma loro assegnato senza tenere conto dello stato dell'esecuzione altrui.

La comunicazione tra processi può essere considerata efficace se questi si trovano nello stato adeguato:

- Anonymous / named pipes
  Necessità dell'altro processo sull'estremità opposta di una pipe, altrimenti deadlock; operazioni bloccanti
- Socket Unix
  Fallimento delle operazioni se l'altro processo non è disponibile; operazioni sia bloccanti che non
- Message queue
  Comunicazioni sia bloccanti che non; più processi sia in lettura che scrittura su una stessa queue

Operazioni non bloccanti → importanza critica assicurarsi che i dati inviati siano effettivamente ricevuti

Sincronizzazione tra processi

Negli esempi di multi-processing visti:

- IPC bloccante
  - Con le pipes non ci sono alternative
  - Socket e message queues bloccanti
- Cooperazione tra processi limitata
  - Solo trasmissione dati
  - Un processo in attesa di dati necessari non può proseguire

Vera utilità del multi-processing:

- Risolvere sotto-problemi indipendenti di un problema comune
- Risolvere parzialmente un problema (ad es. a livello locale) in vista di un ricongiungimento di tutte le soluzioni parziali

Sincronizzazione tra processi: semafori Unix

Per sfruttare a pieno il multi-processing quindi è necessario condividere risorse (ad es. dati comuni) e controllarne l'accesso da parte dei processi.

I meccanismi di IPC visti finora possono essere usati per implementare il controllo dell'accesso, e generalmente più di 2 processi possono accedere in parallelo alle risorse.

Semaforo Unix

- È un contatore con garanzie di atomicità: leggere o aggiornare il valore del contatore è considerata una singola operazione
- L'accesso atomico al valore del semaforo assicura ad un processo che nessun'altra operazione possa interferire (ad es. leggere se il valore è pari ad 1 ed incrementarlo a 2)
- Più processi con accesso allo stesso semaforo possono quindi accedere senza equivoci ad una risorsa condivisa

Semafori Unix

Un processo implementa l'accesso ad una risorsa sfruttando un semaforo nel seguente modo:

1. Controlla il valore del semaforo
2. Se il valore è positivo → può accedere alla risorsa
   a. Decrementa di 1 il contatore → "una unità" della risorsa non è più disponibile perché attualmente in uso
3. Se il valore è zero → blocca l'esecuzione ed attende che la risorsa si liberi
   a. Un processo esce dall'attesa tramite un segnale inviato sul semaforo
   b. Una volta ripresa l'esecuzione, il processo torna al punto 1
4. Se ha avuto accesso alla risorsa ed ha terminato l'uso:
   a. Incrementa il contatore di 1 → "una unità" della risorsa è tornata disponibile per altri processi
   b. Eventuali processi in attesa saranno risvegliati e potranno tentare l'accesso alla risorsa condivisa

L'ordine di accesso alla risorsa non è specificato: un processo che ha rilasciato una risorsa potrebbe riacquisirla immediatamente, facendo andare di nuovo in attesa un processo svegliato appositamente per tentare l'accesso.

Uso di semafori Unix

Librerie: `sys/types.h`, `sys/ipc.h`, `sys/sem.h`

- `int semget(key_t key, int nsems, int semflg)`
  Crea o accede ad un semaforo, molto simile ad una message queue: identificatore del semaforo (`key`), il numero di semafori da creare (`nsems`, tipicamente 1) ed alcune flag (sono accettate le stesse di `open`: permessi Unix ecc.)
  Analogamente alle message queue, `IPC_PRIVATE` può essere fornito come chiave per autonomamente creare un semaforo da condividere con child processes; in alternativa si può usare `ftok` su un file noto ai processi che devono condividere il semaforo.
  Valori di ritorno: ID del semaforo se l'operazione ha successo, in alternativa −1

In realtà `semget` viene usata per ottenere un "insieme di semafori" (semaphore set):

- Raggruppa più di un semaforo in un'unica struttura
- Un processo può sfruttare l'insieme di semafori per implementare transazioni
- In questo contesto, per transazione si intende un insieme di operazioni su uno o più semafori di uno stesso insieme; tutte devono essere eseguite con successo, oppure nessuna viene effettivamente svolta

Uso di semafori Unix

- `int semctl(int semid, int semnum, int cmd, ...)`

  Permette l'esecuzione di vari comandi sul semaforo: per ogni comando viene garantita l'atomicità della sua esecuzione.
  Il comando `cmd` viene eseguito sul semaforo identificato dalla coppia `semid` e `semnum`: rispettivamente l'ID del semaphore set ed il numero dell'effettivo semaforo nell'insieme. Tipicamente viene usata per l'inizializzazione dei semafori tramite il comando `SETVAL`, e l'ultimo argomento (...) rappresenta il valore da usare.
  Valore di ritorno: interpretazione dipende da `cmd`; in caso di errori -1

```c
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short *array;
  struct seminfo *__buf;
};
```

Deve essere definita nel programma che invoca `semctl`: analogo a `msgbuf` per le message queue

```c
struct semid_ds {
  struct ipc_perm sem_perm;
  time_t sem_otime;
  time_t sem_ctime;
  unsigned long sem_nsems;
};
```

```c
struct ipc_perm {
  key_t __key;
  uid_t uid;
  gid_t gid;
  uid_t cuid;
  gid_t cgid;
  unsigned short mode;
  unsigned short __seq;
};
```

`struct` già definite

Uso di semafori Unix

- `int semctl(int semid, int semnum, int cmd, ...)`

  Permette l'esecuzione di vari comandi sul semaforo: per ogni comando viene garantita l'atomicità della sua esecuzione.
  Il comando `cmd` viene eseguito sul semaforo identificato dalla coppia `semid` e `semnum`: rispettivamente l'ID del semaphore set ed il numero dell'effettivo semaforo nell'insieme. Tipicamente viene usata per l'inizializzazione dei semafori tramite il comando `SETVAL`, e l'ultimo argomento (...) rappresenta il valore da usare.
  Valore di ritorno: interpretazione dipende da `cmd`; in caso di errori -1

```
1 union semun {
2   int val;
3   struct semid_ds *buf;
4   unsigned short *array;
5   struct seminfo *__buf;
6 };
```

Valore per il contatore del semaforo, da usare con comando `SETVAL`

Usati per altri comandi

Deve essere definita nel programma che invoca `semctl`: analogo a `msgbuf` per le message queue

Uso di semafori Unix

- `int semop(int semid, struct sembuf *sops, size_t nsops)`

  Usata per gestire l'accesso da parte dei processi ad una risorsa condivisa.
  In realtà esegue atomicamente un array di operazioni sul semaphore set il cui ID è `semid`; l'array è rappresentato dal puntatore `sops` e la dimensione dell'array è specificata da `nsops`.
  Ogni operazione su semaforo è rappresentata da una variabile di tipo `struct sembuf`:
  - `sem_num` → Numero del semaforo all'interno del semaphore set
  - `sem_flg` → Flag relative all'operazione
  - `sem_op` → Valore dipende dal tipo di operazione:
    - Positivo → Rilascio di risorse: viene sommato al contatore
    - Negativo → Richiesta di risorse: se maggiore del contatore, il processo va entra in attesa
    - `0` → Processo in attesa che il contatore diventi 0

Non entra in attesa se viene specificata la flag `IPC_NOWAIT`

```c
struct sembuf {
  unsigned short  sem_num;
  short     sem_op;
  short     sem_flg;
};
```

`struct` già definita

Uso di semafori Unix

Main: creazione e inizializzazione semaforo, setup processi

```c
int main(){
  signal(SIGCHLD, handle_sigchld);

  // Creazione di un semaphore set con 1 solo semaforo
  int semset_id = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600);
  if(-1 == semset_id){
    perror("Error during semaphore creation");
    exit(1);
  }

  // Definizione struct per inviare comandi ai semafori
  union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *__buf;
  };

  // Inizializzazione del semaforo a 0
  union semun sem_val = {.val = 0};
  int n = semctl(semset_id, 0, SETVAL, sem_val);
  if (-1 == n){
    perror("Error during semaphore initialization");
    exit(1);
  }
```

Il contatore del semaforo è inizializzato a 0: qualsiasi processo voglia accedere alla risorsa condivisa entrerà in attesa

```c
  pid_t child_pid = fork();
  switch(child_pid){
    case -1:{
      perror("fork() failed");
      exit(1);
    }

    case 0:{
      child_routine(semset_id);
      exit(0);
    }

    default: break;
  }

  // Parent process
  parent_routine(child_pid, semset_id);
  return 0;
}
```

Creazione dei processi e condivisione dell'ID del semaphore set: completamente analogo alle message queue

Uso di semafori Unix

Child process:

- Richiede l'accesso ad una risorsa condivisa
- Eventualmente entra in attesa
- Una volta che la risorsa è disponibile può proseguire con le proprie operazioni
- Al termine delle operazioni rilascia la risorsa

```c
void child_routine(int semset_id){
  // Tenta l'accesso ad "una unità" della risorsa controllata
  // dal semaforo 0 nel semaphore set
  struct sembuf sem_op = {0, -1, 0};
  int n = semop(semset_id, &sem_op, 1);
  if(-1 == n){
    perror("Child process failed semaphore operation");
    exit(1);
  }

  printf("Child process has had access to the resource!\n");
  sleep(2);

  // Libera "una unità" della risorsa al termine delle operazioni
  sem_op.sem_op = 1;
  n = semop(semset_id, &sem_op, 1);
  if(-1 == n){
    perror("Child process failed semaphore operation");
    exit(1);
  }

  printf("Child process: exit\n");
  exit(0);
}
```

Uso di semafori Unix

Parent process:

- Rende disponibile “un’unità” della risorsa condivisa
- Svolge delle operazioni
- Richiede l’accesso alla risorsa condivisa
- Eventualmente entra in attesa
- Entra in possesso della risorsa una volta disponibile e svolge le proprie operazioni
- Al termine delle operazioni rilascia la risorsa condivisa

```c
void parent_routine(pid_t child_pid, int semset_id){
  // Rende disponibile "una unità" della risorsa --> sblocca il child process
  struct sembuf sem_op = {0, 1, 0};
  int n = semop(semset_id, &sem_op, 1);
  if(-1 == n){
    perror("Parent process failed semaphore operation");
    kill(child_pid, SIGKILL);
    exit(1);
  }

  sleep(1);

  // Tenta l'accesso ad "una unità" della risorsa
  sem_op.sem_op = -1;
  n = semop(semset_id, &sem_op, 1);
  if(-1 == n){
    perror("Parent process failed semaphore operation");
    kill(child_pid, SIGKILL);
    exit(1);
  }

  printf("Parent process has had access to the resource!\n");
  sleep(2);

  // Libera "una unità" della risorsa
  sem_op.sem_op = 1;
  n = semop(semset_id, &sem_op, 1);
  if(-1 == n){
    perror("Parent process failed semaphore operation");
    exit(1);
  }

  printf("Parent process: exit\n");
  exit(0);
}
```

Considerazioni sui semafori

Vantaggi

- Le risorse gestite da semafori possono essere di ogni tipo
- Facile condivisione di un semaforo: basta l'ID del semaphore set
- Esagerati per semplici accessi mutex → utili invece per realizzare pattern producer-consumer
- Implementazione scala a grande numero di processi locali ed anche remoti

Attenzione

- Creazione di un semaforo ed inizializzazione
  Vìola il principio di atomicità: non c'è alternativa ad eseguirli in due tempi

2. Memory-mapped I/O

File I/O

Tutte le operazioni viste su file finora esplicitamente invocavano operazioni I/O su disco:

- Sequenze di (f)read e/o (f)write

Accedere ai contenuti di un file la cui posizione è nota:

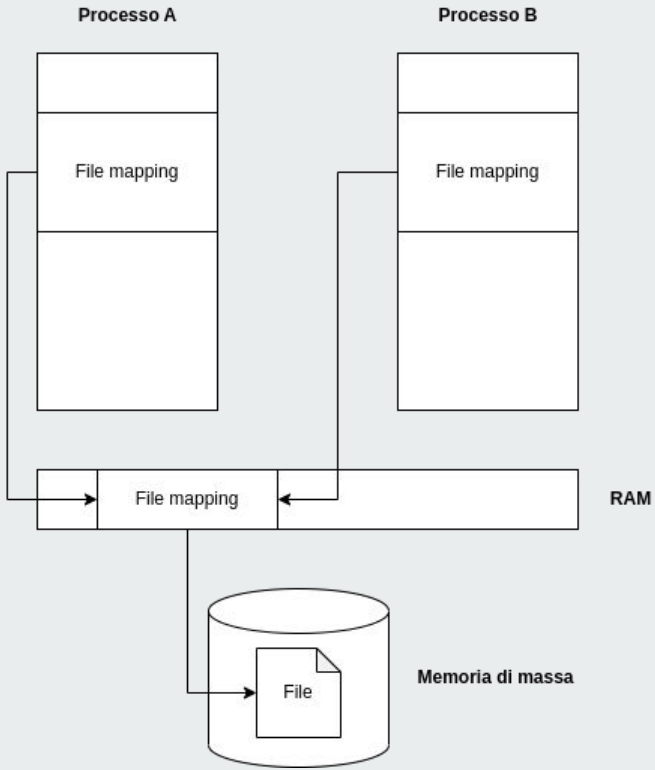
1. Apertura del file dall'inizio (o fine se in modalità append)
2. Per ogni contenuto di interesse:
   a. Spostamento del cursore alla posizione desiderata
   b. Operazioni I/O
3. Chiusura del file

In questo modo, alcune parti del file che non sono di interesse vengono comunque caricate.

Memory-mapped I/O

Invece di effettuare operazioni I/O su parti di file inutili, è possibile mappare un file in una regione di memoria:

- Il file diventa accessibile come se fosse un array
- Solo le porzioni effettivamente utilizzate vengono caricate dal disco
- Sfruttano lo stesso meccanismo del RAM paging
  - È quindi necessario allineare gli indirizzi alla dimensione delle pagine in uso dal sistema



Altri particolari:

- File mappati vengono ereditati dopo `fork`, ma non dopo `exec`
- Più processi possono condividere il mapping (antenato comune*)
- Più efficiente rispetto ad I/O tradizionale e molto veloce per IPC

Uso di memory-mapped I/O

Librerie: `unistd.h` e `sys/mman.h`

- `void* mmap(void *start, size_t len, int prot, int flags, int fd, off_t offset)`
  Crea un file mapping indicando `start` come indirizzo di inizio (può essere `NULL`), con `len` interpretato come dimensione del mapping. `prot` rappresenta la modalità di apertura del mapping: `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, o `PROT_NONE`. `flags` invece può assumere i valori:
  - `MAP_SHARED` → Il file sottostante viene modificato ed i child processes con accesso al mapping vedono le modifiche
  - `MAP_PRIVATE` → Tutte le modifiche sul mapping rimangono locali al processo, il file sottostante rimane inalterato
  - `MAP_FIXED` → Non allinea `start` alla dimensione di paging: alcune implementazioni necessitano l'allineamento

  `fd` rappresenta il file descriptor del file da mappare ed `offset` la distanza dall'inizio del file a cui fare riferimento.
  Valore di ritorno: indirizzo del mapping se l'operazione ha successo, altrimenti `MAP_FAILED`

Perché usare `PROT_NONE`

- Pagine di memoria guardia → protezione da heap buffer overflow
- Intervallo di memoria riservato → ad es. garanzia che `mmap` indipendenti siano contigue (da usare con flag `MAP_FIXED`)

Uso di memory-mapped I/O

Librerie: `unistd.h` e `sys/mman.h`

- `int munmap(void *addr, size_t len)`
  Rimuove il mapping di ogni pagina inclusa anche parzialmente nell'intervallo di memoria che comincia in `addr` e continua per `len` bytes
  Restituisce `0` se l'operazione ha successo, altrimenti `-1`
- `int msync(void *addr, size_t len, int flags)`
  Effettua eventuali scritture sul file sottostante il mapping
  `flags`:
  - `MS_ASYNC`
  - `MS_SYNC`
  - `MS_INVALIDATE`
    Invalida ogni copia in cache di dati inconsistenti

  Ritorna `0` se ha successo, altrimenti `-1`

3. Shared memory

Limiti di memory-mapped I/O

Memory-mapped I/O può essere usata come meccanismo di IPC:

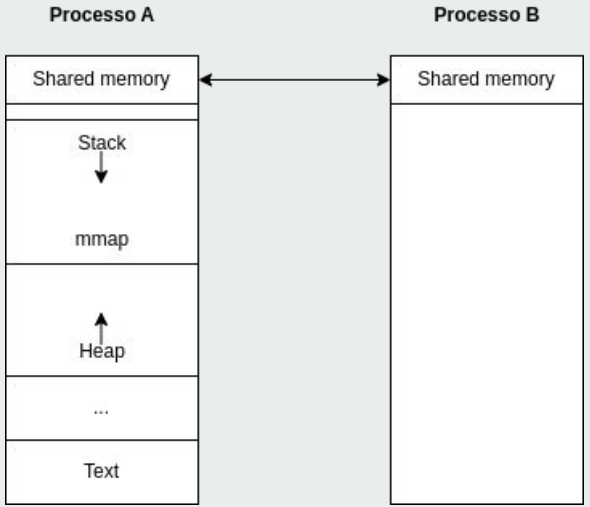
- ➤ Accesso regolato da semafori
- ➤ Porzioni di file dedicate rispettivamente a letture e scritture di un dato processo
- × Necessità di un file su disco

Shared memory

Per superare questo limite, due o più processi possono condividere direttamente un segmento di memoria.

Dopo `fork`, un child process riceve una copia della tabella delle pagine di memoria ma non il loro contenuto.
Le pagine vengono copiate solo al momento di un'eventuale modifica da parte del child process ma soprattutto la modifica rimane locale al child process.

- Molto efficiente se dopo `fork` viene eseguito `exec` per sostituire il programma del child process



Tuttavia, questo comportamento non è utile ai fini dell'IPC; soluzione:

1. Associare un'ID alle pagine da condividere
2. Evitare la copia della pagina e fare direttamente riferimento alla pagina del parent process

Uso di shared memory

Librerie: `sys/types.h`, `sys/ipc.h`, `sys/shm.h`

- `int shmget(key_t key, size_t size, int shmflg)`
  Restituisce un ID per un segmento di memoria condiviso; `key` ha lo stesso significato visto con `msgget` e `semget`, mentre `size` rappresenta la dimensione minima di shared memory (eventuale allineamento alla paging size). `size` può assumere il valore `0` se si sta accedendo ad un segmento condiviso già esistente.
  Analogamente ai semafori, le flag `shmflg` specificano permessi ecc.; tuttavia, la flag `IPC_EXCL` permette di ottenere il successo dell'operazione solo se il segmento non era esistente.
  Ritorna l'ID del segmento se l'operazione ha successo, altrimenti -1.
- `void* shmat(int shmid, const void *shmaddr, int shmflg)`
  "Attacca" un segmento condiviso con ID pari a `shmid` allo spazio gestito dal processo che invoca la funzione. Analogamente a `mmap`, `shmaddr` è l'indirizzo al quale attaccare il segmento in questione (consigliabile `NULL`) e `shmflag` sono flag operative.
  Ritorna l'indirizzo al quale è stato attaccato il segmento, altrimenti `(void *)-1` in caso di errori
- `int shmdt(const void *shmaddr)`
  "Stacca" un segmento condiviso presente al dato indirizzo `shmaddr`.
  Ritorna `0` se l'operazione ha successo, altrimenti -1

Uso di shared memory

Librerie: `sys/types.h`, `sys/ipc.h`, `sys/shm.h`

- `int shmctl(int shmid, int cmd, struct shmid_ds *buf)`
  Esegue il comando `cmd` sul segmento condiviso identificato da `shmid`.
  Alcuni comandi utili:
  - `IPC_STAT`
    Quanti processi hanno accesso al segmento condiviso
  - `IPC_RMID`
    Distrugge il segmento condiviso, anche se in uso da altri processi → avverrà quando tutti avranno effettuato il detach
    Il processo deve avere i permessi adeguati

  Ritorna `0` se l'operazione ha successo, altrimenti `-1`

Un segmento condiviso può essere utilizzato tramite il puntatore restituito da `shmat()` come un qualunque puntatore del processo stesso.

Combinarne l'uso con i semafori permette l'implementazione di un'efficace e flessibile meccanismo di IPC.

Considerazioni su shared memory

- ➤ Nessuna necessità di `malloc` → memoria è già allocata tramite `shmget`
- ➤ Aritmetica dei puntatori: l'unico modo per operare su un segmento condiviso
- ~ Il segmento condiviso deve contenere tutto il necessario affinché gli altri processi possano effettivamente svolgere operazioni su tali dati
  - *Esempio:* array disponibile tramite shared memory
  - La variabile contenente la dimensione dell'array dev'essere anch'essa sul segmento condiviso e non in una variabile locale ad un processo
- × No `malloc` o `realloc`? → dimensione condivisa fissa, non può essere aumentata
- × Allineamento degli indirizzi → i dati possono non allinearsi alla dimensione della pagina: errore `SIGBUS`

Accesso da parte di child processes

- ➤ Come per `mmap`, i segmenti condivisi vengono ereditati dopo `fork` ma non dopo `exec`
- ➤ Un processo lanciato da `exec` deve riaprire eventuali file mappati e segmenti condivisi per poterli usare, conoscendo gli opportuni parametri (ad es. nome file o ID) e possedendo i permessi adeguati

4. Esercizi

Esercizi

1. Riscrivere i programmi dell'esercizio 1 (Lezione 5) con le seguenti modifiche:
   a. Crea un nuovo file vuoto e ne esegue il mapping in memoria; ogni riga letta da un file esistente (ricevuto tramite `argv`) viene scritta sul mapping. Il processo riceve anche un semaphore set ID per gestire l'accesso al mapping.
   b. Apre tramite file mapping lo stesso file creato dal programma a e ne legge le righe da processare; l'accesso è regolato da un semaforo di cui si è ricevuto l'ID tramite `argv`. Inoltre, tramite `argv` viene ricevuto anche l'ID di un segmento condiviso ed il relativo semaforo per regolarne l'accesso; il segmento condiviso viene usato per scrivere i risultati del processing di una data riga.
   c. Questo programma fornisce il nome file che i programmi a e b useranno per il mapping, inizializza un segmento di memoria condivisa su cui avverrà la comunicazione con il programma b, e gestisce anche la creazione e l'inizializzazione dei semafori utilizzati per la sincronizzazione. Infine lancerà i processi che eseguono i due programmi precedenti.

Il resto delle operazioni rimane invariato rispetto all'esercizio originario.

Riferimenti

- http://pelusa.fis.cinvestav.mx/tmatos/LaSumA/LaSumA2_archivos/Paralelizacion/Unix%20Multi-Process%20Programming%20and%20Inter-Process%20Communications%20(IPC).htm
- `semget`, `semctl`, `semop`
- https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html
- `mmap`, `munmap`, `msync`
- `shmget`, `shmat`, `shmdt`, `shmctl`