

# Esercitazione [3]

## Sincronizzazione inter-Processo

Leonardo Aniello - [aniello@dis.uniroma1.it](mailto:aniello@dis.uniroma1.it)

Daniele Cono D'Elia - [delia@dis.uniroma1.it](mailto:delia@dis.uniroma1.it)

Sistemi di Calcolo - Secondo modulo (SC2)

Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2014-2015

# Sommario

- Soluzione esercizio sullo scheduler
- Gestione degli errori in C
- Obiettivi dell'esercitazione
- Sincronizzazione inter-processo

# Esercizio sullo Scheduler

- Al massimo  $N$  thread possono accedere in parallelo in sezione critica
- Una volta in sezione critica, ogni thread esegue un certo numero di work item e poi ne esce
- Esercizio  
Invece di eseguire tutti i work item di seguito, ogni thread deve
  - Eseguirne due alla volta (se sono dispari, all'ultima iterazione ne eseguirà solo uno)
  - Uscire dalla sezione critica
  - Rimettersi in coda per accedere alla sezione critica

# Soluzione Esercizio sullo Scheduler

- Esistono più modi di risolvere l'esercizio
- La soluzione proposta prevede
  - Un ciclo while con condizione  $i < \text{numero work item}$ , con  $i$  incrementato ad ogni iterazione
  - In ogni iterazione
    - Acquisizione risorsa
    - Esecuzione di un work item
    - Se non sono ancora stati eseguiti tutti i work item, esecuzione di un altro work item
      - Nell'ultima iterazione, non viene eseguito solo se il numero di work item è dispari!!
    - Rilascio risorsa
- Per verificare la correttezza della vostra soluzione, eseguite il codice sia con un numero di work item pari che con un numero dispari

# Gestione degli Errori in C

- In caso di errore, una funzione (system call o di libreria) ritorna un certo valore (es: -1)
- Può essere utile avere maggiori informazioni sull'errore stesso
- errno
  - Variabile di tipo intero definita in `errno.h`
  - In caso di errore, viene settata con un opportuno codice che consente di risalire alla causa dell'errore stesso
  - È possibile ottenere una descrizione del codice di errore tramite la funzione `strerror` definita in `string.h`

# Gestione degli Errori in C - Esempio

```
#include <errno.h>
#include <string.h>
...
int ret = function(arg1, ..., argN);
if (ret) {
    printf("Error: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}
```

- Per fini di debug, può essere necessario testare il codice di gestione dell'errore; due alternative
  - Forzare il lancio di un errore
    - Dipende dalla specifica della funzione, potrebbe essere complicato
  - Cambiare temporaneamente la condizione dell'`if` in `(!ret)`
    - Molto più semplice

# Obiettivi Esercitazione [3]

- Sincronizzazione inter-processo
  - Come usare i named semaphores per implementare meccanismi di sincronizzazione tra processi diversi
- Esercizio di riepilogo sugli argomenti trattati finora nella seconda parte

# Sincronizzazione inter-Processo

- Come sincronizzare processi diversi con i semafori?
  - Nella `sem_init` va impostato `pshared ≠ 0`
  - Per essere accessibile ad altri processi, il semaforo va allocato in un'area di memoria condivisa
    - Accedendo a quest'area di memoria, un altro processo può recuperare il puntatore al semaforo
- Una soluzione più semplice consiste nell'usare i `named semaphores`...

# Named Semaphore

- È identificato univocamente dal suo nome
  - Stringa (con terminatore) che inizia con uno slash (es. `/semaforo`)
  - Processi diversi riescono ad accedere allo stesso semaforo usando lo stesso nome
- È creato con `sem_open` invece che con `sem_init`
- Le operazioni `semWait` e `semSignal` sono eseguite allo stesso modo con `sem_wait` e `sem_post`
- Quando un processo ha terminato di lavorare con un semaforo, deve invocare `sem_close`
- Quando tutti i processi hanno terminato di lavorare con un semaforo, va invocata `sem_unlink`

# Named Semaphore in C - `sem_open` (1/3)

- Due possibili signature

- `sem_t *sem_open(const char *name, int oflag);`
- `sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);`

- Parametri

- Nome del semaforo (stringa con terminatore che inizia con /)
- Flag che controllano la open (definiti in `fcntl.h`)
  - `O_CREAT`: il semaforo viene creato se non esiste già
  - `O_CREAT|O_EXCL`: se il semaforo già esiste viene lanciato un errore
  - In caso di creazione, user e group ID del semaforo sono quelli del processo chiamante
- Se `O_CREAT` compare nei flag, vanno specificati altri due parametri.....

# Named Semaphore in C - `sem_open` (2/3)

- Il parametro `mode` specifica i permessi del semaforo
  - Maschera nella forma `0xyz`
    - `x` specifica i permessi per il proprietario
    - `y` specifica i permessi per il gruppo
    - `z` specifica i permessi per gli altri utenti
  - `x`, `y`, `z` sono costruiti «sommando» i seguenti valori
    - 0: nessun permesso
    - 1: permesso di esecuzione
    - 2: permesso di scrittura
    - 4: permesso di lettura
  - es: `0640` significa che
    - Il proprietario può leggere e scrivere
    - Gli utenti del gruppo possono solo leggere
    - Tutti gli altri non possono né leggere né scrivere

# Named Semaphore in C - `sem_open` (3/3)

- Per il `mode` è anche possibile usare le macro definite in `sys/stat.h`
  - es: `S_IRUSR` è il permesso di lettura per il proprietario
- L'ultimo parametro è il valore con cui inizializzare il semaforo stesso
  - Deve essere non negativo
  - Analogo all'omonimo parametro della `sem_init`
- Se viene specificato `O_CREAT` (non in OR con `O_EXCL`) e il semaforo già esiste, gli ultimi due parametri vengono ignorati

# Named Semaphore in C

`sem_close` e `sem_unlink`

- `sem_close` chiude il semaforo e consente al sistema di liberare tutte le risorse allocate al processo chiamante per quel semaforo
  - prende come argomento un puntatore al named semaphore «aperto» da `sem_open`
- `sem_unlink` rimuove il semaforo immediatamente; quando tutti i processi che avevano aperto il semaforo lo avranno chiuso, il semaforo verrà distrutto
  - prende come argomento la stringa che identifica univocamente il named semaphore
  - va utilizzata dopo che un semaforo è stato «chiuso»!

# Named Semaphore - Esercizio

- Scheduler con sincronizzazione inter-processo secondo il paradigma client-server
  - Semantica analoga allo scheduler della scorsa esercitazione
  - Il server crea il semaforo per consentire l'accesso in sezione critica al massimo a `NUM_RESOURCES` thread per volta
  - Il client lancia `THREAD_BURST` thread per volta che si sincronizzano tramite il semaforo creato dal server
- Sorgenti: `server.c` `client.c`
- Completare il codice nelle parti contrassegnate TODO!!
- Compilazione tramite `Makefile`
  - Il server richiede `util.c` e `util.h`
  - Client e server vanno entrambi linkati alla libreria `pthread`
- Esecuzione: lanciare prima il server, poi in un altro terminale avviare una istanza del client

# Funzione sem\_getvalue

- Consente di leggere il valore corrente di un semaforo
- `int sem_getvalue(sem_t *sem, int *sval);`
  - Parametri di input
    - Puntatore al semaforo
    - Puntatore ad un int che verrà settato al valore del semaforo
  - Ritorna 0 in caso di successo, -1 in caso di errore
- Se la coda di attesa del semaforo non è vuota, `*sval` su sistemi Linux sarà sempre settato a 0 o
  - su altri sistemi operativi POSIX-compliant può invece assumere valore negativo (pari al numero di thread in coda)

# Esercizio proposto (1/4)

- Esercizio di riepilogo su ciò che è stato visto finora
- Sviluppare un'applicazione in C con questa semantica
  - Il processo main crea N processi figlio tramite fork
  - Tutti i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main
  - L'attività dei processi figlio consiste nel lanciare M thread per volta
    - Competono per l'accesso in sezione critica, gestito dal processo main
    - Una volta in sezione critica, devono scrivere in append su un file l'identità del processo scrivente
  - Passati T secondi, il processo main deve notificare i processi figlio di cessare la loro attività e terminare (usare la `sem_getvalue`)
    - Prima di terminare, un processo deve attendere la fine dei thread attualmente in esecuzione
  - Infine, il processo main deve identificare il processo che ha effettuato più accessi in sezione critica

# Esercizio proposto (2/4)

- Processo main
  - Crea N processi figlio
  - Notifica gli N processi figlio di avviare la loro attività
  - Attende T secondi
  - Notifica gli N processi figlio di cessare la loro attività e terminare
  - Attende il termine degli N processi figlio
  - Identifica il processo che ha acceduto in sezione critica più volte
  - Termina

# Esercizio proposto (3/4)

- Processo figlio
  - Attende la notifica di avvio dal processo main
  - Ciclo
    - Lancia M thread
    - Attende il termine degli M thread
    - Verifica se il processo main ha notificato di cessare l'attività
      - In caso positivo, esce dal ciclo
  - Termina

# Esercizio proposto (4/4)

- Thread di un processo figlio
  - Richiede l'accesso in sezione critica
  - Una volta in sezione critica
    - Apre il file in append
    - Scrive l'identità del processo figlio
    - Chiude il file
    - Esce dalla sezione critica
  - Termina