POLITECNICO DI MILANO





# Sincronizzazione e IPC

**Laboratorio Software 2008-2009**
**M. Grotto F. Farina**

# Sommario

1. **Sincronizzazione e ItC**
   - Variabili globali
   - Mutex
   - Semafori
   - Condition variables
2. **Sincronizzazione e IPC**
   - Segnali
   - Memoria condivisa
   - Semafori
   - Mappedmemory
   - Pipe
   - Socket

# Sincronizzazione e ItC

## Variabili globali

- La maggior parte dei bug nei programmi multithread deriva dal fatto che più thread accedono agli stessi dati
  - L'operazione critica è la modifica
- Race condition
  - "Gara a chi modifica prima i dati"
  - Correttezza del funzionamento dipendente dallo scheduling dei thread
  - Errori tipici:
    - Più thread svolgono lo stesso lavoro
    - Segmentation fault
- Soluzione: rendere atomiche le operazioni

**Esempio**
`job-queue1.c`

# Sincronizzazione e ItC

## MUTEX

- **MUT**ual **EX**clusion locks
- Speciale “lucchetto” che consente ad un solo thread di avere accesso alla risorsa in mutua esclusione
  - Esempio: porta del bagno…
- Meccanismo BLOCCANTE
  - Se un thread tenta di “chiudere” un lucchetto già chiuso rimane in uno stato blocked fino alla sua apertura

## Uso dei MUTEX

- Usare il tipo di dato `pthread_mutex_t`
- Passare un puntatore a `pthread_mutex_init`
  - Il secondo argomento è un puntatore a un oggetto mutex attribute, `NULL` per i valori di default
- È possibile usare `PTHREAD_MUTEX_INITIALIZER`
- `pthread_mutex_lock` per acquisire il diritto d'accesso
  - BLOCCANTE se il mutex è già bloccato
- `pthread_mutex_unlock` per risvegliare i thread bloccati
  - Invocata dallo stesso thread che ha eseguito il lock

**Esempio**
`job-queue2.c`

# Sincronizzazione e ItC

## Possibili problemi e soluzioni

- Ad ogni lock far corrispondere un unlock
  - Se il thread termina prima di aver eseguito una unlock il mutex rimarrà bloccato
- Deadlock
  - Uno o più thread sono in attesa di un evento che non si verificherà mai
  - Caso classico: un thread esegue il lock di un mutex che esso stesso ha già bloccato
    - Dipende dal tipo di mutex scelto
- Possibilità di usare chiamate non bloccanti
  - `pthread_mutex_trylock` su un mutex bloccato restituisce `EBUSY`

# Sincronizzazione e ItC

## Tipi di MUTEX

- Fast mutex (default)
  - Possibilità di generare deadlock se il thread tenta di effettuare il lock di un mutex che ha già bloccato
- Recursive mutex
  - Tiene traccia del numero di lock
  - Necessario effettuare lo stesso numero di unlock per sbloccare il mutex
- Error-checking mutex
  - Una seconda chiamata a `pthread_mutex_lock` restituisce il valore `EDEADLK`
    - Generalmente usati in fase di debug

# Sincronizzazione e ItC

## Cambiare il tipo di mutex

- Si usa l'oggetto `pthread_mutexattr_t`
- Inizializzazione con `pthread_mutexattr_init`
- `pthread_mutexattr_setkind_np` per cambiare tipo di mutex
  - `PTHREAD_MUTEX_RECURSIVE_NP`
  - `PTHREAD_MUTEX_ERRORCHECK_NP`
- Si invoca `pthread_mutex_init` con il puntatore all'oggetto mutex attribute
- `pthread_mutexattr_destroy` per distruggere l'oggetto
- NP significa Not Portable: specifici di GNU/Linux

# Sincronizzazione e ItC

## Semafori per i thread

- Contatore usato per la sincronizzazione di thread
- Garanzia che la lettura e la modifica del valore di un semaforo avviene in maniera sicura
- Due operazioni base
  - Wait: decrementa di 1 il valore del semaforo. Se il valore è 0 l'operazione è BLOCCANTE
  - Post: incrementa di 1 il valore del semaforo, causando il risveglio di eventuali thread bloccati in wait
- Due implementazioni
  - POSIX per la comunicazione tra thread
  - Implementazione per la comunicazione tra processi

# Sincronizzazione e ItC

## Uso dei semafori

- Si usa il tipo `sem_t`
- Inizializzazione con `sem_init`
  - Il secondo parametro dovrebbe essere 0
    - Non supportata la condivisione di questo tipo di semafori in GNU/Linux
  - Il terzo argomento è il valore iniziale
- `sem_wait` attende che un semaforo assuma un valore non nullo, quindi lo decrementa
  - `sem_trywait` restituisce `EAGAIN` se il valore è 0
- `sem_post` incrementa il valore del semaforo specificato
- `sem_getvalue` per leggere il valore del semaforo
  - Non usare il valore letto per prendere decisioni
    - Possibilità di race condition

**Esempio**
`job-queue3.c`

# Sincronizzazione e ItC

## Condition variables

- Terzo meccanismo di sincronizzazione per l'implementazione di condizioni di esecuzione più complesse
  - Quando la condizione non è verificata il thread non rimane in "polling" ma va in uno stato blocked
  - Quando la condizione è segnalata il thread è risvegliato
- Simile ai semafori nel meccanismo di attesa
- Differente dai semafori poichè non ha memoria
  - Se la condizione è segnalata ma nessun thread è in attesa il segnale è perso

# Sincronizzazione e ItC

## Possibile scenario

- Il thread controlla un flag di esecuzione
  - Se non è impostato si mette in attesa della condition variable
- Il thread che imposta la condition variable prima modifica il flag, poi segnala la condizione
- Problema: race condition
  - Se il primo thread è interrotto dallo scheduler durante il controllo…
- Soluzione: lock del flag e della condition variable con un singolo mutex
  - In GNU/Linux ogni condition variable è usata congiuntamente ad un mutex

## La soluzione in pratica

- Lock del mutex e controllo del valore
- Se impostato, unlock del mutex ed esecuzione
- Se non impostato, unlock del mutex ed attesa della condizione di esecuzione
  - Possibile problema se queste ultime operazioni non sono svolte atomicamente
    - Un altro thread potrebbe cambiare il valore del flag e segnalare la condizione tra il test e l'attesa
    - GNU/Linux consente di svolgere il terzo passo in maniera atomica

# Sincronizzazione e ItC

## Uso delle condition variables

- Si usa il tipo `pthread_cond_t`
  - Accompagnata da un `pthread_mutex_t`
- Inizializzazione con `pthread_cond_init`
  - Il secondo argomento è ignorato in GNU/Linux
  - Necessario inizializzare il mutex separatamente
- `pthread_cond_signal` per segnalare la condizione
  - `pthread_cond_broadcast` per sbloccare TUTTI i thread in attesa della specifica condizione
- `pthread_cond_wait` per rimanere in attesa
  - Il secondo argomento è il mutex da sbloccare
  - Quando la condizione è segnalata il lock viene ripreso

**Esempio**
`condvar.c`

## Segnali

- Alternativa a `sigaction`: uso della system call `signal` per installare un nuovo handler per il segnale
  - `SIG_IGN, SIG_DFL`, handler specifico
  - Restituisce un puntatore al vecchio handler o `SIG_ERR`
  - Comportamento non standard
    - In alcuni casi l'handler è resettato al valore di default
    - Preferibile usare `sigaction`
- Possibilità di mascherare i segnali annidati
  - `sigemptyset, sigaddset, sigdelset`
  - Operano sul campo `sa_mask di sigaction(sigset_t)`
- `pause` per rimanere in attesa di un segnale

# Sincronizzazione e IPC

## Memoria condivisa

- Possibilità di condividere una zona di memoria tra più processi
  - Meccanismo di paginazione
- In fase di creazione, un processo alloca il segmento, tutti gli altri lo aprono (attach)
  - Se si tenta di allocare un segmento esistente si ottiene un riferimento ad esso
  - I segmenti sono allocati come multipli interi della dimensione di pagina definita dal sistema
    - In Linux solitamente è 4KB, `getpagesize();`
- In fase di eliminazione, tutti i processi lo chiudono (detach), un processo deve deallocare il segmento

## Creazione di un segmento

- `shmget(…);` per l'allocazione (**S**hared **M**emory **GET**)
  - Necessario specificare una chiave intera per l'accesso
    - `IPC_PRIVATE` per garantire univocità
  - Restituisce il `shmid`
  - Necessario specificare la dimensione del segmento
    - Automaticamente arrotondata (paginazione)
  - Il terzo argomento è un or bitwise di opzioni:
    - `IPC_CREAT` per creare il segmento
    - `IPC_EXCL` per abortire se il segmento esiste
    - Mode flags: permessi d'accesso simili a quelli per i file
      `S_IRUSR, S_IWUSR, … (sys/stat.h)`
      Permessi di esecuzione ignorati

## Attach e detach

- `shmat(…);` (**S**hared **M**emory **AT**tach)
  - Necessario specificare la chiave d'identificazione
  - Puntatore che specifica dove mappare il segmento
    - Con `NULL` GNU/Linux ne sceglie uno disponibile
  - Terzo argomento
    - `SHM_RND` per l'allineamento automatico con la pagina
    - `SHM_RDONLY` per indicare che sarà solo letto
  - Restituisce il puntatore al segmento
- `shmdt(…);` (**SH**ared **M**emory **D**e**T**ach) per la chiusura
  - Automatico con chiamata a `exit();` o `exec();`
  - Segmento rimosso se già deallocato e l'ultimo processo che lo usa invoca la chiusura

# Sincronizzazione e IPC

## Controllo e rimozione

- `shmctl(…);` (**S**hared **M**emory **C**on**T**ro**L**)
  - Per ottenere informazioni passare `IPC_STAT` come secondo parametro ed un puntatore a `shmid_ds`
  - Per rimuovere il segmento passare `IPC_RMID` come secondo parametro e `NULL` come terzo
    - Rimosso quando l'ultimo processo richiede detach
- I segmenti vanno deallocati esplicitamente per evitare di raggiungere il numero massimo di segmenti
  - `ipcs -m` per vedere i segmenti di memoria condivisa
  - `ipcrm shm <id>` per rimuovere un segmento

**Esempio**
`shm.c`

# Sincronizzazione e IPC

## Memoria condivisa

- Meccanismo di IPC più semplice e più veloce
- Possibilità di comunicazione bidirezionale
  - Rischio di race conditions: stabilire un “protocollo d’accesso”
    - Non è garantito accesso esclusivo neanche con `IPC_PRIVATE`
- Il problema dello scambio della chiave…

# Sincronizzazione e IPC

## Semafori e processi

- Anche chiamati System V semaphores
- I semafori per processi sono organizzati in insiemi
- `semget(…);` per l'allocazione
  - Chiave di identificazione
  - Numero di semafori dell'insieme
  - Flag per i permessi
- `semctl(…);` per il controllo e la rimozione
  - Necessaria deallocazione esplicita per non saturare il sistema
    - A differenza della memoria condivisa i semafori vengono deallocati immediatamente

**Esempio**
`sem_all_deall.c`

## Inizializzazione

- Necessario definire una `union semun`
- Si invoca `semctl(…);`
  - `0` come secondo parametro
  - `SETALL` come terzo
  - Quarto parametro: si crea una `union semun` e si riferisce il campo array ad un vettore di valori `unsigned short`

**Esempio**
`sem_init.c`

## Operazioni sui semafori

- Chiamata di sistema `semop(…);` sia per `wait` che per `post`
- Necessario creare un vettore di strutture `sembuf`
  - `sem_num`: numero del semaforo
  - `sem_op`: intero che specifica l'operazione sul semaforo
    - Se positivo è aggiunto al valore corrente
    - Se negativo, il valore assoluto è sottratto
    - Se questa operazione renderebbe negativo il valore del semaforo il processo è bloccato fino a quando il valore del semaforo non consente l'operazione
    - Se è zero il processo si blocca fino a quando il semaforo non vale 0

## Operazioni sui semafori

- `sem_flag`
  - `IPC_NOWAIT` per evitare il blocco
    - Se la chiamata porterebbe il processo in uno stato di blocco non viene eseguita e restituisce un errore
  - `SEM_UNDO` per far in modo che GNU/Linux annulli automaticamente tutte le operazioni sul semaforo quando il processo termina
    - Se il processo termina (volontariamente o no) il valore del semaforo è ripristinato annullando gli effetti del processo sul semaforo stesso
- `ipcs -s` per informazioni sugli insiemi di semafori
- `icprm sem <id>` per rimuovere un insieme

**Esempio**

`sem_pv.c`

# Sincronizzazione e IPC

## Mapped memory

- Consente la comunicazione tramite un file condiviso
- Differente dalla memoria condivisa
  - Usata sia per IPC che per accedere al contenuto di un file
- Associazione tra un file e la memoria di un processo
  - Il contenuto è diviso in chunks di dimensione pari a quella delle pagine di memoria ed è caricato in memoria virtuale
  - Lettura e scrittura in memoria
  - Il SO gestisce lettura e scrittura sul file trasparentemente

## mmap

- `mmap(…);` (**M**emory **MAP**ped) per effettuare il mapping
  - `#include <sys/mman.h>`
  - Primo parametro: indirizzo al quale mappare il file
    - NULL permette a Linux di scegliere il primo disponibile
  - Secondo parametro: lunghezza del mapping in bytes
  - Terzo parametro: flag di protezione
    - `PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE`
  - Quarto parametro: opzioni addizionali
  - Quinto parametro: file descriptor aperto
  - Sesto parametro: offset dall'inizio del file con il quale effettuare il map
  - Restituisce l'indirizzo del mapping o `MAP_FAILED`

## `mmap`: opzioni addizionali

- `MAP_FIXED`
  - Usa l'indirizzo specificato per il mapping
  - Deve essere allineato (page-aligned)
- `MAP_PRIVATE`
  - Le scritture saranno effettuate su una copia privata del file, non su quello originale (copy-on-write)
  - Nessun altro processo si accorge delle scritture
  - Da non usare con `MAP_SHARED`
- `MAP_SHARED`
  - Le scritture sono effettuate IMMEDIATAMENTE sul file originale, non bufferizzate (IPC)

## `munmap`

- Necessario effettuare l'unmapping: `munmap(…);`
- Due parametri:
  - Indirizzo di inizio del mapping
    - Multiplo della dimensione delle pagine
  - Lunghezza della sezione
- Ulteriori referenze ad indirizzi interni al range generano "invalid reference" (`SIGSEGV`)
- Unmapping automatico quando il processo termina
- Unmapping NON automatico quando si chiude il file descriptor

**Esempio**

`mmap-write.c`

`mmap-read.c`

## mmap

- Mapped memory con `mmap(…);` viene preservata a fronte di una `fork(…);`, con gli stessi attributi
- `MAP_SHARED` quando più processi accedono allo stesso file
- Linux supporta altri flag oltre a quelli POSIX
  - `MAP_DENYWRITE` (storico)
    - Scritture falliscono con `ETXTBUSY` (attacchi DoS)
  - `MAP_ANONYMOUS` (`MAP_ANON`, deprecato)
    - Non associata ad un file, i parametri fde offset sono ignorati
  - `MAP_32BIT`
    - Posiziona il mapping nei primi 2GB dello spazio di indirizzi del processo (ignorata con `MAP_FIXED`)

## `msync`

- Linux potrebbe bufferizzare le scritture
- `msync(…);` forza il flush delle scritture sul file
  - Primi due parametri analoghi a `munmap`
  - Terzo parametro:
    - `MS_ASYNC`: l'aggiornamento è schedulato ma non necessariamente eseguito prima che la funzione termini
    - `MS_SYNC`: aggiornamento immediato, bloccante
    - `MS_INVALIDATE`: richiesta di invalidazione degli altri mapping per l'aggiornamento con le nuove modifiche
  - Restituisce `EINVAL` o `ENOMEM`

# Sincronizzazione e IPC

## Mapped memory

- Necessario stabilire un protocollo d'accesso per evitare race conditions(semaforo)
- Usata non solo per IPC
  - Strutture dati (`struct`) in memoria mappata
  - Alla terminazione del programma ho un file con tutte le strutture
  - Al successivo riavvio del programma, rimappando il file in memoria riottengo tutta la struttura
    - Necessario mappare allo stesso indirizzo altrimenti i puntatori perdono di significato
  - Mapping dei file speciali (`/dev/zero`)

# Sincronizzazione e IPC

## Pipes

- Mezzo di comunicazione UNIDIREZIONALE e SERIALE
- Solitamente usato per la comunicazione tra thread o tra processi padre e figlio
- Esempio lampante: simbolo | della shell
  - Connette lo stdout del primo processo allo stdin del secondo
- Capacità di memorizzazione limitata
  - Se si tenta di scrivere in una pipe piena o di leggere da una pipe vuota si va in blocked
    - Sincronizzazione automatica

## Creazione pipes

- Si invoca `pipe(…);`
- L'argomento è un array di interi di dimensione 2
  - `0`: file descriptor di lettura
  - `1`: file descriptor di scrittura
  - I dati scritti in 1 vengono riletti tramite 0
  - I descrittori creati sono validi solo all'interno del processo corrente e dei suoi figli
- `fdopen(…);` per convertire i file descriptors in `FILE*`
  - Possibilità di usare le API di alto livello (`printf(…);`, `fgets(…);`)

## Redirezione

- Possibilità di redirigere `stdin`, `stdout` e `stderr` di un programma
- Chiamata a `dup2(…);`
  - File descriptor identici condividono la STESSA posizione all'interno del file e gli stessi flag
    - Non condividono il flag `close-on-exec`
  - `STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO`
    - `dup2(fd, STDIN_FILENO)` per redirigere lo standard input

**Esempio**
`dup2.c`

# Sincronizzazione e IPC

## `popen` e `pclose`

- Eliminano la necessità di invocare `pipe(…);`, `fork(…);`, `dup2(…);`, `exec(…);`, `fdopen(…);`
- `popen(…);` crea una pipe, esegue la `fork(…);` ed invoca la shell
  - Comando da eseguire (passato a `/bin/sh` con `-c`)
  - Operazione (lettura, "`r`", o scrittura, "`w`")
  - Restituisce un'estremità della pipe o `NULL`
  - L'altra estremità è connessa allo `stdin` del processo figlio
- `pclose(…);` attende la terminazione del processo figlio
  - Restituisce lo stato d'uscita

**Esempio**
`popen.c`

## FIFO

- Pipe con un nome nel filesystem (named pipes)
- Comunicazione tra processi non in relazione
- Comando `mkfifo` per la creazione
  - Si cancella come un file normale
- Funzione `mkfifo(…);`
  - Da includere: `sys/types.h`, `sys/stat.h`
  - Primo parametro: percorso di creazione
  - Secondo parametro: permessi
  - Restituisce -1 se non si può creare la pipe

**Esempio**

## Utilizzo delle FIFO

- Si usano come file normali
  - Un processo la apre in scrittura, l'altro in lettura
  - Sia tramite primitive di basso livello (`open(…);`, `close(…);`, …) sia funzioni di alto livello (`fopen(…);`, `fprintf (…);`, `fscanf(…);`, …)
- Si possono avere più processi che scrivono/leggono
- Dati scritti atomicamentefino a `PIPE_BUF`
  - 4KB in Linux
  - Scritture/letture simultanee possono risultare in sezioni sovrapposte (interleaving)

**Esempio**
`fifo.c`