Software Lab

Synchronization, IPC and ItC

Roberto Farina
roberto.farina@cefriel.it

Summary

- Synchronization and ItC
  - Global variables
  - Mutex
  - Semaphores
  - Condition variables
- Synchronization and IPC
  - Signals
  - Shared memory
  - Semaphores
  - Mapped memory
  - Pipes
  - Sockets

Race condition

- Most of bugs in multithread programs happens because different threads access same data
  - The critical operation is the modification
- Race condition
  - “Race to be the first who modifies data”
  - Correctness depends on the thread scheduling
  - Typical errors:
    - Different threads do the same job
    - Segmentation fault
- Solution: make operations atomic

Mutex

- MUTual EXclusion locks
- Special “lock” that allows a single thread to access a resource in mutual exclusion
  - E.g.: bathroom door
- BLOCKING mechanism
  - If a thread tries to “close” an already closed lock it goes in a blocked state until the lock is reopened

Mutex use

- Use the data type `pthread_mutex_t`
- Pass a pointer to `pthread_mutex_init`
  - The second argument is a pointer to a mutex attribute object; NULL for default values
- It is possible to use `PTHREAD_MUTEX_INITIALIZER`
- `pthread_mutex_lock` to acquire the access right
  - BLOCKING if the mutex is already locked
- `pthread_mutex_unlock` to unlock the mutex waking up blocked threads
  - It must be invoked by the same thread which acquired the lock

Problems and solutions

- Every lock must have the corresponding unlock
  - If a thread terminates before unlocking a mutex, it will remain blocked
- Deadlock
  - One or more threads are waiting for an event that will never occur
- It is possible to use non blocking calls
  - `pthread_mutex_trylock` on a blocked mutex returns `EBUSY`

Three mutex types

- Fast mutex (default)
  - Can generate deadlock if the thread tries to lock a mutex it has already blocked
- Recursive mutex
  - It keeps track of the number of locks
  - It is necessary to execute the same number of unlock to free the mutex
- Error-checking mutex
  - A second call to **`pthread_mutex_lock`** returns **`EDEADLK`**

Changing mutex type

- Use the `pthread_mutexattr_t` object
- Initialize it with `pthread_mutexattr_init`
- `pthread_mutexattr_setkind_np` to change the mutex type
  - `PTHREAD_MUTEX_RECURSIVE_NP`
  - `PTHREAD_MUTEX_ERRORCHECK_NP`
  - `pthread_mutexattr_settype`
- Invoke `pthread_mutex_init` with the pointer to the mutex attribute object
- `pthread_mutexattr_destroy` to destroy the object
- NP means Not Portable: GNU/Linux specific!

Thread semaphores

- Counter used for thread synchronization
- It is granted that reading and writing a semaphore value is secure
- Two basic operations
  - Wait: decrease the semaphore value by 1. If the value is 0, the operation is BLOCKING
  - Post: increase the semaphore value by 1, causing the awakening of a blocked thread if any
- Two implementations
  - POSIX for thread communication
  - Specific for process communication

Semaphore usage

- Use the data type `sem_t`
- Initialize with `sem_init`
  - The second parameter should be 0
    - GNU/Linux doesn't support sharing of this kind of semaphore
  - The third argument is the initial value
- `sem_wait` to wait for a semaphore
  - `sem_trywait` returns `EAGAIN` if the value is 0
- `sem_post` to increase the semaphore value
- `sem_getvalue` to read the semaphore value
  - Don't use the value to take decisions

Condition variables

- A synchronization mechanism to implement more complex execution conditions
  - When the condition is false, the thread doesn't poll but transits in a blocked state
  - When the condition becomes true, the thread is awakened
- Similar to semaphores in the waiting mechanism
- Different from semaphores because it doesn't have memory
  - If the condition is signaled but there's no thread waiting the signal is lost

Possible scenario

- The thread checks the execution flag
  - If it is not set, waits for the c.v.
- The thread which sets the c.v., modifies the flag and then it signals the new condition
- Problem: race condition
  - If the first thread is interrupted by the scheduler during control…
- Solution: lock on the flag and the c.v. with a single mutex
  - In GNU/Linux, every c.v. is used in conjunction with a mutex

Practical solution

- Mutex lock and check of the value
- If it is set, mutex unlock and execution
- If it is not set, mutex unlock and wait for the execution condition
  - Possible problem if those operations are not performed atomically
    - Another thread could change the flag value and signal the condition between the test and the wait
    - GNU/Linux allows to perform this step atomically

Condition variables

- Use the data type `pthread_cond_t`
  - In conjunction with `pthread_mutex_t`
- Initialization with `pthread_cond_init`
  - The second argument is ignored in GNU/Linux
- `pthread_cond_signal` to signal the condition
  - `pthread_cond_broadcast` to unblock EVERY thread that is waiting for the specific c.v.
- `pthread_cond_wait` to wait a c.v.
  - The second argument is the mutex to unlock
  - When the condition is signaled, the lock is gained again

Summary

- Synchronization and ItC
  - Global variables
  - Mutex
  - Semaphores
  - Condition variables
- Synchronization and IPC
  - Signals
  - Shared memory
  - Semaphores
  - Mapped memory
  - Pipes
  - Sockets

Signals

- `sigaction` alternative: use the `signal` system call to install a new signal handler
  - SIG_IGN, SIG_DFL, specific handler
  - Returns a pointer to the old handler or SIG_ERR
  - Non standard behavior
    - In some cases the handler is reset to the default value
- It is possible to mask nested signals
  - `sigemptyset`, `sigaddset`, `sigdelset`
    - Work on the `sa_mask` field of sigaction (`sigset_t`)
- `pause` to wait for a signal

Shared memory

- It is possible to share a memory region between processes
- Upon creation, a process allocates the segment, other processes open it (attach)
  - Trying to allocate an existing segment results in a reference to it
  - Segments are integer multiples of the page dimension defined by the system
    - In Linux usually it is 4KB, `getpagesize()`
- Upon deletion, every process closes the segment (detach), a single process has to deallocate it

Creation of a segment

- `shmget` to allocate it (SHared Memory GET)
  - It is necessary to specify an integer key to be able to access it
    - IPC_PRIVATE to grant unique keys
  - Returns the shmid
  - It is necessary to specify the segment dimension
    - Automatically rounded (paging)
  - The third argument is a bitwise OR of options:
    - IPC_CREAT to create the segment
    - IPC_EXCL to abort if the segment already exists
    - Mode flags: access rights similar to those for files
      - S_IRUSR, S_IWUSR, … (`sys/stat.h`)
      - Exec rights ignored

Attach and detach

- `shmat` (SHared Memory ATtach)
  - It is necessary to specify the id key
  - Pointer to specify where to map the segment
    - NULL to let Linux choose an available one
  - Third argument
    - SHM_RND to automatically align with the page
    - SHM_RDONLY to indicate it will be only read
  - Returns a pointer to the segment
- `shmdt` (SHared Memory DeTach) to close
  - Automatic when calling `exit` or `exec`
  - The segment is removed if it is already deallocated and the last process closes it

Control and removal

- `shmctl` (SHared Memory ConTroL)
  - To obtain information use IPC_STAT as the second param and a pointer to `shmid_ds` as the third one
  - To remove the segment use IPC_RMID as second param and NULL as the third one
    - Removed when the last process requests detach
- Segments must be explicitly deallocated to avoid reaching the max number of segments
  - `ipcs -m` to view shared memory segments
  - `ipcrm shm <id>` to remove a segment

Shared memory

- Probably the quickest and simplest IPC mechanism
- Suitable for bidirectional communication
  - Risk of race conditions: an “access protocol” must be established
    - Exclusive access not granted even with IPC_PRIVATE
- The problem of id key exchange…

Process semaphores

- Also called System V semaphores
- They are organized in sets
- `semget` to allocate them
  - Identification key
  - Number of semaphores in the set
  - Flags
    - IPC_CREAT, IPC_EXCL, …
- `semctl` to control and remove them
  - It is necessary to explicitly deallocate them to not saturate the system
    - Differently from the shared memory, semaphores are immediately deallocated

Initialization

- It is necessary to define a `union semun`
- Invoke `semctl`
  - semid as the first parameter
  - 0 as the second parameter
  - SETALL as the third one
  - Fourth parameter: create a `union semun` with the `array` field referring to an array of `unsigned short`

Operations on semaphores

- `semop` system call for both wait and post operations
- It is necessary to create a vector of `sembuf` structures
  - `sem_num`: semaphore number
  - `sem_op`: integer that specifies the operation on the semaphore
    - If positive, it is added to the current value
    - If negative, the absolute value is subtracted
      - If this operation would cause the semaphore value to become negative, the process is blocked until the semaphore value allows the operation to be performed

Operations on semaphores

- `sem_flag`
  - IPC_NOWAIT to avoid to be blocked
    - If the call will take the process to a blocked state, it is not performed and an error is returned
  - SEM_UNDO to let Linux undo every operation on the semaphore when the process terminates
    - If the process terminates (cleanly or not) the semaphore value is restored undoing the effects of the process
- `ipcs -s` for info on semaphore sets
- `icprm sem <id>` to remove a set

Mapped memory

- It allows the communication through a shared file
- Different from shared memory
  - Used both as IPC and to access the content of a file
- Association between a file and a process' memory
  - The content is divided in chunks with dimension equal to memory pages and it is loaded in virtual memory
  - Memory read and write operations
    - The OS handles read and write operation on the file transparently

mmap

- `mmap` (Memory MAPped) to make the mapping
  - `#include <sys/mman.h>`
  - First parameter: address to map the file to
    - NULL lets Linux choose the first one available
  - Second parameter: mapping length in bytes
  - Third parameter: protection flag
    - PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
  - Fourth parameter: additional options
  - Fifth parameter: opened file descriptor
  - Sixth parameter: offset from the beginning of the file to which make the map
  - It returns the mapping address or MAP_FAILED

munmap

- It is necessary to remove the mapping: **`munmap`**
- Two parameters:
  - Mapping starting address
  - Section length
- Further references to addresses within the range generate “invalid reference” (SIGSEGV)
- Automatic unmapping when the process terminates
- **NO** automatic unmapping when closing the file descriptor

mmap: additional options

- MAP_FIXED
  - Use the specified address for the mapping
  - It must be page-aligned
- MAP_PRIVATE
  - Write operations will be performed on a private copy of the file, not on the original one (copy-on-write)
  - Other processes are not aware of modifications
  - To not be used with MAP_SHARED
- MAP_SHARED
  - Shares the mapping with other processes that map the object
  - Write operations are performed on the original file (`msync` or `munmap` to be sure the file is updated)

mmap: additional options

- Mapped memory with **`mmap`** is preserved upon a **`fork`**, with same attributes
- Linux supports its own flags, not just POSIX ones
  - MAP_DENYWRITE
    - Write operations fail with ETXTBUSY (DoS attacks)
  - MAP_ANONYMOUS (MAP_ANON, deprecated)
    - Not associated to a file, fd and offset params are ignored
  - MAP_32BIT
    - Mapping in the first 2GB of the process address space
  - MAP_LOCKED
    - Lock the pages of the mapped region into memory (mlock)

msync

- Linux could buffer write operations
- `msync` forces the flush on the file
  - First two parameters similar to `munmap`
  - Third parameter:
    - MS_ASYNC: the update is scheduled but not necessarily executed before the function terminates
    - MS_SYNC: immediate update, blocking
    - MS_INVALIDATE: invalidation request for other mappings to update with fresh values
  - It returns EINVAL or ENOMEM in case of errors

Mapped memory

- It is necessary to establish an access protocol to avoid race conditions (semaphore)
- Not only for IPC
  - Data structures (`struct`) in mapped memory
  - When program terminates there's a file with all the data structures
  - When the program starts again, data structures are immediately available remapping the file
    - It is necessary to map to the same address to not invalidate pointers
  - Mapping of special files (`/dev/zero`)

Pipes

- UNIDIRECTIONAL and SERIAL communication mechanism
- Usually used between threads or related processes
- | symbol in the shell
  - It connects the stdout of the first process to the stdin of the second one
- Limited memory
  - Writing on a full pipe or reading from an empty pipe results in a blocked state
    - Automatic synchronization

Creation

- Invoke **`pipe`**
- The argument is an integer array of dimension 2
  - 0: reading file descriptor
  - 1: writing file descriptor
  - Data written in 1 are read through 0
  - The descriptors created this way are valid in the current process and in its sons
- **`fdopen`** to convert file descriptors in **`FILE*`** in order to use the high level API (**`printf`**, **`fgets`**)

Redirection

- It is possible to redirect stdin, stdout and stderr streams of a program
- Call to **`dup2`**
  - Identical file descriptors share the SAME position within the file and the same flags
    - They don't share the close-on-exec flag
  - STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
    - **`dup2(fd, STDIN_FILENO)`** to redirect the standard input

popen and pclose

- To avoid to call `pipe`, `fork`, `dup2`, `exec`, `fdopen`
- `popen` creates a pipe, executes the `fork` and invokes the shell
  - Command to be executed
  - Operation (read, `"r"`, or write, `"w"`)
  - It returns one end of the pipe or NULL
    - The other end is connected to the stdin of the child process
- `pclose` waits for child termination
  - It returns the exit status

FIFO

- Pipe with a name in the filesystem (named pipes)
- Communication between unrelated processes

- `mkfifo` command for the creation
  - Can be cancelled as a normal file

- `mkfifo` function
  - `sys/types.h`, `sys/stat.h`
  - First parameter: creation path
  - Second parameter: rights
  - It returns -1 if the pipe cannot be created

FIFO usage

- They are used as normal files
  - A process opens it for writing, the other for reading
  - Both low level primitives (`open`, `close`, …) and high level ones (`fopen`, `fprintf`, `fscanf`, …) can be used
- More than one process as reader/writer
- Data are atomically written up to PIPE_BUF
  - 4KB in Linux
  - Simultaneous read/write operations can result in interleaving sections

Differences with Win32

- In Windows, named pipes are similar to sockets
  - It is possible to connect processes on different machines
- It is possible to avoid interleaving
- It is possible to make a bidirectional communication
- Only WinNT can create named pipes, Win9x can only create client connections

Close-on-exec

Process 1 → `fork()` → Process 1

File descriptor 5
open
With close-on-exec

File descriptor 5
open/shared

Process 2 → `exec()` → Process 2

File descriptor 5
open/shared

File descriptor 5
closed/not shared

Bibliography

- *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*, Prentice Hall, 1998
- *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications*, Prentice Hall, 1999
- http://mij.oltrelinux.com/devel/unixprg/

