Process spawning and synchronisation basics.

Jiří Zahradník

Digiteq Automotive

2024

What is a process

- General definition
  - A series of actions or steps taken in order to achieve a particular end.
- Definition in computing
  - An instance of a computer program, executed by one or more threads.

POSIX and QNX processing

Both POSIX and QNX use so called `POSIX thread model` AKA `pthreads`. This contains API for:

- Thread management
- Mutexes
- Condition variables
- Synchronisation primitives (RW lock, barriers)
- Spinlocks (excellent for quick thread swapping, bad for prolonged waits)

Some synchronisation methods and it's uses

- Spinlock
- Mutex
- Semaphore (Not included in `pthread`)
- Condition variable
- Monitor

Spinlock - Simplest synchronisation primitive.

Uses active waiting loop and atomic operations upon an atomic flag. E.g. `test_and_set`, `clear` upon `std::atomic_flag`

Pros

- Simple implementation.
- Lightweight implementation - does not take lot of instructions to perform.
- Good for very short critical sections and quick thread swapping.

Spinlock - Simplest synchronisation primitive.

Cons

- Uses busy loop wait.
- Bad for prolonged waits, since the thread is active, takes resources from other threads.
- Does not synchronise order of operations.
- To use in interprocess blocking, must be created in shared memory - not used unless necessary.

Spinlock - example

```
class SpinLock {
    std::atomic_flag locked = ATOMIC_FLAG_INIT ;
public:
    void lock() {
        while (locked.test_and_set(
                std::memory_order_acquire
                )
                ) { ; }
    }
    void unlock() {
        locked.clear(std::memory_order_release);
    }
};
```

Spinlock - example

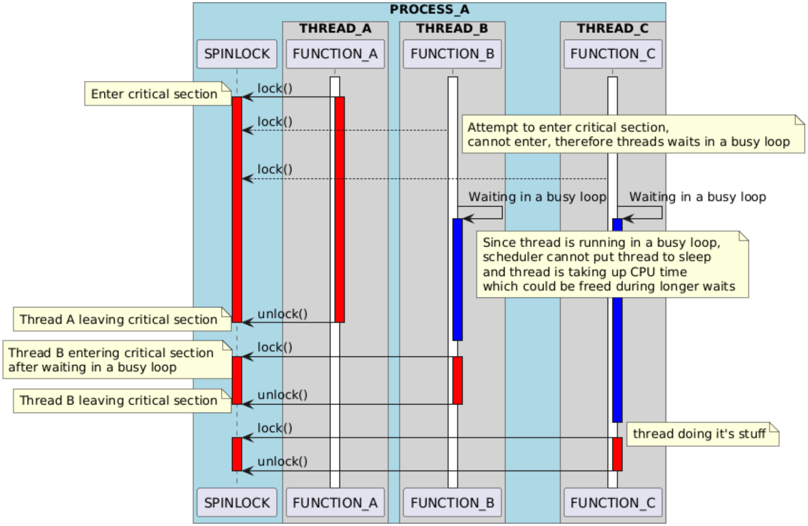

PROCESS_A
THREAD_A
THREAD_B
THREAD_C
SPINLOCK
FUNCTION_A
FUNCTION_B
FUNCTION_C
Enter critical section
lock()
lock()
Attempt to enter critical section,
cannot enter, therefore threads waits in a busy loop
lock()
Waiting in a busy loop
Waiting in a busy loop
Since thread is running in a busy loop,
scheduler cannot put thread to sleep
and thread is taking up CPU time
which could be freed during longer waits
Thread A leaving critical section
unlock()
Thread B entering critical section
after waiting in a busy loop
lock()
Thread B leaving critical section
unlock()
lock()
thread doing it's stuff
unlock()
SPINLOCK
FUNCTION_A
FUNCTION_B
FUNCTION_C

Figure: Spinlock sequence diagram

Mutex

Mutual exclusion mechanism to secure critical section.

Pros

- Easy usage with excellent abstraction.
- Easy usage in single process, multi threaded applications.
- Goes well with condition variable.
- (C++) Is accepted by various objects like `std::lock_guard`, `std::unique_lock`, `std::shared_lock`, `std::lock` etc.

Cons

- Only owning thread can unlock the mutex, unlocking in different thread leads to undefined behaviour.
- A bit biger overhead, but still nothing that should concern us unless we are building real time low power embedded systems.
- For interprocess comunication requires special regime.

Mutex - example

```
// entering critical section
mutex.lock(); // if mutex is locked, thread blocks
... // do stuff with shared resource
mutex.unlock();
```

Mutex - better example utilizing RAII in C++

```
foo function() {
    // entering critical section
    // Resource Allocation Is Initialisation.
    // Acquiring mutex locks it
    // and going out of scope unlocks it
    // in destructor.
    std::lock_guard<std::mutex> lock(mutex);

    ... // do stuff with shared resource

} // destructor releases the mutex (unlocks)
```

Mutex - example

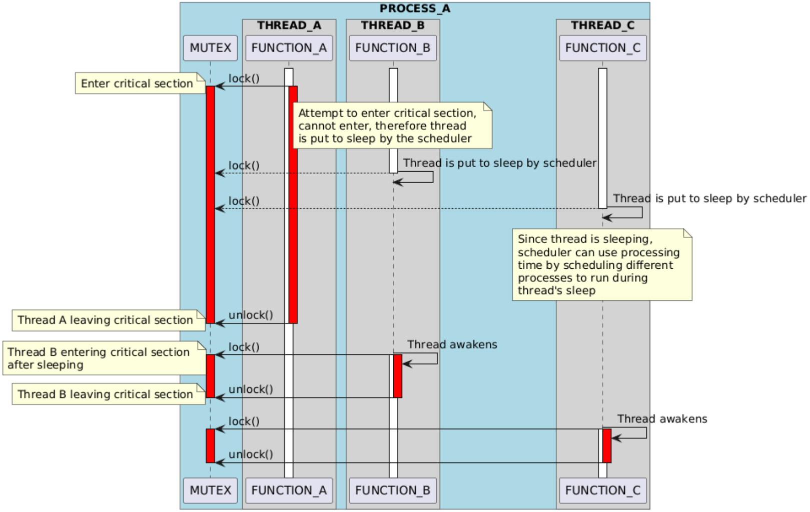

PROCESS_A
THREAD_A
THREAD_B
THREAD_C
MUTEX
FUNCTION_A
FUNCTION_B
FUNCTION_C
Enter critical section
lock()
Attempt to enter critical section, cannot enter, therefore thread is put to sleep by the scheduler
lock()
Thread is put to sleep by scheduler
lock()
Thread is put to sleep by scheduler
Since thread is sleeping, scheduler can use processing time by scheduling different processes to run during thread's sleep
Thread A leaving critical section
unlock()
Thread B entering critical section after sleeping
lock()
Thread awakens
Thread B leaving critical section
unlock()
lock()
Thread awakens
unlock()
MUTEX
FUNCTION_A
FUNCTION_B
FUNCTION_C

Figure: Mutex sequence diagram

Mutex - std::shared_mutex

Contains 2 locks: `exclusive` and `shared`

- Acquiring exclusive lock prevents threads from acquiring both shared and exclusive locks.
- Acquiring shared lock prevents acquiring the exclusive lock, but allows acquiring more shared locks.
- Great solution for readers - writers problem.

Semaphore

- Signaling primitive to synchronise multiple threads and processes.
- Consists of a counter, mutex and wait queue.
- Binary semaphore vs generic semaphore.
- Named semaphore vs unnamed semaphore.
- 2 operations
  - `P()` acquires semaphore (e.g. `sem_wait()`).
  - `V()` releases semaphore (e.g. `sem_post()`).
- A semaphore, by definition, does not have an owner.

IMPORTANT!

Do not mistake binary semaphore for mutex.

Semaphore pt.2

Pros

- Does not have an owner. Therefore one thread can `wait`, second can `post`.
- Easy to store named semaphore in shared memory to ensure interprocess communication.
- Named semaphore can synchronise two completely different processes which do not have to share same ancestor.
- We can decide between binary semaphore or general semaphore.
- Can protect a resource from being used by more than N processes/threads.
- Native C++ support as of `C++20`

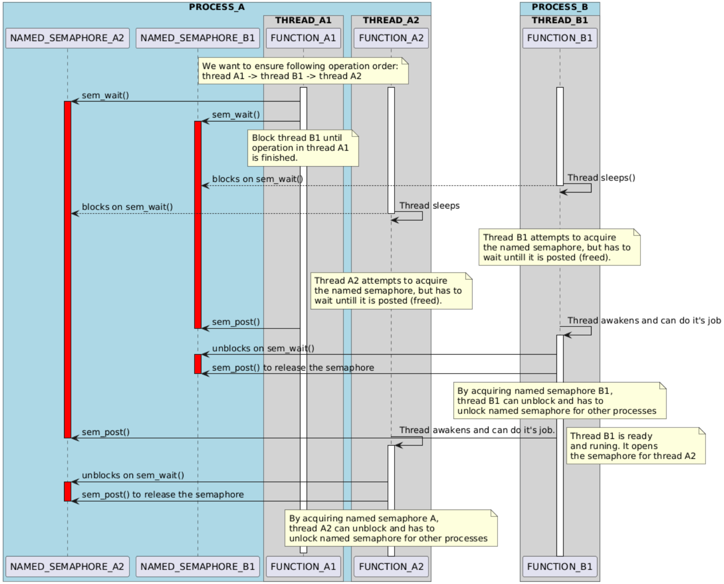
Semaphore pt.3

Cons

- Complexity - challenging to use correctly.
- Priority inversion - tasks with greater priority can be superseded by tasks with lower priority - important in real time tasks.
- They are expensive - CPU time and memory requirements.
- Requirement for correct access control list.

Priority inversion almost ended the Mars Pathfinder mission in 1977. More here.
More on how to solve priority inversion later in the presentation.

Semaphore example



PROCESS_A
THREAD_A1
THREAD_A2
PROCESS_B
THREAD_B1
NAMED_SEMAPHORE_A2
NAMED_SEMAPHORE_B1
FUNCTION_A1
FUNCTION_A2
FUNCTION_B1
We want to ensure following operation order: thread A1 -> thread B1 -> thread A2
sem_wait()
sem_wait()
Block thread B1 until operation in thread A1 is finished.
blocks on sem_wait()
Thread sleeps()
blocks on sem_wait()
Thread sleeps
Thread B1 attempts to acquire the named semaphore, but has to wait untill it is posted (freed).
Thread A2 attempts to acquire the named semaphore, but has to wait untill it is posted (freed).
sem_post()
Thread awakens and can do it's job
unblocks on sem_wait()
sem_post() to release the semaphore
By acquiring named semaphore B1, thread B1 can unblock and has to unlock named semaphore for other processes
sem_post()
Thread awakens and can do it's job.
Thread B1 is ready and runing. It opens the semaphore for thread A2
unblocks on sem_wait()
sem_post() to release the semaphore
By acquiring named semaphore A, thread A2 can unblock and has to unlock named semaphore for other processes
NAMED_SEMAPHORE_A2
NAMED_SEMAPHORE_B1
FUNCTION_A1
FUNCTION_A2
FUNCTION_B1

Condition variable

- Synchronisation primitive to enable threads to wait until a certain condition is satisfied.
- Used with mutexes to protect critical section.
- Primary use is in multithreading, not multiprocessing.
- Basically works as a queue for waiting threads.
- Is crucial for construction of monitors.
- In some cases can be better than semaphore.

Condition variable pt.2

Pros

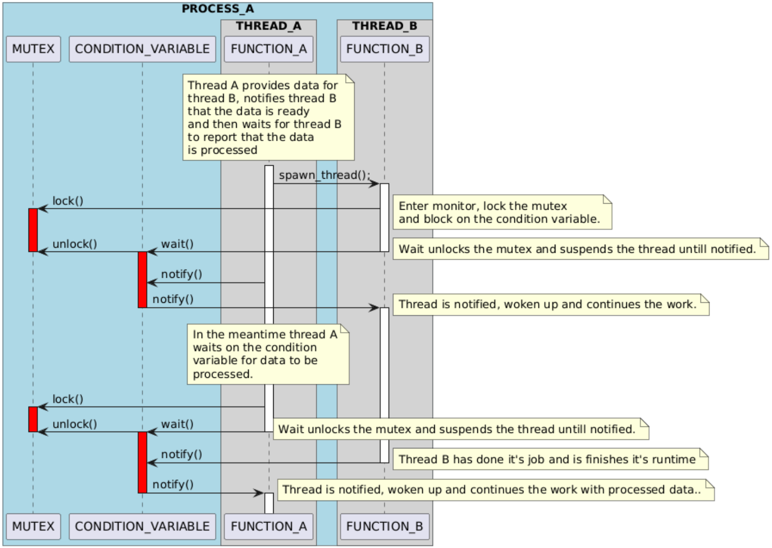
- Simple synchronisation of threads.
- Combined with mutex solves consumer/producer problem.
- Native use in C/C++.
- User defined condition for wakeup.
- Can be used to wake up all threads.
- Solves spurious wakeups.

Condition variable pt.3

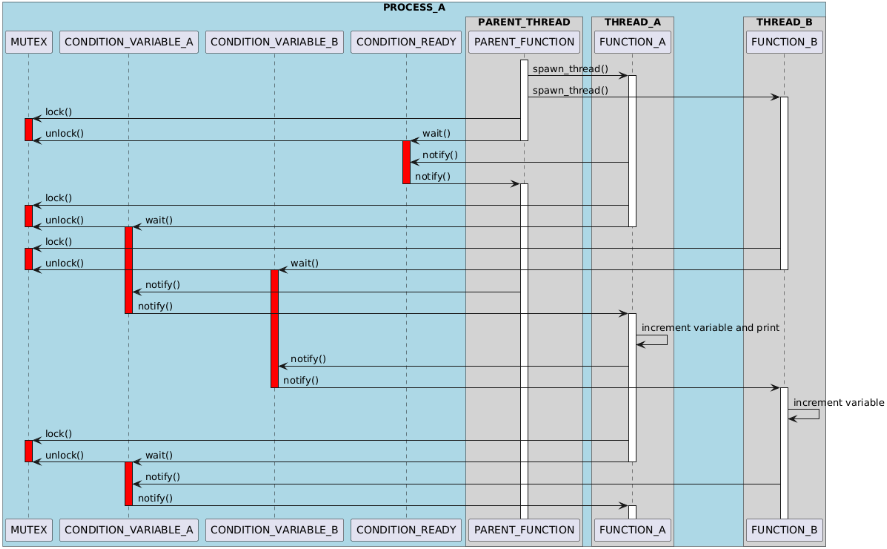
Cons

- Can be only used in monitors.
- Primarily for multithreading synchronisation, multiprocess synchronisation requires further configuration and must be paired with shared mutex.
- `std::condition_variable` available only for multithreading.
- C version also available for multiprocessing, but requires further configuration.

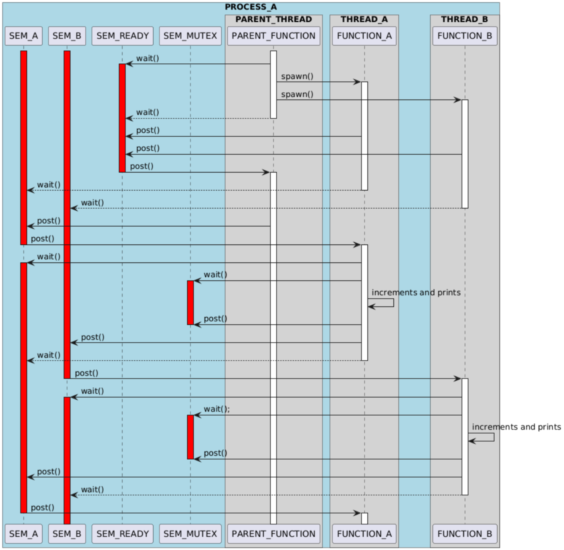
Condition variable example


PROCESS_A
THREAD_A
THREAD_B
MUTEX
CONDITION_VARIABLE
FUNCTION_A
FUNCTION_B
Thread A provides data for thread B, notifies thread B that the data is ready and then waits for thread B to report that the data is processed
spawn_thread();
lock()
Enter monitor, lock the mutex and block on the condition variable.
unlock()
wait()
Wait unlocks the mutex and suspends the thread untill notified.
notify()
notify()
Thread is notified, woken up and continues the work.
In the meantime thread A waits on the condition variable for data to be processed.
lock()
unlock()
wait()
Wait unlocks the mutex and suspends the thread untill notified.
notify()
Thread B has done it's job and is finishes it's runtime
notify()
Thread is notified, woken up and continues the work with processed data..
MUTEX
CONDITION_VARIABLE
FUNCTION_A
FUNCTION_B

Condition variable vs Semaphore for synchronisation



Condition variable vs Semaphore for synchronisation



Monitor

- High level abstraction.
- Protection and synchronisation mechanism.
- Great for encapsuling protected data.
- Uses Mutex and at least one Condition variable to protect data inside.

KEEP IN MIND

Rules still apply for mutex and condition variable.

Spawning a new process.

Common ways to spawn a process

- `fork()`
- `fork() + exec()`
- `clone()`
- `clone() + exec()`
- `posix_spawn() == clone() + exec()`

Spawning a new process.

Fork() and fork() + exec()

- `fork()` clones entire process using Copy on Write method, including parent process' memory map - in reality it's `clone()` with small preset overhead.
- `exec()` overwrites old memory map with new binary's memory map.

Spawning a new process.

clone() and clone() + exec()

- `clone()` permits more precise control over what pieces of execution context are shared.

Spawning a new process.

spawn_posix()

- wrapper for `clone()` and `exec()` with advantages and disadvantages.
    - Pros
      Scheduling configuration passed as an argument
    - Cons
      AFAIK core affinity configuration not supported (I didnt find it in documentation)

Spawning a new thread

OS creates new execution context within a running process which shares code section and memory. Stack, registers including program counter are sole property of the thread and are not shared. This is called Thread Control Block

Possible issues.

Deadlock

Definition

Deadlock is any situation in which no member of some group of entities can proceed because each waits for another member, including itself, to take action, such as sending a message or, more commonly, releasing a lock.

Typical representation

E.g. when two diggers share single showel, but digger A holds the showel, but can start digging after digger B finishes his hole, for which digger B needs the exact showel.

Deadlock prevention

Invalidate Coffman conditions

For deadlock to occur, following necessary conditions must be satisfied.

- Mutual exclusion.
- No preemption.
- Hold and wait.
- Circular wait.

Race condition

Definition

Condition where the system's substantive behaviour is dependent on the sequence or timing of other uncontrollable events, leading to undefined behaviour.

Typical representation

Data race during reader/writer problem.

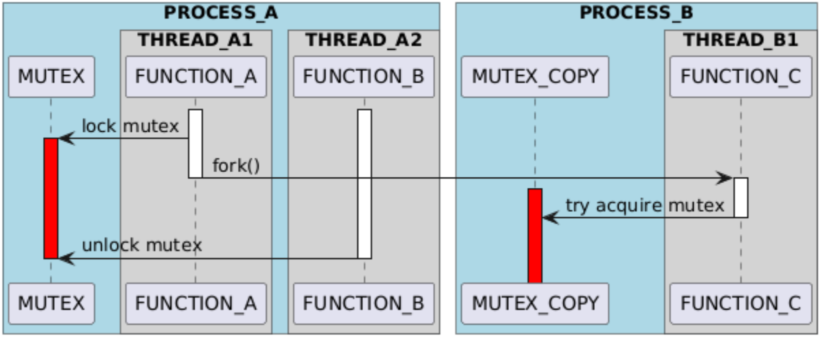
Priority inversion

Priority inversion is a bug that occurs when a high priority task is indirectly preempted by a low priority task. For example, the low priority task holds a mutex that the high priority task must wait for to continue executing.

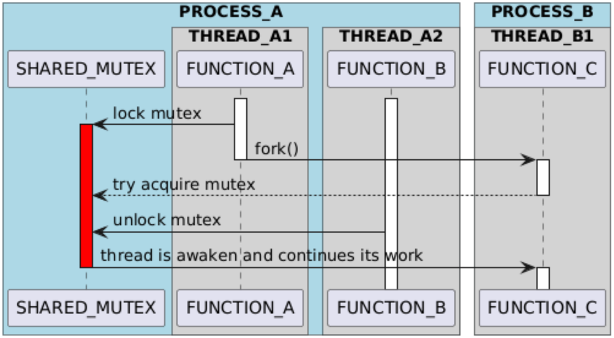
Solutions

- Disabling all interrupts.
- Priority ceiling
- Priority Inheritance
- etc.

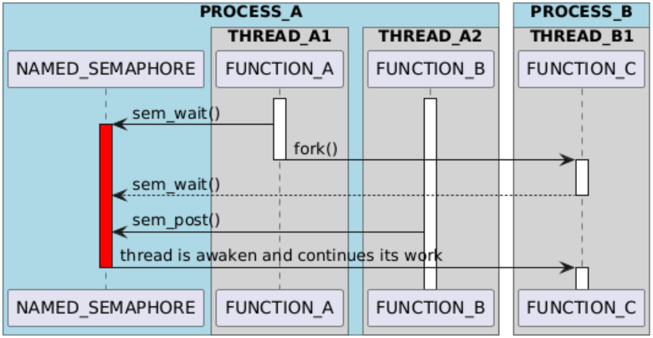
Using wrong synchronisation primitive pt. 1



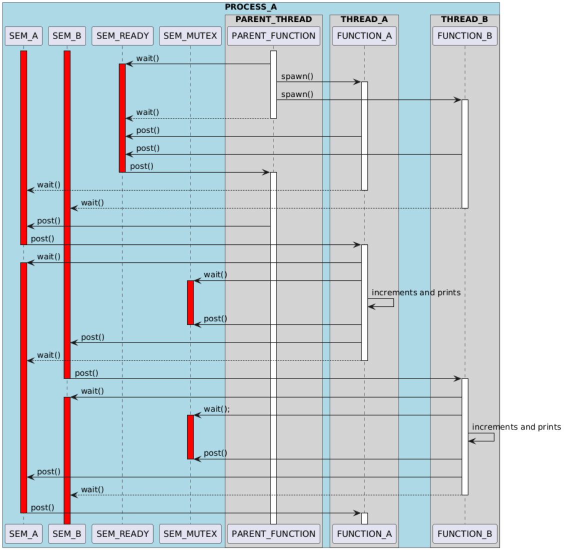
Using wrong synchronisation primitive pt. 2



Using correct synchronisation primitive pt. 1



Using correct synchronisation primitive pt. 2


PROCESS_A
PARENT_THREAD
THREAD_A
THREAD_B
SEM_A
SEM_B
SEM_READY
SEM_MUTEX
PARENT_FUNCTION
FUNCTION_A
FUNCTION_B
wait()
spawn()
spawn()
wait()
post()
post()
post()
wait()
wait()
post()
post()
wait()
wait()
increments and prints
post()
post()
wait()
post()
wait()
wait();
increments and prints
post()
post()
wait()
post()