Synchronization Primitives

From OSDev Wiki

All the techniques presented here are basic building blocks to address the problem of *process synchronization*. E.g. given programs that are running independently from each other on the same machine, how can one ensure some properties about what combination of operations are allowed and what combinations are not.

Among other examples of real-world problems, we're looking for technique that can grant:

- Mutual exclusion of processes: a portion of code cannot be executing by two process simultaneously.
- Rendezvous: one process must not perform one operation (e.g. generating a summary) before other processes have completed their operations.
- Shared readers/Single writer approach of resource locking: many process may be reading a table at the same time, but only one can write at a time and it should prevent readers to access the table until the table has returned to a consistent state.

  Note: A good synchronization implementation should not only guarantee **correctness***, but also* **fairness** *(all process have equal chance to get the access) and* **non-starvation** *(any waiting process will eventually have the resource).*

Contents

Semaphores

Semaphores are one of the oldest and most widely used methods of ensuring Mutual Exclusion between two or more processes. A semaphore is a special integer variable which is (usually) initialized to 1, and can only altered by a pair of functions. Each of these functions, historically called *p* and *v* (from the Dutch words *proberen*, to try, and *verhogen*, to increment), must be an Atomic operation. Each semaphore has an associated queue for processes waiting on the resource it guards.

- The function *p*, also called `wait()` (or `test()`), decrements the value of the semaphore, and if the semaphore is negative, puts the process on the waiting queue until the semaphore is released by the process holding it.

- The function *v*, also called `signal()` (or `release()`), increments the semaphore and, if it is still negative, indicates to the scheduler to wake the next waiting process in the queue.

Note that a general semaphore can do much more than just guaranteeing mutual exclusion. Some FIFO queue (single reader and single writer) can for instance be implemented by using one semaphore counting "how many messages are available" and another one counting "how many free slots are available"

```
Message queue[N];
Semaphore slots=new Semaphore(N);
Semaphore messages=new Semaphore(0);
int last_read=0, last_written=0;

Message get() {
  Message m;
  messages.wait();
  m=queue[last_read]; last_read=(last_read+1)%N;
  slots.signal();
  return m;
}

void put(Message m) {
  slots.wait();
  queue[last_written]=m; last_written=(last_written+1)%N;
  messages.signal();
}
```

Mutexes

A variant on this, called a *binary semaphore* uses a boolean value instead of an integer. In that case, *p* tests the value of the semaphore, and if it is true, sets it to false, and if false, waits. The binary *v* function checks the waiting queue, and if it is empty, set the semaphore to true; otherwise, it indicates to the scheduler to wake the next queued process.

In either form, it is important that a process release a semaphore once it has finished using the resource it guards, otherwise the resource could be left inaccessible.

Note that while "semaphore" is a globally-unique semantic items, "mutex" is a fuzzy name and system designers tends to have "their own mutex" which may look more like a spinlock or like a binary semaphore or like a general semaphore...

Spinlocks

Spinlocks try to address the same problem of Mutual Exclusion, but without relying on a scheduler infrastructure to make the process sleep if the resource is busy. Instead, a spinlock will keep checking the value until it has changed and usually relies on some atomic `test_and_set` instruction on the CPU to perform its task (See Intel Manuals (http://developer.intel.com/design/pentium4/manuals/index_new.htm) to see how `xchg` can be used to mimmic `test_and_set` virtual operation).

While poorly used spinlocks will lead to severe performance penalty in single-cpu systems, wise use on multi-cpu may achieve higher throughput.

If you need more information on spinlocks, you're suggested to walk through these documents:

- Spinlocks 1 (http://osdev.berlios.de/spinlock_part1.html)
- Spinlocks 2 (http://osdev.berlios.de/spinlock_part2.html)
- Spinlocks 3 (http://osdev.berlios.de/spinlock_part3.html)

***Note for IA-32 programmers*: If you consider to use spinlocks, be aware that the P4 / Xeon CPUs will falsely detect a possible memory order violation as the spinloop finishes, resulting in an large additional performance penalty. Place a PAUSE instruction into the spinlock to avoid this undesirable behavior. Refer to the Intel Manuals (http://developer.intel.com/design/pentium4/manuals/index_new.htm) for more information.**

See Also

Threads

- Userland only Semaphores
- Spinlocks that disable interrupts
- SMP compatibility
- Mutex Implementation
- Mutexs, Spinlocks and all that jazz
- Spinlocks & Semaphores

Retrieved from "http://wiki.osdev.org/index.php?title=Synchronization_Primitives&oldid=12886"
Categories: IPC | Synchronization

- This page was last modified on 2 March 2012, at 07:04.
- This page has been accessed 32,984 times.