

# Synchronization without locks

M1 MoSIG : Operating Systems

Poupin Pierre

Rouby Thomas

18/11/2014

Synchronization primitives are costly :

- spinlocks :
  - atomic instruction that locks the memory bus
  - active wait
- semaphore/monitor :
  - risk of being unscheduled : cost of switch
  - entering in the kernel

Notice that actual implementation of locks in system such as Linux is a mix between spinlock and unscheduling of the thread.

Other solutions :

- algorithmic changes : no more critical sections

Example : Producer/Consumer

	size buffer head	
Producer		Consumer
while(size==N)		while(size==0)
buffer[(head+size)%N]= object		result = buffer[head]
		head=(head+1)%N
size++		size --

- forget about size
- use head and tail
- single producer/consumer
- SC

So we get :

	size buffer head	
Producer		Consumer
while((tail+1)%N==head)		while(head==tail)
buffer[tail]= object		result = buffer[head]
tail = (tail+1)%N		head=(head+1)%N

- take advantage of atomic instructions

Example :

```
Compare_and_Swap(address,old,new)
{
    if(*address ==old) {
        *address=new;
        return 1;
    }
    else return 0;
}
```

can be used to write an atomic stack :

```
void atomic_push(stack* s, int value) {
    item = malloc(sizeof(node));
    item->val = value;
    do {
        item->next = *s;
    } while (!compare_and_swap(s, item->next, item));
}
```

Similar for atomic\_pop (try to set s to \*s->next atomically)

Other performance issues include :

- locks locality: with non uniform memories or with non uniform caches. => more elaborate locking structures (hierarchical, or distributed and made coherent)
- bottlenecks: if a single lock is used by all the threads => it's better to separate the work into independant parts and use several locks.