Operating Systems

Advanced Thread Synchronization

Thomas Ropars

thomas.ropars@imag.fr

Équipe ERODS – LIG/IM2AG/UJF

2015

References

The content of these lectures is inspired by:

- The lecture notes of Prof. André Schiper.
- The lecture notes of Prof. David Mazières.
- The lectures notes of Arnaud Legrand.
- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau

Other references:

- *Modern Operating Systems* by A. Tanenbaum
- *Operating System Concepts* by A. Silberschatz et al.

Agenda

Efficiency of concurrent code

Advanced Spinlocks

Deadlocks

Non-blocking synchronization

Agenda

Efficiency of concurrent code

Advanced Spinlocks

Deadlocks

Non-blocking synchronization

First a reminder

Threads have two main purposes:

- Overlapping computation and I/Os
- Taking advantage of multicore processors for compute-intensive code

The performance improvement that can be expected from using multiple threads is not infinite.

Amdahl's Law

Speedup of parallel code is limited by the sequential part of the code:

$$T(n) = (1 - p) \cdot T(1) + \frac{p}{n} \cdot T(1)$$

where:

- $T(x)$ is the time it takes to complete the task on x cores
- p is the portion of time spent on parallel code in a sequential execution
- n is the number of cores

Even with massive number of cores:

$$\lim_{n \to \infty} T(n) = (1 - p) \cdot T(1)$$

Amdahl's Law

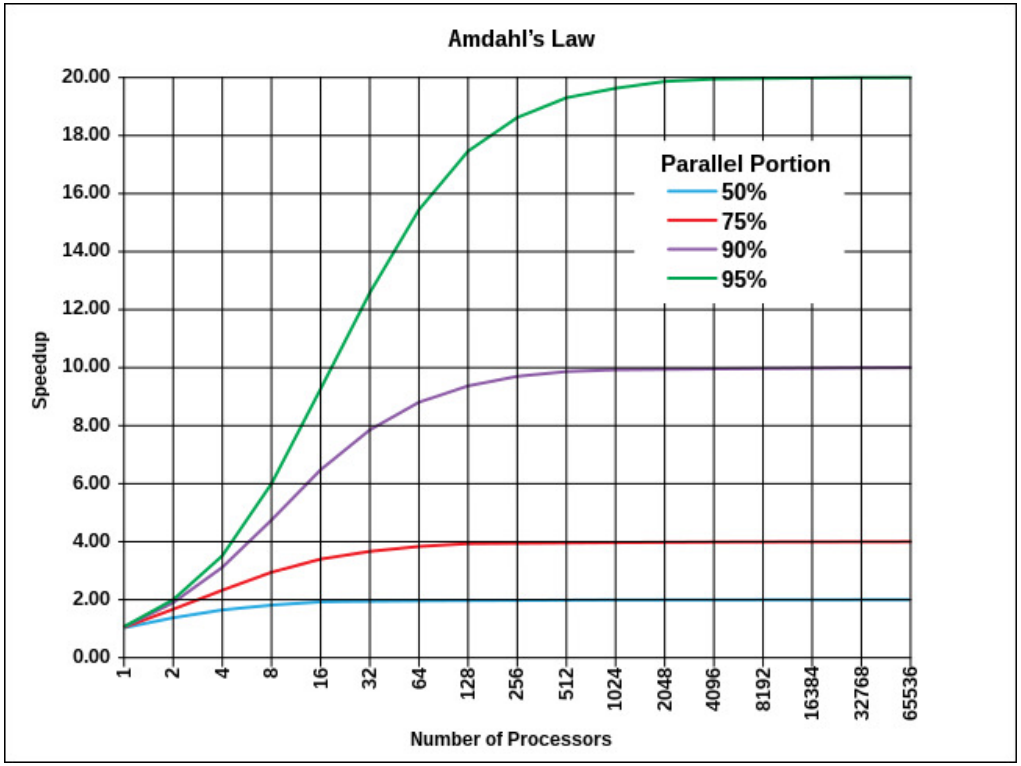

Figure: Speedup as a function of the number of cores[1]

[1] "AmdahlsLaw" by Daniels220 at English Wikipedia

Efficient code

Critical sections are pieces of code that are executed sequentially!

Finding the right granularity for critical sections:

- Correctness always comes first!
- Try to reduce the size of the critical sections that are executed most often
- Do not forget that lock/unlock operations also take time

Agenda

Efficiency of concurrent code

Advanced Spinlocks

Deadlocks

Non-blocking synchronization

Limits of simple spinlocks

Recall

In a basic spinlock algorithm, all contending threads spin by trying to execute a read-modify-write operation on the same variable :

- It does not ensure bounded waiting (fairness)
- It implies a large number of message on the interconnection network

Improved spinlocks

Ticket lock

- Principle:
  - The lock includes two counters (*next*, *current_num*). Threads execute FAA on *next* counter to get a unique *ticket* and wait until *current_num* $==$ *next*
  - The `unlock()` operation increments *current_num*.
- This solution ensures fairness

Improved spinlocks

MCS lock

- Principle:
  - The lock is a list of nodes (a node represent one thread trying to acquire the lock) where the first node correspond to the thread holding the lock. Nodes are added to the list using CAS. Each node spins on a `waiting flag` in its local node.
  - The `unlock()` operation resets the `waiting flag` on the next node in the list.
- This solution ensures fairness and limits communication on the interconnection network (local spinning)

Agenda

Efficiency of concurrent code

Advanced Spinlocks

Deadlocks

Non-blocking synchronization

The deadlock problem

```
mutex_t m1, m2;

void p1 (void *ignored) {
  lock (m1);
  lock (m2);
  /* critical section */
  unlock (m2);
  unlock (m1);
}
void p2 (void *ignored) {
  lock (m2);
  lock (m1);
  /* critical section */
  unlock (m1);
  unlock (m2);
}
```

Problem?

The deadlock problem

```
mutex_t m1, m2;

void p1 (void *ignored) {
  lock (m1);
  lock (m2);
  /* critical section */
  unlock (m2);
  unlock (m1);
}
void p2 (void *ignored) {
  lock (m2);
  lock (m1);
  /* critical section */
  unlock (m1);
  unlock (m2);
}
```

Problem? The program can stop making progress

Deadlocks (with condition variables)

```
mutex_t m1, m2;
cond_t c1;

void p1 (void *ignored) {
  lock (m1); lock (m2);
  while(!ready){wait(c1,m2);}
  /* do something */
  unlock (m2); unlock (m1);
}
void p2 (void *ignored) {
  lock (m1); lock (m2);
  /* do something*/
  signal (c1)
  unlock (m2);unlock (m1);
}
```

One lesson: Dangerous to hold locks when crossing abstraction barriers!

- e.g., `lock(a)` then call function that uses condition variable

Deadlock characterization

Coffman et al. (1971)

1. Mutual exclusion condition
   - Each resource is either currently assigned to exactly one process or is available
2. Hold and wait condition
   - Processes currently holding resources that were granted earlier can request new resources
3. No preemption condition
   - Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition
   - There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Dealing with deadlocks

Prevention

Eliminating one of the conditions:

- Non-blocking (wait-free) algorithms (1)
- Wait on all resources at once (2)
- Use optimistic concurrency control (transactions) (3)
- Always lock resources in the same order (4)

Avoidance

Prevent the system from entering an unsafe state (requires knowing a priori the resource requirements of each process)

Dealing with deadlocks

Detection + corrective action

Problem: What corrective action?

- Process termination
- Rollback (Possible? Cost?)

Ignore the problem

Solution chosen by most OSes (including UNIX systems)

- Assume it occurs rarely enough to avoid the need for a complex solution

Read "Operating Systems: Three Easy Pieces", chapter 32 for a more detailed discussion on deadlocks

Agenda

Efficiency of concurrent code

Advanced Spinlocks

Deadlocks

Non-blocking synchronization

Progress condition

Concurrent algorithms without locks

Progress condition

- Lock freedom: If the algorithm is run long enough, it is guaranteed that some operations will finish

Concurrent algorithms without locks

Progress condition

- Lock freedom: If the algorithm is run long enough, it is guaranteed that some operations will finish
- Wait freedom: Each operation takes a finite number of steps to complete.
  - Newcomers need to help *older* requests before executing their own operation

Concurrent algorithms without locks

Progress condition

- Lock freedom: If the algorithm is run long enough, it is guaranteed that some operations will finish
- Wait freedom: Each operation takes a finite number of steps to complete.
  - Newcomers need to help *older* requests before executing their own operation

These algorithms strongly rely on *compare_and_swap()* operations.

A lock-free stack

```
typedef struct {
  node* head;
}stack_t;

push(stack_t *stack, node* new_node){
  while(true){
    new_node->next=stack->head;
    if(CAS(&stack->head, new_node->next, new_node)){break;}
  }
}

node* pop(stack_t *stack){
  node* item=NULL;
  while(true){
    item=stack->head;
    if(item==NULL){return NULL;}
    node* new_head=item->next; /*A*/
    if(CAS(&stack->head, item, new_head)){return item;} /*B*/
  }
}
```

Comments on previous algorithms

- This algorithm is lock free
  - A thread cannot prevent other threads from progressing
- This algorithm is not wait free
  - A thread could repeatedly fail on CAS
- The algorithm is not correct because it does not deal with two problems:
  - Memory reclamation: How to free memory without introducing a risk of segfault at line (A)?
  - The ABA problem: What is a thread takes a long time between execution of lines (A) and (B) such that *item* has been taken by another thread, and the corresponding memory reused for another item? (CAS at (B) could complete but *new_head* be garbage)