Threads

M1 MoSIG : Operating Systems

Poupin Pierre Rouby Thomas

04/11/2014

1 Introduction

A thread is an execution context that belong to a process. A process might contain several threads which share some resources :

- memory
- file descriptors

Sometimes called lightweight process.

process with single thread

code	data	files
register	⋮	stack

process with multiple threads

code	data	files
registers stack ⋮	registers stack ⋮	registers stack ⋮

Advantages

- Lighter management (especially context switch)
- Take advantage of concurrency within a process (eg can perform a computation during a blocking system call in another thread).
- Communication between threads is easier/more efficient than IPC (Inter Process Communication) between processes.

Example : Webserver

The main thread can listen to connextions while other threads handle requests Accesses to Webserver data can be performed concurrently and overlapped with computations

2 Thread Models

Threads might be :

- Preemptive : threads might be interrupted asynchronously to switch to another thread.
- Cooperative : the thread itself release the PCU to let another thread be scheduled.

Advantages / Drawbacks

- For preemptive :
  - Insensitive to misbehaving threads.
  - Can take advantage of multiple CPUS.
  - higher cost (context switch)
- For cooperative :
  - easier to programm and debug because not sensitive to data races (will be explained later)
  - can only take advantage of a single CPU
  - more efficient

3 Implementation

The implementation can be performed by using the kernel, in that case the implementation is similar to processes implementation :

- the kernel itself performs threads scheduling.
- threads management : creation, destruction and preemption managed by the kernel.
- can be implemented almost like processes, eg in Linux :
  - processes and threads are tasks.
  - they have different attributes, memory is shared for threeads
  - scheduled using the same scheduler

The implementation might also be completely in a user-level process under the form of a library :

- creation, destruction are just fonctions of the library (which manage an internal list of threads)
- context switch :
  - preemptive model : rely of preemption mechanism provided by the kernel such as signals (setitimer/sigalarm)
  - cooperative model : rely on a function 'yield' provided by the library and called by the thread itself
  - both cases : most blocking functions are redefined to put blocking thread in a waiting queue and let the scheduler decide on another thread to execute. This includes : read, write, accept, connect, ...

4 Mixed models

Another possibility is to use n user threads mapped to m kernel threads.

Idea :

Take advantage of any number of threads to express parallel work or to overlap computation with communications, limit the number of resources (Cores, CPUS) to limit the cost of context switches between them.

The library is more diffucult to implement as it implies synchronization between kernel threads

5 Memory & Execution issues related to threads

Threads share the same memory space :

Shared variable

Count

Thread 1	Thread 2
count++	count- -

Example :

```
int count;
void *t1 (void *ignored) {
     count ++;
}

void *t2 (void *ignored) {
     count --;
}

int main() {
     thread_create(t1,0);
     t2();
}
```

Example :

```
int flag1 = 0, int flag2 =0;

void *p1 (void *ignored){
     flag1=1;
     if (!flag2) { function_1(); }
}

void *p2 (void *ignored){
     flag2=1;
```

```
     if (!flag1) { function_2(); }
}

int main() {
. . .
create_thread(p1,NULL);
p2();
}
```

Which function 1 and 2 are executed ?

- none ?
- one of them ?
- both ?

All these 3 choices are possible it depends on the hardware :

- CPUS can reorder operations especially memory operations.
  - concurrent independant writes
  - pre-fetching
  - ...
- compiler can reorder operations
  - register promotion
  - common sub-expressions elimination
  - loop blocking
  - ...

Definition : Sequential consistency (SC)

The execution of the program has the same result at is if the execution has been performed according to some sequential order and the order of instructions on each processor will be the same as the order in the program.

From the hardware point of view :

- atomic instructions and locked instructions
- memory barriers : special instruction to complete pending memory operations

The programmer can use them to avoid data races : the program will then have SC.

Definition : There is a **data race** if :

- two threads are accessing the same memory location
- one of them is a write
- none are related to synchronization operations

Example :

```
int Money_Transfer(int amount, account from, account to) {
int  bol;
atomic {bol = from_balance}
if (bol < amount) return ERROR;
- - if there is another money transfer here, problems !
atomic {from_balance -= amount}
atomic {to_balance += amount}
}
```

Imagine that two threads perform transfers to/from the same account. Here atomic means: completely executed without the possibility for other threads to access the same memory locations.

The fix is make everything atomic :

```
int Money_Transfer(int amount, account from, account to) {
int  bol;
     atomic{
            bol = from_balance;
            if (bol < amount) return ERROR;
            from_balance -= amount;
            to_balance += amount;
          }
}
```

Race Condition :

A flaw in the program that makes the correctness of the result depends on the ordering of external events.

External events :

- context switches
- signals
- interrupts
- memory writes from other processors
- ...

6 Critical sections

A critical setion is a part of a program that might be the source of a race condition with critical sections in other threads.

The critical section problem :

- ensure mutual exclusion between critical sections
- ensure progression : if some threads wait to enter their critical section and no thread is execution its critical section, one of them should be allowed to enter in a finite time.
- bounded waiting : if some thread wants to enter its critical section, the number of threads that enter before him should be bounded.

6.1 Software solution

Assuming :

- sequential consitancy
- threads loop over :
  - critical section
  - remaining of the program

Here is the Petterson solution :

flag[1]= false ; flag[2] = false

turn= 1 or 2

flag[1] = true	flag[2] = true
Thread 1	Thread 2
turn =2	turn =1
while(turn !=1) && flag[2]{}	while(turn!=2) && flag[1]{}
critical section	critical section
flag[1] = false	flag[2] = false
remaining	remaining

It can be extended to N threads but it is not efficient in practice (because of SC and the number of operations)

6.2 Hardware support

lock = 0

Thread 1	Thread 2
while(test_and_set(&lock,1)){}	while(test_and_set(&lock,1)){}
lock = 1;	lock = 1;
critical section	critical section
lock = 0;	lock = 0;
remaining	remaining

Spinlock

New Instruction :

```
int test_and_set(int *adress, int value) {
     atomic {
          int result;
          result = *adress;
          *adress = value;
          return result;
     }
}
```

Other possibilities : Compare_and_swap, or swap.

Works with any number of threads.

Costly :

- such an atomic instruction locks the memory bus for its duration
- active wait : a loop that does nothing while waiting for the lock

Is it used ?

- in the kernel, on multiprocessor systems, to synchronize accesses to shared memory