Threads Of Computation - I

Threads within a Process

- So far we have looked at a process as a program in execution.
- We assumed that there is only one execution sequence or thread of computation of the code within a process.
- But it is possible to have more than one execution sequence running concurrently or in parallel[a] within a process.

[a] Say on a multi-core processor.

Threads within a Process

- There are computational jobs with distinct logical parts and can have almost independent flow of computation.
- We have seen how child processes are created to do the parts of a whole job in parallel. And then combine the outcome of different child processes if necessary.

Threads within a Process

- But instead of creating child processes with separate address spaces, it is possible to have multiple threads of execution within a process sharing the same address spece.
- Each thread can be scheduled independently.
- Each thread has its own identification, thread ID (TID), CPU state (program counter (PC) and other registers), and stack.

Multiple Threads in a Process

- But all threads of a process share the same code, global data, heap area[a], open files etc.
- Data sharing is easy between threads.
- A software may have different kinds of activity e.g. user interfaces to different users, different computations, different database access etc.

[a] There are different stacks for different threads of a process. But they live within the same virtual address space of the process.

Multiple Threads in a Process

- This may be achieved by running different threads within a process.
- A blocking I/O suspends a single-thread process. But in a multi-thread process even when a thread is blocked for I/O, other threads may continue.
- There is also a possibility of concurrent I/O by different threads.

Multiple Threads in a Process

- It is also claimed that creating and switching thread is order of magnitude faster than creating and switching process.
- This may be due to the fact that creation of a thread does not require the creation of a new address space and its page table.
- It also does not require different data copy to child process.

Multiple Threads in a Process

- A process can communicate either through shared memory or by message passing. Both requires some dialog with the OS. But a threads of a process can communicate through the common area of global data.

Advantages

- Better responsiveness in a interactive system.
- Easy sharing of data.
- Faster creation of thread.
- Non-blocking of the complete process due to slow I/O.

Amdahl's Law of Performance Gain

- Let there be n processing units, and the fraction of task that is to be performed sequentially be s.
- The speedup is limited by the formula:

$$\text{speedup} \leq \frac{1}{s + (1 - s)/n}$$

Amdahl's Law of Performance Gain

- If $s = 0.2$ and there are 4-cores, the speedup cannot exceed $\frac{1}{0.2+0.8/4} = 2.5$.
- The main observation is that with 20% sequential work load, the speedup cannot exceed $1/0.2 = 5$ with any number of processor.

Types of Threads

- The implementation of thread may be at the user level known as a user thread or at the OS level, known as a kernel thread.
- User threads are managed at the user level. The kernel is not aware of it. So it can also be implemented on a single CPU system.

Types of Threads

- Kernel threads are managed and scheduled by the OS kernel.
- But at the lower level any thread runs on a kernel thread. Following are three different mapping models.

Types of Threads

- Many-to-one model: maps many user threads to one kernel thread.
- One-to-one model: each user thread is mapped to a kernel thread.
- Many-to-many model: the set of user threads are mapped to a set (smaller or same size) of kernel threads.

Many-to-One Model

- Threads are managed at the user space by the thread library, so there is no overhead of transition from user mode to kernel mode during thread switching.
- But user threads cannot take the advantage of the multiprocessor or multi-core architecture.

Many-to-One Model

- Any blocking system call will block the underlying kernel thread, resulting the blocking of all user level threads mapped to it.
- User level threads are used for fine grain parallelism where system calls are often not required.

Many-to-One Model

- Normally user level threads are small computation intensive code. Each thread has its CPU state and a thread control block to cooperate and manage the scheduling of different threads.
- It should have its own mechanism to manage the atomic of critical sections of code and synchronization.

Many-to-One Model

- As it does not require any OS support, it can be implemented on any OS.
- But most OS today support kernel level thread and most processors are multi-core. That possibly makes user level thread less popular.

Many-to-One Model

Think of the following issues in connection to user-level threads:

- How does one thread gets suspended and another is scheduled?
- Who decides about the scheduling policy?
- What will happen if there is an exception or blocking system call in one thread?

One-to-One Model

- Each user level thread is mapped to a kernel level thread.
- Threads can run in parallel on a multiprocessor or multi-core architecture.
- Blocking of one thread does not affect the execution of another thread.

One-to-One Model

- For every user thread there is a kernel thread. So the thread creation overhead and the presence of large number of kernel threads may be a problem.
- Thread creation time may be comparable to process creation time.
- This model is good for coarse-grained parallelism.

One-to-One Model

- It require full support from the OS e.g. creation, scheduling, blocking and termination of threads.
- OS must support data structures like thread control block (TAB) etc.

Many-to-Many Model

- This is a middle-path between the first two models where an user can create as many threads as he wishes.
- But the OS can create number of kernel threads depending on the architecture (number of processor or core).

Many-to-Many Model

- If an user thread issues a blocking system call, the kernel can schedule a ready user thread on the kernel thread.
- It is also possible to nail some particular user thread to a kernel thread.

Thread Library

- An API to create and manage threads is provided by a thread library.
- The library may work at the user space or at the kernel level.
- We shall talk about POSIX Threads known as pthread. The API is defined by POSIX standard (IEEE Std 1003.1c-1995).

An Example

```
/*
 Programming with pthread: pthread1.c++
    one thread computes factorial and
    the other thread computes fibonacci
 $ g++ -Wall pthread1.c++ -lpthread
 $ ./a.out 5
*/
#include <iostream>
using namespace std;
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define loop(X) {for(int i=0; i<=(X); ++i);}

void * thread1(void *) ;
void * thread2(void *) ;
int eS1, eS2;

int fact(int n){
```

```
    if(n == 0) return 1 ;
    return n*fact(n-1) ;
}

int fib(int n) {
    int f0 = 0, f1 = 1, i ;

    if(n == 0) return f0 ;
    if(n == 1) return f1 ;
    for(i=2; i<=n; ++i) {
        int temp = f0 ;
```

```
        f0 = f1 ;
        f1 = f0 + temp ;
    }
    return f1 ;
}

int main(int count, char *vect[]) {
    pthread_t thID1, thID2; // thread ID
    int n ; // pthread1.c++
    int err, *esP1, *esP2;
```

```
if(count < 2) {
    cerr << "No argument for function\n" ;
    exit(1) ;
}

n = atoi(vect[1]) ;
cout << "main thread: n = "
     << n << "\n" ;

err = pthread_create(&thID1, NULL, thread1,
```

```
                                                // 1st child thread1
    if(err != 0){
      cerr << "Thread 1 creation problem\n";
      exit(1);
    }
    err = pthread_create(&thID2, NULL, thread2,
                                                // 2nd child thread2
    if(err != 0){
      cerr << "Thread 2 creation problem\n";
      exit(1);
    }
```

```
    pthread_join(thID2, (void **)&esP2);// 2nd th
    pthread_join(thID1, (void **)&esP1);// 1st th
    cout << "Thread 1: " << *esP1 << endl;
    cout << "Thread 2: " << *esP2 << endl;

    return 0 ;
}
void *thread1(void *vp) {// Address of parameter
    int i, *p ;          // to pass
```

```
    p = (int *) vp ;
    for(i=0; i<=*p; ++i) {
        cout << "Th1: fib(" << i
             << ") = " << fib(i) << endl ;
        loop(5000000);
    }
    eS1 = 1;
    pthread_exit((void *)&eS1) ;
}
void *thread2(void *vp) {// Address of parameter
    int i, *p;              // to pass
```

```
    p = (int *) vp ;
    for(i=0; i<=*p; ++i) {
        cout << "Th2:" << i
             << "! = " << fact(i) << endl ;
        loop(5000000);
    }
    eS2 = 2;
    pthread_exit((void *)&eS2) ;
}
```

Creating Threads

```
int pthread_create(pthread_t *thread,
const pthread_attr_t *attr, void
*(*start_fun) (void *), void *arg);
```

- Creates a thread with the identifier in `*thread`.
- `attr` is used to set thread attributes. A `NULL` is for default attribute values.
- `start_fun` is a function that the thread will

execute once created.

- `arg` is the single argument passed to `start_fun` as a `(void *)` pointer.

Wait for Termination

- `int pthread_join(pthread_t tid, void **ret)` waits for the thread with `tid` to terminate.
- `void pthread_exit(void *retval);` terminates the thread and returns status information through `*retval`. This information is available through `ret`, where `*ret` is the value of `retval` in `pthread_join`.

Terminating Thread

- `pthread_cancel(thread)` sends cancellation request to the `thread`.
- But whether the thread will be canceled or not depends on the threads cancelability state and type.

Cancelability State

- `int pthread_setcancelstate (int state, int *oldstate)` sets the cancelability state of a thread either to `THREAD_CANCEL_ENABLE`, receive cancel request, or to `THREAD_CANCEL_DISABLE`, ignores cancel request.
- The second parameter is the old state pointer, may be put to NULL.

- Also see `pthread_setcanceltype()`.

Terminating Thread: An Example

```
/*
 Programming with pthread: cancelability state:
 pthread2.c++
 $ g++ -Wall pthread2.c++ -lpthread
 $ ./a.out
*/
#include <iostream>
using namespace std;
#include <unistd.h>
```

```
#include <pthread.h>
#define MAXLOOP 15

void * thread(void *) ;
int tS, *tSP;

int main() {
     pthread_t tid; // pthread2.c++

     pthread_create(&tid, NULL, thread, NULL);
     pthread_cancel(tid);
```

```
    pthread_join(tid, (void **)&tSP);
    // cout << "Thread status: " << *tSP << endl

    return 0 ;
}
void *thread(void *vp) {
    pthread_setcancelstate(PTHREAD_CANCEL_DISABL
    for(int i=1; i<= MAXLOOP; ++i) {
        sleep(1);
        if(i==10)
            pthread_setcancelstate(
```

```
                    PTHREAD_CANCEL_ENABLE, NULL);
            cout << "Thread Running: " << i << endl;
        }
        tS = 1;
        pthread_exit((void *)&tS) ;
}
```

Different Stacks

```
// pthread4.c++ different stacks
// $ c++ -Wall pthread4.c++ -lpthread
#include <iostream>
using namespace std;
#include <pthread.h>
#include <cstdio>
#include <unistd.h>

void *thread1(void *p){
```

```
    int n = 20;
    sleep(1);
    cout << "Child Thread 1 - n: " << dec << n
         << ", &n: " << hex << &n << endl;
    return NULL;
}
void *thread2(void *p){
    int n = 30;
    sleep(2);
    cout << "Child Thread 2 - n: " << dec << n
         << ", &n: " << hex << &n << endl;
```

```
    return NULL;
}

int main() {
    pthread_t thID1, thID2; // thread ID
    int n=10; // pthread4.c++

    cout << "Main Thread - n: " << n
         << ", &n: " << hex << &n << endl;
    pthread_create(&thID1, NULL, thread1, NULL);
    pthread_create(&thID2, NULL, thread2, NULL);
```

```
    pthread_join(thID1, NULL);
    pthread_join(thID2, NULL);

    return 0;
}
```

Different Stacks

```
$ a.out
Main Thread - n: 10, &n: 0x7fff8ff7b50c
Child Thread 1 - n: 20, &n: 0x7f49609b9efc
Child Thread 2 - n: 30, &n: 0x7f49601b8efc
```

Race on Global Variable

```
/*
 pthread3.c  Race condition
 $ g++ -Wall -lpthread pthread3.c++
 $ ./a.out 500000
 output:  0, +ve and -ve
*/
#include <iostream>
using namespace std;
#include <stdio.h>
```

```
#include <stdlib.h>
#include <pthread.h>

int times, n = 0 ;
void * thread1(void *) ;
void * thread2(void *) ;
void inc() {n=n+1;}
void dec() {n=n-1;}

int main(int count, char *vect[]) { // argument i
     pthread_t thID1, thID2;
```

```
if(count < 2) {
      perror("No argument for times\n") ;
      exit(1) ;
}
times = atoi(vect[1]) ;
pthread_create(&thID1, NULL, thread1, NULL)
pthread_create(&thID2, NULL, thread2, NULL)
pthread_join(thID1, NULL) ;
pthread_join(thID2, NULL) ;
```

```
    cout << "n: " << n << "\n" ;
    return 0 ;
}

void *thread1(void *vp) {
    int i ;

    for(i=1; i<=times; ++i) inc();
    return NULL ;
}
```

```
void *thread2(void *vp) {
    int i ;

    for(i=1; i<=times; ++i) dec() ;
    return NULL ;
}
```

Race on Global Variable

- The global variable is initialized to 0.
- One thread increments it 5×10^6 times.
- The other thread decrements it 5×10^6 times.
- At the end the expected result is 0 again.
  But different runs give different results.

Race on Global Variable

```
$ a.out 5000000
n: -2203358
$ a.out 5000000
n: 3156188
$ a.out 5000000
n: 4050120
```

Linux `clone()`

- The library function `clone()` and the corresponding system call `clone()` is specific to Linux and is not portable.
- The system call or its `glibc` wrapper function is used to create a child process.
- But it can also be used to create kernel level threads.

Linux `clone()`

- A call to `clone()` also creates the child process almost as the copy of the parent.
- But unlike `fork()`, the child process does not start execution at the point of the call.
- It calls the function specified as argument in the call along with parameters.

Linux `clone()`

- The interface of glibc wrapper function of `clone()` is

  ```
  int clone(int (*f)(void *), void
  *child_stack, int flags, void *rag);
  ```

- The child process starts executing the function `f` with `void *rag` as the parameter i.e. `f(rag)`.

Linux `clone()`

- The child process created by `clone()` terminates when `f(rag)` returns or there is a call to `exit()` within it.
- The exit code of the child is the integer returned by `f()`. The parent process may wait for the completion of the child as usual.

Linux `clone()`

- A cloned child, unlike forked child, shares some execution context of the parent.
- The memory space, the file descriptor table etc. are shared.
- As the memory space is shared, the stack of the parent cannot be used by the child.
- The second parameter of the call specifies the bottom of child's stack[a].

[a] Which often grows from higher address to lower address.

Linux `clone()`

- The least significant byte of `flags` specifies the termination signal from the child to the parent.
- Other bits are used to control the effects of call to `clone()`.
- CLONE_VM - parent and child share the virtual memory, CLONE_FILES - parent and child share the file descriptor table.

Thread Creation by `clone()`

```
/*
 clone1.c++ Creation of new thread by clone()
*/
#include <iostream>
using namespace std;
#include <stdio.h>
#include <sched.h>
#include <sys/types.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAXSTACK 4096

int fact ; // global data
int what(void *p) ;

int main() {  // clone1.c++
    int chPID, status, n ;
```

```
  char *chStack ;


  cout << "Enter a +ve integer: " ;
  cin >> n;
  chStack = (char *) malloc(MAXSTACK);
            // Memory for new stack
  chStack = chStack  + MAXSTACK;
            // Stack grows towards lower
            // address. Bottom of stack


 chPID = clone(what, chStack, CLONE_VM,
```

```
                                (void *)&n);
                // Cloned process will execute
                // 'what(NULL)'.
                // CLONE_VM - same memory space
                // &n parameter to 'what'
                // chPID - cloned process id
cout << "Inside proc: pid = " << getpid() << "
cout << "Inside proc: cpid = " << chPID << "\n
waitpid(chPID, &status, __WCLONE) ;
                // __WCLONE - wait for
                // cloned process
```

```
    cout << "Inside proc:" << n << "! = " << fact

    return 0 ;
}


int what(void *p) {
    int n = *(int *)p, i;

    for(fact=i=1;i<=n;++i) fact *= i;
    return 0 ;
}
```

Thread in Python

- Import the `thread` module.
- Start the method `thread.start_new_thread(function, rags)`.
- The first parameter is the function name, the second parameter is a tulle of arguments to the function.
- There is a third parameter that we ignore.

A Simple Thread

```
#!/usr/bin/python
# sorting.py reads a string of integers seperated
#              blanks. split them in three lists
#              sort them by running three threads
#              finally merge them
import thread
import time
def merge(l1, l2):
    if l1 == []: return l2
```

```
    if l2 == []: return l1
    if l1[0] < l2[0]: return [l1[0]]+ \
                      merge(l1[1:], l2)
    else: return [l2[0]]+merge(l1, l2[1:])
def mySort(l, n):
    global l1g, l2g, l3g
    l.sort()
    if n==1: l1g = l
    elif n==2: l2g = l
    elif n==3: l3g = l
```

```
s = raw_input("Enter +ve integers: ")
l = []
for i in s.split(): l = l + [int(i)]
llen = len(l)
l1, l2, l3 = l[:llen/3], l[llen/3:2*llen/3], \
             l[2*llen/3:]
print l1, l2, l3
try:
   thread.start_new_thread(mySort, (l1, 1,))
   thread.start_new_thread(mySort, (l2, 2,))
   thread.start_new_thread(mySort, (l3, 3,))
```

```
except: print "Thread creation error"
time.sleep(1) # bad use
print merge(merge(l1g, l2g), l3g)
```

Thread in C++11

- Include `<thread>` header and compile with following options:
  `g++ -Wall -std=c++11 c++thread1.c++ -pthread`
- Following is a very simple example. But there are many features.

A Simple Thread

```
/*
 * c++thread1.c++ basic c++ thread
 * $ g++ -Wall -std=c++11 c++thread1.c++ -pthread
 */
#include <iostream>
#include <thread>
using namespace std;
#include <unistd.h>
```

```
int fact;
void factorial(int n){
     fact=1;
     for(int i=1; i<=n; ++i) fact *= i;
     sleep(1);
     cout << "child thread ID: " << this_thread::
}

int main(){
    int n;
```

```
    cout << "Enter a +ve integer: ";
    cin >> n;
    std::thread t(factorial, n);
    cout << "main thread ID: " << this_thread::ge
    cout << "child thread ID (in parent): " << t.
    t.join();
    cout << n << "! = " << fact << endl;

    return 0;
}
```