Concurrent Programming

Some important concepts in concurrent programming

- Multithreading
- Shared memory communication
- Message passing communication
- Race conditions
- Testing
- Synchronization
- Deadlock
- Livelock
- Safety properties
- Liveness properties
- Ahmdahl's law and Gustafson's law
- Nonblocking algorithms

Things you should be familiar with:

- Processes (fork, exec, wait)
- Interprocess communication (pipes, TCP sockets)

Outline of this section

- Threads of control
- Threads and processes
- Why program concurrently?
- Concurrent programming architectural patterns
- Architectural patterns for concurrent processes
- Communication between threads.
- Shared Memory Programming
- Threads and Mutexes with PThreads
- Designing sharable objects
- Conditions
- Conditions in PThreads
- Semaphores

Reading: Chapter 12 of *Computer Systems: A Programmers Perspective*

Threads of control

In the simplest computer systems there is a single **thread of control**

- After an instruction x is executed,
- the next instruction is either
  - the instruction that follows x or,
  - if x is a branch, an instruction that x branches to.

However, modern computers allow various forms of **multithreading** — i.e. multiple threads of control existing at the same time.

Why?

- *Interrupts*: Interrupts cause the main thread of control to be suspended while the interrupt handler executes.
- *Time slicing:* After an interrupt, we can arrange that the CPU starts executing a different thread of control.
- *Simultaneous multithreading:* The CPU might switch between two (or more) threads on its own. (Intel calls this **hyperthreading**).
- *Multiprocessors:* There may be more than one CPUs, each executing a thread of control.
- *Distributed systems:* When multiple computer systems cooperate, each runs its own set of threads.

Henceforth: “thread of control” will be abbreviated by “**thread**”.

See animations for time slicing and simultaneous multithreading.

Threads and Processes

In the early Unix operating system:

- Each process had one thread
- The fork system call was the only way to create a new thread.

Modern operating systems allow multiple threads within the same process.

In modern Unix:

- fork creates a process with one thread
- threads can request the OS to create more threads
- all threads in the same process share the same global (static) and heap memory.

Concurrent application programs and systems

A concurrent application is a program that is written to use multiple threads at the same time.

Reasons to do this:

- Programming simplicity
- Speed
- Distribution and reliability

Programming simplicity:

When there are multiple streams of inputs, it may be simplest to use a separate thread to deal with each one.

- ∗ Consider a music player application: It need to deal with:
  - ∗ · Zero or more songs being downloaded.
  - · Playback of zero or one song.
  - · Interactions with the user
- Each of these is a distinct activity.
- We can use a separate thread to manage each activity

For example, our music player

```
concurrently
    while not done
        wait for an input action
        change the state of the music player in reaction
||
    while not done
        wait for a part of a song to be delivered via the network
        if not timed out then add that part to a local file
||
    while not done
        if playing back a song
            read a part of the song from disk
            if not end of file
                    play that part through the speakers
```

Speed

If we have multiple processors, we can speed up a calculation by expressing it concurrently

Consider computing a dot product: $a \cdot b = \sum_{i\in\{0,..n\}} a_i b_i$

```
**concurrently**
    $sum0 := 0$
    **for** $i \in \{0, .. \lfloor n/2 \rfloor\}$
        $sum0 := sum0 + a[i] \times b[i]$
||
    $sum1 := 0$
    **for** $i \in \{\lfloor n/2 \rfloor, ..n\} \cdot sum1 := sum1 + a[i] \times b[i]$
**end concurrently**
$result := sum0 + sum1$
```

If we have p processors, we might use a **concurrent for loop** to execute p threads at the same time.

```
**concurrently for** $k \in \{0, ..p\}$
    $sum[k] := 0$
    **for** $i \in \left\{\left\lfloor \frac{k\times n}{p} \right\rfloor, .. \left\lfloor \frac{(k+1)\times n}{p} \right\rfloor\right\}$
        $sum[k] := sum[k] + a[i] \times b[i]$
$result := 0$
**for** $k \in \{0, ..p\}$
    $result := result + sum[k]$
```

A cautionary tale about speed

I tried sorting 10,000,000 numbers using a concurrent version of the quicksort algorithm

On my older single-processor laptop:

- 1 thread took about 14 seconds,
- 2 threads about 15 seconds.

No surprise there.

But ...

On my partner’s new dual-core (i.e. 2 CPU) laptop:

- 1 thread took about 11 seconds,
- 2 threads took about 20 seconds!

Why?

- The threads communicated a lot.
- When all communication was between threads on the same CPU, this was efficient.
- When communication was between threads on different CPUs, it took more time.
  - ∗ Data had to be copied from one processor’s cache to the other’s, and back, over and over.

After tuning the threads to use less communication, the algorithm did indeed run almost twice as fast with 2 threads on the dual-core.

Lessons from this:

- Don’t assume that concurrent programs will speed up just because you add more processors.

- In fact, they may slow down.
- Communication takes time, even when it is through shared memory.
- Performance problems, once identified, can be solved.

Looking back at our concurrent dot product.

- We should ensure that $sum[k]$ and $sum[k+1]$ are not stored on the same cache-line.
- Otherwise the speed may be severely compromised.

Speed and peripheral devices

Consider a system with one CPU, a magnetic disk (hard drive) and an optical disk. We wish to copy a file from the optical disk (100 MB/s) to the hard drive (100 MB/s).

Suppose both devices have block sizes of 16 kB.

Sequential version:

```
var $buffer$ : array $2^{14}$ of byte
var $len := read(\ rfd, buffer, 2^{14})$
while $len >$ do
    $write\_blocking(wfd, buffer, len)$ // Assume a write
    that writes all len bytes.
    $len := read(\ rfd, buffer, 2^{14})$
```

Speed is about 50MB/s.

Synchronous Pipelined version: Overlap the read and the write

```
var $buffer$ : array $2$ of array $2^{14}$ of byte
var $len$ : array $2$ of int
var $i$ : int $:= 0$
var $j$ : int $:= 1$
$len[i] := read(\ rfd, buffer[i], 2^{14})$
while $len[i] > 0$ do
    concurrently
        $write\_blocking(wfd, buffer[i], len[i])$
    ||
        $len[j] := read(\ rfd, buffer[j], 2^{14})$
    end concurrently
    $i, j := j, i$ // Swap $i$ and $j$.
```

Speed is about 100MB/s.

So multithreading can improve performance even with a single CPU by better utilizing other hardware.

Asynchronous pipelined (producer consumer) version

If there is variation in the speeds of reads and writes, we can do slightly better by decoupling the reading and writing.

We pick a number m to be the number of buffers.

```
var $buffer$ : array $m$ of array $2^{14}$ of byte
var $len$ : array $m$ of int
var $q$ : queue of int := $empty$
var $s$ : set of int := $\{0, ..m\}$
concurrently
    var $i : int$
    while $true$
        remove an item from $s$ and store it in $i$
        $len[i] := read(\ rfd, buffer[i], 2^{14})$
        if $len[i] = 0$ break
        add $i$ to the queue
    end while
    add -1 to the queue
||
    var $j : int := 0$
    while true
        take an item out of the queue and put it in $j$
        if $j < 0$ break
        $write\_blocking(wfd, buffer[j], len[j])$
        add $j$ to $s$
    end while
end concurrently
```

Some history In this example the CPU is idle most of the time. But we can also use multithreading to keep the CPU doing useful work while it was also waiting for peripherals.

- E.g. we could add a processing stage to our pipeline above

$\boxed{\text{reading thread}} \rightarrow \boxed{\text{computing thread}} \rightarrow \boxed{\text{writing thread}}$

Historically, concurrent use of peripherals and the CPU was the reason that operating systems first implemented multithreading.

- In the 1950s, 1960s, and 1970s a “batch” of sequential programs would be run at once, which allowed many devices (e.g. printers, card-readers, tape drives) and the CPU to be kept busy.
- By allowing one CPU and several terminals to operate concurrently, it was possible to allow interactive time-sharing of a mainframe among many simultaneous users.
  - ∗ Pioneered in the 1960s — e.g. MULTICS
  - ∗ Common in the 1970s and 1980s — e.g. Unix, VMS
- Further changes to the OS allowed multiple threads within one OS process.

This is one reason that multithreading operating systems were common even when most computer systems had only one CPU.

Distribution

When we have a distributed application, multithreading is unavoidable.

- Each node must run at least one thread

We looked at reasons to make distributed applications earlier.

Concurrent application architectural patterns

- Iterative Parallelism (data parallel)
  - ∗ Multiple loop iterations executed in parallel
- Recursive parallelism (data parallel)
  - ∗ Recursive subroutine calls executed in parallel
- Producers and Consumers (task or data parallel)
  - ∗ One thread feeds output to the next
- Client/Server (task parallel)
  - ∗ Clients make requests, servers respond.
- Peers
  - ∗ Similar threads communicate directly to each other.

Iterative Parallelism

- Execute iterations of loops in parallel
- Typical for scientific computations.

Example: Matrix Multiplication

Compute `a := b×c`, where `a`, `b` and `c` are `n` by `n` matrices. ($\mathtt{n}^2$ inner products)

double a[n,n], b[n,n], c[n,n];

Sequential version:

```
for [i := 0 to n-1] {
    for [j := 0 to n-1] {
        c[i,j] := 0.0;
        for [k := 0 to n-1]
            c[i,j] := c[i,j] + a[i,k] * b[k,j]; } }
```

In the matrix multiplication algorithm each of the n^2 iterations of the dot product computation is independent of all the others. So:

```
concurrently for [i := 0 to n-1] { # All rows in ||
    concurrently for [j := 0 to n-1] { # All columns in ||
        c[i,j] := 0.0;
        for [k := 0 to n-1]
            c[i,j] := c[i,j] + a[i,k] * b[k,j]; } }
```

But if there are less than n^2 processors then the above is wasteful. Having many more threads than processors will slow down computation.

We can divide the work among $P < n$ threads thus

```
thread worker[w = 0 to P-1] {
    int first := $\lceil (\mathsf{w} * \mathsf{n}) \div \mathsf{P} \rceil$ ; # first row of strip
    int last := $\lceil ((\mathsf{w} + 1)\ * \mathsf{n}) \div \mathsf{P} \rceil - 1$; # last row of strip
    for [i := first to last] {
        for [j := 0 to n-1] {
            c[i,j] := 0.0;
            for [k := 0 to n-1]
                    c[i,j] := c[i,j] + a[i,k] * b[k,j]; } } }
```

Recursive Parallelism

Independent recursive procedures:

When a sequence of calls (recursive or not) are independent, they can run in parallel.

Example: Adaptive Quadrature

Estimate the area under a curve, $f(x)$, on an interval $[left, right]$.

```
double quad(double left, right, fleft, fright, area) :
    double mid := (left + right) / 2;
    double fmid := f(mid);
    double larea := (fleft + fmid) * (mid - left) / 2;
    double rarea := (fmid + fright) * (right - mid) / 2;
    if( abs((larea+rarea) - area)) > EPSILON )
        larea := quad(left, mid, fleft, fmid, larea)
        rarea := quad(mid, right, fmid, fright, rarea)
    return larea + rarea; }
```

Since recursive calls only use local variables and value parameters, we can do them in parallel.

```
double quad(double left, right, fleft, fright, area) :
    double mid = (left + right) / 2;
    double fmid = f(mid);
    double larea = (fleft + fmid) * (mid - left) / 2;
    double rarea = (fmid + fright) * (right - mid) / 2;
    if( abs((larea+rarea) - area) > EPSILON )
        concurrently
            larea := quad(left, mid, fleft, fmid, larea)
        ||
            rarea := quad(mid, right, fmid, fright, rarea)
    return larea + rarea;
```

Producers and Consumers (pipelines)

- Threads may act as filters — consuming output from upstream threads and producing for downstream.
- Example: Unix pipelines

```
                    sed -
f Script $* | tbl | eqn | groff Macros -
```

Pipes acts as bounded FIFO queues.

Clients & Servers

- Dominant pattern for distributed systems.
- Distributed analog to procedure call.
- Examples: Remote file systems, http, ftp, telnet.
- Also OS kernels: Kernel is a set of kernel-mode threads that services system calls on behalf of user-level threads.
- Servers may service multiple clients, possibly concurrently.

Simple multithreaded server pseudocode:

```
thread server[ s = 1 to n ] {
    while( system is not shutdown ) {
        await new client
        while true
            receive request from client
            process request
            send reply
            if( client requested quit ) break
        clean up } }
```

Peers

Similar distributed threads cooperate to accomplish a task.

Example: Distributed Matrix Multiplication

Assume an n by n matrix and n distributed workers arranged in a ring. Each worker is responsible for computing one column of matrix C.

```
thread worker[i = 0 to n-1] :
    double a[n]; # row i of a
    double b[n]; # one column of b
    double c[n]; # row i of c (result)
    receive a ; # row i from coordinator
    receive b ; # col i from coordinator
    int j = i;
    ##Inv: b holds column j of matrix B.
    loop
        c[j] := 0.0 ;
        for [k = 0 to n-1] c[j] += a[k] * b[k];
        j := (j-1) mod n; # subtract 1 from j
        break if j=i
        send b to worker[(i+1) mod n];
        receive b; # col j
    send (i,c) to coordinator
```

The coordinator

```
thread coordinator :
    for[ i = 0 to n-1 ] send A[i][*] to worker[i]
    for[ j = 0 to n-1 ] send B[*][j] to worker[j]
    for[ i = 0 to n-1 ] receive C[i][*] from worker[i]
```

- First each row of A is sent to a worker.
- Each column of B is sent to a worker.
- The workers pass the columns of B among themselves (in a ring) until each worker has seen all n columns of B.
- The rows of C are now sent from the workers to the coordinator.

Connectivity required

- Workers in a (1-way) ring.
- All workers connected (2-way) to the coordinator.

Communication between threads in a concurrent application

Generally threads communicate

- Through shared memory.
- By passing messages between them.

	Message Passing	**Shared memory**
Threads in the same process	Uncommon	Yes
Processes on the same machine	Yes	Possible
Processes on different machines	Yes	Not possible

Threads in same process

- Sharing memory: Since threads in the same process share global and heap memory, communicating by shared memory is the usual way for such threads to communicate.
- Message passing: This is less efficient and rarely done.

Processes on the same machine:

- Sharing memory (in UNIX)
  - By default processes do not share memory.
  - The `fork` system call creates a private address space for each process.
  - But it is possible for processes to create memory segments later and share them with other processes on the same machine.
- Message passing (in UNIX)
  - Pipes can be used.
  - FIFOs are like pipes, but with names.
  - So can TCP or UDP sockets

Processes on different machines:

- Sharing memory:
  - Not possible. Otherwise we'd consider it the same machine.
- TCP or UDP sockets are commonly used.
- So are higher-level protocols such as HTTP and HTTPS

I'm going to focus on sharing memory on one computer within one process.

End of slide set.