Concurrent Programming

January 8, 2015

Introduction

Introduction to using threads and thread management techniques.

Problem

- Why would we want to use threads?
- Lets say we have a service that performs a simple task. Take in sets of number sequences and sorts them using quicksort. $f(S) = S'$ such that $S = \{s_i | s_i = \{a_1, ...a_n\}, 1 <= i <= m; m, n \in \mathbb{N}\}$ and $S = \{S' | s_i \text{ is sorted}\}$. $f(S)$ will then call $Q(s_i)$ on each sequence m times.

Serial Example

- A simple program may contain a single loop in f that calls the quicksort function at each iteration.

```
int main() {
  // A vector with 10,000 vectors of size 1,000 with unsorted integers.
  std::vector< std::vector< int > > S = { std::vector<int>(1000),
                                          /*...*/
                                          std::vector<int>(1000) };

  for(std::vector< std::vector< int > >::iterator s_i = S.begin();
      s_i != S.end(); ++s_i)
  {
    sort(*s_i);
  }
}

void sort(std::vector< int > & to_sort) { /* .... */ }
```

Serial Example

Advantages

- Easy to write.

Disadvantages

- Slow, takes about 2.2 seconds to complete the operation on a 3GHz Core 2 Duo.
- Most computers have at least two cores.
- Only single core is being used.
- Other cores wasted.

Parallel Example

- A program that utilizes threads to speed up the process would spawn multiple threads.

```
int main() {
  // A vector with 10,000 vectors of size 1,000 with unsorted integers.
  std::vector< std::vector< int > > S = { std::vector<int>(1000),
                                          /*...*/
                                          std::vector<int>(1000) };

  std::vector< std::thread > threads;
  //Spawn thread to sort each subvector.
  for(std::vector< std::vector< int > >::iterator s_i = S.begin();
      s_i != S.end(); ++s_i)
  {
    threads.push_back(std::thread(sort, std::ref((*s_i))));
  }

  // Wait for each thread to complete its work.
  for(std::vector< std::thread >::iterator t = threads.begin();
      t != threads.end(); ++t)
  {
    (*t).join(); // or t->join();
  }
}
```

Parallel Example

Advantages

- Takes less time to execute. 1.3 seconds on a 3GHz Core 2 Duo.

Disadvantages

- More complex.
- Keep track of threads.
- New errors may arise.

Use Cases

- Operating Systems: Linux, Windows, and Unix.
- Graphical interfaces use event driven multithreading to preserve responsiveness.
- Games, separation of input, physics, and rendering.
- Web server technologies such as databases, search engines, and web servers.
- HMMER
- Bioinformatics.

History

Early Multithreading & Multitasking Systems

- Early Machines
  - Single process model.
  - Batch processing.
- Berkeley Timesharing System
  - Give processes **time-slots** of execution.
  - Memory is shared.
  - Computer remains usable for other operators.
- Unix
  - Processes now have dedicated memory.
  - Later, threading support added. Subprocesses that share memory with the processes.

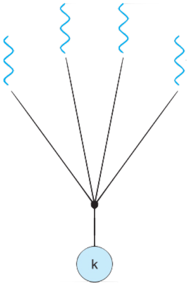
Threading Architecture

Software

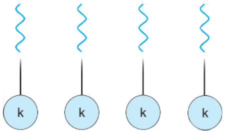
- Operating systems have built in thread management.
  - Distributed operating systems.
- Processes run separately.
- Pipes and sockets used for process communication.
- Subprocess support (threads) that share memory with processes.

Threading Models

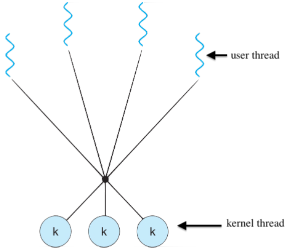
- kernel threads (most kernel threads on Linux are processes).
- user threads (threads that processes spawn).



(a) n:1



(b) 1:1

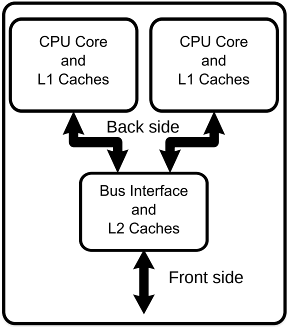


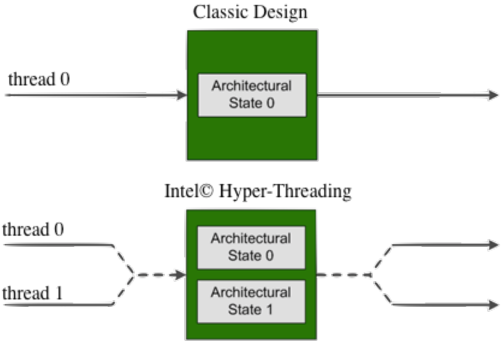
(c) n:m

Figure 1: Threading Models: Retrieved from *Operating System Concepts*

Hardware

- Duplication of registers to store multiple states (Intel Hyper-threading).
  - Threads can still lock while waiting for CPU resources.
- Intel Hyper-Threading.
- Multiple cores.
- Multiple sockets.





Multithreaded Programming

Overview

- Minimal code example.
- Spawning a thread from a function.
- Terminating a thread (join, detach).
- Atomic Operation.
- Mutex (Locks).
- Semaphore.
- Lock-free data structure.

Minimal Working Example

```
void sort(std::vector< int > & to_sort) {
  std::sort(to_sort.begin(), to_sort.end());
}

void parallel_sorting(std::vector< std::vector<int> > & T) {
  std::vector< std::thread > threads;
  for(std::vector< std::vector< int > >::iterator s_i = T.begin();
        s_i != T.end(); ++s_i)
  {
    threads.push_back(std::thread(sort, std::ref(*s_i)));
  }

  for(std::vector< std::thread >::iterator t = threads.begin();
        t != threads.end(); ++t)
  {
    (*t).join();
  }
}

int main(int argc, char * argv[]) {
    std::vector< std::vector<int> > T;
    fill_with_vectors(T, 10000, 1000);
    parallel_sorting(T);
    return 0;
}
```

Threads in C++11

- After we have created `T` with vectors of random integers. We pass `T` to the `parallel_sorting` function.
- The first line of code `std::vector< std::thread > threads` is a vector that contains thread objects. If thread needs to be terminated with later then it is necessary to keep track of it.
- Second we iterate through the vectors in `T` and spawn a thread to sort each vector with:
  - `threads.push_back(std::thread(sort, std::ref(*s_i)));`
- `sort` is the function that defines the instructions for the thread instance.
- `std::ref(*s_i)` tells the thread constructor that `sort` requires `std::vector<int> &` as a parameter.

Defining a Thread Function

- Threads require a function to execute since they are subprocess.
- The thread function `sort` is a wrapper for `std::sort` from the C++ Standard Library (stdlib)

```
void sort(std::vector< int > & to_sort) {
    std::sort(to_sort.begin(), to_sort.end());
}
```

Terminating a Thread

- Going back to the `parallel_sorting` lets take a look at the section of code where the `join` method is called.
- There are two methods dealing with thread termination. First we can **join** a thread.
  - Joining a thread will cause the thread calling the join method to block until the thread completes its task.
- If the termination of the thread is not important to the state of the program then **detaching** a thread will cause the thread to continue executing until the OS destroys it when the program quits.

Thread Communication

Thread Communication

Suppose we have an application that records the number of clients serviced. What problem may arise from the code?

```
void handle_request(int & requests_served) {
  std::this_thread::sleep_for(std::chrono::milliseconds(100));
  /* Do stuff */
  requests_served++;
}

int main(int argc, char * argv[]) {
  int requests_served = 0;
  std::vector< std::thread > threads;
  for(int i = 0; i < 1000; i++) {
    threads.push_back(std::thread(handle_request,
      std::ref(requests_served)));
  }
  for(std::vector< std::thread >::iterator t = threads.begin();
        t != threads.end(); ++t)
  {
    t->join();
  }
  std::cout << "Handled " << requests_served << " requests." << std::endl;
  return 0;
}
```

Race Condition

- When `requests_served++` is executed a race condition may occur.
- `requests_served++` is not an atomic operation. Expanding it to machine code would result in:

$$register_1 = requests\_served \tag{1}$$
$$register_1 = register_1 + 1 \tag{2}$$
$$requests\_served = register_1 \tag{3}$$

- If a context change were to arise between lines 1 and 2 or 1 and 3. Then there is the possibility that *requests_served* will have changed due to another thread. Which would make $register_1$ inconsistant with *requests_served*.

Solution?

- Ensure the threads don't overstep other threads with atomic operations.
- atomic operations guarantee consistency across threads when a variable is modified.

Atomic Operations

- Atomic operations used to only use features provided by the operating system to guarantee process syncronization.
- Multicore processors now provide special instructions to aid the operating system.
- An atomic operation allows the modification of a single variable.
- C++11 stdlib provides atomic types.
- Rundown of the different atomic operations.

Atomic Operations

Code updated to make use of atomics.

Atomic Operation Example

```
void handle_request(std::atomic<int> & a_requests_served) {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    /* Do stuff */
    a_requests_served++;
}

int main(int argc, char * argv[]) {
  std::atomic<int> a_requests_served(0);
  std::vector< std::thread > threads;
  for(int i = 0; i < 1000; i++) {
    threads.push_back(std::thread(handle_request,
      std::ref(requests_served)));
  }
  for(std::vector< std::thread >::iterator t = threads.begin();
      t != threads.end(); ++t)
  {
    t->join();
  }
  std::cout << "Handled " << a_requests_served << " requests." << std::endl;
  return 0;
}
```

Critical Section

- Any section of code that reads or writes to data that is shared amongst threads.
- Must satisfy three requirements ensure consistency.
    1. **Mutual Exculsion**: If a thread is in a critical section then other threads must wait for it to exit the section.
    2. **Progress**: A thread cannot wait inside of a critical section. Waiting can cause a *deadlock*.
    3. **Bounded Waiting**: Threads shall not hoard the critical section.

Mutexes

- Atomic operations are simple, but you cannot lock multiple lines of code.
- Mutexes allow you to declare a critical section and limit access to a single thread.
- Mutexes use locks to identify a critical section of code.

Mutex Example

- The *Atomic Operation Example* has been modified to use a mutex instead.

```
std::mutex mlock;

void handle_request(int & requests_served) {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    mlock.lock();
    /* Do stuff */
    requests_served++;
    mlock.unlock();
}
```

Semaphores

- A mutex is a semphore that only allows a single thread to access a critical section.
- More advanced data structure that provides mutual exclusion access to a critical section.
- Unlike a classic mutex a semaphore keeps count of the threads that want to access a resource.
- Designed to allow multiple threads access a critical section.
- Two operations used.
  - `wait(semaphore)`: Thread is blocked until another thread calls signal.
  - `signal(semaphore)`: Thread calls signal to indicate exit of critical section and allow another thread to enter.

Building a Semaphore in C++11

- Tools Needed
  - mutex: See example in previous slides.
  - condition_variable: Class that manages the execution of threads that call wait on a given lock.
  - primitive to keep track of number of threads in the critical zone.

Building a Semaphore in C++11

Semaphore C++11

```
class semaphore {
    private:
        std::mutex    _mtx;
        std::condition_variable _cv;
        int _count;

    public:
        semaphore(int count = 1): _count(count) {}
        void signal() {
            std::lock_guard<std::mutex> lck(_mtx);
            _count++;
            _cv.notify_one();
        }
        void wait() {
            std::unique_lock<std::mutex> lck(_mtx);
                            /* C++11 Anonymous (lamda) Function  */
            _cv.wait(lck, [this](){ return _count > 0; });
            _count--;
        }
};
```

Readers-Writers Problem

- The semaphore can then be used to synchronize communication between a writer thread and set of reader threads.

Reader

```
void writer(somedata & shared_data, semaphore & wrt) {
  while(1) {
    wrt.wait();
    \\\< Write to shared data.
    wrt.signal();
    total_writes++;
  }
}
```

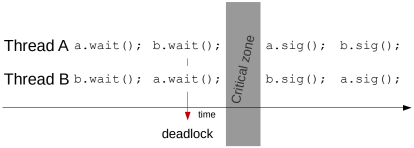
Readers-Writers Problem

Writer

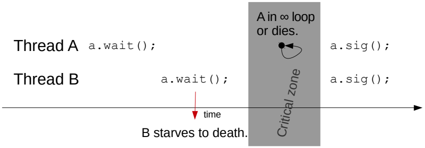
```
void reader(somedata & shared_data, semaphore & wrt,
    semaphore & mtx) {
  while(1) {
    mtx.wait();
    read_count++;
    if(read_count == 1) {
      wrt.wait();
    }
    mtx.signal();

    rd.wait();
    \\\< Read the shared data.
    rd.signal();

    mtx.wait();
    read_count--;
    if(read_count == 0) {
      wrt.signal();
    }
    mtx.signal();
  }
}
```

Deadlocks

- What can cause deadlocks? There exists four conditions.
  1. **Mutual exclusion**: There exists at least a single resource that can be held in a non-sharable mode.
  2. **Hold and wait**: Thread holds onto a resource and waits for another resource to be freed.
  3. **No preemption**: Threads cannot be stripped of their resources by other threads.
  4. **Circular wait**: When two or more threads are holding and waiting on shared resources.

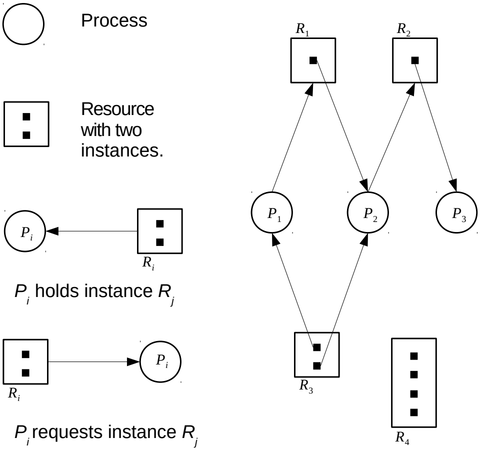


(c) Mutual Exclusion



(d) No Preemption | Hold & Wait

Resource-Allocation Graph



(e) Key

Figure 2: Adopted from *Operating System Concepts*

Lock-Free Programming

- Another method to prevent deadlocks is to use lock-free constructs.
- A lock-free structure guarantees throughput, but doesn't prevent starvation.
- Need to make use of atomic operations to construct the lock free data structure.
  - Atomic types, for example `std::atomic<T>`.
  - Atomic compare and swap (CAS). Such as `std::atomic_compare_exchange_*`

Advantages

- Guarantees no deadlocks.
- Scalable.

Disadvantages

- May be slower than lock-based structures.
- More difficult to implement.

Wait-Free Programming

- Subset of lock-free programming that ensures all threads complete their task within a finite set of steps.

Advantages

- Eliminates thread starvation.
- Scalable.

Disadvantages

- Runs slower than locked-free structures.

Inter-Process Communication

Forking

- How about process level parallelism? Use the `fork` command.
- *forking* a process will create a *child* (new) process that is an exact copy of the *parent* (calling) process except:
  - Locks are not preserved (including file locks).
  - Other threads from the parent process are not copied. Only the thread that forked the process is copied.
  - Process IDs are not preserved. The child will be assigned new IDs.
  - For more exceptions refer to the POSIX.1-2008 specifications for `fork()`.

Forking

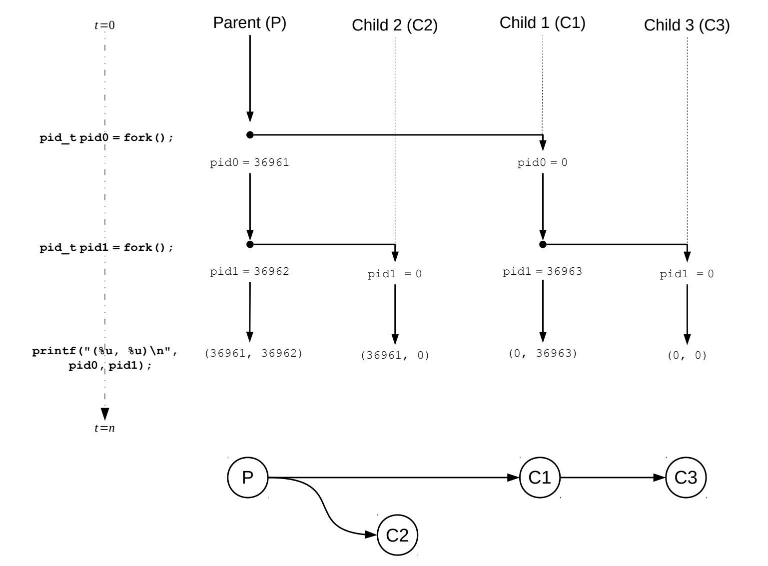
- How many times will `printf` be called? How will we know what `printf` was called by the root process?

Forking Example

```
int main(int argc, char * argv[])
{
  pid_t pid0 = fork(); // fork returns 0 to the child process.
  pid_t pid1 = fork();

  printf("(%u, %u)\n", pid0, pid1); // print two unsiged numbers.
}
```

Forking

Forking Diagram



Pipes

- Shared between the parent and child process (or multiple children). Different processes cannot share pipes.
- Enables communication between the parent and child process.
- Ordinary pipes provide unidirectional communication so that one end can be written to (write-end) and the other read from (read-end).

Pipe

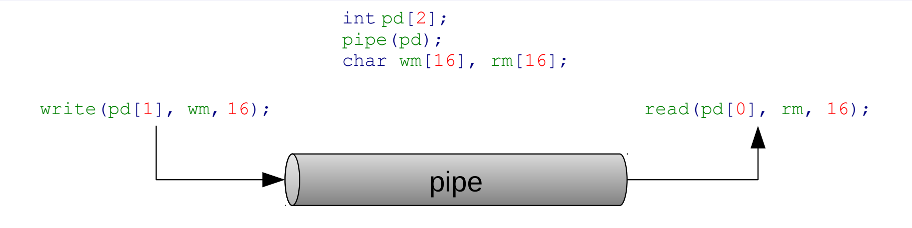


Figure 3: Visual Representation of a Pipe: Adopted from *Operating System Concepts.*

Named Pipes: FIFO

- Referred to as First-In First-Out (FIFO) on POSIX/Unix sytems.
- Enables communication between separate processes.
- Represented as a special file handle that points to a location in memory.
- Functionality similar to pipes; except bidirectional communiction is possible. Unlike a pipe, reading and writing from the same file descriptor is possible.
- More overhead.
  1. Create the FIFO file.
  2. Open the FIFO.
  3. Read/Write to the FIFO.

Sockets

- Sockets provide full-duplex communication streams.
- Remote connections across the network.
- Primary tool to setup client-server communication model.

Advantages

- Dynamic: Allows the distribution of processes across multiple machines.
- More abstracted since the machines could be running their own operating system.

Disadvantages

- More overhead to set up.
  - Create a socket.
  - Bind the socket to an address.
  - Connect to the socket.
- Slower than FIFOs and Pipes since the data passes through the network stack.

Threading Revisited

Thread Patterns Revisited

- Event driven designs.
- Thread pools
- Schedulers

Event Driven Designs

- Performance is not always a priority when using threads.
- Responsiveness may be another reason.
- Graphical User Interfaces, servers, and other producer-consumer patterns.

Thread Pools

- Solution for when system resources are limited and thrashing may happen.
- It is expensive to create threads. Therefore create a set of threads for later use.
- Pass jobs to the thread pool which then hands the jobs over to the threads.
- Thread pools are scalable, depending on the system resources the number of threads can be increased or decreased.

Schedulers

- Pools of threads do not guarantee optimal execution.
- Different threads will have varying execution times.
- Use a scheduler to ensure the desired optimal performance is achieved.
- Using a defined set of heuristics the scheduler will dequeue and run the desired threads from the pool.

Limitations

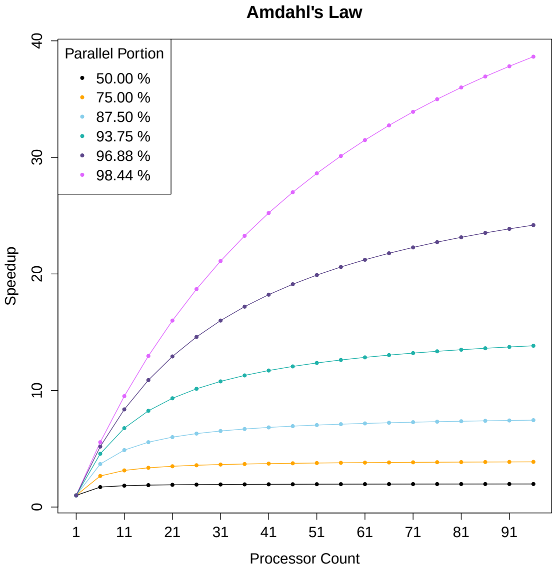
Amdahl's Law

- Determine the potential code speedup with Amdahl's Law.
  - This version assumes that a case of parallelization.

$$T(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

- P is a value between 0 and 1. P is the fraction of the program that is executed in parallel.
- As P approaches 1 then the program becomes more parallelized.
- If $P == 1$ then the program is solving an *embarrassingly parallel* problem. The speedup is linear to the number of cores.
- N is the count of processors.
- Amdahl's law assumes a fixed problem size. Which causes a diminishing returns effect as the number of cores increase.

Amdahl's Law


Amdahl's Law
Parallel Portion
50.00 %
75.00 %
87.50 %
93.75 %
96.88 %
98.44 %
Speedup
0
10
20
30
40
1
11
21
31
41
51
61
71
81
91
Processor Count

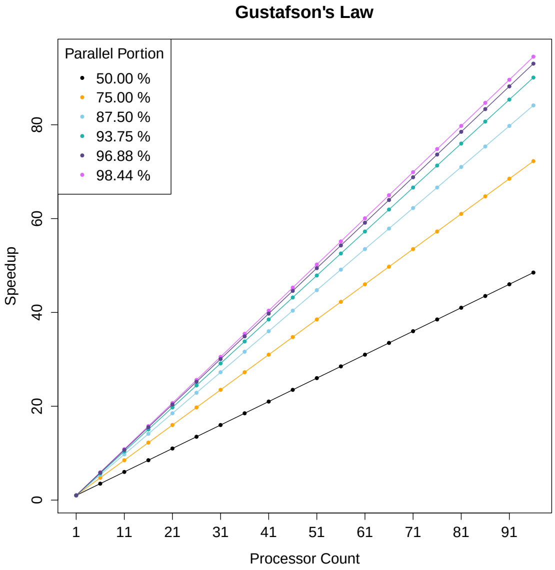
Gustafson's Law

- Unlike Amdahl's Law, Gustafson's Law assumes that the program will work on larger problems to utilize all of the processors.

$$S(N) = N - P(N - 1)$$

- P is the fraction of the program that is parallel.
- N is the number of processors.

Gustafson's Law



Thrashing / IO / Starvation / Busses

- If an application spawns too many threads then the CPU will spend more time on context switches between the threads instead of executing the threads.
- Threads that perform a lot of I/O bound operations will be limited to the speed of the resources that they share.
  - Data that needs to be passed through a bus will limit thread performance.