Concurrent Programming

Introduction

Introduction to using threads and thread management techniques.

NOTE: To whom ever is reading this document. Pandoc processes divs that inherit from the 'notes' class as a note. These will not show up in your slides. To compile slides with notes run `make notes`. You will need to be running pandoc –version $>=$ 1.12.A

- All of the examples in these lectures will make use of C++11 features. C++11 provides thread abstractions that should in theory work on any OS.

Concurrent Programming

Problem

- Quicksorting sets of subsequences is a good example since the subsequences can be of any size and the quicksort's performance may vary.

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.

Concurrent Programming

Serial Example

- Code is in `Concurrency/parallel_soring_example`.

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) { /* .... */ }
```

Concurrent Programming

Parallel Example

- Spawning 10,000 threads probably isn't a smart idea.

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();
  }
}

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

Concurrent Programming

Use Cases

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.

- `http://www.valvesoftware.com/publications/2007/GDC2007_SourceMulticore.pdf`
- Web server technologies such as Apache, Nginx, and Microsoft ISS emplay multithreading to separate requests.
- `http://en.wikipedia.org/wiki/Event-driven_programming`

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.

- `http://www.faqs.org/faqs/os-research/part1/section-10.html` provides a nice and simple intro to the history of threading.
- Chapter 4 in *Operating System Concepts* contains additional information about threads.
- `http://en.wikipedia.org/wiki/Unix`

Threading Models

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

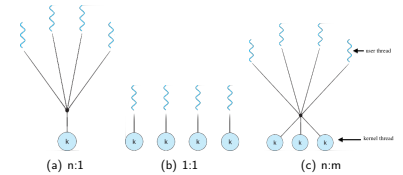
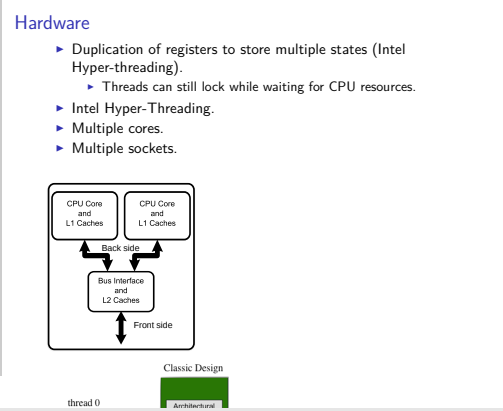


Figure 1: Threading Models: Retrieved from *Operating System Concepts*

- Refer to Chapter 4: **Operating System Concepts 8e.** for more information on threading models.
  - Book states that Linux uses the one-to-one model.



- Intel Hyper-Threading Image adopted from `https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology?language=es`.
- Multi-core processor diagram was retrieved from `https://en.wikipedia.org/wiki/Multi-core_processor`

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;
}
```

- Includes at the top of the example.
  - `stdlib.h`, `iostream`, `algorithm`, `thread`, and `utils.h`.
- `utils.h` contains `fill_with_vectors(std::vector< std::vector<int> >, int, int)`.

- `#include<thread>` a C++11 wrapper for various system threads. In the case of Linux and Unix it is a PThread (POSIX) thread wrapper.

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.

- 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());
}
```

- The STL is not the C++ Standard Library. It just stands for Standard Library which was created by Alexander Stepanov.
  - `http://stackoverflow.com/questions/5205491/whats-this-stl-vs-c-standard-library-fight-all-about`

- What problems may arise from the code?
  - Race conditions. *Critical Section*
  - Operations that appear atomic may not be once compiled.
  - Provide Assembly example of race condition.

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:

$$register_1 = requests\_served \quad (1)$$
$$register_1 = register_1 + 1 \quad (2)$$
$$requests\_served = register_1 \quad (3)$$

- 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*.

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.

- Refer to page 228 chapter 6 in *Operating System Concepts* 8th Edition.

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.

- Explanation and definition of semaphores came from lecture 6 in the lecture materials from *CSE 120: Principles of Operating by Systems Alex C. Snoeren* (also included in the resources directory).
- Page 189 from the *Art of Multiprocessor Programming* by *Maurice Herlihy & Nir Shavit*.

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--;
        }
};
```

- It most cases it is recommended to use an existing library for semaphores.
- Such as boost or the POSIX defined `semaphore.h`
- lock_guard will lock within the scope using a mutex.
- unique_lock allows the condition variable to associate a set of threads to a common lock and defer their execution.
- condition_variable
  - `notify_one()` will wake up a sleeping thread.
  - `wait(unique_lock &, predicate)` will put a thread to sleep. `predicate` determines if a thread should go back to sleep after a spurious wakeup.
  - `predicate` is a C++11 anonymous function.

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++;
  }
}
```

- Refer to page 241 from *Operating System Concepts 8 e.* or slide 10 in the lecture slides provided by Alex C. Snoeren CSE 120.
- Refer to `main.cpp` in the `semaphore_example` for a working demonstration of the readers/writers problem.

Readers-Writers Problem

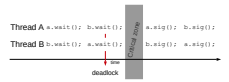
Writer

```
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();
  }
}
```

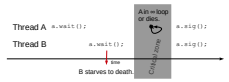
- Refer to page 241 from *Operating System Concepts 8 e.* or slide 10 in the lecture slides provided by Alex C. Snoeren CSE 120.
- Refer to `main.cpp` in the `semaphore_example` for a working demonstration of the readers/writers problem.

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



- Chapter 7 from *Operating System Concepts* covers deadlocks and deadlock avoidance.
  - Section 7.2 covers the four conditions necessary for a deadlock.
  - `http://nob.cs.ucdavis.edu/classes/ecs150-1999-02/dl-cond.html`
- Figure (d) Thread A goes into the critical zone and holds a resource and waits for another resource to be freed. Thread B starves while waiting.

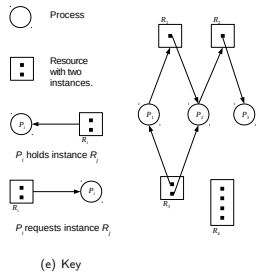


Figure 2: Adopted from *Operating System Concepts*

- Chapter 7.4 Deadlock Prevention from *Operating System Concepts* and the Chapter 7 deadlock slides has more information.
- Represent resource usage as a directed acyclic graph.
  - If any cycles exist then there may exist a deadlock.

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.

- Page 60 from *The Art of Multiprocessor Programming* provides a nice definition of lock-free programming.
- Dr. Dobbs has a nice article about lock-free programming.
  - `http://www.drdobbs.com/lock-free-data-structures/184401865`.
- Lock-Free programming is a difficult subject.
- libcds provides a set of concurrent data structures `http://libcds.sourceforge.net/`
- Nice set of Youtube videos about lock-free programming.
  - CppCon by Herb Sutter Part 1: `https://www.youtube.com/watch?v=c1gO9aB9nbs`
  - CppCon by Herb Sutter Part 2:
  -

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()`.

- Provide simple code example and image.
- Section 4.4.1 in *Operating System Concepts* mentions that there exists fork implementations that will duplicate all threads when `fork` is called. Most versions of `fork` will only duplicate the thread that called the function. The book doesn't provide any references so it may be safe to assume that `fork` only duplicates the calling thread.
- `man 2 fork` on OS X and Linux will provide usage details.
- The Open Group provides the specifications of `fork` on a POSIX.1-2008 compatable system. `http://pubs.opengroup.org/onlinepubs/9699919799/`

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.
}
```

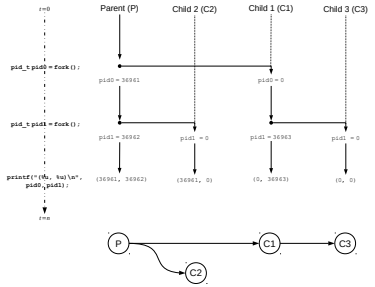
- `fork` will return the child pid to the parent and 0 to the child.
- `printf` requires `#include <stdio.h>`
- `fork` requires `#include <unistd.h>`
- Program Output:

  (36961, 36962)

  (36961, 0)

  (0, 36963)

  (0, 0)



- A Visual example of the process behind `fork()`.
- The graph below shows the inheritance order.

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

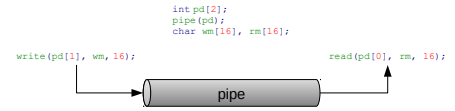


Figure 3: Visual Representation of a Pipe: Adopted from *Operating System Concepts*.

- Provide simple code example and image.
- Refer to *Operating System Concepts* section 3.6.3: Pipes.
- Refer to the man pages
  - `man 2 pipe`
  - `man 2 read`
  - `man 2 write`

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.

- Provide simple code example and image.
- Refer to the operating systems book for more information.
- Refer to the man pages.
  - `man 2 mkfifo`
  - `man 2 open`
  - `man 2 read`
  - `man 2 write`
  - `man 2 close`

- Refer to the man pages:
  - `man 2 socket`

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.

- 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.

- There isn't any defined thread pool class in C++11.

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.

- `http://en.wikipedia.org/wiki/Amdahl%27s_law`
- `http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980`
- Amdahl's Law assumes the dataset that the program is working on is static in size.
- For example a program that only parallelizes a fixed sized problem would follow this rule. Let's say that a program will always multiply two 1000x1000 matrices. Then the speedup of that program will follow Amdahl's law as the number of processors increase.

- 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.

- http://en.wikipedia.org/wiki/Gustafson%27s_law
- http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980