Lecture 28: Concurrency
Continued...

Lecture Participation Poll #28

Log onto pollev.com/cse374
Or
Text CSE374 to 22333
Administrivia

- HW 5 (final HW) posted
- Final review assignment posted!
- End of quarter due date Wednesday December 16th @ 9pm
Concurrency vs Parallelism

- **Parallelism** refers to running things simultaneously on separate resources (ex. Separate CPUs)
- **Concurrency** refers to running multiple threads on a shared resources
- Concurrency is one person cooking multiple dishes at the same time.
- Parallelism is having multiple people (possibly cooking the same dish).
- Allows processes to run ‘in the background’
  - Responsiveness – allow GUI to respond while computation happens
  - CPU utilization – allow CPU to compute while waiting (waiting for data, for input)
  - Isolation – keep threads separate so errors in one don’t affect the others
Concurrency

- A search engine could run concurrently:
  - Example: Execute queries one at a time, but issue I/O requests against different files/disks simultaneously
  - Could read from several index files at once, processing the I/O results as they arrive
  - Example: Web server could execute multiple queries at the same time
    - While one is waiting for I/O, another can be executing on the CPU
- Use multiple “workers”
  - As a query arrives, create a new “worker” to handle it
  - The “worker” reads the query from the network, issues read requests against files, assembles results and writes to the network
  - The “worker” uses blocking I/O; the “worker” alternates between consuming CPU cycles and blocking on I/O
  - The OS context switches between “workers”
    - While one is blocked on I/O, another can use the CPU
    - Multiple “workers’” I/O requests can be issued at once
- So what should we use for our “workers”?
Threads

- In most modern OS’s threads are the *unit of scheduling.*
  - Separate the concept of a process from the “thread of execution”
  - Threads are contained within a process
  - Usually called a thread, this is a sequential execution stream within a process

- Cohabit the same address space
  - Threads within a process see the same heap and globals and can communicate with each other through variables and memory
  - Each thread has its own stack
  - But, they can interfere with each other – need synchronization for shared resources

- Advantages:
  - They execute concurrently like processes
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores

- Disadvantages:
  - If threads share data, you need locks or other synchronization
    - Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - Lock contention, context switch overhead, and other issues
  - Need language support for threads

A *Process* has a unique: address space, OS resources, and security attributes
A *Thread* has a unique: stack, stack pointer, program counter, and registers
Threads are the *unit of scheduling* and processes are their *containers;* every process has at least one thread running in it
Address Spaces

**Before creating a thread**
- One thread of execution running in the address space
  - One PC, stack, SP
- That main thread invokes a function to create a new thread

\[
pthread\_create()\]

**After creating a thread**
- *Two* threads of execution running in the address space
  - Original thread (parent) and new thread (child)
  - New stack created for child thread
  - Child thread has its own *values* of the PC and SP
- Both threads share the other segments (code, heap, globals)
  - They can cooperatively modify shared data

---

Single threaded address space

<table>
<thead>
<tr>
<th>OS kernel [protected]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack _parent</td>
</tr>
<tr>
<td>Shares Libraries</td>
</tr>
<tr>
<td>Heap (malloc/free)</td>
</tr>
<tr>
<td>Read/Write Segments _data, _bss</td>
</tr>
<tr>
<td>Read-Only Segments _text, _rodata</td>
</tr>
</tbody>
</table>

Multi-threaded address space

<table>
<thead>
<tr>
<th>OS kernel [protected]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack _child</td>
</tr>
<tr>
<td>Shares Libraries</td>
</tr>
<tr>
<td>Heap (malloc/free)</td>
</tr>
<tr>
<td>Read/Write Segments _data, _bss</td>
</tr>
<tr>
<td>Read-Only Segments _text, _rodata</td>
</tr>
</tbody>
</table>
POSIX Threads and Pthread functions

- The POSIX APIs for dealing with threads
  - Declared in pthread.h
  - Not part of the C/C++ language (cf. Java)
  - To enable support for multithreading, must include -pthread flag when compiling and linking with gcc command
  - POSIX stands for Portable Operating System Interface, pthread conforms to POSIX standard for threading

```
gcc -g -Wall -std=c11 -pthread -o main main.c
```

- Example Usage
  - `pthread_t thread ID;`
    - the threadID keeps track of to which thread we are referring
  - `pthread_create` takes a function plинтер and arguments to trigger separate thread
    - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start routing) (void*), void *arg);`
    - note – pthread_create takes two generic (untyped) pointers
      - interprets the first as a function pointer and the second as an argument pointer
  - `int pthread_join(pthread_t thread, void **value_ptr);`
    - puts calling thread 'on hold' until 'thread' completes – useful for waiting to thread to exit

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html
Creating and Terminating Threads

**Creating a Thread**

- Use `pthread_create` to create a new thread.

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

- Creates a new thread into `*thread`, with attributes `*attr` (NULL means default attributes).
- Returns 0 on success and an error number on error (can check against error constants).
- The new thread runs `start_routine(arg)`.

**Terminating a Thread**

- Use `pthread_exit` to terminate a thread.

```c
void pthread_exit(void* retval);
```

- Equivalent of `exit(retval)`; for a thread instead of a process.
- The thread will automatically exit once it returns from `start_routine()`.
Multi Threaded Example

```c
#include <stdio.h>
#include <pthread.h>

void do_one_thing(int *pnum_times) {
    int i, j, x;
    for (i = 0; i < 4; i++) {
        printf("doing one thing\n");
        for (j = 0; j < 10000; j++) x = x + i;
        (*pnum_times)++;
    }
}

void do_another_thing(int *pnum_times) {
    int i, j, x;
    for (i = 0; i < 4; i++) {
        printf("doing another \n");
        for (j = 0; j < 10000; j++) x = x + i;
        (*pnum_times)++;
    }
}

int main() {
    pthread_t thread1, thread2;
    int r1 = 0, r2 = 0;
    pthread_create(&thread1, NULL, (void *) do_one_thing, (void *) &r1);
    pthread_create(&thread2, NULL, (void *) do_another_thing, (void *) &r2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    do_wrap_up(r1, r2);
}
```

```c
void do_wrap_up(int one_times, int another_times) {
    int total;
    total = one_times + another_times;
    printf("All done, one thing %d, another %d for a total of %d\n", one_times, another_times, total);
}
```
Parallel Processing

- common pattern for expensive computations (such as data processing)

1. split up the work, give each piece to a thread (fork)
2. wait until all are done, then combine answers (join)
   - to avoid bottlenecks, each thread should have about the same about of work

- performance will always be less than perfect speedup

- what about when all threads need access to the same mutable memory?
After forking threads

- Waits for the thread specified by thread to terminate
- The thread equivalent of `waitpid()`
- The exit status of the terminated thread is placed in `*retval`

```c
int pthread_join(pthread_t thread, void** retval);
```

- Mark thread specified by thread as detached – it will clean up its resources as soon as it terminates

```c
int pthread_detach(pthread_t thread);
```
Race Conditions

- **race condition** happens when the result of a computation depends upon scheduling of multiple threads, i.e., the order in which the processor executes instructions.

- **Bad interleavings** is when the code exposes bad intermediate state.
  - Example: the `getBalance()` -> `setBalance()` calls exposed intermediate state.
  - Bad interleavings are incorrect from the programmatic logical perspective:
    - In the bank example, we lost money or allowed balances to go below 0.

- **Data races** - Even if we can't have a line-by-line interleaving, we can still have race conditions
  - What seems like an "atomic" operation, like setting "balance_ = amount" or "return balance_"., is actually NOT guaranteed to be an atomic operation at the compiled machine-code level.

  Whenever you have the potential to read+write or write+write on different threads, you MUST synchronize access to the shared memory (with a lock or similar).
Data Races

- Two memory accesses form a data race if different threads access the same location, and at least one is a write, and they occur one after another
  - Means that the result of a program can vary depending on chance (which thread ran first?)

- Data races might interfere in painful, non-obvious ways, depending on the specifics of the data structure

  **Example:** two threads try to read from and write to the same shared memory location
  - Could get “correct” answer
  - Could accidentally read old value
  - One thread’s work could get “lost”

  **Example:** two threads try to push an item onto the head of the linked list at the same time
  - Could get “correct” answer
  - Could get different ordering of items
  - Could break the data structure!
A Data Race

- Two threads are running at the same time, and therefore, because we cannot guarantee the exact speed at which each thread runs, we could get into a bad situation.

- Have a bank account x with a balance of $150.

- Thread T1 calls x.withdrawal(100) and thread T2 calls x.withdrawal(100) right afterwards.
  - Two transactions are attempting to happen on the same account.
  - What SHOULD happen is that one of the transactions succeeds in withdrawing 100, and the other throws an exception because the remaining balance of $50 is insufficient.

- T1 reads the balance (150) and stores it in variable b.

- T2 executes completely, deducting 100 from the account to leave a balance of 50.

- Rest of the function on T1 executes, comparing 150 with 100 (ok) and then setting the balance to $50.

- We've lost a transaction!

https://courses.cs.washington.edu/courses/cse374/18sp/lectures/24-concurrency-1.html
Synchronization

- **Synchronization** is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - “Let me go first, then you can go”
    - Many different coordination mechanisms have been invented

- **Goals of synchronization:**
  - **Liveness** – ability to execute in a timely manner
    (informally, “something good happens”)
  - **Safety** – avoid unintended interactions with shared data structures (informally, “nothing bad happens”)
Lock Synchronization

- Use a “Lock” to grant access to a critical section so that only one thread can operate there at a time
  - Executed in an uninterruptible
  - an operation we want to be done all at once
  - operation must be the right size (atomic unit)
    - too big program runs sequentially
    - too small program has data races

- Lock Acquire
  - Wait until the lock is free, then take it

- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

```c
// non-critical code
lock.acquire(); // critical section
lock.release();
// non-critical code
```
Example

- If your fridge has no milk, then go out and buy some more
  - What could go wrong?

- If you live alone:

  ```
  if (!milk) {
    buy milk
  }
  fridge.unlock()
  milk_lock.lock()
  if (!milk) {
    buy milk
  }
  milk_lock.unlock()
  ```

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?

- If you live with a roommate:

  ```
  fridge.lock()
  if (!milk) {
    buy milk
  }
  fridge.unlock()
  ```

- For performance reasons, only put what is necessary in the critical section
  - Only lock the milk
  - But lock all steps that must run uninterrupted (i.e. must run as an atomic unit)

  ```
  milk_lock.lock()
  if (!milk) {
    buy milk
  }
  milk_lock.unlock()
  ```
pthreads and Locks

- Another term for a lock is a mutex ("mutual exclusion")
  - pthread.h defines datatype pthread_mutex_t

- pthread_mutex_init()
  - Initializes a mutex with specified attributes

```c
int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr);
```

- pthread_mutex_lock()
  - Acquire the lock – blocks if already locked

```c
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

- pthread_mutex_unlock()
  - Releases the lock

```c
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

- pthread_mutex_destroy()
  - "Uninitializes" a mutex – clean up when done

```c
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```
Synchronization Example

- https://courses.cs.washington.edu/courses/cse374/20sp/lectures/cppcode/BankAccountThread.h
- https://courses.cs.washington.edu/courses/cse374/20sp/lectures/cppcode/BankAccountThread.cpp