ECE650 lecture 1 Notes

Concurrency and Synchronization

Part1 Concurrency/Multiprogramming/Multiprocessing

Q1. Multiprogramming VS uniprogramming?

Ans: Multiprogramming: More than one "unit of execution" at a time, Supported by most all current operating systems.

Uniprogramming: no concurrency, A characteristic of early operating systems, e.g., MS-DOS.

Q2. The important things related to a process?

1. A process contains many threads. A process is a unit of allocation.
2. Different processes have different memory addresses
3. The important components of a process are: execution context(program counter, stack pointer, registers), code, data, stack, separate memory views by virtual memory abstraction.
4. Each process created or managed by the operating system, process can create child processes.
5. Each process contains process ID number, process state, execution context(pc, sp,register), memory management, files, IO info.
6. The process is created via the system calls with os, fork(): make exact copy the current process and run. Return value of fork() differs between child and parent, child-> 0, parent-> child's PID. exec(): can follow fork() to run a different program, Exec takes filename for program binary from disk, Loads new program into the current process's memory.
7. A process may also create & start execution of threads, clone(), pthread_create(), etc.

Q3. The important things related to a thread?

1. A thread is a part of one process. A thread is a unit of execution.
2. If different threads belong to same process, they share same memory address, data(variables, array locations,objects).
3. Each thread has unique execution context(program counter, stack pointer, registers).
4. Threads can execute in one of the following ways:
   4.1 Multiple threads time-slice on 1 CPU with 1 hardware (HW) thread
   4.2 Multiple threads at the same time on 1 CPU with n HW threads
      4.2.2 Could be 1 CPU having many cores
      4.2.3 Or 1 CPU having 1 core that supports simultaneous multi-threading, Requires some duplicate HW and advanced scheduling of resources

   4.3 Multiple threads at the same time on m CPUs with n HW threads

5. Threads are usually created to cooperate (benefiting from a shared virtual memory of their process) to perform a task

6. Threads usually communicate with each other through reading from or writing to shared memory address or variables.

Part 2 Handling Race Conditions

Q1. What is a race condition?

"The condition where the system's substantive behavior is dependent on the sequence ortiming of other uncontrollable events. It becomes a bug when one or more of the possible behaviors is undesirable."

Q2. Why Race condition may not good?

Depends on execution timing, Non-deterministic result, Testing doesn't always reveal the problem.

Q3. How to fix race condition?

1. Do not add sleep() function call!
2. Use mutual exclusion(one thread at a time in critical section(a set of operations we want to execute atomically).
3. Mutual exclusion provided by lock operations.
4. This isn’t only an issue on machines with >1 HW thread

Part 3. Synchronization

Q1. What is Global Event Synchronization?

1. When using multiple threads, we often need barriers to synchronize their execution
2. Barriers may be use when one threads need to wait until all other threads to finish the execution task to start its own work.
3. use barrier synchronization only when needed and design to try to avoid using it

Q2. What is Point to Point Event Synchronization?

1. A thread notifies another thread so it can proceed
2. “producer-consumer” behavior
3. Shared memory parallel programs use semaphores, monitors, or flag variables to control synchronization and access

Q3. Lower level understanding of synchronization.

1. The assembly-version implementation of synchronization are not atomic, allows several threads to enter the critical section simultaneously.
2. We may use software-only solutions, Petersons algorithm, mutual exclusion for 2 threads.
3. In Petersons algorithm, Exit from lock() happens only if: the other process has not competed for the lock or the other process is competing, has set the turn to itself, and will be blocked in the while() loop.
4. Software-only solutions may be tricky to implement and different solutions for different memory consistency models need to be considered.

5. atomic operations that we can use for atomically locking/unlocking may help.
   5.1 Part of different processors' ISA/design
   5.2 Can execute in one cycle
   5.3 Test-and-set, compare-and-swap, fetch-and-increment
   5.4 Provide atomic processing for a set of steps, such as Read a location, capture its value, write a new value

Part 4. Multi-threaded Programming

Q1. How can we create multiple threads within a program?

1. C: pthreads, C++: std::thread or boost::thread

Q2. What will the threads execute?

1. Typically spawned to execute a specific function

Q3. What is shared vs. private per thread?

1. Shared: Recall address space
2. Private: Thread-local storage.

Q4. How to use POSIX pthreads?

1. #include<pthread.h> to your C source code
2. gcc -o p_test p_test.c -lpthread

Q5. What can pthread do?

1. Create threads, Wait for threads to complete, Destroy threads, Synchronize across threads, Protect critical sections.

Q6. How to create pthread?

1. int pthread_create(pthread_t* thread, pthread_attr_t* attr, void *(*start_routine)(void *), void* arg);
   1.1 pthread_t *thread_name – thread object (contains thread ID)
   1.2 pthread_attr_t *attr – attributes to apply to this thread
   1.3 void *(*start_routine)(void *) – pointer to function to execute
   1.4 void *arg – arguments to pass to above function

Q7. How to destroy a pthread?

1. pthread_join(pthread_t thread, void** value_ptr)
   1.1 Suspends the calling thread
   1.2 Waits for successful termination of the specified thread
   1.3 value_ptr is optional data passed from terminating thread's exit
2. pthread_exit(void *value_ptr)
   2.1 Terminates a thread

2.2 Provides value_ptr to any pending pthread_join() call

Q8. The use of pthread mutex?

1. 2 ways to initialize a mutex
   1.1 int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* mutex_attr);
   1.2 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
      1.2.1 Initialized with default pthread mutex attributes
2. Operate on the lock?
   2.1 int pthread_mutex_lock(pthread_mutex_t* mutex);
   2.2 int pthread_mutex_trylock(pthread_mutex_t* mutex);
   2.3 int pthread_mutex_unlock(pthread_mutex_t* mutex);

Q9. Read/write locks?

1. Declaration
   1.1 pthread_rwlock_t x = PTHREAD_RWLOCK_INITIALIZER;
2. Operations
   2.1 Acquire Read Lock: pthread_rwlock_rdlock(&x);
   2.2 Acquire Write Lock: pthread_rwlock_wrlock(&x);
   2.3 Unlock Read/Write Lock: pthread_rwlock_unlock(&x);
   2.4 Destroy Lock: pthread_rwlock_destroy(&x)

Q10. Read/Write lock behavior?

1. Lock has 3 states: unlocked, read-locked, write-locked
2. pthread_rwlock_rdlock(&x)
   2.1 If state = unlocked: thread proceeds & state becomes read-locked
   2.2 If state = read-locked: thread proceeds & state remains read-locked
      2.2.1 Internally, a counter increments to track # of readers
   2.3 If state = write-locked: thread blocks until state becomes unlocked
      2.3.1 Then state becomes read locked
3. pthread_rwlock_wrlock(&x)
   3.1 If state = unlocked: thread proceeds & state becomes write-locked
   3.2 If state = read-locked or state = write-locked
      3.2.1 Thread blocks until state becomes unlocked
      3.2.2 Then state becomes write-locked

Q11. Pthread barrier?

1. int pthread_barrier_init(pthread_barrier_t* barrier, const pthread_barrierattr_t* barrier_attr, unsigned int count);
2. pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(count);
3. int pthread_barrier_wait(pthread_barrier_t* barrier);

Q12. C++ thread?

1. C++11, Support for similar features as pthreads

Q13. Thread local storage?

1. Mechanism to allocate variables such that there is 1 per thread
2. Can be applied to variable declarations that would normally be shared
3. Indicated with the __thread keyword:

Q14. C++ synchronization?

1. #include<mutex>
2. Barriers: use boost::barrier