Thread

* A new abstraction for a single-running process
* Multi-threaded program
  + A multi-threaded program has more than one point of execution
  + Multiple PCs (program counters)
  + They share the same address space

Context switch between threads

* Each thread has its own program counter abd set of registers
* One or more thread control blocks (TCBs) are needed to store the state of each thread
* When switching from running one (T1) to running the other (T2):
  + The register state of T1 is saved
  + The register state of T2 is restored
  + The address space remains the same

The stack

* There is one stack per thread

Why use threads?

* **Parallelism**
  + Single-threaded program: the task is straightforward but slow
  + Multi-threaded program: a natural and typical way to make programs run faster on modern hardware
  + Parallelization: the task of transforming a standard single-threaded program into a program that does this sort of work on multiple CPUs
* Avoid blocking program progress due to slow I/O
  + Threading enables overlap of I/O with other activities within a single program
  + It is much like multiprogramming did for processes across programs

Logical view of threads

* Threads associated with process form a pool of peers
  + unlike processes which form a tree hierarchy

threads vs. processes

* how threads and processes are similar
  + each has its own logical control flow
  + each can run concurrently with others (possibly on different cores)
  + each is context switched
* how threads and processes are different
  + threads share all code and data (except local stacks)
    - processes typically do not
  + threads are less expensive than processes
    - process control (creating and reaping) is twice as expensive as thread control
  + linux numbers:
    - 20k cycles to create and reap a process
    - 10k cycles to create and reap a thread

Thread creation in c code, and thread trace:

* Slide 10-13

Problems with shared data

* Two threads
* Add a number to the shared variable i
  + Do so 10 million times in a loop
* Expected result:
  + i = 20 million
* actual result:
  + i < 20 million, and it is different for each thread
* this is because of **race condition**

Race condition

* increasing the value of a variable
  + i++
* assume:
  + the variable i is in address x
  + instructions (x86):
    - mov instruction takes 5 bytes of memory
    - add instruction takes 3 bytes of memory
    - see **slide 16** for the actual assembly code
* see **slide 17 for thread trace**, showing why the expected result is not achieved
  + but basically thread 1 is pulling i from memory and incrementing it in temporary storage, then thread 2 is pulling the same i from memory and increasing it before thread 1 has written the changes back to memory. So both threads result in the same i and one of them is not contributing anything.

Terminology

* **critical section**
  + a piece of code that accesses a **shared variable** and must not be concurrently executed by more than one thread
  + needs to support **atomicity** for critical sections (**mutual exclusion**)
* multiple threads executing a critical section -> race condition
* **race condition**
  + multiple threads of execution enter the critical section at roughly the same time
    - attempt to update the shared data structure -> leading to undesirable outcome
  + the results depend on the timing execution of the code
  + result is indeterminate

Atomicity

* ideal approach: make the critical section of code into a single assembly instruction
* in general, we cant do that because the hardware is not setup for that
* instead, we can use **lock**
  + ensure that any such critical section executes as if it were a single atomic instruction
    - execute a series of instructions atomically
  + check **slide 19 for lock sy**ntax

Thread APIs

Thread creation

* **slide 21-23** has thread syntax with commands and stuff

Waiting for a thread to complete

* **slide 24-25**

examples

* bad code: **slide 26**
* simpler argument passing to a thread **slide 27**

Locks

* they provide mutual exclusion to a critical section
  + interface, basic usage: **slide 28**
  + no other thread holds the lock
    - the thread will acquire the lock and enter the critical section
  + if another thread holds the lock
    - the thread will not return from the call until it has acquired the lock
* all locks must be properly initialized
* static way: using PTHREAD\_MUTEX\_INITIALIZER
* dynamic way: using pthread\_mutex\_init()
* error checking: use a simple wrapper
* these two calls are used in lock acquisition
  + trylock: return failure of the lock is already held
  + timelock: return after a timeout

condition variables

* **condition variables** are useful when some kind of **signaling** must take place between threads
* pthread\_cond\_wait
  + put the calling thread to sleep
  + wait for some other thread to signal it
* pthread\_cond\_signal
  + unblock at least one of the threads that are blocked on the condition variable
* a thread calling wait routine
  + the wait call **releases the lock** when putting said caller to sleep
  + before returning after being woken, the wait call **re-acquires the lock**
* a thread calling signal routine
  + slide 33 bruh
* **slide 34 shows what not to do** for wait and signal

compiling and running

* to compile: you must include the header pthread.h
  + explicitly link with the pthreads library by adding the -pthread flag when compiling the program

Remarks

* keep it simple
  + any code to lock or signal between threads should be as simple as possible
  + minimize thread interactions
    - try to keep the number of ways in which threads interact to a minimum
    - tricky thread interactions lead to bugs
* initialize locks and condition variables
  + failure to do so will lead to code that may work or fail in very strange ways
  + always use condition variables to signal between threads
* be careful about arguments passing and return values
  + passing a reference to a variable allocated on the stack wont save changes
* each thread has its own stack
  + to share data between threads, the values must be in the heap or globally accessible

Thread Synchronization

The producer / consumer (**bound buffer**) problem

* producer
  + produce data items
  + wish to place data items in a buffer
* consumer
  + grab data items out of the buffer and consume them in some way
* example: multi-threaded web server
  + a producer puts HTTP requests into a work queue
  + consumer threads take requests out of this queue and process them
* bounded buffer is a shared resource
  + **synchronized access** is required

the put and get routines (version 1)

* put data into the buffer when count is zero (when the buffer is empty)
* get data from the buffer when count is one (when the buffer is full)
* producer puts an integer into the shared buffer i number of times
* consumer gets the data out of that shared buffer

producer / consumer: single CV & if statement

* a single condition variable cond and associated lock mutex
* p1-p3: the producer waits for the buffer to be empty
* count is a global variable
* c1-c3: the consumer waits for the buffer to be full
* this code works for a single producer and a single consumer
* does it work for multiple producers and consumers?
  + No

Thread trace: broken solution (version 1)

* The problem arises for a simple reason:
  + After the producer woke Tc1, but before Tc1 ever ran, the state of the bounded buffer changed by Tc2
* There is no guarantee that when the woken thread runs

Producer / consumer: single CV and while

* Consumers wake up and re-checks the state of the shared variable
* If the buffer is empty, the consumer goes back to sleep
* This code is still broken though
* A consumer should not wake other consumers, only producers and vice-versa

The single buffer producer / consumer solution

* Use two condition variables, and while
  + Producer threads wait on the condition empty, and signals fill
  + Consumer threads wait on fill and signal empty

New synchronization primitive: semaphore

What is semaphore

* An object with an integer value
  + We can manipulate with two routines: sem\_wait() and sem\_post()
* Initialization
  + Declare a semaphore s and initialize it to the value 1
  + The second argument 0, indicates that the semaphore is shared between threads in the same process

Interacting with Semaphore

* Sem\_wait()
  + If the value of the semaphore was one or higher
    - Return right away
  + Otherwise,
    - It will cause the caller to suspend execution and wait for a subsequent post
    - When negative, the value of the semaphore is equal to the number of waiting threads
* Sem\_post()
  + Increments the value of the semaphore
  + If there is a thread waiting to be woken, wakes one of them up

Semaphore as a lock (binary semaphore)

* What should x be?
  + The initial value should be 1, because when we call sem\_wait() it will be 0, then it will wait for the sem value to become positive again when sem\_post() is called

**Slide 22 & 24 & 25: practice the trace as an exercise**

Semaphore as a condition variables

* What should x be
  + The value of semaphore should be set to 0
    - Sem\_wait() will decrease the value to -1, which will tell main to sleep
    - Then when the child call finishes, it will bring the value back up to 0 and main can finish.
    - So basically we want main to sleep while child is running

Synchronization problems

The dining philosophers

* Assume there are five philosophers sitting around a table
  + Between each pair of philosophers is a single fork (five total)
  + The philosophers each have times when they think, and don’t need any forks, and times when they eat
  + In order to eat, a philosopher needs two forks, both the one on the left and the one on the right
  + The contention is for the forks
* Key challenges
  + Make sure there is no **deadlock**
    - No philosopher starves and never gets to eat
  + We need high **concurrency**
    - As many philosophers can eat at the same time as possible
* Philosopher p wishes to refer to the for on their left -> call left(p)
* Philosopher p wishes to refer to the for on their right -> call right(p)
* We need some semaphores, one for each fork: sem\_t forks[5]
  + But we get deadlock
    - If each philosopher happens to grab the fork on their left before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another forever
  + Solution: breaking the dependency
    - Change how forks are acquired
      * Lets assume that philosopher 4 acquires the forks in a different order
      * There is no situation where each philosopher grabs one fork and is stuck waiting for another

Semaphore application thread throttling

* How can a programmer prevent “too many” threads from doing something at once and bogging the system down?
  + Assume there are many threads performing the same memory intensive problem
  + Your system will get fucked up so you need to throttle how many threads can work at the same time
  + So create a semaphore initialized to 5, which will ensure at most 5 threads can work in parallel at a time

Common concurrency problems

Atomicity-violation bugs

* Two different threads access the field proc\_info in the struct thd
* What is the problem with the code (slide 33)
* The desired **serializability** among multiple memory accesses is violated
  + If the first thread performs the check but then is interrupted before the call to fputs, the second thread could run in-between, thus setting the pointer to NULL
  + When the first thread resumes, it will crash
    - As a NULL pointer will be dereferenced by fputs
* Solution?
  + Add locks around the shared-variable references

Deadlock bugs

* The presence of a cycle
  + Thread1 is holding a lock L1 and waiting for another one, L2
  + Thread2 that holds lock L2 is waiting for L1 to be released
* Prevention: circular wait
  + Provide a total ordering on lock acquisition
    - This approach requires careful design of global locking strategies
  + Example:
    - There are two locks in the system (L1 and L2)
    - We can prevent deadlock by always acquiring L1 before L2

Avoiding deadlock bugs via scheduling

* Deadlock avoidance
  + Get the information about the locks various threads might grab during their execution
  + Schedule the threads in a way that guarantees no deadlock can occur
* In some scenarios, deadlock avoidance is preferable
* Problem: global knowledge is required
* Example #1:
  + We have 2 processors and 4 threads
    - Lock acquisition demands of the threads: table on slide 38
  + A scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could arise
* Example #2:
  + Limit the concurrency
  + Drawback is we get a very inefficient scheduler