CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska

Winter 2010

Lecture 26 – Threads and Concurrency Control

Final Exam

- Monday, March 15th @ 8:30-10:20, MGH 241
- Content: Lecture 7 and following
  - There will be questions about tools: svn, debugger, makefiles
- CLOSED book and closed notes
- EXCEPT for **two** 8.5''x11'' pages
  - 7pt font or higher or written manually
  - Both sides

Motivation for Concurrency

- Imagine a software system such as a web server or a database management system (DBMS)
- A Web server works as follows
  - Client requests a page (URL)
  - Web server locates and reads page from disk
  - Web server sends content of page back to client
- A DBMS works as follows
  - Client submits a query
  - DBMS reads from disk the data that satisfies the query
  - DBMS sends the data back to the client

How to Achieve High Performance?

- Many clients submit requests at the same time
- Approach 1: put requests in a queue and serve one request at the time
  - But... reading data from disk is very slow
  - And while reading from disk, the CPU is idle
  - This is very slow, very inefficient. Can we do better?
- Approach 2: serve multiple requests simultaneously
  - While reading data from disk for one client
  - Start parsing request for second client
  - Send results from previous request to third client
  - Use multiple cores if available
  - All resources are fully utilized. This is **much more efficient**

Enabling Concurrency

- How to serve many requests at the same time?

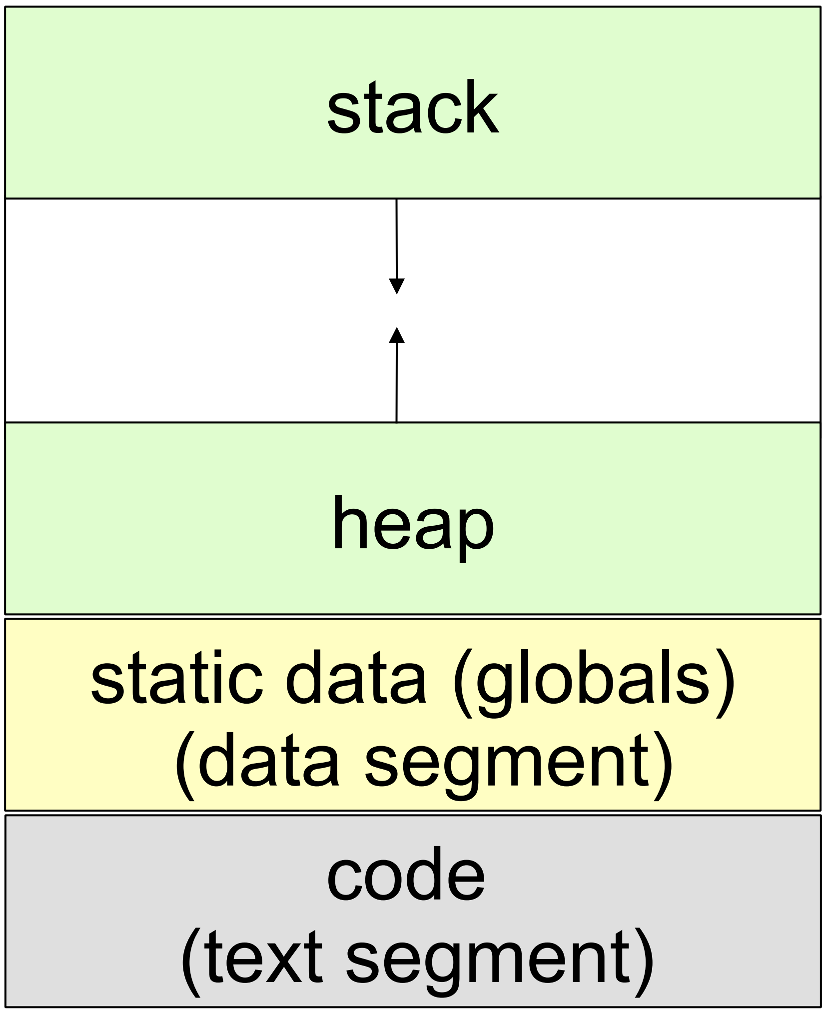
- Design 1: Launch one process per client request
  - One web server process or one database system process
  - Each process has its own address space with a stack, a heap, code, and global variables
  - OS takes turn running processes on processor(s)
  - Processes can communicate with each other (in our example they communicate through the filesystem)
  - This approach is quite “heavyweight”

Enabling Concurrency

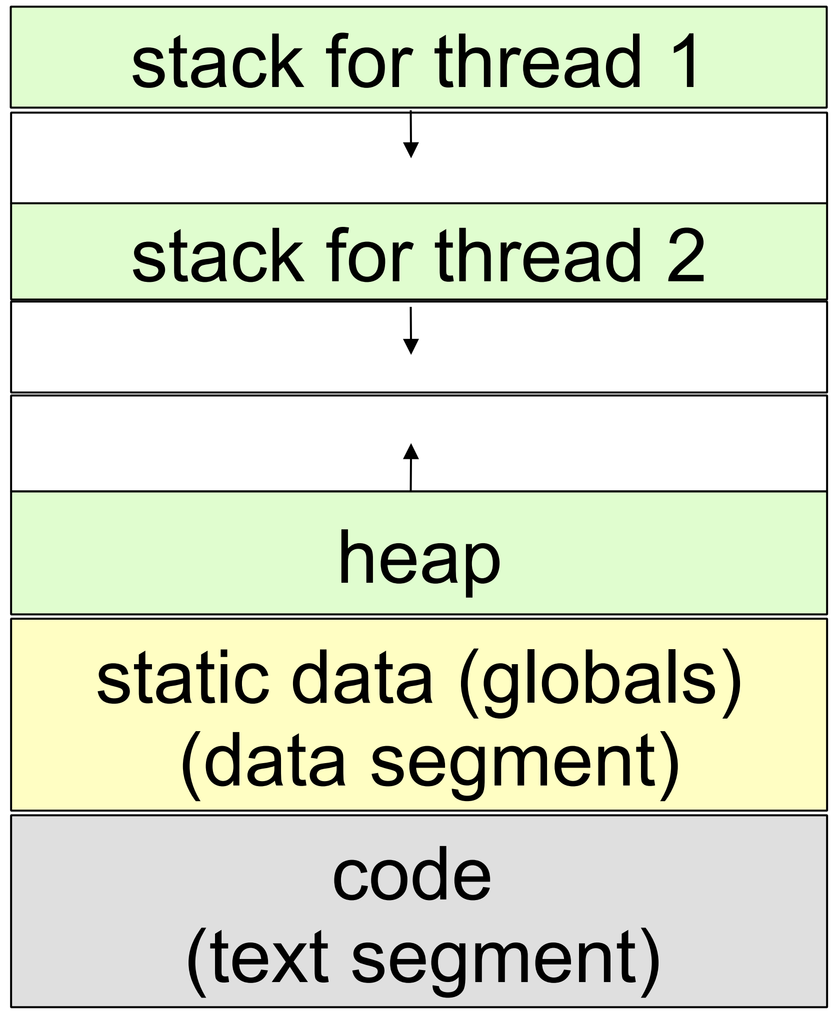
- How to serve many requests at the same time?

- Design 2: Launch one **thread** per client request
  - Launch a single process with multiple threads
  - Each thread has its own stack
  - A scheduler runs threads one-or-more at a time
  - This time, **threads share an address space: same heap and same global variables**
  - This approach is more “lightweight”

Address Space of a Process



One process with one thread



One process with two threads

Plan for Today

- Today, we will talk about writing programs with threads
  - What can go wrong?
  - How to avoid problems?
- Concurrency is a difficult concept
  - Focus on the key challenges and solutions
  - You do not need to learn the programming syntax
- In later classes
  - You will learn more about the tradeoffs between threads and processes (and the history)
  - You will learn about design issues regarding how to leverage concurrency (these are hard systems issues)

Pthreads

- In Java, syntax for threads is quite easy
  - You should learn it on your own
- In C, threads are messier and often not portable
- For UNIX systems, there exists a standardized C language threads programming interface
- Implementations that adhere to this standard are referred to as POSIX threads or Pthreads
- We will use Pthreads in our examples but
  - Concepts and principles are language independent
- Our first example: `bank.cc`

Creating a New Thread

- Initially, program comprises a single, default thread
- Other threads must be created explicitly
- Function: `pthread_create`
  - Creates a new thread and makes it executable
- Example from `bank.cc`

```
pthread_t spender_thread;
pthread_create(&spender_thread, // identifier
               NULL,            // attributes
               spender,         // start function
               (void*)p_nb_transfers // arguments
               );
```

Creating a New Thread

- Arguments to `pthread_create`
  - thread: opaque, unique id for the new thread returned by the subroutine
  - attr: serves to specify thread attributes or NULL for the default values (we will use NULL)
    - Example attribute is the thread max stack size
  - start_routine: the C function that the thread will execute once it is created
  - arg: a single argument that may be passed to start_routine.

Terminating a Thread

- If process terminates, all threads terminate
- Can also terminate a single thread
  - By returning from start_routine
  - By calling `pthread_exit` explicitly inside the thread
  - By calling `pthread_cancel` from outside the thread

- It is possible to wait for a thread to terminate
  - By calling `pthread_join`

- Example `bank.cc`

Race Conditions

- Threads communicate through shared memory
- This makes communication nice and easy BUT
- This leads to a problem known as a race condition
  - Two threads can access the same memory at the same time, and at least one access is a write

Thread 1	Thread 2	Value of X
`int a = x`	`int a = x`	10
`a = 2*a`		
	`a = 2*a`	
`x = a`		20
	`x = a`	20

- Example: in `bank.cc`, simultaneous transfers by the two threads can cause money to disappear

Locking

- To avoid race conditions, typical solution is to use locks
- Lock is either available or held by a thread
- Before modifying a shared data item
  - A thread tries to acquire a lock
  - If lock is available, thread acquires and holds lock
  - Otherwise, thread blocks until lock is available
- After the modification, the thread releases the lock
  - Lock becomes available again

Locking Example

Thread 1	Thread 2	Value of X
Lock X	Lock X -> Block	10
int a = x		
a = 2*a		
x = a		20
Unlock X		
	Lock X	
	int a = x	
	a = 2*a	
	x = a	40
	Unlock X	

Pthread Mutexes

- With PThreads, special mutex variables are used for locking. Mutex is an abbreviation for "mutual exclusion"
- Example from bank-fixed.cc:

```
pthread_mutex_t mutex_bank;
pthread_mutex_init(&mutex_bank, NULL);      — Only need to do once
 ...

pthread_mutex_lock (&mutex_bank);           — For each access to data
// perform operations on bank accounts
// ...
pthread_mutex_unlock (&mutex_bank);
//...
pthread_mutex_destroy(&mutex_bank);         — When mutex is no longer needed
```

Pthread Mutexes

- Note: with Pthreads, when multiple threads are waiting for the same lock, there is no guarantee which thread will acquire the lock next
- Notice the performance decrease once we added locks

More About Race Conditions

- Any one of the following is **sufficient** to avoid races
  - Keep data thread-local (keep data reachable only by one thread or at least accessed only by one thread)
  - Keep data read-only (make your objects immutable)
  - Use locks consistently (always acquire a lock before accessing an object)

- Easy to forget about any of these and get bugs that are very hard to reproduce

Deadlocks

- Locks reduce concurrency
  - Because threads must wait for each other
- To maximize concurrency, want to use 1 lock/data item
  - Threads that access different data items can then still run in parallel by acquiring different locks
- But existence of multiple locks can cause deadlocks:

Thread 1	**Thread 2**
`Lock X`	
	`Lock Y`
`Lock Y -> Block`	
	`Lock X -> Block`
`Deadlock`	`Deadlock`

Avoiding Deadlocks

- Ensure that all threads acquire locks in the same order
- Deadlock examples:
  - bank-deadlock.cc and bank-nodeadlock.cc
  - Famous deadlock example: dinning philosophers
- Can also use deadlock detection (e.g. database systems)
  - Time-outs
  - Wait-for graphs

Summary

- Multithreaded programming can improve performance
  - Helps keep resources busy
  - Can take advantage of existence of multiple processors
- Multithreaded programming is difficult
  - There are multiple stacks in one address space
  - There are potential races and deadlocks
  - Need to use locks carefully to avoid these problems
- A lot more to this topic than we have covered today