

Thread Synchronization

Daniel Zappala

CS 360 Internet Programming
Brigham Young University

Caution!

A Simple Example

```
1 void echo()  
2 {  
3     chin = getchar();  
4     chout = chin;  
5     putchar(chout);  
6 }
```

- shared method among multiple threads
- what can happen?

A Simple Example

Thread 1

```
1 void echo()  
2 {  
3     chin = getchar();  
4     chout = chin;  
5     putchar(chout);  
6 }
```

Thread 2

```
1 void echo()  
2 {  
3     chin = getchar();  
4     chout = chin;  
5     putchar(chout);  
6 }
```

- Thread1 executes line 3, then interrupted
 - Thread1 chin = 'x'
- Thread2 executes completely through the procedure
 - Thread2 chin = 'y', chout = 'y'
- Thread1 starts again
 - Thread1 chin = 'y'!

Example Code

- see example code problem

▶ [GitHub](#)

Concurrency Problems

- can't predict the speed with which threads will execute and therefore when a resource will be accessed
- if synchronization is not used, errors will be rare but they will occur
- errors are hard to duplicate and debug since they are nondeterministic

Mutual Exclusion

Mutual Exclusion

- need to protect shared resources (e.g. global variable, shared data structures) among multiple processes or threads
- may involve processes or threads interleaved in time on a single processor or running in parallel on a multiprocessor machine
- result of process or thread must be independent of the speed of execution of other concurrent processes

Mutual Exclusion

- **critical section**: shared portion of code that must be executed by one thread at a time
 - thread must mark the critical section because OS doesn't know where it is
- **starvation**: one or more threads are prevented from ever executing critical section
- **deadlock**: situation in which no thread can make progress because they are all waiting for a critical section
- must ensure data coherence, e.g. atomic access to a database

Mutual Exclusion Requirements

- only one thread has access to critical section at a time
- halting in non-critical section must not interfere with other threads
- no indefinite wait for critical section, i.e. no starvation or deadlock
- if no thread in critical section, then no wait to enter
- no assumptions about process speeds or number of processors
- thread may only spend finite time within critical section

Solutions

- software
 - assume no support from OS, hardware, or language
 - historic algorithms: Dekker, Peterson, Lamport
 - difficult to get right, to generalize
- hardware
 - disable interrupts: single processor machines
 - no other process can run until they are re-enabled
 - limits flexibility of OS to schedule threads, doesn't work for multiprocessors
 - atomic machine instructions: compare-and-swap
- operating system support

Operating System Support

- mutexes and condition variables
 - mutex: lock that allows only one thread into a critical section
 - condition variable: signal conditions between threads
- monitors and thread-safe classes
 - monitor: programming language construct that makes it easier to declare and use a critical section
 - thread-safe class: construct a class with private variables, use a lock on each thread so that only one caller is active at a time
- semaphores
 - when one thread is in the critical section, others may **wait** by sleeping
 - when thread is done with critical section, it wakes one other thread with a **signal**
- message passing
 - synchronization by explicitly exchanging messages