CMSC 412
Spring 2007

Concurrency and Synchronization

Announcements

- Reading
  - Chapter 6
- Project #1
  - Due Wed, 2/14

Systems = Objects + Activities

- Safety is a property of objects, and groups of objects, that participate across multiple activities.
  - Can be a concern at many different levels: objects, composites, components, subsystems, hosts, …
- Liveness is a property of activities, and groups of activities, that span across multiple objects.
  - Levels: Messages, call chains, threads, sessions, scenarios, scripts workflows, use cases, transactions, data flows, mobile computations, …

Violating Safety

- Data can be shared by threads
  - Scheduler can interleave or overlap threads arbitrarily
  - Can lead to *interference*
    - Storage corruption (e.g. a *data race/race condition)*
    - Violation of representation invariant
    - Violation of a protocol (e.g. *A* occurs before *B*)

How does this apply to OSs?

- Any resource that is shared could be accessed inappropriately
  - Shared memory
    - Kernel threads
    - Processes (shared memory set up by kernel)
  - Shared resources
    - Printer, Video screen, Network card, …
- OS must protect shared resources
  - And provide processes a means to protect their own abstractions

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

Start: both threads ready to run. Each will increment the global count.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

y = 0

T1 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

y = 0

y = 0

T1 is pre-empted. T2 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state **cnt = 1**

y = 0

y = 0

T2 executes, storing the incremented cnt value.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state **cnt = 1**

y = 0

y = 0

T2 completes. T1 executes again, storing the old counter value (1) rather than the new one (2)!

But When I Run it Again?

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

Start: both threads ready to run. Each will increment the global count.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

y = 0

T1 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state **cnt = 1**

y = 0

T1 executes again, storing the counter value

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 1

y = 0

y = 1

T1 finishes. T2 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state **cnt = 2**

y = 0

y = 1

T2 executes, storing the incremented cnt value.

What happened?

- In the first example, **t1** was preempted after it read the counter but before it stored the new value.
  - Depends on the idea of an *atomic action*
  - Violated an object invariant
- A particular way in which the execution of two threads is interleaved is called a *schedule*. We want to prevent this undesirable schedule.
- Undesirable schedules can be hard to reproduce, and so hard to debug.

Question

- If you run a program with a race condition, will you always get an unexpected result?
  - No! It depends on the scheduler
  - ...and on the other threads/processes/etc that are running on the same machine
- Race conditions are hard to find

Synchronization

```
static int cnt = 0;
struct Mutex lock;
Mutex_Init(&lock);
void run() {
  Mutex_Lock (&lock);
  int y = cnt;
  cnt = y + 1;
  Mutex_Unlock (&lock);
}
```

***Lock**, for protecting The shared state*

***Acquires** the lock; Only succeeds if not held by another thread*

***Releases** the lock*

Java-style **synchronized** block

```
static int cnt = 0;
struct Mutex lock;
Mutex_Init(&lock);
void run() {
  synchronized (lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

***Lock**, for protecting The shared state*

***Acquires** the lock; Only succeeds if not held by another thread*

***Releases** the lock*

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 0

T1 acquires the lock

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 0

y = 0

T1 reads cnt into y

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 0

y = 0

T1 is pre-empted. T2 attempts to acquire the lock but fails because it's held by T1, so it blocks

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state **cnt = 1**

y = 0

T1 runs, assigning to cnt

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

y = 0

Shared state cnt = 1

T1 releases the lock and terminates

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

y = 0

Shared state cnt = 1

T2 now can acquire the lock.

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

y = 0

y = 1

Shared state cnt = 1

T2 reads cnt into y.

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

y = 0

y = 1

Shared state **cnt = 2**

T2 assigns cnt, then releases the lock

Mutexes (locks)

- Only one thread can "acquire" a mutex
  - Other threads block until they can acquire it
  - Used for implementing critical sections
- A critical section is a piece of code that should not be interleaved with code from another thread
  - Executed atomically
- We'll look at other ways to implement critical sections later …

Mutex Policies

- What if a thread already holds the mutex it's trying to acquire?
  - Re-entrant mutexes: The thread can reacquire the same lock many times. Lock is released when object unlocked the corresponding number of times
    - This is the case for Java
  - Non-reentrant: Deadlock! (defined soon.)
    - This is the case in GeekOS
- What happens if a thread is killed while holding a mutex? Or if it just forgets to release it
  - Could lead to deadlock

5

GeekOS: interrupts

- The kernel provides two useful routines for avoiding races
  - Disable_Interrupts()
  - Enable_Interrupts()
- These can be used to define a critical section:

  Disable_Interrupts();
  Actions here to perform atomically
  Enable_Interrupts();

Reentrancy

- But beware reentrancy! Will interrupts be disabled when executing block B?

```
Foo() {
  DI(); codeblock A; EI();
}
Bar() {
  DI(); Foo(); codeblock B; EI();
}
```

Saving Interrupt State

- Can rewrite code to save/restore state:

```
Foo() {
  bool flag = Begin_Int_Atomic();
  codeblock A;
  End_Int_Atomic(flag);
}
Bar() {
  DI(); Foo(); codeblock B; EI();
}
```

Java Synchronized statement

- **synchronized (obj) {** *statements* **}**
- Obtains the lock on **obj** before executing statements in block
  - **obj** can be any Object
- Releases the lock when the statement block completes
  - Either normally, or do to a return, break, or exception being thrown in the block
- Can't forget to release the lock!

Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called *deadlock*

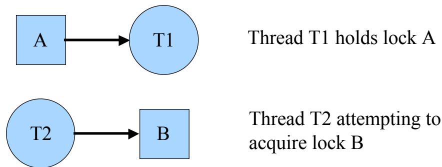
```
Object A = new Object();
Object B = new Object();
T1.run() {
  synchronized (A) {
    synchronized (B) {
      …
    }
  }
}
```

```
T2.run() {
  synchronized (B) {
    synchronized (A) {
      …
    }
  }
}
```

Deadlock

- Quite possible to create code that deadlocks
  - Thread 1 holds lock on **A**
  - Thread 2 holds lock on **B**
  - Thread 1 is trying to acquire a lock on **B**
  - Thread 2 is trying to acquire a lock on **A**
  - Deadlock!
- Not easy to detect when deadlock has occurred
  - other than by the fact that nothing is happening

Deadlock: Wait graphs

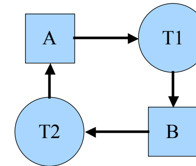


Thread T1 holds lock A

Thread T2 attempting to acquire lock B

Deadlock occurs when there is a cycle in the graph

Wait graph example



T1 holds lock on **A**
T2 holds lock on **B**
T1 is trying to acquire a lock on **B**
T2 is trying to acquire a lock on **A**

Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or can be scheduled on a single processor
    - A running thread can be pre-empted at any time
- Threads can share data
  - Need to prevent interference
    - Synchronization is one way, but not the only way
  - Overuse use of synchronization can create deadlock
    - Violation of liveness

Implementing Synchronization

- Next we're going to revisit the issues of synchronization for the producer/consumer problem.
- Another way of looking at what we've just gone over.
- Will prepare us to see possible solutions next time.

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

- Producer process

```
item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Bounded-Buffer

- Consumer process

```
item nextConsumed;

while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Bounded Buffer

- The statements

  counter++;
  counter--;

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

- The statement **"count++"** may be implemented in machine language as:

  register1 = counter
  register1 = register1 + 1
  counter = register1

- The statement **"count--"** may be implemented as:

  register2 = counter
  register2 = register2 - 1
  counter = register2

Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded Buffer

- Consider the three address code for the counter

Counter Increment	Counter Decrement
reg_1 = counter	reg_2 = counter
$reg_1 = reg_1 + 1$	$reg_2 = reg_2 - 1$
counter = reg_1	counter = reg_2

- Now consider an ordering of these instructions

T_0	producer	reg_1 = counter	{ $reg_1 = 5$ }
T_1	producer	$reg_1 = reg_1 + 1$	{ $reg_1 = 6$ }
T_2	consumer	reg_2 = counter	{ $reg_2 = 5$ }
T_3	consumer	$reg_2 = reg_2 - 1$	{ $reg_2 = 4$ }
T_4	producer	counter = reg_1	{ counter = 6 }
T_5	consumer	counter = reg_2	{ counter = 4 } ← This should be 5!

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

8

The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Implementing Critical-Section

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the *n* processes.

Simpler Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
    entry section
        critical section
    exit section
        reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.