(I love) Concurrency

2/14/24

Recall: Threads and parallelism/concurrency

- Parallelism: Using more resources to complete job faster

- Concurrency: Managing access to shared resources

$$\text{Speedup} = \frac{\text{Serial (non-parallel) running time}}{\text{Parallel running time}}$$

Recall: Why not linear speedup? (1)

If B = fraction of program that must run serially

T_1 = total time on 1 processing element

What is best possible time on p elements?

A. $T_1/p + B$
B. $T_1 B/p$
C. $T_1(1-B)/p + B$
D. $T_1(1-B)/p + T_1 B$ (called Amdahl's Law)
E. None of the above

Why not linear speedup? (2)

- Poor load balance:

Why not linear speedup? (3)

- Overhead
  - Extra instructions needed for running in parallel
  - Examples:
    - creating and destroying threads
    - calls needed to coordinate threads or communicate between them
    - changes to algorithm needed to expose parallelism or improve load balance

Multicore programming

- So far, focused on speedup and why linear speedup might not be achieved

- Today, looking at concurrency problems
  - What kind of coordination might be needed and how can it be done?

Setting of concurrency problems

- Each thread/process runs serially

- Relative to each other, they can run at arbitrary speed, allowing very general interactions

Race conditions

- Logic errors caused by interactions through shared variables
- Example: processing ATM withdrawal

Operation	Balance
Read current value (100)	$100
Perform calculation (80)	$100
Store new value	$80

Race conditions

- Logic errors caused by interactions through shared variables
- Example: processing ATM withdrawal

Operation 1	Operation 2	Balance
Read current value (100)		$100
Perform calculation (80)	Read current value (100)	$100
Store new value	Perform calculation (80)	$80
	Store new value	$80

Solving race conditions

- One solution: locks
  - acquire: block if lock is held, mark lock as held
  - release: mark lock as not held, unblock one waiting thread (if any)

Solving race conditions

- One solution: locks
  - acquire: block if lock is held, mark lock as held
  - release: mark lock as not held, unblock one waiting thread (if any)
- Usage:

  acquire lock

  do critical section

  release lock

Construction blocks one lane of a two-lane highway so that all traffic must use the other lane.

What parallelism/concurrency concept does this illustrate?

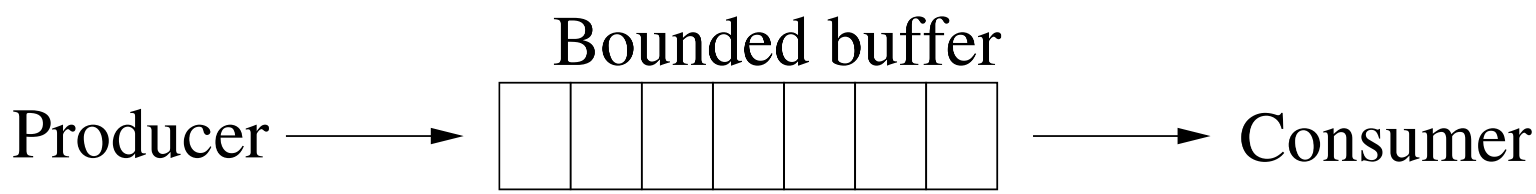
A. Threads
B. Race condition
C. Critical section
D. Parallel overhead
E. I hate construction

Construction blocks one lane of a two-lane highway so that all traffic must use the other lane.

What parallelism/concurrency concept does this illustrate?

A. Threads
B. Race condition
C. Critical section
D. Parallel overhead
E. I hate construction

Producer-consumer problem



- Producer writes into buffer while not full
- Consumer reads from buffer while not empty
- Each blocks if it can’t work
- Example: I/O buffers

What is wrong with the given code?

What is wrong with the given code?

```
void producer() {
    …
    if(count == N) sleep();
```

If the other thread removes an item between the check and going to sleep, the producer sleeps forever

Similar issue in consumer as well

Deadlock

- Situation in which group of threads/processes all block forever
- Typically, each holds a resource that others are blocking on

Yes. My traffic example did happen

Posted by "netchicken" at http://xmb.stuffucanuse.com/xmb/viewthread.php?tid=4848, where it is attributed to an article on Reddit.

More than once

http://minutillo.com/steve/weblog/2003/1/21/deadlock/, where it is attributed to "Chuck @ China" (http://chake.chinatefl.com/)

Does it work to move the troublesome line into the critical section?

```
acquire_lock();            //moved from below next line
if(count == N) sleep();
insert_item(item);
...
```

A. Yes. The code works correctly with just changing the producer code
B. Yes. The code works correctly if this change is made to both the producer and consumer
C. No. This doesn't prevent an interruption between reading count and calling sleep
D. No. This creates a different deadlock
E. No. Something else breaks

Does it work to move the troublesome line into the critical section?

```
acquire_lock();              //moved from below next line
if(count == N) sleep();
insert_item(item);
...
```

A. Yes. The code works correctly with just changing the producer code
B. Yes. The code works correctly if this change is made to both the producer and consumer
C. No. This doesn't prevent an interruption between reading count and calling sleep
D. No. This creates a different deadlock
E. No. Something else breaks

What if we make the producer give up the lock right before going to sleep?

```
acquire_lock();
if(count == N) { release_lock();  sleep();  acquire_lock(); }
insert_item(item);
…
```

A. Yes. The code works correctly with just changing the producer code
B. Yes. The code works correctly if this change is made to both the producer and consumer
C. No. This doesn't prevent an interruption between reading count and calling sleep
D. No. This creates a different deadlock
E. No. Something else breaks

What if we make the producer give up the lock right before going to sleep?

```
acquire_lock();
if(count == N) { release_lock();  sleep();  acquire_lock(); }
insert_item(item);
...
```

A. Yes. The code works correctly with just changing the producer code
B. Yes. The code works correctly if this change is made to both the producer and consumer
C. No. This doesn't prevent an interruption between reading count and calling sleep
D. No. This creates a different deadlock
E. No. Something else breaks

Semaphore

(Dijkstra 1965)

- Integer with two atomic operations:
  - down: if 0, sleep until positive

    when positive, decrease by 1
  - up: increase by one

    (if processes were sleeping, wake one up)

- Can be used as a lock, but more powerful. Typically for more complicated inter-process communication (IPC)

Semaphore-based solution to producer-consumer

2 semaphores:

empty: initial value n full: initial value 0

producer:

```
down(empty);
insert_item(); (w/ lock to protect data structure)
up(full);
```

consumer:

```
down(full);
remove_item();
up(empty);
```

Using a semaphore as a lock

binary semaphore: called a mutex
    can implement a lock if initial value is 1

producer:
```
down(empty);
down(mutex);
insert_item();
up(mutex);
up(full);
```

consumer:
```
down(full);
down(mutex);
remove_item();
up(mutex);
up(empty);
```

Using a semaphore as a lock

binary semaphore: called a mutex
  can implement a lock if initial value is 1

producer:
```
down(empty);
down(mutex);
insert_item();
up(mutex);
up(full);
```

consumer:
```
down(full);
down(mutex);
remove_item();
up(mutex);
up(empty);
```

Does the order of the calls to down matter?
(Just here, not in both methods.)

A. Yes. Swapping them creates a race condition
B. Yes. Swapping them allows deadlock
C. Yes. Swapping them creates a different problem
D. No. Swapping them works fine
E. You can't tell without more information

Using a semaphore as a lock

binary semaphore: called a mutex

can implement a lock if initial value is 1

producer:

```
down(empty);
down(mutex);
insert_item();
up(mutex);
up(full);
```

consumer:

```
down(full);
down(mutex);
remove_item();
up(mutex);
up(empty);
```

Does the order of the calls to down matter? (Just here, not in both methods.)

A. Yes. Swapping them creates a race condition
B. Yes. Swapping them allows deadlock
C. Yes. Swapping them creates a different problem
D. No. Swapping them works fine
E. You can't tell without more information