Concurrency

10/9/15
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?

```c
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?

```c
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?

```c
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?

```c
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?

```c
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);
  critical region (w/ lock to protect data structure)
  up(full);

consumer:
  down(full);
  critical region
  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);
critical region
up(mutex);
up(full);

consumer:
down(full);
down(mutex);
critical region
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);
critical region
up(mutex);
up(full);

consumer:
down(full);
down(mutex);
critical region
up(mutex);
up(empty);

Does the order of the calls to down matter?
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);
    critical region
    up(mutex);
    up(full);

consumer:
    down(full);
    down(mutex);
    critical region
    up(mutex);
    up(empty);

Does the order of the calls to down matter?
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
High-level primitive: Monitors

• Group of functions such that only one can run at a time

• Example: Java’s synchronized methods
  – acquires lock on object before entering method
Readers and writers

• Processes share a common database
• Some want read access (readers) while others want ability to write (writers)
• Readers should be able to share the database, but all other processes must block if a writer gets access
Solving readers and writers

semaphore mutex = 1;  //control access to database

void read() {
    down(mutex);
    //perform read
    up(mutex);
}

void write() {
    down(mutex);
    //perform write
    up(mutex);
}

Does this successfully implement readers and writers?
A. Yes.
B. Yes.
C. No. It allows deadlock
D. No. It creates some other problem
E. What are up and down again?
Solving readers and writers

semaphore mutex = 1;  //control access to database

void read() {
    down(mutex);
    //perform read
    up(mutex);
}

void write() {
    down(mutex);
    //perform write
    up(mutex);
}

Does this successfully implement readers and writers?
A. Yes.
B. Yes.
C. No. It allows deadlock
D. No. It creates some other problem (doesn’t allow more than 1 reader)
E. What are up and down again?
Solving readers and writers

semaphore mutex = 1; //control access to database
int numR = 0;       //number of active readers

void read() {
    numR++;
    if(numR == 1) down(mutex);
    //perform read
    numR--;
    if(numR == 0) up(mutex);
}

void write() {
    down(mutex);
    //perform write
    up(mutex);
}

Does this successfully implement readers and writers?
A. Yes.
B. I sure hope so
C. No. It allows deadlock
D. No. It creates some other problem
E. It’s Friday and I can’t think this hard anymore
Solving readers and writers

semaphore mutex = 1;    //control access to database
int numR = 0;           //number of active readers

void read() {
    numR++;
    if(numR == 1) down(mutex);
    //perform read
    numR--;
    if(numR == 0) up(mutex);
}

void write() {
    down(mutex);
    //perform write
    up(mutex);
}

Does this successfully implement readers and writers?

A.  Yes.
B.  I sure hope so
C.  No. It allows deadlock
D.  No. It creates some other problem
E.  It’s Friday and I can’t think this hard anymore