



**CS 241 Spring 2007
System Programming**

CS241 Systems Programming

Synchronization Problems

Lawrence Angrave

CS241 Administrative

This week

HW due Friday

Next week

Midterm Monday in class

This lecture

Goals:

Introduce classic synchronization problems

Topics

Producer-Consumer

Dining Philosophers

Producer Consumer

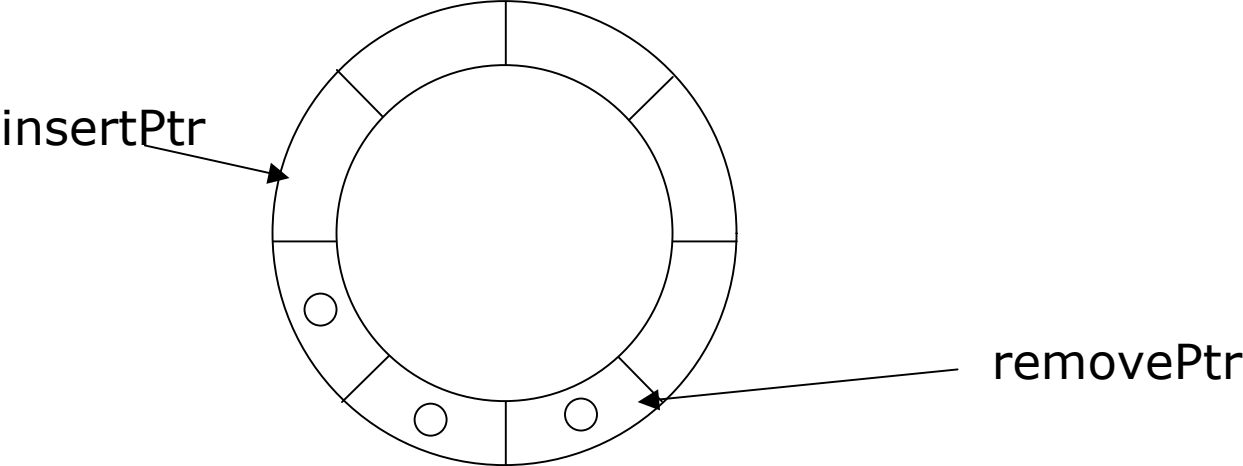
Problem occurs in system & application programming

e.g. Web Server dispatches incoming web requests to a waiting process(es)

e.g. GUI events from keyboard, & mouse are queued by O/S and consumed by applications.

Pipelines (Hardware & software examples)

Producer-Consumer



Producer-Consumer Problem

Producers insert items

Consumers remove items

Shared bounded buffer *

* Efficient implementation is a circular buffer with an insert and a removal pointer.

Challenge?

What the abstract requirements that our solution must satisfy?

Challenge

Prevent buffer overflow

Prevent buffer underflow

Proper synchronization

- Mutual Exclusion

- Progress

- Bounded wait

- i.e. Prevent deadlock

Buffer underflow

Imagine:

Producer inserts an item

Consumer removes an item

Consumer removes another item

Conclusions:

Make sure consumer only retrieves valid items
from the buffer

Consumer should block (or return an error code)
if there are no items available

Buffer overflow

Producer inserts too many items and the buffer overflows

Conclusion:

Block Producer if the buffer is full

Mutual Exclusion

Producer inserts items. Updates insertion pointer.

Consumer executes destructive reads on the buffer. Update removal pointer

Both update information about how full/empty the buffer is.

Solution should allow multiple readers/writers

What could possibly go wrong?

Check the tricky "Edge cases"

e.g. Producer is waiting(blocked) but the buffer is already full. Now Consumer reads an item.

What could go wrong at this point?

What could possibly go wrong?

Check the tricky "Edge cases"

e.g. Producer is waiting(blocked) but the buffer is already full. Now Consumer calls remove()

What could go wrong at this point?

- Consumer cannot continue because Producer is in critical section => deadlock
- Producer is never unblocked => deadlock
- Consumer unblocks Producer too early. Failed mutual exclusion => corrupt data.
- Suppose *another* consumer/producer arrives... will our implementation still work?

Other edge cases

- Two producers call insert() at the same time
- Consumer(s) is/are waiting on an empty buffer. Producer tries to insert one item.

Implementation?

What do we need to prevent buffer underflow?

What do we need to prevent buffer overflow?

What do we need to protect updates to buffer?

2 Counting Semaphores and a mutex

Counting semaphore to count # items in buffer

Counting semaphore to count # free slots

Mutex to protect accesses to shared buffer & pointers.

Assembling the solution

```
sem_wait(slots), sem_post(slots)  
sem_wait(items), sem_post(items)  
mutex_lock(m) mutex_unlock(m)
```

```
insertptr=(insertptr+1) % N  
removalptr=(removalptr+1) % N
```

Initialize semaphore *slots* to size of buffer
Initialize semaphore *items* to zero.

Pseudocode getItem()

Error checking/EINTR handling not shown

sem_wait(items)

mutex_lock(mutex)

 result=buffer[removePtr];

 removePtr=(removePtr +1) % N

mutex_unlock(mutex)

sem_post(slots)

so what about insertItem()?

Pseudocode putItem(*data*)

Error checking/EINTR handling not shown

sem_wait(slots)

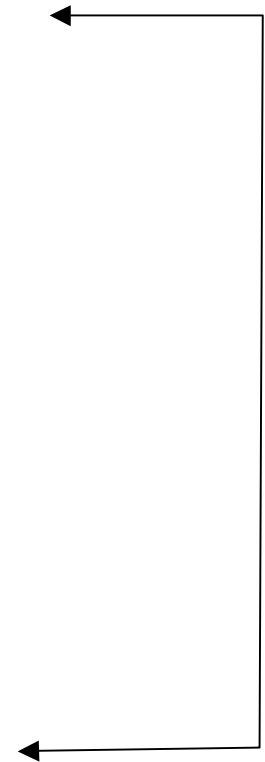
mutex_lock(mutex)

 buffer[insertPtr]= data;

 insertPtr=(insertPtr + 1) % N

mutex_unlock(mutex)

sem_post(items)



Analysis#1 What's the precise problem?

```
putItem(data) {
```

```
mutex_lock(mutex)
```

```
sem_wait(slots)
```

```
buffer[ insertPtr]= ...
```

```
insertPtr=...
```

```
sem_post(items)
```

```
mutex_unlock(mutex)
```

```
}
```

Underflow? Overflow?

```
getItem() {
```

```
mutex_lock(mutex)
```

```
sem_wait(items)
```

```
result=buffer[ removePtr ];
```

```
removePtr=...
```

```
sem_post(slots)
```

```
mutex_unlock(mutex)
```

```
}
```

Deadlock? Failed Mut Excl?

Deadlock e.g Consumer waits for producer to insert a new item but Producer is waiting for Consumer to release mutex

```
putItem(data) {
```

```
mutex_lock(mutex) #2
```

```
sem_wait(slots)
  buffer[ insertPtr]= ...
  insertPtr=...
sem_post(items)
mutex_unlock(mutex)
}
```

```
getItem() {
```

```
mutex_lock(mutex)
```

```
sem_wait(items) BLOCKS #1
  result=buffer[ removePtr ];
  removePtr=...
sem_post(slots)
mutex_unlock(mutex)
}
```

Analysis#2

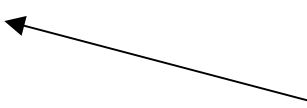
```
putItem(data) {  
  
    sem_wait(slots)  
  
    mutex_lock(mutex)  
    buffer[ insertPtr]= ...  
    insertPtr=...  
    sem_post(items)  
  
    mutex_unlock(mutex)  
}
```

```
getItem() {  
  
    sem_wait(items)  
    sem_post(slots)  
  
    mutex_lock(mutex)  
    result=buffer[ removePtr ];  
    removePtr=...  
    mutex_unlock(mutex)  
}
```

Buffer overflow when reader removes item from a full buffer: Producer inserts item too early

```
putItem(data) {
    sem_wait(slots)
    mutex_lock(mutex)
    buffer[ insertPtr]= ...
    insertPtr=...
    sem_post(items)
    mutex_unlock(mutex)
}

getItem() {
    sem_wait(items)
    sem_post(slots)
    mutex_lock(mutex)
    result=buffer[ removePtr ];
    removePtr=...
    mutex_unlock(mutex)
}
```



Other considerations

Early cancelation using signals?

Limited Production?

(Possibly no items)

... Consumers wait forever

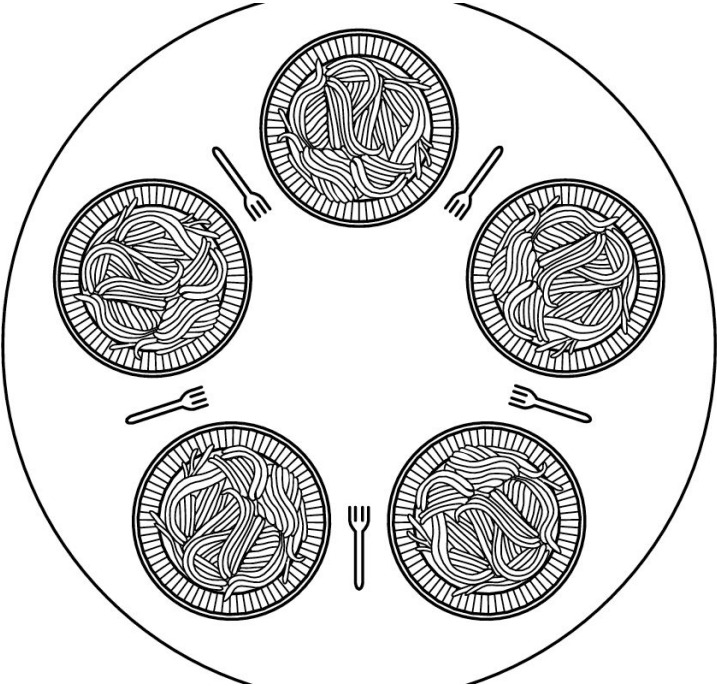
... Consumers quit too early?

One implementation: Insert special end-value into queue which consumers read (but may not consume)

Other reasonable implementations (condition variables; not covered in CS241)

Priority of Consumers & Producers

Dining Philosophers



History

Dijkstra 1971

Originally set as a exam question

Illustrates Starvation, Deadlock

Test shared resource allocation algorithms

See Stallings Ch.6.6 p.276

Dining Philosopher Challenge

{ Think | Eat }

N Philosophers circular table with N chopsticks

To eat the Philosopher must first pickup two chopsticks

i^{th} Philosopher needs i^{th} & $i+1^{\text{th}}$ chopstick

Only put down chopstick when Philosopher has finished eating

Devise a solution which satisfies mutual exclusion but avoids starvation and deadlock

Seems simple enough ...?

```
while(true) {  
    think()  
    sem_wait(chopstick[i])  
    sem_wait(chopstick[(i+1) % N])  
    eat()  
    sem_post(chopstick[i])  
    sem_post(chopstick[(i+1) % N])  
}
```

... Deadlock (Each P. holds left fork)

```
while(true) {  
    think()  
    sem_wait(chopstick[i])  
    sem_wait(chopstick[(i+1) % N])  
    eat()  
    sem_post(chopstick[i])  
    sem_post(chopstick[(i+1) % N])  
}
```

What if?

Made picking up left and right chopsticks an atomic operation?

or,

N-1 Philosophers but N chopsticks?

What if?

Made picking up left and right chopsticks an atomic operation?

or,

N-1 Philosophers but N chopsticks?

... both changes prevent deadlock.

Formal requirements for deadlock

Mutual exclusion

Hold and wait condition

No preemption condition

Circular wait condition

Original scenario & our proposed ritual had all four of these properties.

Formal requirements for deadlock

Mutual exclusion

Exclusive use of chopsticks

Hold and wait condition

Hold 1 chopstick

No preemption condition

Cannot force another P. to undo their hold

Circular wait condition

N Philosophers N chopsticks