CS 333
Introduction to Operating Systems

Class 5 – Classical IPC Problems

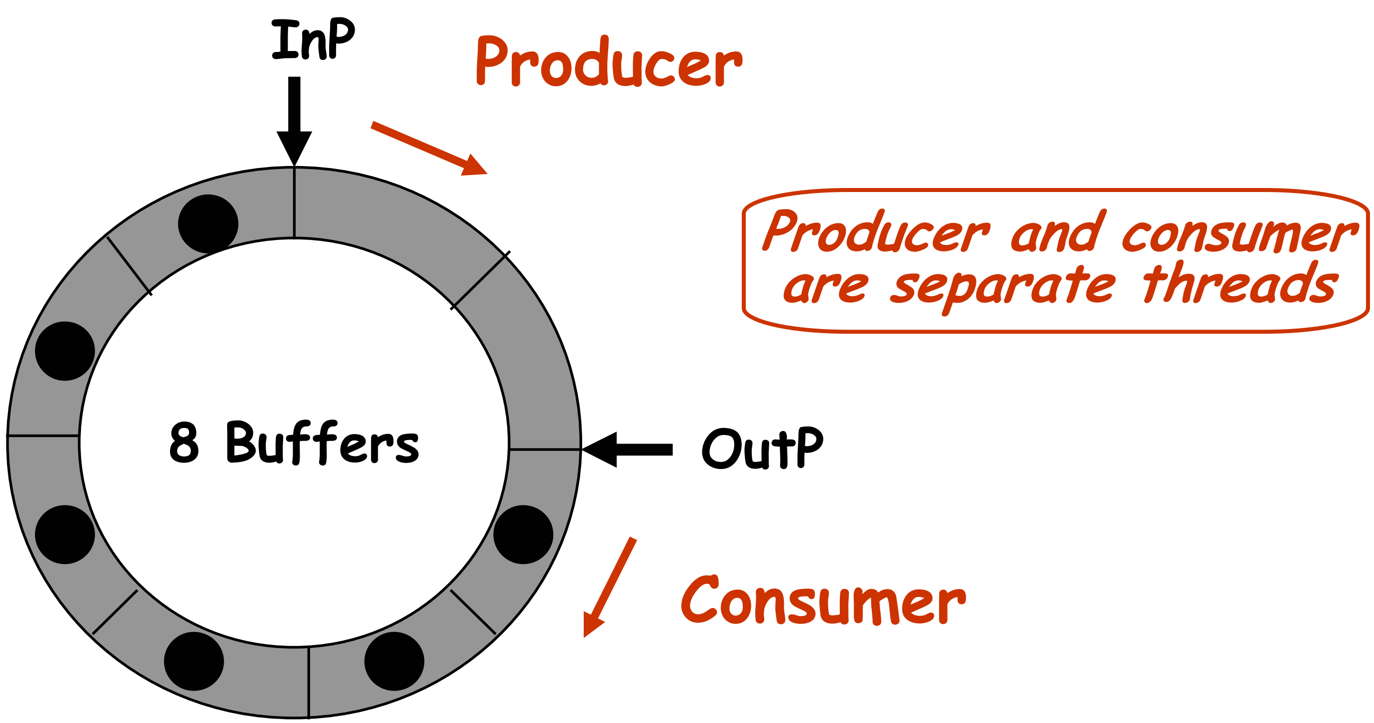
Jonathan Walpole
Computer Science
Portland State University

Classical IPC problems

- Producer Consumer (bounded buffer)
- Dining philosophers
- Sleeping barber
- Readers and writers

Producer consumer problem

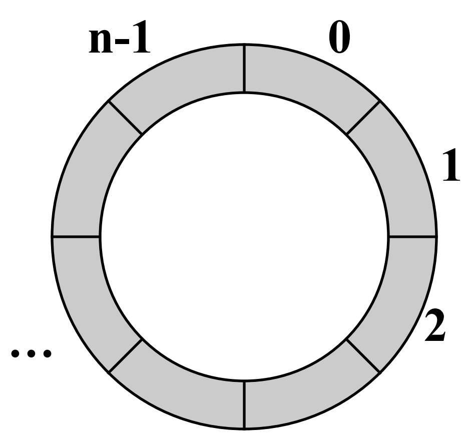
- Also known as the bounded buffer problem



Is this a valid solution?

```
thread producer {
    while(1){
      // Produce char c
      while (count==n) {
        no_op
      }
      buf[InP] = c
      InP = InP + 1 mod n
      count++
    }
}
```

```
thread consumer {
    while(1){
      while (count==0) {
        no_op
      }
      c = buf[OutP]
      OutP = OutP + 1 mod n
      count--
      // Consume char
    }
}
```

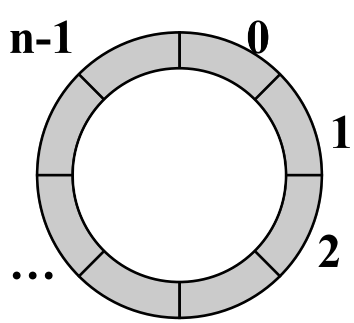



```
Global variables:
    char buf[n]
    int InP = 0    // place to add
    int OutP = 0   // place to get
    int count
```

How about this?

```
thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
    }
  }
```

```
thread consumer {
  while(1) {
    while (count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}
```




```
Global variables:
    char buf[n]
    int InP = 0    // place to add
    int OutP = 0   // place to get
    int count
```

Does this solution work?

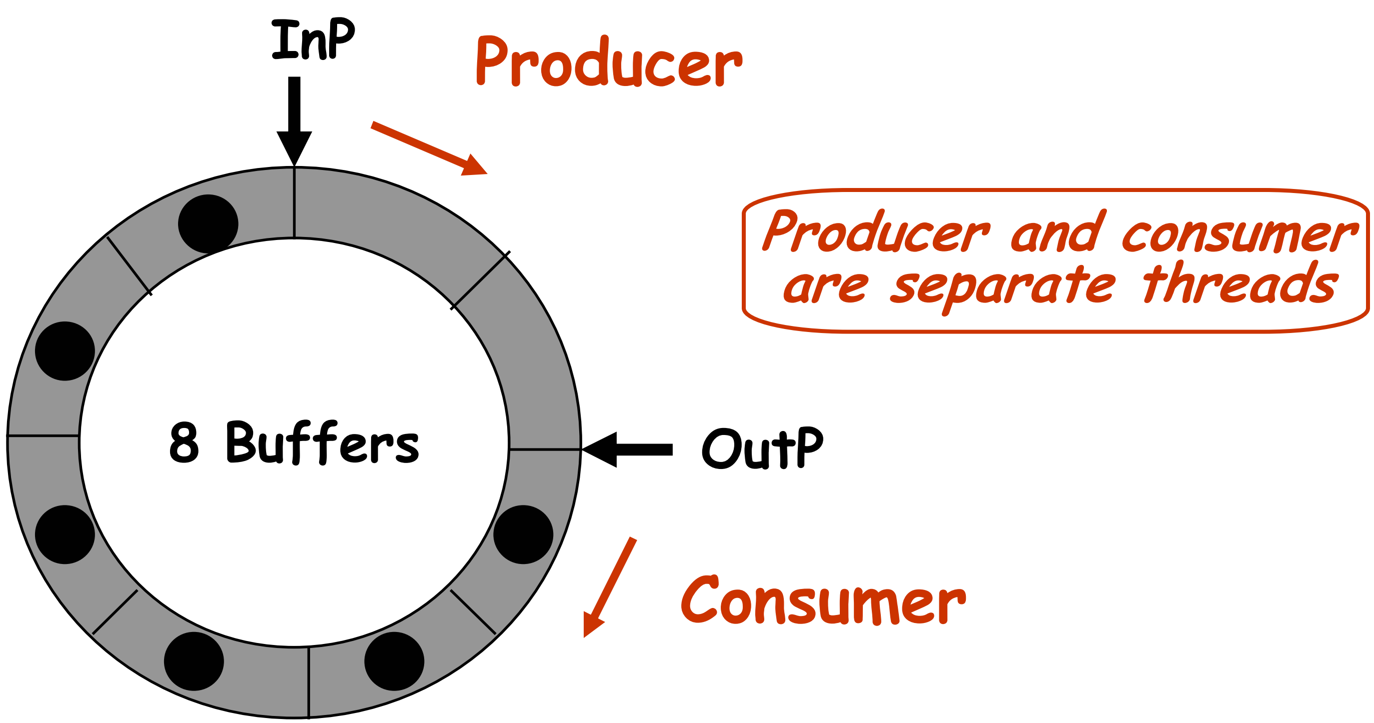
```
Global variables
  semaphore full_buffs = 0;
  semaphore empty_buffs = n;
  char buff[n];
  int InP, OutP;
```

```
thread producer {
  while(1){
    // Produce char c...
    down(empty_buffs)
    buf[InP] = c
    InP = InP + 1 mod n
    up(full_buffs)
  }
}
```

```
thread consumer {
  while(1){
    down(full_buffs)
    c = buf[OutP]
    OutP = OutP + 1 mod n
    up(empty_buffs)
    // Consume char...
  }
}
```

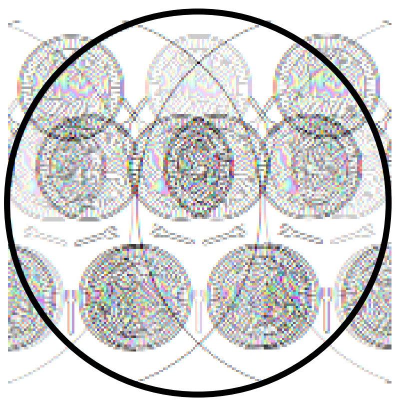
Producer consumer problem

- What is the shared state in the last solution?
- Does it apply mutual exclusion? If so, how?



Dining philosophers problem

- Five philosophers sit at a table
- One fork between each philosopher



Each philosopher is modeled with a thread

```
while(TRUE) {
  Think();
  Grab first fork;
  Grab second fork;
  Eat();
  Put down first fork;
  Put down second fork;
}
```

- *Why do they need to synchronize?*
- *How should they do it?*

Is this a valid solution?

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_fork(i);
    take_fork((i+1)% N);
    Eat();
    put_fork(i);
    put_fork((i+1)% N);
  }
}
```

Working towards a solution …

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_fork(i);
    take_fork((i+1)% N);
    Eat();
    put_fork(i);
    put_fork((i+1)% N);
  }
}
```

take_forks(i)

put_forks(i)

Working towards a solution …

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_forks(i);
    Eat();
    put_forks(i);
  }
}
```

Picking up forks

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```
take_forks(int i) {
  down(mutex);
  state [i] = HUNGRY;
  test(i);
  up(mutex);
  down(sem[i]);
}
```

```
// only called with mutex set!

test(int i) {
 if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING){
   state[i] = EATING;
   up(sem[i]);
 }
}
```

Putting down forks

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```
put_forks(int i) {
  down(mutex);
  state [i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(mutex);
}
```

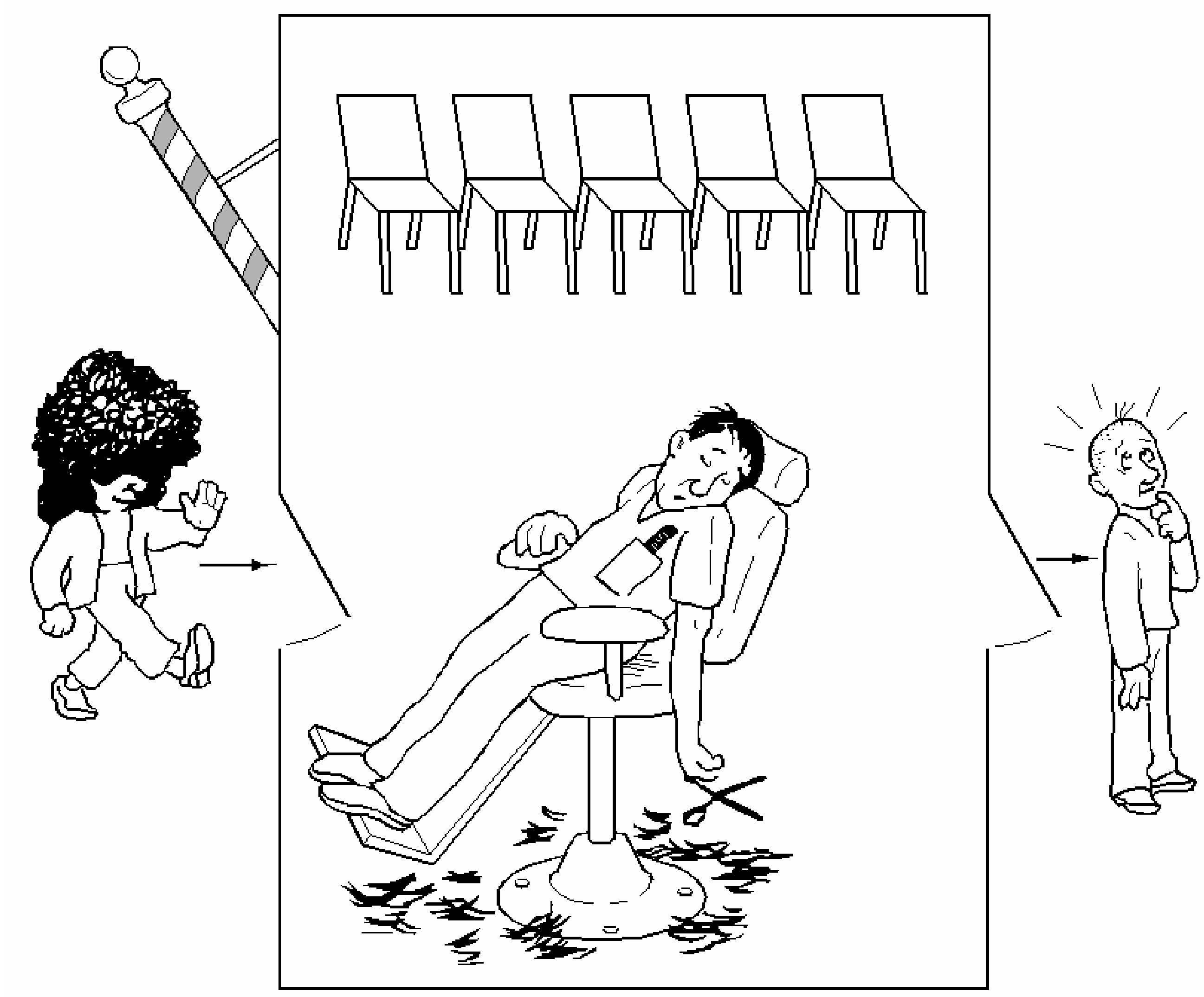
```
// only called with mutex set!

test(int i) {
 if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING){
    state[i] = EATING;
    up(sem[i]);
 }
}
```

Dining philosophers

- Is the previous solution correct?
- What does it mean for it to be correct?
- Is there an easier way?

The sleeping barber problem



The sleeping barber problem

- ***Barber:***
  - While there are people waiting for a hair cut, put one in the barber chair, and cut their hair
  - When done, move to the next customer
  - Else go to sleep, until someone comes in
- ***Customer:***
  - If barber is asleep wake him up for a haircut
  - If someone is getting a haircut wait for the barber to become free by sitting in a chair
  - If all chairs are all full, leave the barbershop

Designing a solution

- **How will we model the barber and customers?**
- **What state variables do we need?**
  - .. and which ones are shared?
  - …. and how will we protect them?
- **How will the barber sleep?**
- **How will the barber wake up?**
- **How will customers wait?**
- **What problems do we need to look out for?**

Is this a good solution?

```
const CHAIRS = 5
var customers: Semaphore
    barbers: Semaphore
    lock: Mutex
    numWaiting: int = 0
```

Barber Thread:

```
while true
  Down(customers)
  Lock(lock)
  numWaiting = numWaiting-1
  Up(barbers)
  Unlock(lock)
  CutHair()
endWhile
```

Customer Thread:

```
Lock(lock)
if numWaiting < CHAIRS
  numWaiting = numWaiting+1
  Up(customers)
  Unlock(lock)
  Down(barbers)
  GetHaircut()
else  -- give up & go home
  Unlock(lock)
endIf
```

The readers and writers problem

- **Multiple readers and writers want to access a database (each one is a thread)**
- **Multiple readers can proceed concurrently**
- **Writers must synchronize with readers and other writers**
  - *only one writer at a time !*
  - *when someone is writing, there must be no readers !*

Goals:

- Maximize concurrency.
- Prevent starvation.

Designing a solution

- **How will we model the barber and customers?**
- **What state variables do we need?**
  - .. and which ones are shared?
  - …. and how will we protect them?
- **How will the barber sleep?**
- **How will the barber wake up?**
- **How will customers wait?**
- **What problems do we need to look out for?**

Is this a valid solution to readers & writers?

```
var mut: Mutex = unlocked
    db: Semaphore = 1
    rc: int = 0
```

Writer Thread:

```
  while true
    ...Remainder Section...
    Down(db)
    ...Write shared data...
    Up(db)
  endWhile
```

Reader Thread:

```
  while true
    Lock(mut)
    rc = rc + 1
    if rc == 1
      Down(db)
    endIf
    Unlock(mut)
    ... Read shared data...
    Lock(mut)
    rc = rc - 1
    if rc == 0
      Up(db)
    endIf
    Unlock(mut)
    ... Remainder Section...
  endWhile
```

Readers and writers solution

- **Does the previous solution have any problems?**
  - is it "fair"?
  - can any threads be starved? If so, how could this be fixed?
  - … and how much confidence would you have in your solution?

Quiz

- When faced with a concurrent programming problem, what strategy would you follow in designing a solution?
- What does all of this have to do with Operating Systems?