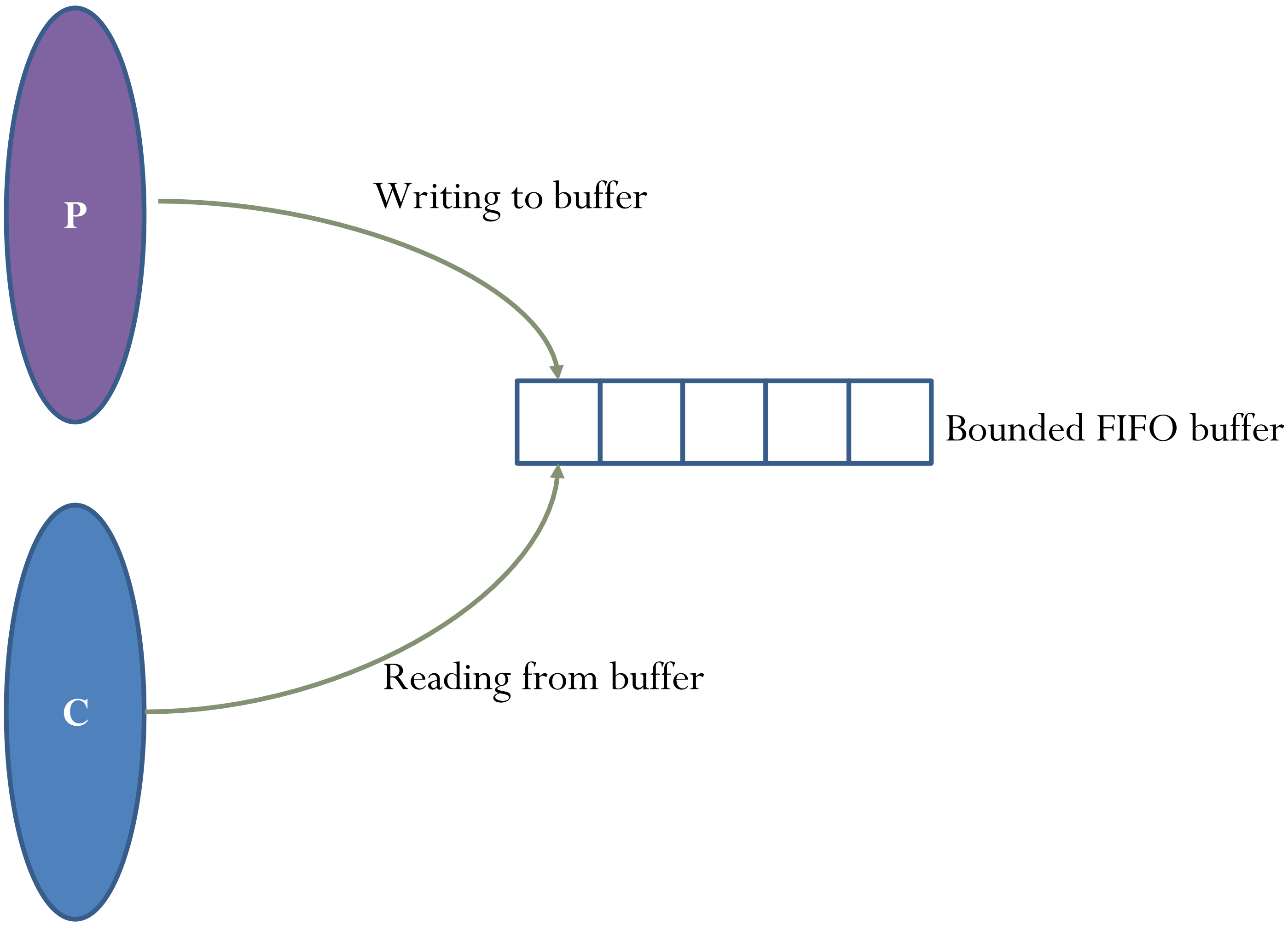
Synchronization

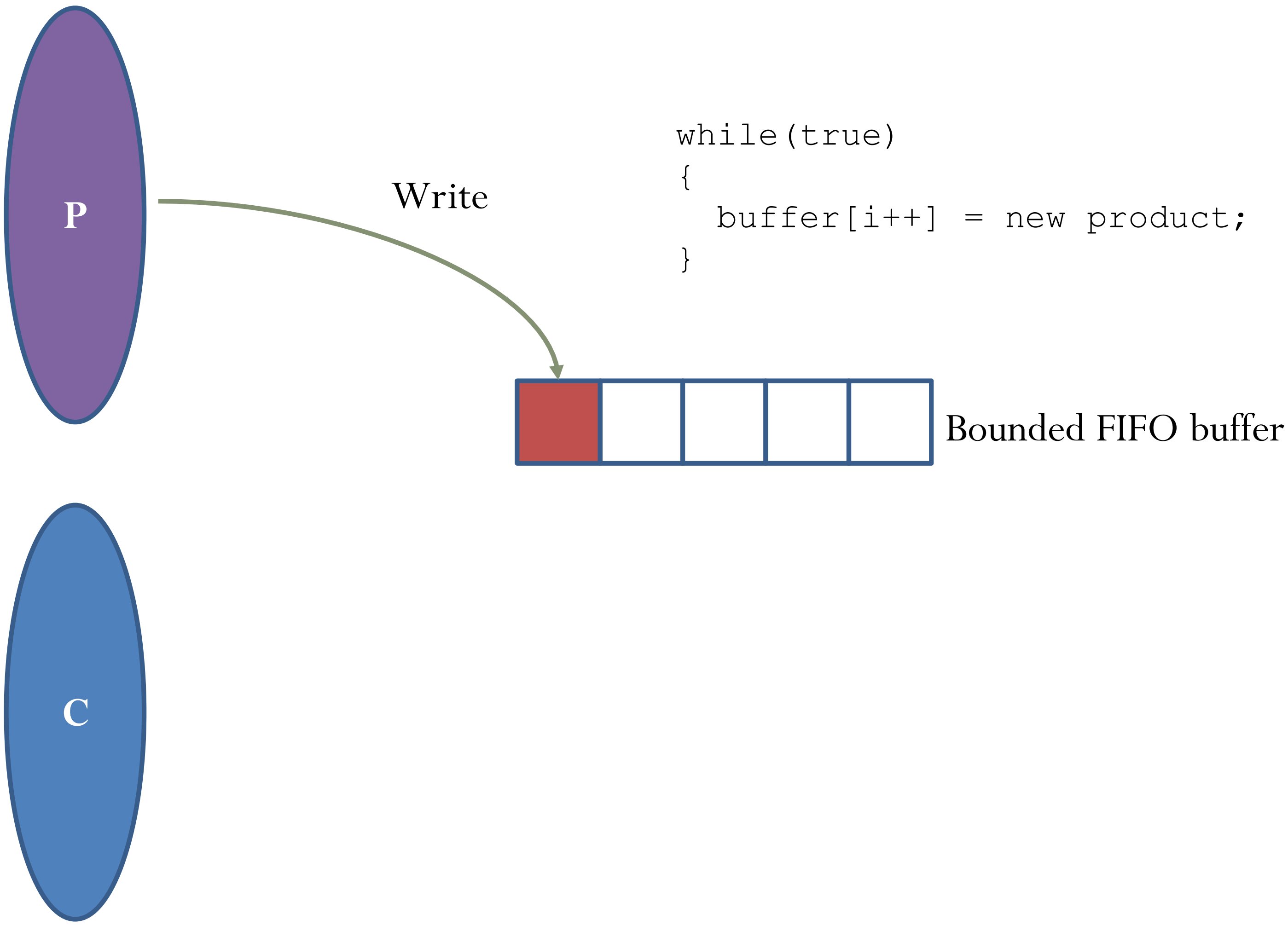
Instructor: William Zheng
Email:
william.zheng@digipen.edu
PHONE EXT: 1745

Producer – Consumer Problem

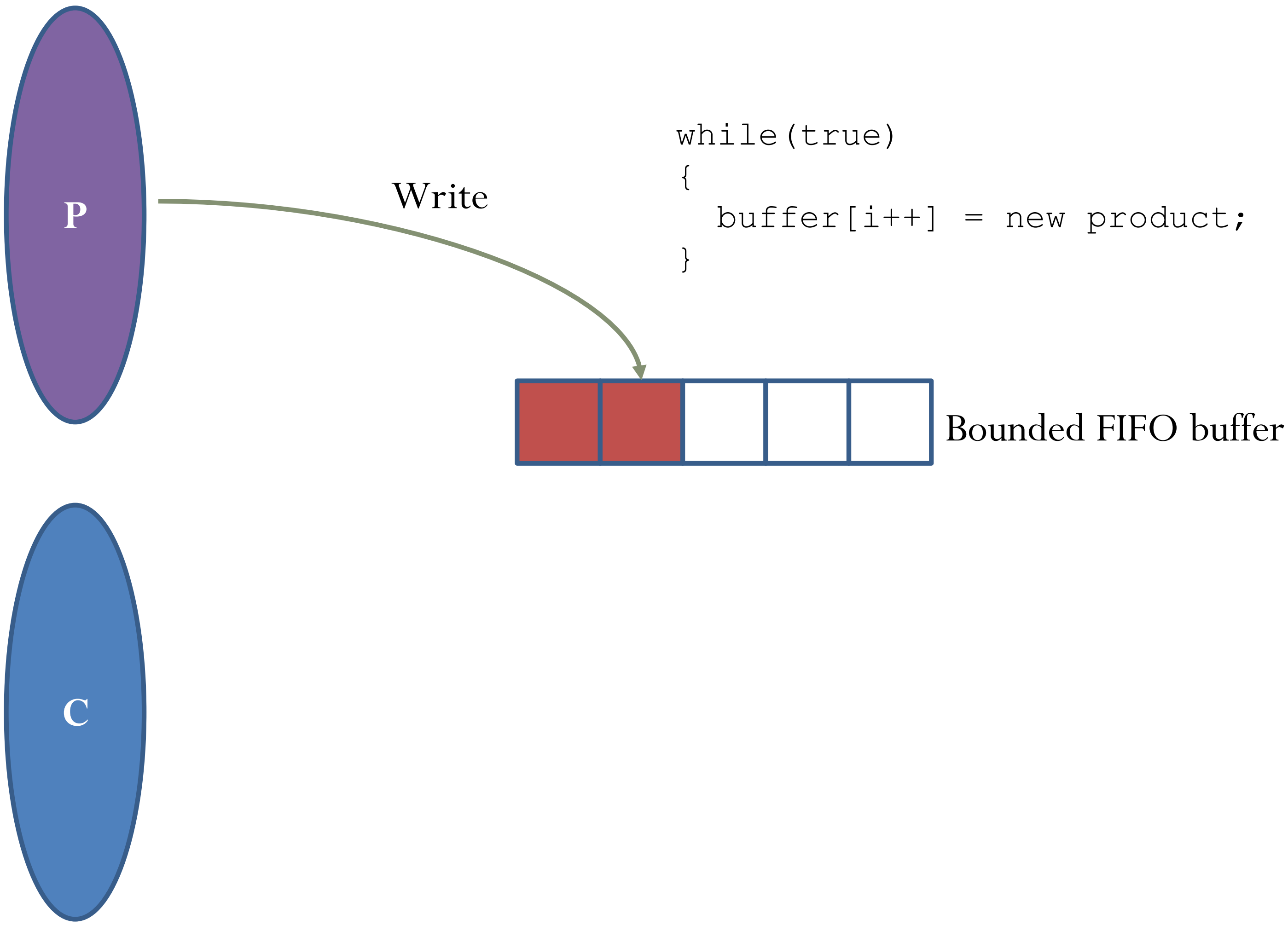


Producer writes to buffer

```
while(true)
{
  buffer[i++] = new product;
}
```

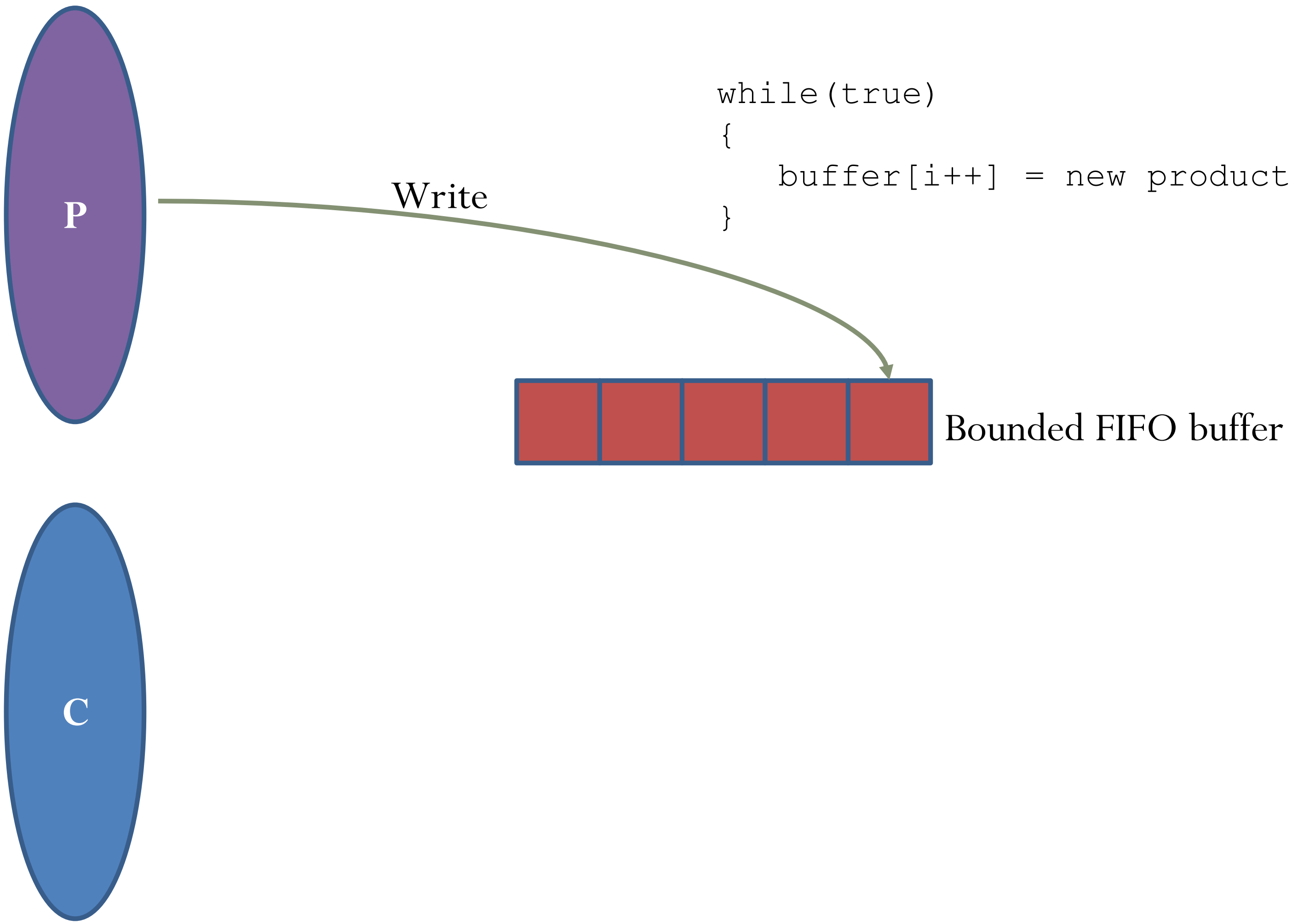


Buffer filling




```
while(true)
{
  buffer[i++] = new product;
}
```

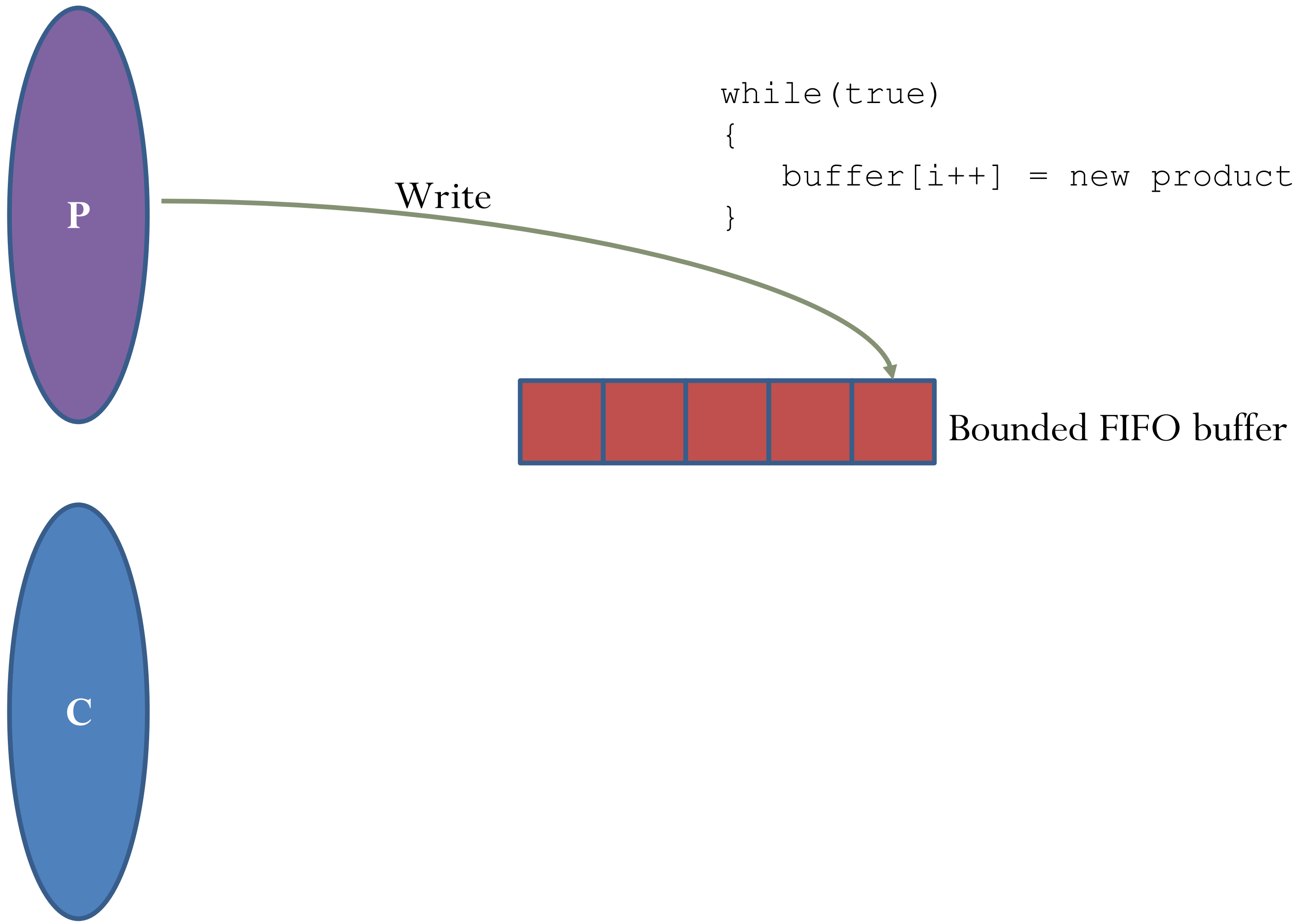
Stop when buffer is full!




```
while(true)
{
   buffer[i++] = new product;
}
```

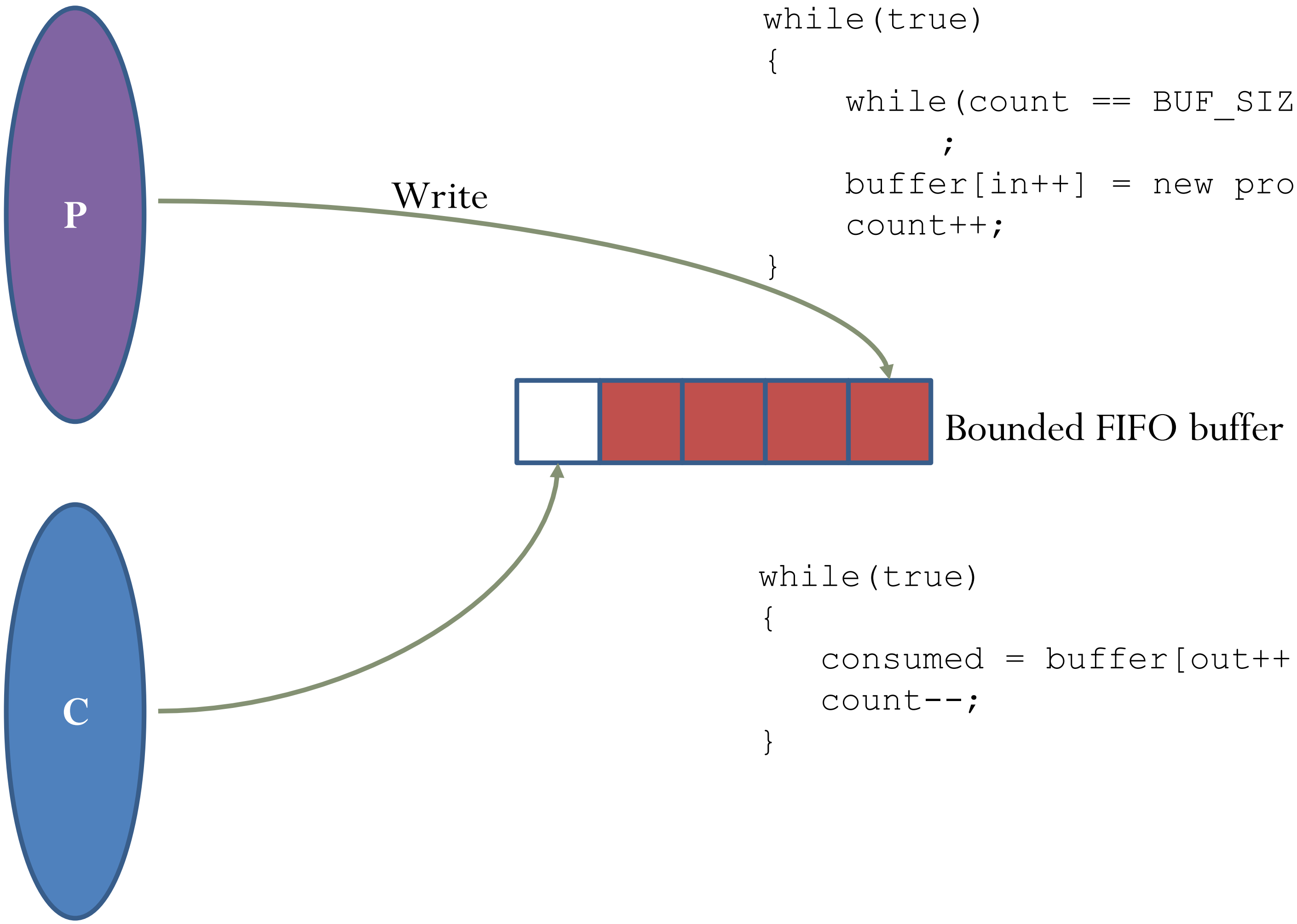
Stop when buffer is full!

```
while(true)
{
   buffer[i++] = new product;
}
```



Consumer reads buffer

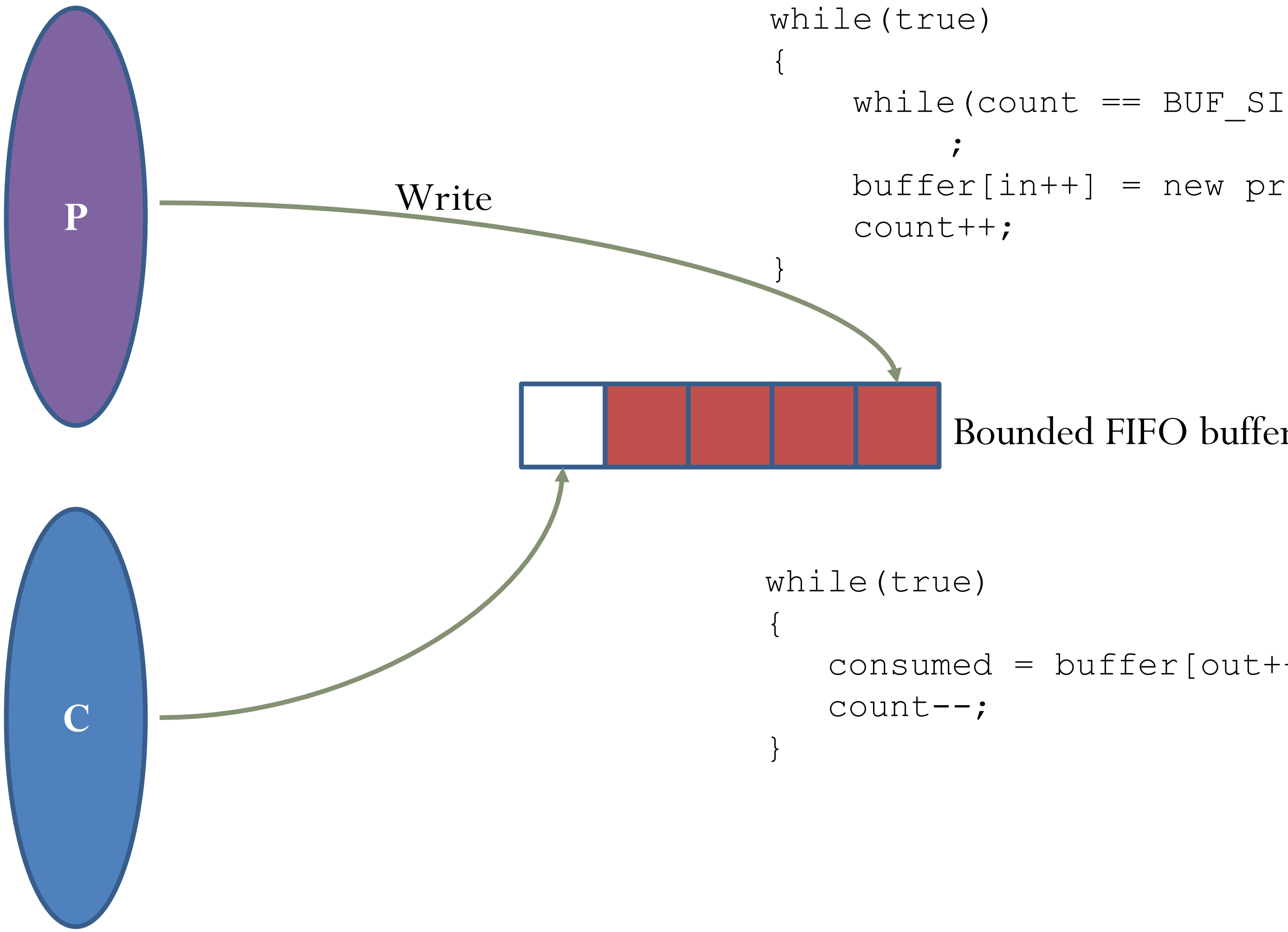
```
while(true)
{
    while(count == BUF_SIZE)
        ;
    buffer[in++] = new product;
    count++;
}
```




```
while(true)
{
    consumed = buffer[out++];
    count--;
}
```

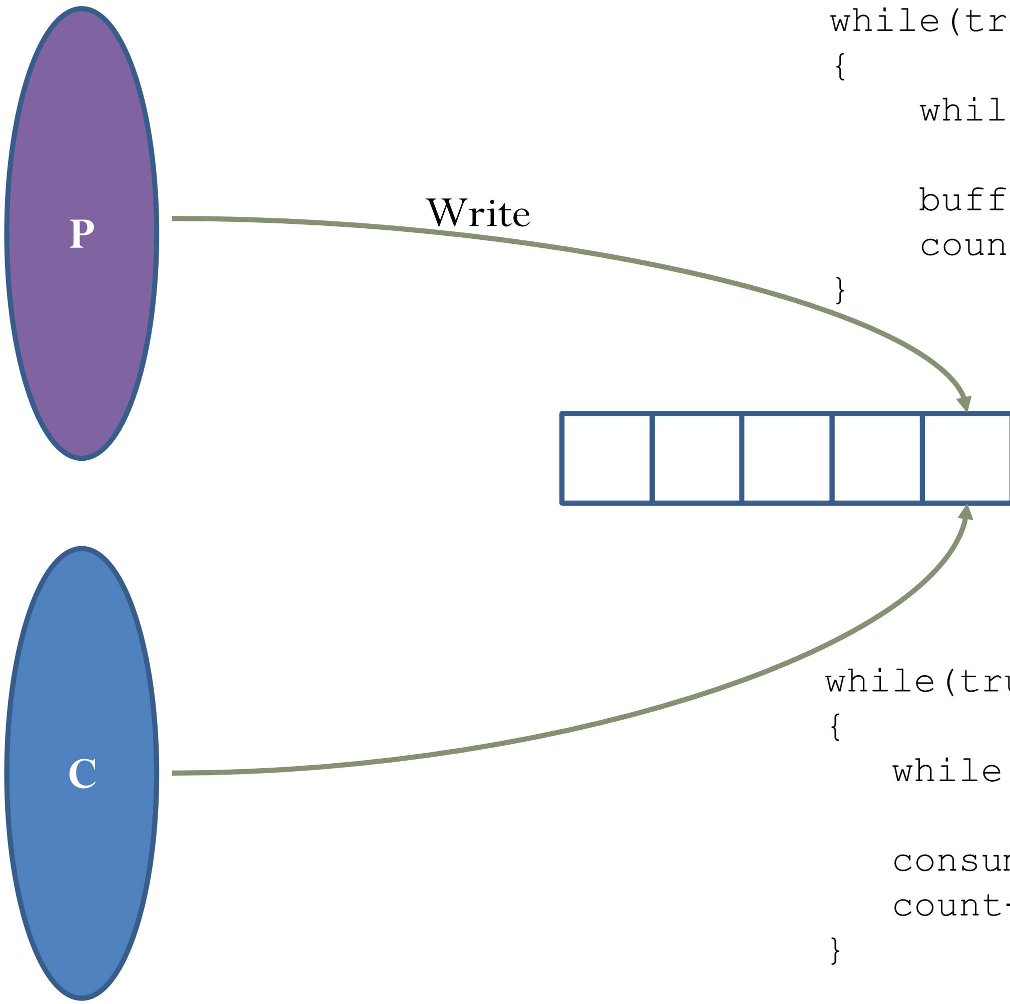
Consumer reads buffer

```
while(true)
{
    while(count == BUF_SIZE)
        ;
    buffer[in++] = new product;
    count++;
}
```




```
while(true)
{
   consumed = buffer[out++];
   count--;
}
```

Stop if buffer is empty!

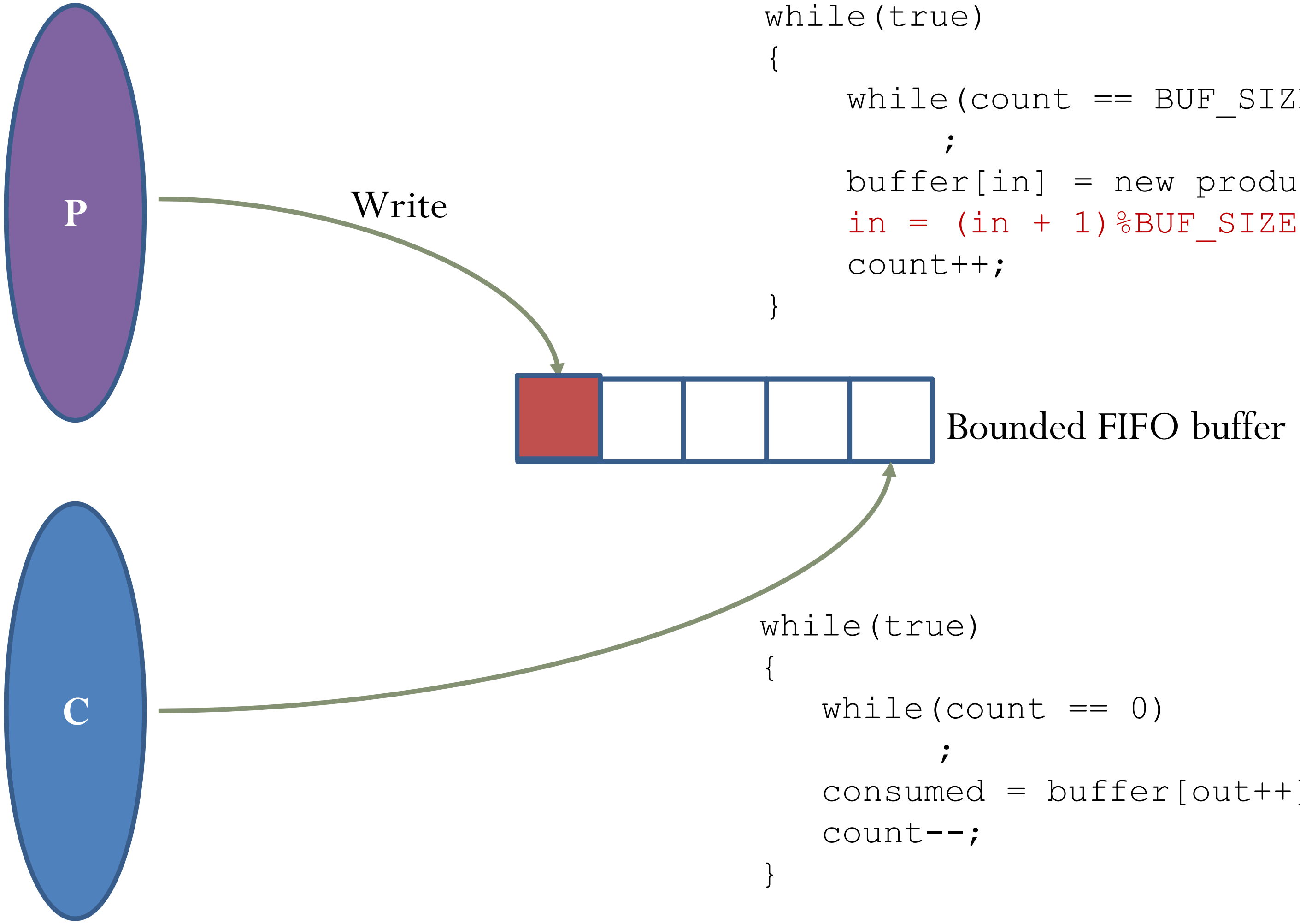



```
while(true)
{
    while(count == BUF_SIZE)
        ;
    buffer[in++] = new product;
    count++;
}
```

```
while(true)
{
   while(count == 0)
        ;
   consumed = buffer[out++];
   count--;
}
```

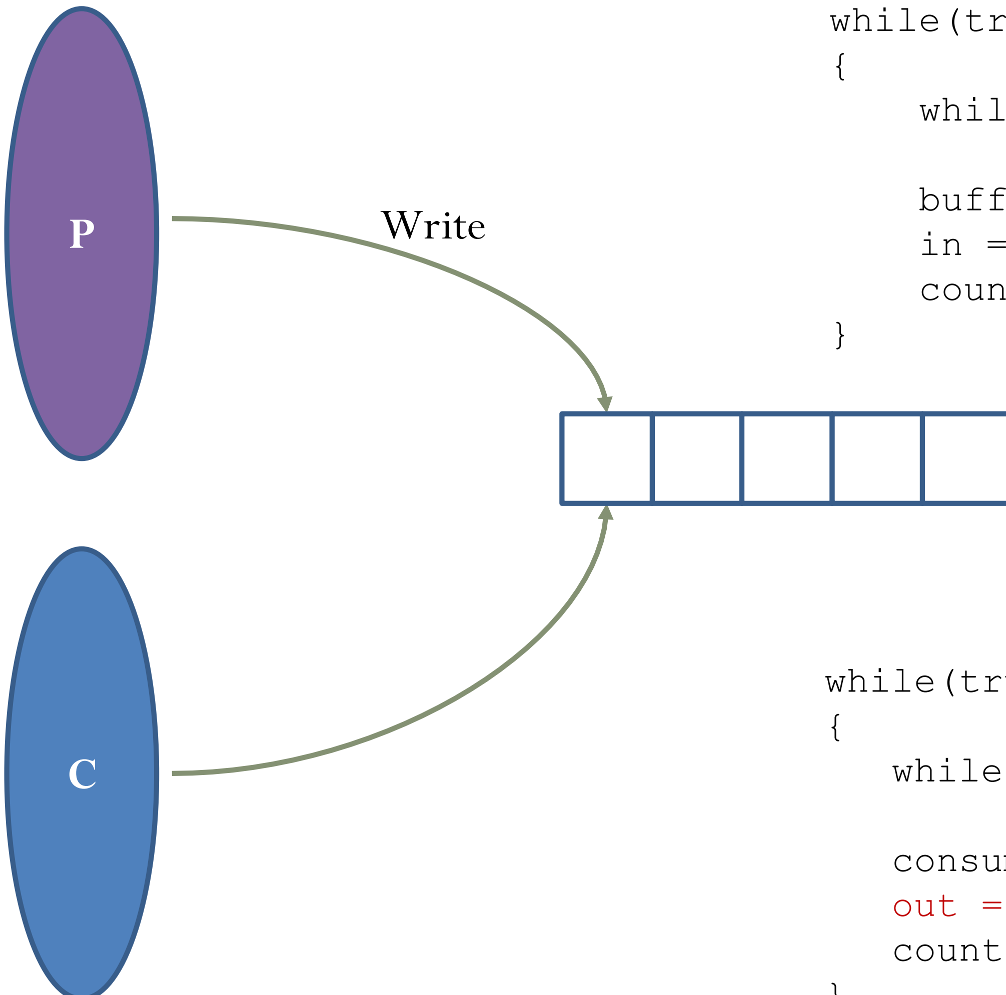
Wrap around - writing

```
while(true)
{
     while(count == BUF_SIZE)
           ;
     buffer[in] = new product;
     in = (in + 1)%BUF_SIZE;
     count++;
}
```




```
while(true)
{
   while(count == 0)
          ;
   consumed = buffer[out++];
   count--;
}
```

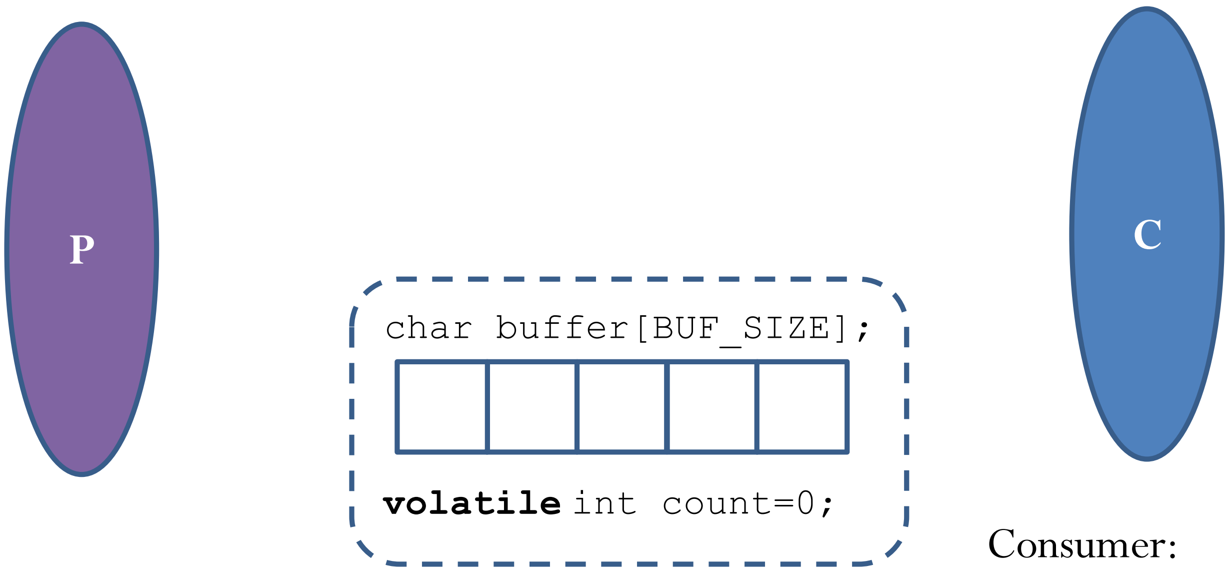
Wrap around - reading




```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

```
while(true)
{
   while(count == 0)
        ; //do nothing
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   count--;
}
```

Producer – consumer Problem



Shared Memory

Producer:

1. Insert item (wrap around)
2. Should stop when buffer is full

Consumer:

1. Read item (wrap around)
2. Should stop when buffer is empty

Examples

- Traffic simulation
- Event queues
- Streaming (e.g., video)
- Pipes between 2 processes

What’s the problem?

Consumer

```
while(true)
{
   while(count == 0)
         ; //do nothing
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   count--;
}
```

```
load count, r0      r0<-count
sub r0, r0, 1       r0<-r0-1
store r0, count     count<-r0
```

Producer

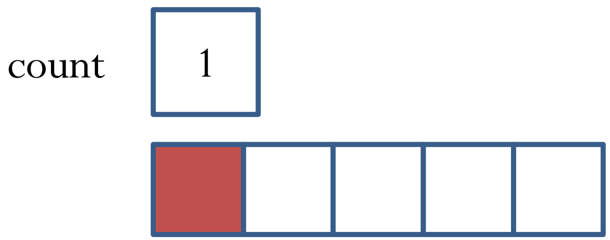
```
while(true)
{
    while(count == BUF_SIZE)
         ;
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

```
load count, r0      r0<-count
add r0, r0, 1       r0<-r0+1
store r0, count     count<-r0
```

count is in the memory. r0 is the register.

Main problem: updating of shared memory “count” is non-atomic

Illustration of problem - I



Scenario:

Producer has just put one item in buffer
count is 1
Consumer's turn

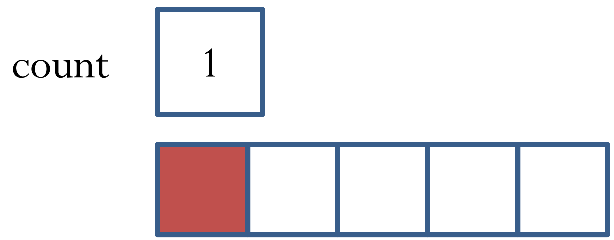
Consumer

```
(1)   load count, r0
(2)   sub r0, r0, 1
      store r0, count
```

Producer

```
load count, r0
add r0, r0, 1
store r0, count
```

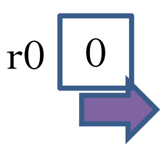
Illustration of problem - II



Scenario continued:

- Consumer has executed 2 instructions
- Before storing back, interrupt happens

Consumer




```
(1)   load count, r0
(2)   sub r0, r0, 1
      store r0, count
```

Producer

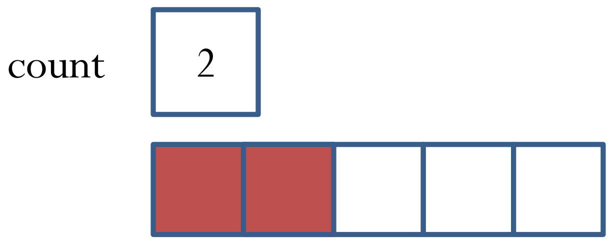
```
(3)   load count, r0
      add r0, r0, 1
      store r0, count
```

Illustration of problem - III

Scenario continued:

Producer successfully incremented count to 2

Interrupt occurs and switch back to consumer



Consumer		Producer	
(1)	`load count, r0`	(3)	`load count, r0`
(2)	`sub r0, r0, 1`	(4)	`add r0, r0, 1`
(6)	`store r0, count`	(5)	`store r0, count`

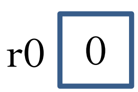
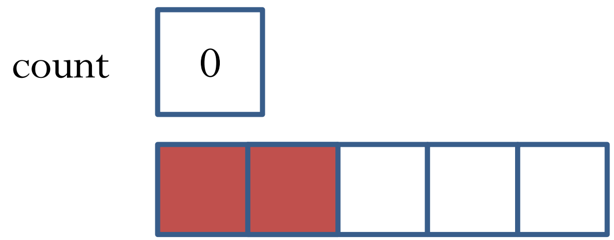


Illustration of problem - IV

Scenario continued:

Consumer writes 0 into count

Wrong value!



Consumer

r0 0

```
(1)   load count, r0
(2)   sub r0, r0, 1
(6)   store r0, count
```

Producer

```
(3)   load count, r0
(4)   add r0, r0, 1
(5)   store r0, count
```

The problem

- Non- atomicity
  - Divisible operation
  - Interruptible
- Instruction interleaving

Critical Section (CS)

Consumer

```
while(true)
{
   while(count == 0)
         ; //do nothing
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   count--;
}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
         ;
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

critical section:
Segments of code (in different threads or processes) sharing access to shared objects (e.g., files, variables, arrays etc)

There's more than 1 CS here! How?

General structure for CS solutions

```
while (true) {
      entry section e.g, entryCS()
      critical section
      exit section e.g., leaveCS()
      remainder section
}
```

Code for entry request

Code that "may" release someone to enter CS

Solution criteria

- Criteria
  - Mutual exclusion
  - Progress
  - Bounded Waiting
- Assumptions
  - Speed independence
  - Progress assumed in CS

Mechanisms for CS solution

- No enhanced support
  - Peterson's algorithm
- Hardware mechanisms
  - TestAndSet and Swap
- Operating system support
  - Semaphores

SW Solution 1: Peterson's algorithm

```
int turn;
int interested[2];
void EnterCS(int proc)
{
  int other;
  other = proc ^ 0x1; //toggle
  interested[proc] = true;
  turn = other;
  while((turn==other) && interested[other]);
}


void LeaveCS(int proc)
{
  interested[proc]= false;
}
```

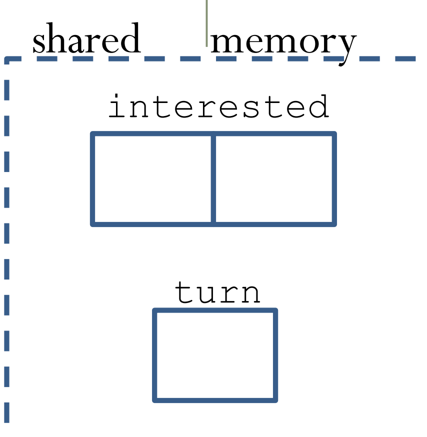
Peterson's algorithm

P_0 proc=0

```
void EnterCS(…)
{
  int other;
  other = 1;
  interested[0] = true;
  turn = 1;
  while((turn==1) &&
          interested[1]);
}


void LeaveCS(…)
{
   interested[0]= false;
}
```

P_1 proc=1

```
void EnterCS(…)
{
  int other;
  other = 0; //toggle
  interested[1] = true;
  turn = 0;
  while((turn==0) &&
          interested[0]);
}


void LeaveCS(…)
{
   interested[1]= false ;
}
```



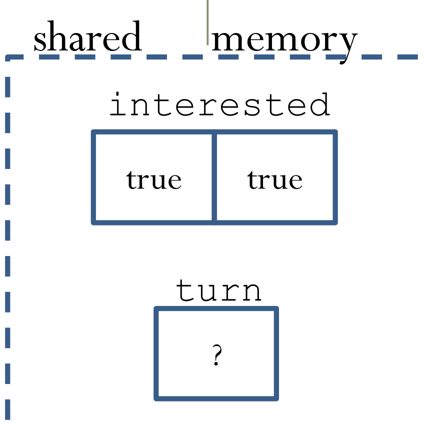
Can both go through?

P_0 proc=0

```
void EnterCS(…)
{
  int other;
  other = 1;
  interested[0] = true;
  turn = 1;
  while((turn==1) &&
          interested[1]);
}


void LeaveCS(…)
{
   interested[0]= false ;
}
```

P_1 proc=1

```
void EnterCS(…)
{
  int other;
  other = 0; //toggle
  interested[1] = true;
  turn = 0;
  while((turn==0) &&
         interested[0]);
}

void LeaveCS(…)
{
   interested[1]= false ;
}
```



Memory order

P_0 `proc=0`

Expected ordering

```
store 1, interested[0];
store 1, turn;
load r0, interested[1];
Load r1, turn;
```

Hardware messes it up during run-time!

```
load r0, interested[1];
store 1, interested[0];
store 1, turn;
load r1, turn;
```

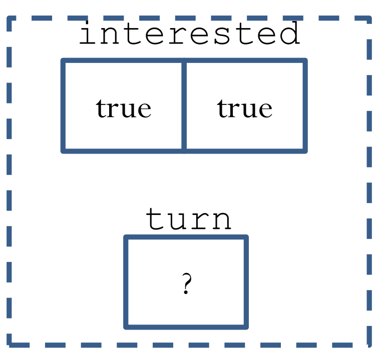
P_1 `proc=1`

Expected ordering

```
store 1, interested[1];
store 0, turn;
load r0, interested[0];
Load r1, turn;
```

```
load r0, interested[0];
store 1, interested[1];
store 0, turn;
load r1, turn;
```

shared memory



Peterson’s algorithm (Modern version)

```
int turn;
Int interested[2];
void EnterCS(int proc)
{
  int other;
  other = proc ^ 0x1; //toggle
  interested[proc] = true;
  turn = other;
  __asm__ (“mfence”);
  while((turn==other) && interested[other]);
}


void LeaveCS(int proc)
{
  interested[proc]= false ;
}
```

Memory barrier:
No memory reordering before this instruction

HW Solution 1: Disable Interrupts

```
while(true)
{
   Disable interrupts;
   //Critical Section
   …
   Enable interrupts;
   //Remainder Section

}
```

- Requirements
  - Computer Instruction
  - Usually kernel-mode
- Caveat
  - Uniprocessor only
  - Scalability issues

HW Solution 2: Test And Set

```
bool TestAndSet(bool *tgt)
{
  bool rv = *tgt;
  *tgt = true;
  return rv;
}
```

```
do
{
   while(TestAndSet(&lock));

   // critical section

   lock = false;

   // remainder section
} while (true);
```

HW Solution 3: Compare and Swap

```
int CompareAndSwap(
  int *value,
  int expected
  int new_value)
{
  int temp = *value;

  if(*value == expected)
      *value = new_value;

  return temp;
}
```

```
do {
  while(CompareAndSwap(&lock,
  0, 1) !=0);
  // critical section

  lock = 0;

  // remainder section
} while (true);
```

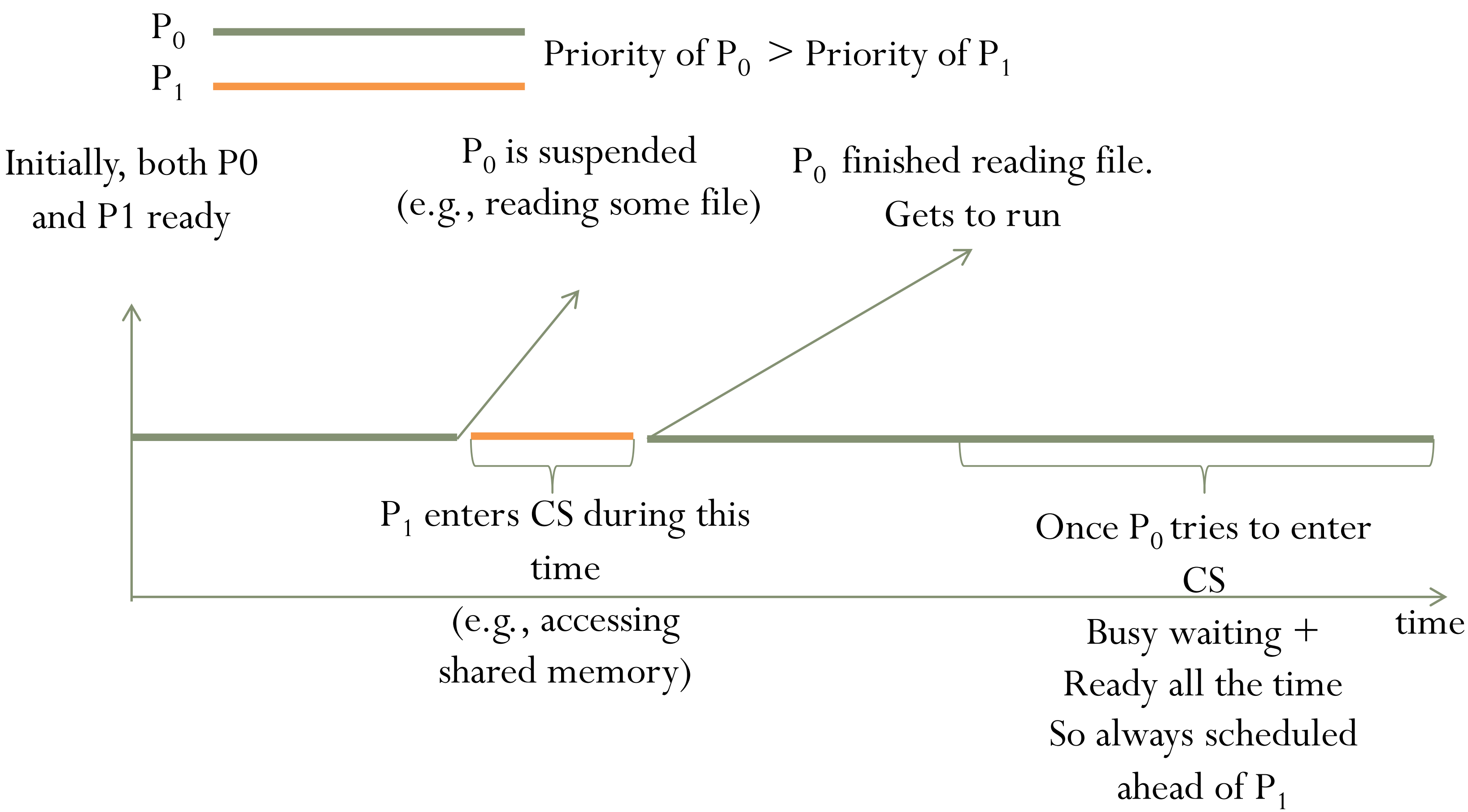
HW Solutions

- TestAndSet and CompareAndSwap are HW instructions!
- Atomic
- Require hardware design
- Fails bounded waiting requirement (Refer to textbook chap 6.4)

Busy waiting

- Who's busy waiting?
  - Peterson, TestAndSet, CompareAndSwap
- Cons
  - Low CPU Utilization
  - Priority Inversion Problem (Uniprocessor)

Priority Inversion (Uniprocessor version)

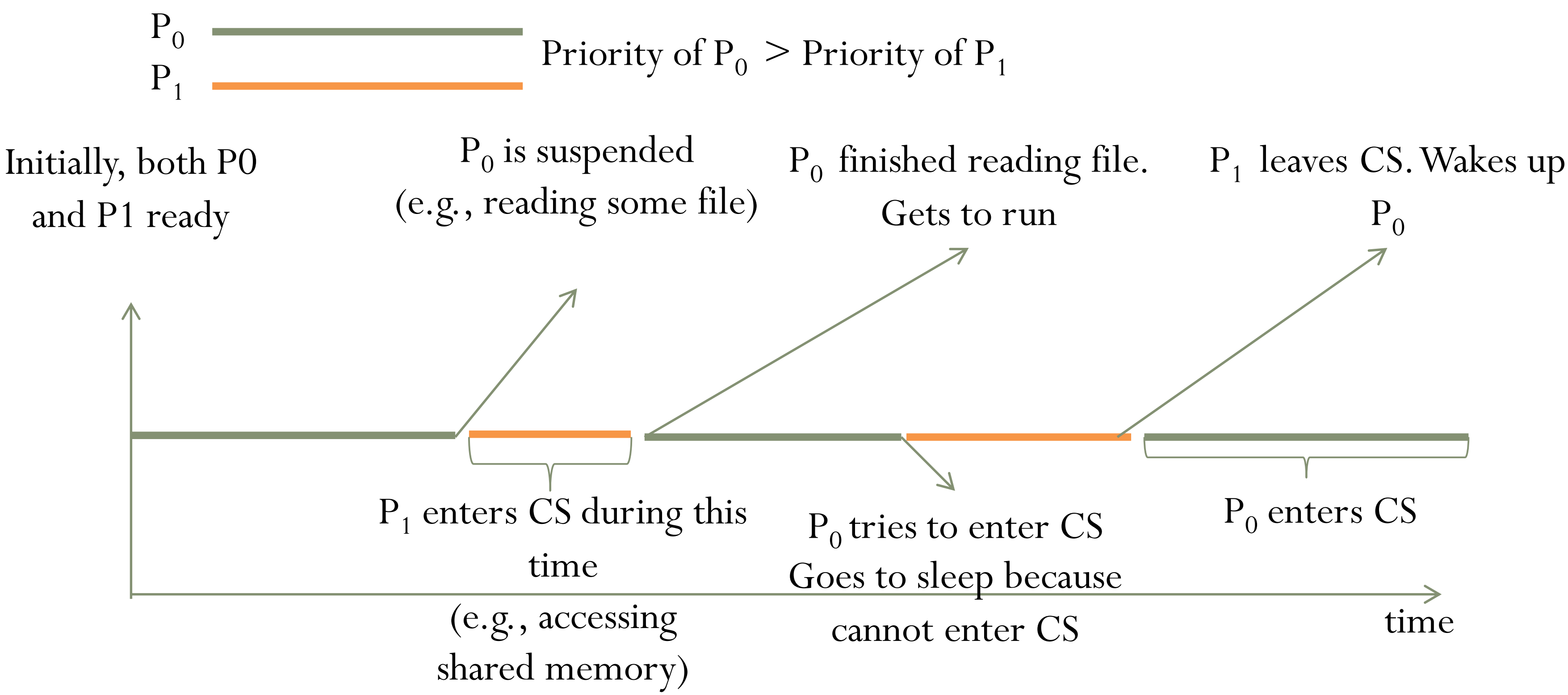


Sleep and wakeup

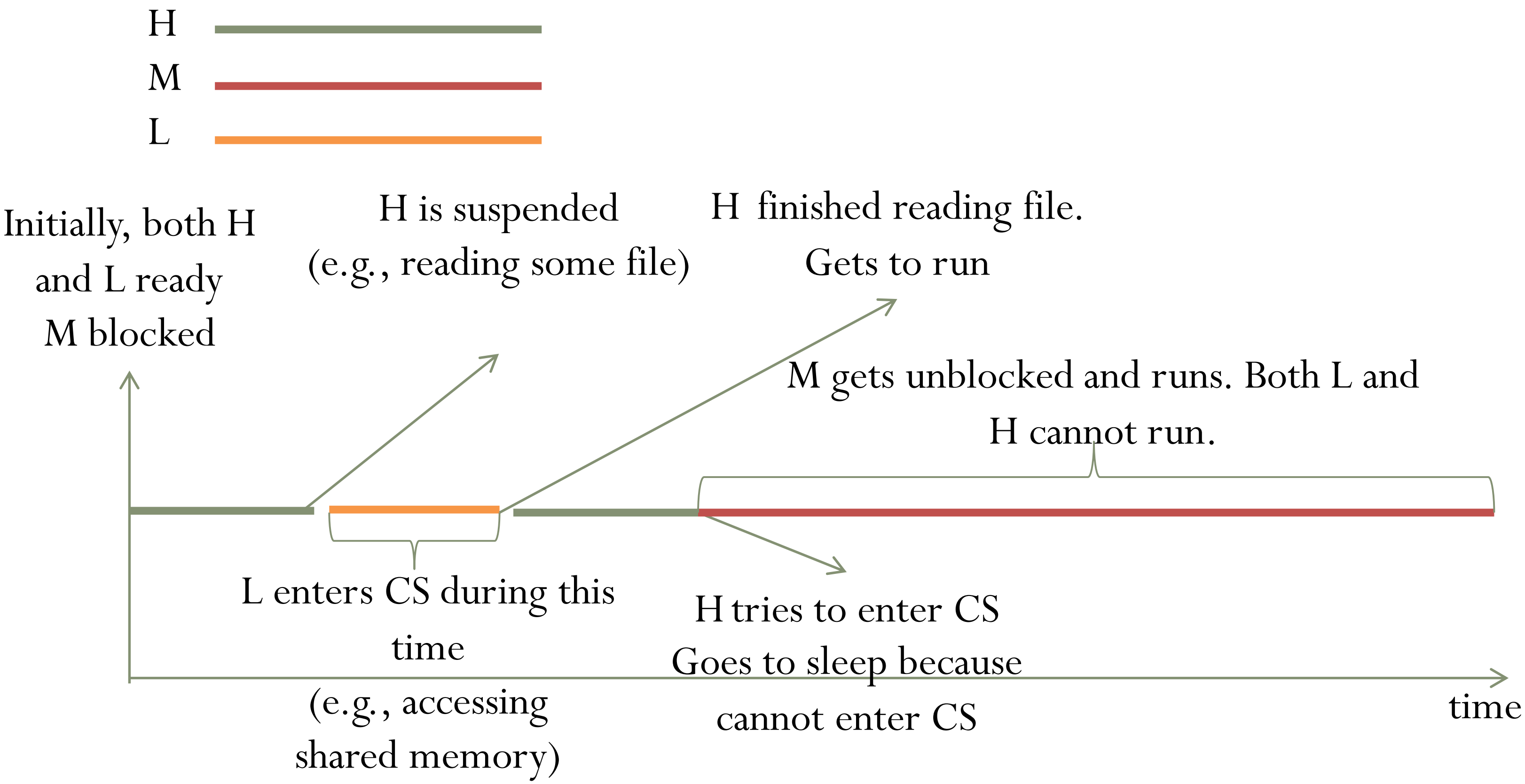
```
EnterCS()
{
   if cannot enter CS
     block calling process;
}

LeaveCS()
{
   When leaving CS,
   wakeup one of any
   blocking process in
   EnterCS
}
```

- Pros
  - Better CPU Utilization
  - Avoids priority inversion (?!)
- Cons
  - Less Efficient
  - Requires OS support

Priority Inversion avoided (Uniprocessor version)



Priority Inversion comes back with a vengeance!



Semaphores – initial definition

```
class Semaphore
{
  void wait ()
  {
      while(s <=0) ;
      s--;
  }
  void signal()
  {
       s++;
  }
  private:
       int s;
}
```

Busy-waiting

```
class Semaphore
{
  void wait ()
  {
      if(s <= 0)
          block this process
      s--;
  }
  void signal()
  {
       s++;
       wake up one of blocking
       processes
  }
  private:
       int s;
}
```

Sleep-wakeup

Types of semaphore

- Counting/General semaphores
  - value of semaphore initialized to N
- Binary semaphore
  - value semaphore initialized to 1

Using semaphores

```
Semaphore s(1);

while(true)
{
    s.wait();
    //critical section
    s.signal();
    //remainder section;
}
```

Only 1 in CS each time

```
Semaphore s(N);

while(true)
{
    s.wait();
    //critical section
    s.signal();
    //remainder section;
}
```

Up to N in CS each time

Producer-Consumer Problem Revisited

Consumer

```
while(true)
{
   while(count == 0)
         ; //do nothing
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   count--;
}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
         ;
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

Producer-Consumer Problem Revisited

Consumer

```
while(true)
{
   while(count == 0)
         ; //do nothing
   mutex.wait();
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   mutex.signal();
   count--;
}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
         ;
    mutex.wait();
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    mutex.signal();
    count++;
}
```

Not in the same process!

Process 1

```
while(true)
{
    S1;
    synch.signal();
}
```

Process 2

```
while(true)
{
    synch.wait();
    S2;
}
```

Shared memory

```
Semaphore synch;
```

Consumer should block when empty…

Consumer

```
while(true)
{
   while(count == 0)
         ; //do nothing
   mutex.wait();
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   mutex.signal();
   count--;
}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
         ;
    mutex.wait();
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    mutex.signal();
    count++;
}
```

Consumer should block when empty...

Consumer

should block when buffer is empty!

```
while(true)
{
    empty.wait();
    mutex.wait();
    consumed = buffer[out];
    out = (out + 1) %BUF_SIZE;
    mutex.signal();
    count--;
}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
        ;
    mutex.wait();
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    mutex.signal();
    empty.signal();
}
```

increment available buffer count

Producer should block when buffer is full!

Consumer

```
while(true)
{
    empty.wait();
    mutex.wait();
    consumed = buffer[out];
    out = (out + 1) %BUF_SIZE;
    mutex.signal();
    full.signal();
}
```

increment available buffer count

Producer

should block when buffer is full!

```
while(true)
{
    full.wait();
    mutex.wait();
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    mutex.signal();
    empty.signal();
}
```

Semaphores – sleep and wakeup

```
class Semaphore
{
  void wait ()
  {
       if(s <=0)
      {
         add self-process to queue;
         block();
      }
      s--;
  }
  void signal()
  {
      s++;

      if queue is not empty

          wake up any process

          waiting in queue
  }
  private:
         int s;
}
```

- So can we eliminate busy waiting entirely?

Semaphores - critical section

```
class Semaphore
{
  void wait ()
  {
       if(s <=0)
       {
          add self-process to queue;
          block();
       }
       s--;
  }
  void signal()
  {
       s++;
       if queue is not empty
            wake up any process
            waiting in queue
  }
  private:
          int s;
}
```

- So can we eliminate busy waiting entirely?

Implementation of Semaphores (REVISED)

```
void wait()
{
  while(TestAndSet(&lock));
  if(s <= 0)
  {
      add self-process to
      queue;
      lock = false;
      block(&lock);
  }
  s--;
  lock = false;
}
```

```
void signal()
{

  while(TestAndSet(&lock));
  s++;
  if (queue is not empty)
     wake up any process
     waiting in queue
  lock = false;
}
```

Sleeping barber problem

- A barbershop has N chairs and one barber
- The barber does one of two things
  - If there is at least one customer, he chooses one and cuts his hair
  - If there are no customers, he goes to sleep (waits for a customer)
- When a customer enters
  - If there is at least one seat available, he seats himself and tells the barber he would like a haircut, and waits for a haircut
  - If there are no seats available, he leaves

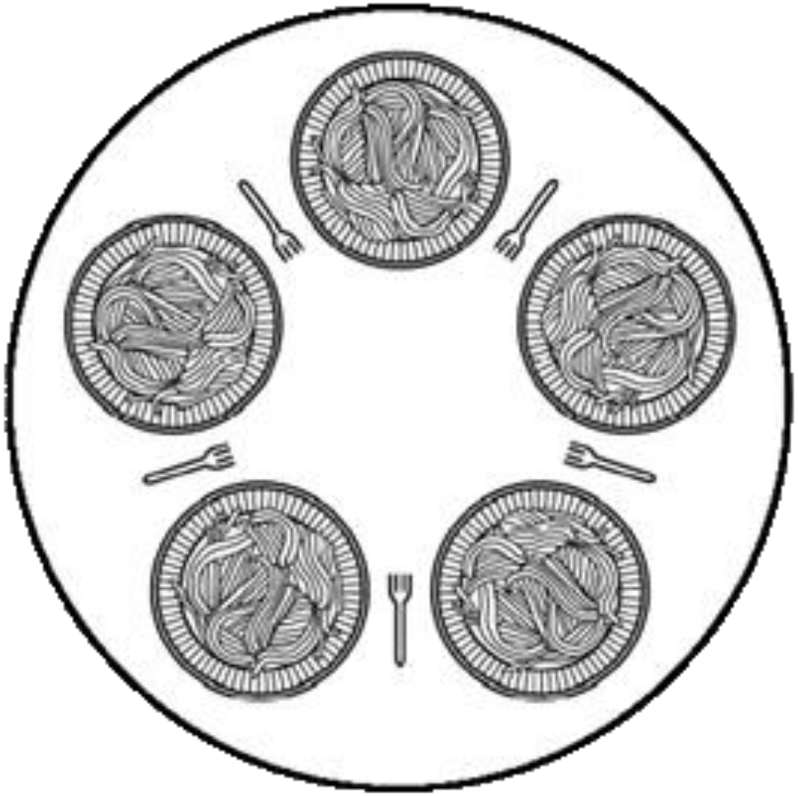
Solution for sleeping barber

```
Semaphore barbers = 0;
Semaphore mutex = 1;
Semaphore customers = 0;
int nFreeWRSeats = N;

void Barber(){
     while (1) {
          wait(customers);
          wait(mutex);
          nFreeWRSeats ++;
          signal(barbers);
          signal(mutex);
          haircut();
      }
}
```

```
void Customer() {
    wait(mutex);
    if (nFreeWRSeats > 0) {
      nFreeWRSeats --;
      signal(customers);
      signal(mutex);
      wait(barber);
      get_haircut();
    }
    else {
      signal(mutex);
      //leaveWR();
    }
}
```

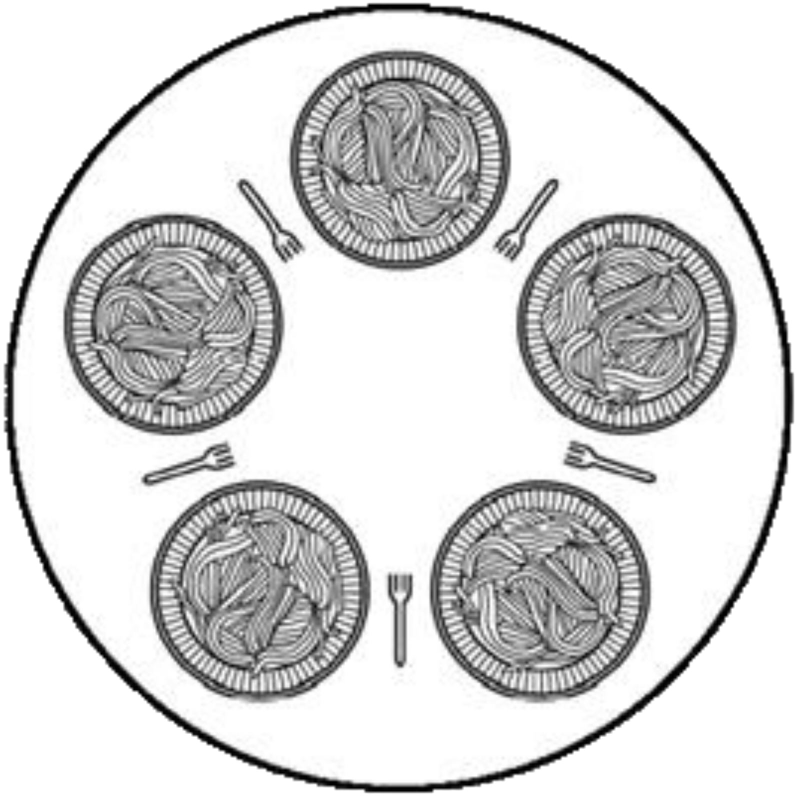
Dining Philosopher's Problem

```
while (true)
{
  get_left();
  get_right();
  eat();
  think();
}
```



Dining Philosopher's Problem

```
Semaphore chopsticks[N];

while (true)
{
   wait(chopsticks[i]);
   get_left();
   wait(chopsticks[(i+1)%N]);
   get_right();
   eat();
   signal(chopsticks[i]);
   signal(chopsticks[(i+1)%N]);
   think();
}
```

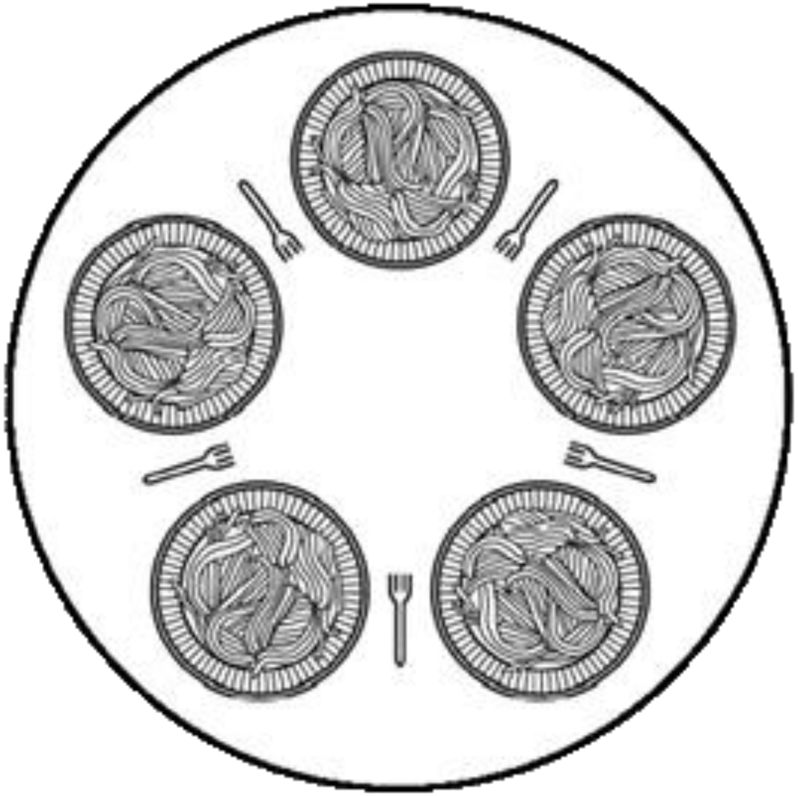


Resource Hierarchy solution for Dining Philosopher's Problem

- Assigns a partial order to the forks and all forks will be requested in order
- forks will be numbered 1 through 5
- Each philosopher will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks they plan to use.
- The order of putting down forks does not matter.
- If 4 of the five philosophers simultaneously pick up their lower-numbered fork, only the highest-numbered fork will remain on the table,
    - 5^{th} philosopher not able to pick up any fork
    - only one philosopher will have access to that highest-numbered fork

Resource Hierarchy solution (REVISED)

```
Semaphore chopsticks[N];

while (true)
{
  if ((i+1)%N > i){
    wait(chopsticks[i]);
    get_left();
    wait(chopsticks[(i+1)%N]);
    get_right();
  }
  else {
    wait(chopsticks[(i+1)%N]);
    get_right();
    wait(chopsticks[i%N]);
    get_left();
  }
   eat();
   signal(chopsticks[i]);
   signal(chopsticks[(i+1)%N]);
   think();
}
```



Deadlock and Starvation

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation – **indefinite blocking**

A process may never be removed from the semaphore queue in which it is suspended

Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Deadlock and Livelock

- **DEADLOCK** a condition in which a task waits indefinitely for conditions that can never be satisfied
  - task claims exclusive control over shared resources
  - task holds resources while waiting for other resources to be released
  - tasks cannot be forced to relinquish resources
  - a circular waiting condition exists
- **LIVELOCK**
  - State of the processes involved in the livelock constantly change with regard to one another
- Live lock happens when process are in ready or running state while deadlock deals with waiting state

Livelock Example

```
/* Thread 0*/

While (1) {
     Acquire lock for A

     Yield CPU cycles

     If (Trying to lock for B
     fails)
         continue;
     Else
     {
         /*CS*/
         unlock for B;
     }
}
```

```
/* Thread 1*/

While (1) {
     Acquire lock for B

     Yield CPU cycles

     If (Trying to lock for A
     fails)
         continue;
     Else
     {
         /*CS*/
         unlock for A;
     }
}
```

/*Main*/
Creat two threads (Thread 0 and Thread 1)
Sleep for some time
Send TERM signal to both threads

Race condition

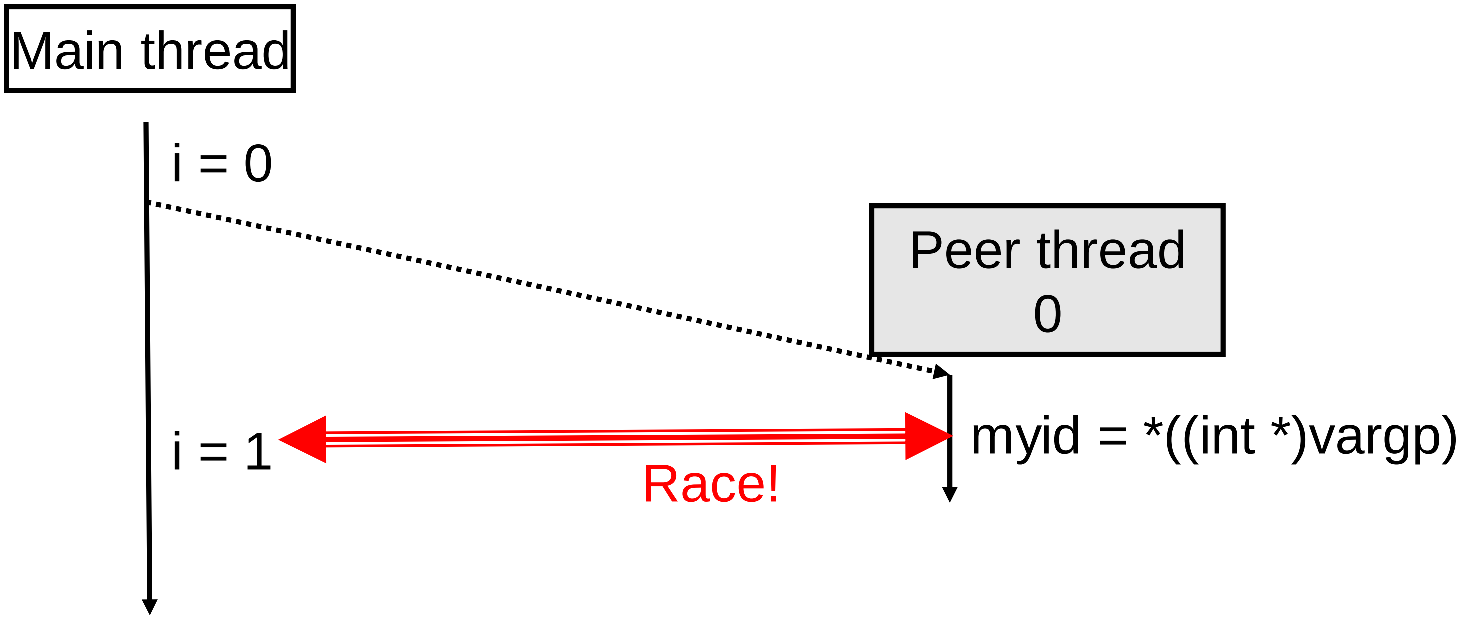
```c
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;

    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

N threads are sharing i

A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

Race Illustration

```
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, &i);
```



- Race between increment of i in main thread and deref of vargp in peer thread:
    - If deref happens while i = 0, then OK
    - Otherwise, peer thread gets wrong id value