Operating Systems and Concurrency

Lecture 10: Concurrency

G52OSC/COMP2007

Geert De Maere
(Isaac Triguero)
{Geert.DeMaere,Isaac.Triguero}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2018

Recap

Last Lecture

- Concurrency using **semaphores** and **mutexes**
  - **Mutex** is **a spinlock** (busy waiting)
  - **Semaphore** puts process to **sleep**

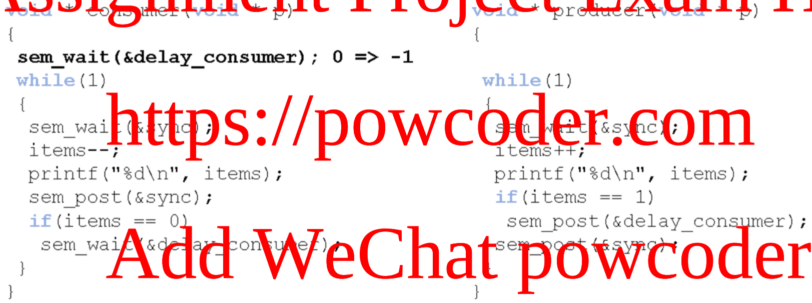
- Practical examples on how to **use (code) semaphores**
- Concurrency in practice is **difficult** (performance, deadlocks, priority inversion)

Goals For Today

- The **bounded buffer** problem
- The **dining philosophers** problem

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



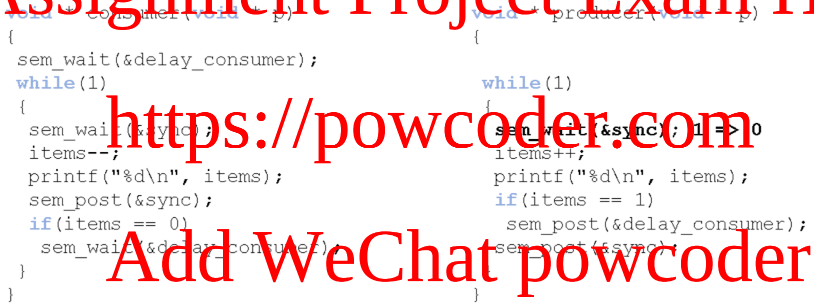
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer); 0 => -1
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



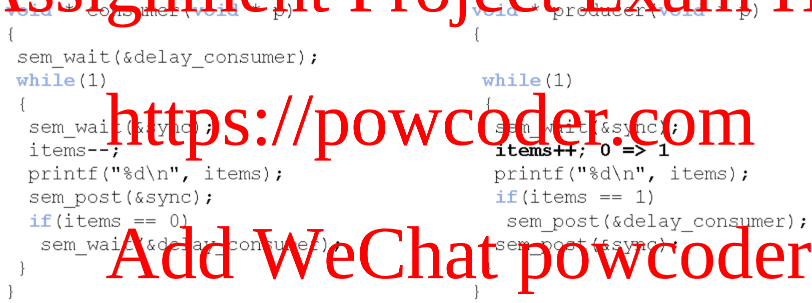
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync); 1 => 0
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



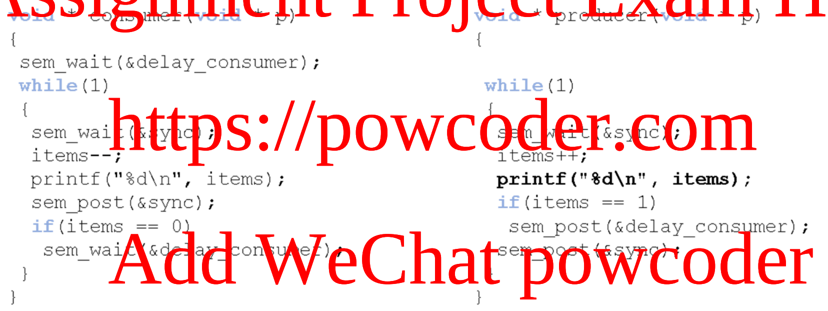
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++; 0 => 1
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

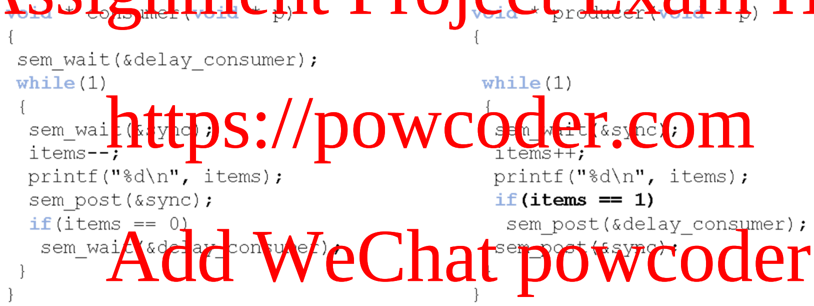


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



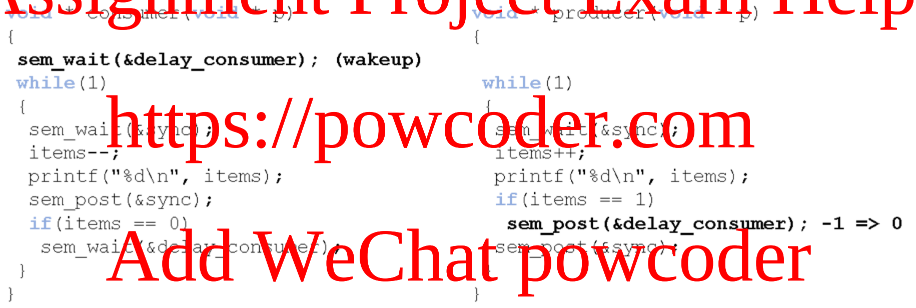
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



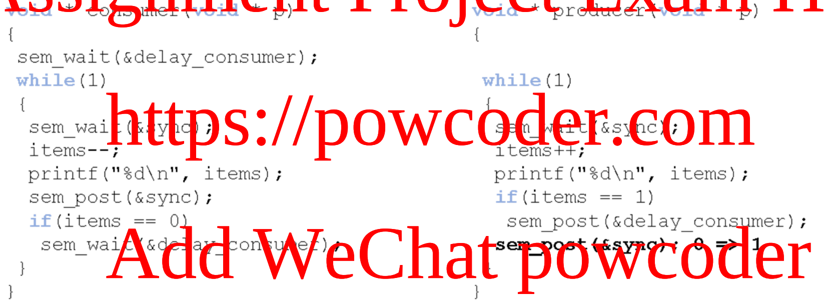
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);  (wakeup)
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer); -1 => 0
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

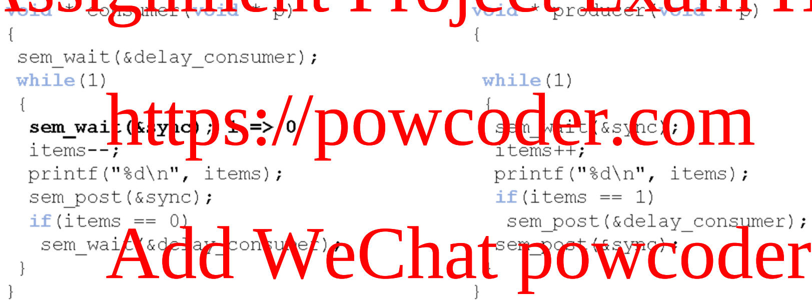


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync); 0 => 1
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



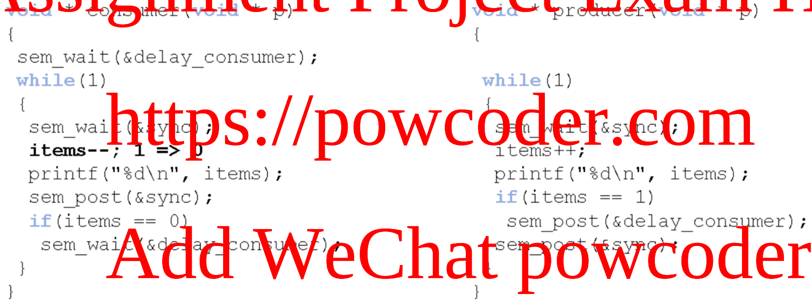
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync); 1 => 0
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



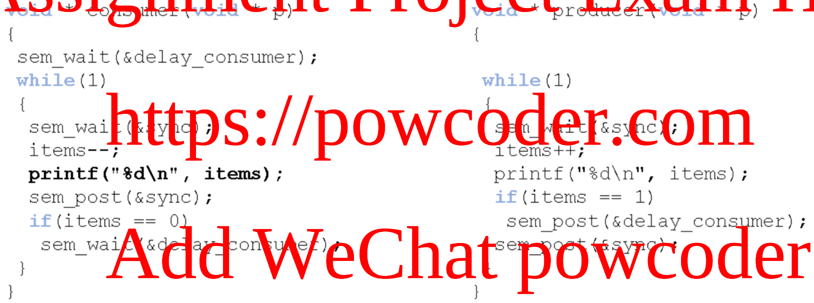
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--; 1 => 0
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

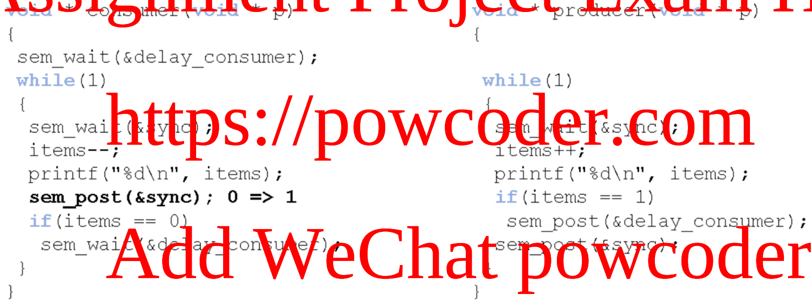


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



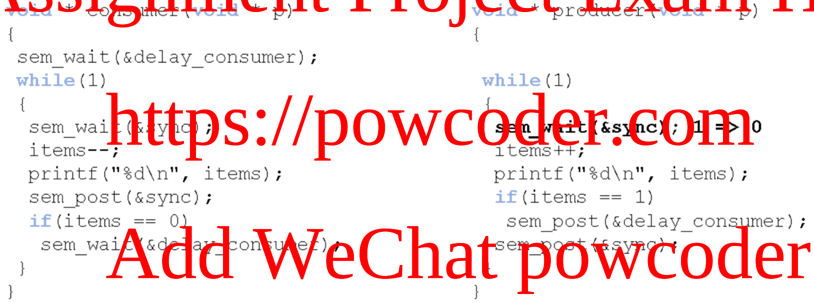
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync); 0 => 1
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



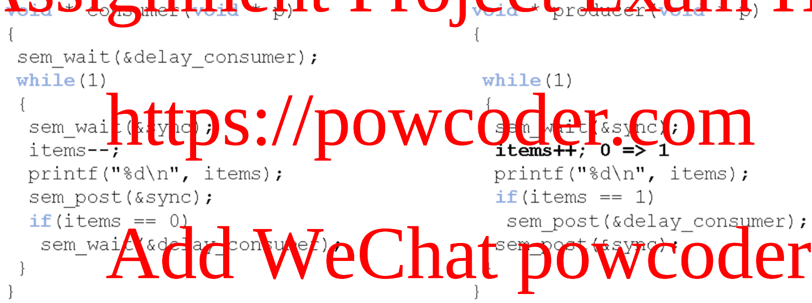
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync); 1 => 0
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



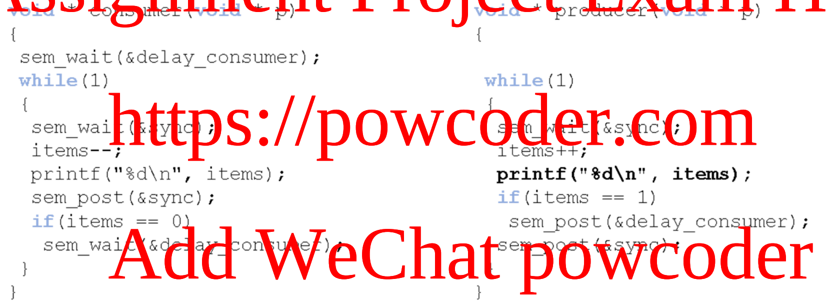
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++; 0 => 1
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

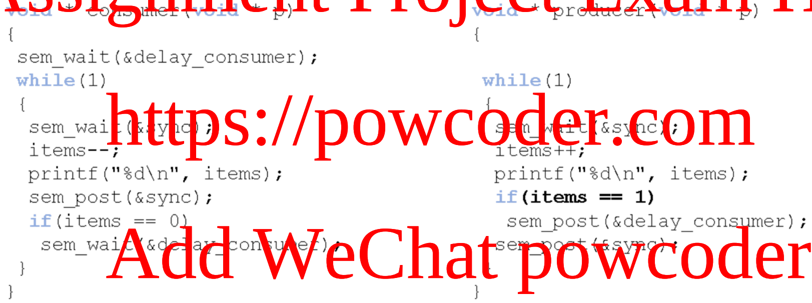


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

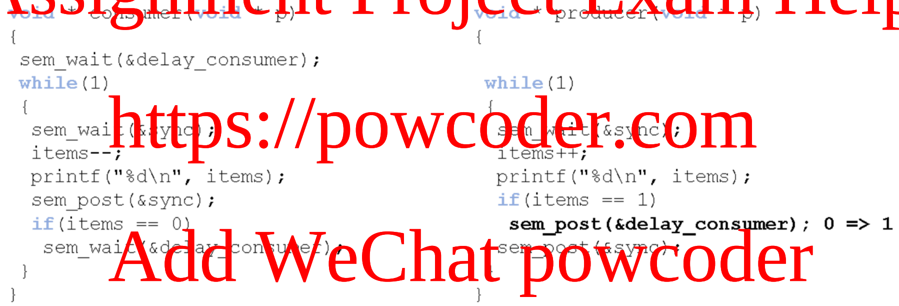


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



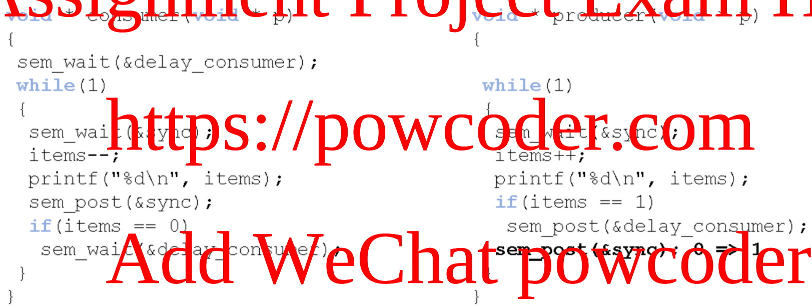
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer); 0 => 1
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

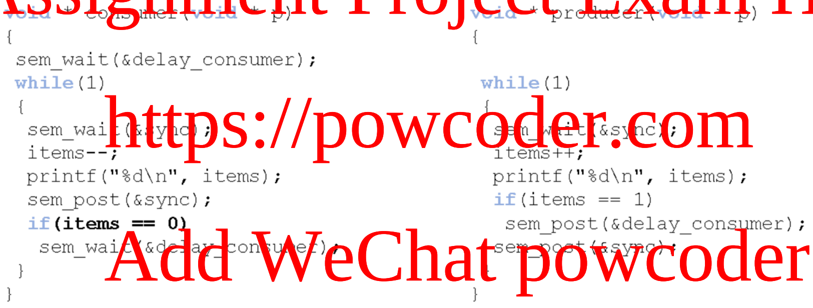


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync); 0 => 1
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

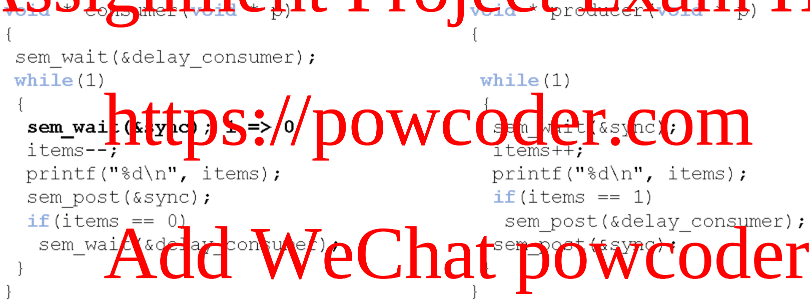


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



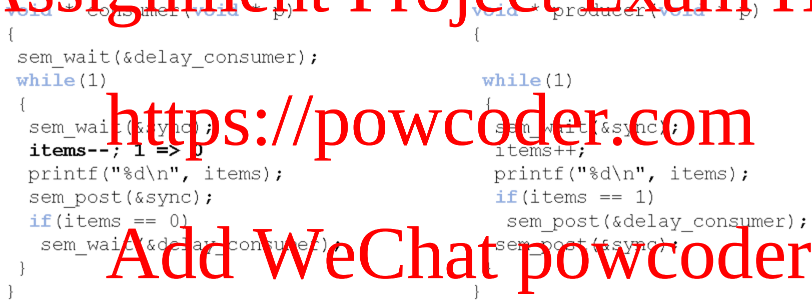
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync); 1 => 0
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



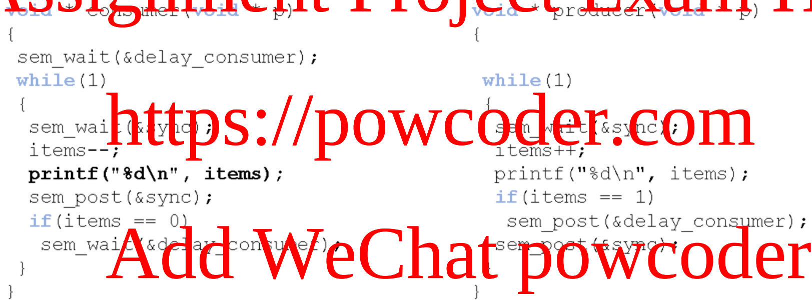
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--; 1 => 0
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



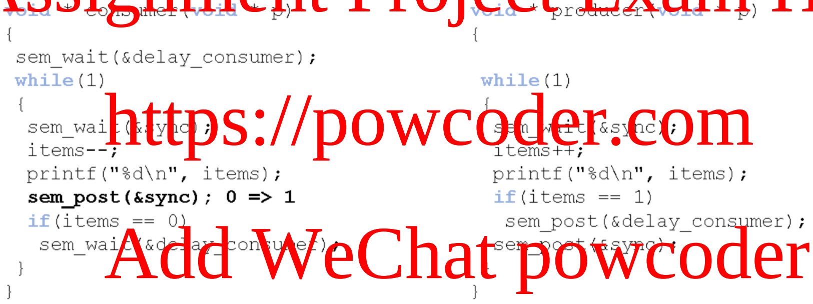
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



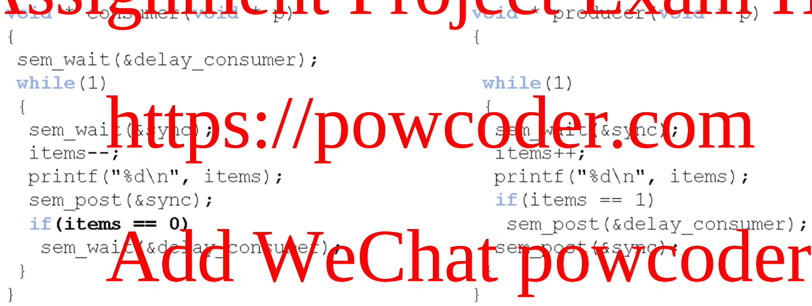
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync); 0 => 1
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer); 1 => 0
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

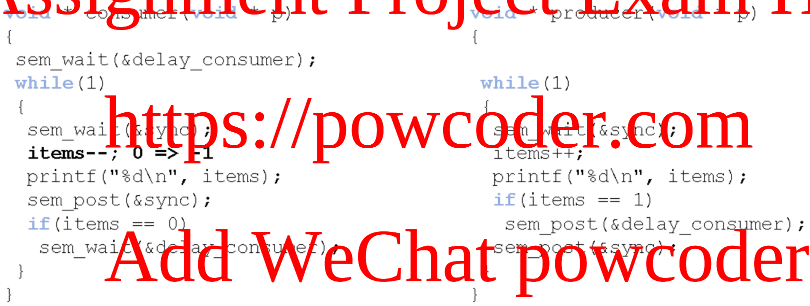
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync); 1 => 0
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

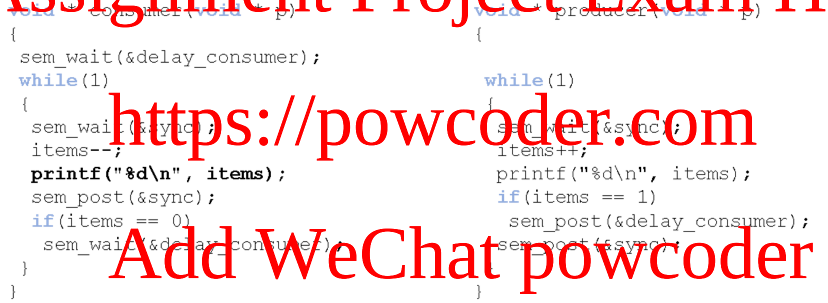


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--; 0 => -1
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



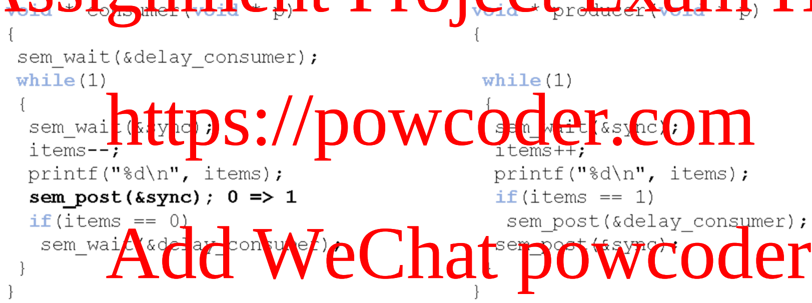
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



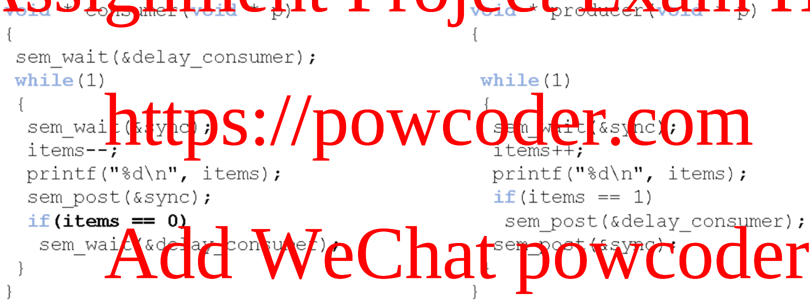
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync); 0 => 1
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items



```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  sem_post(&sync);
  if(items == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

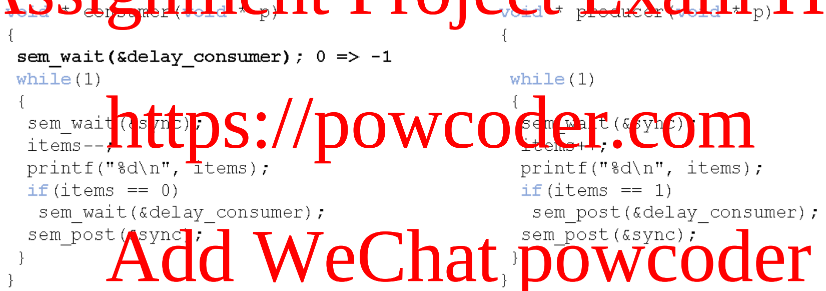
The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer

- It is obvious that any **manipulations of "items"** will have to be **synchronised**

- **Race conditions** still exist:
    - When the consumer has **exhausted the buffer**, should have gone to sleep, but the **producer increments** `items` **before the consumer checks** it

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks



```
void * consumer(void * p)
{
 sem_wait(&delay_consumer); 0 => -1
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync) : 1 => 0
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

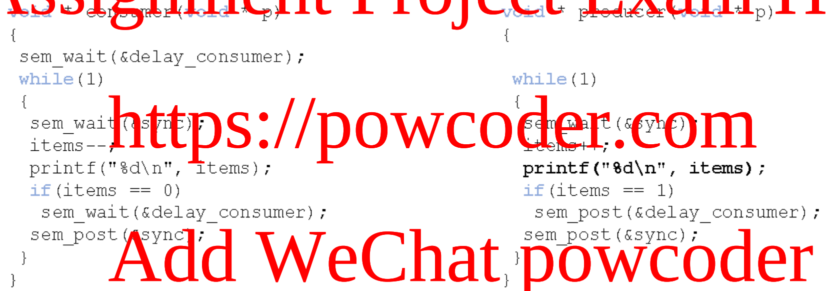
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++; 0 -> 1
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

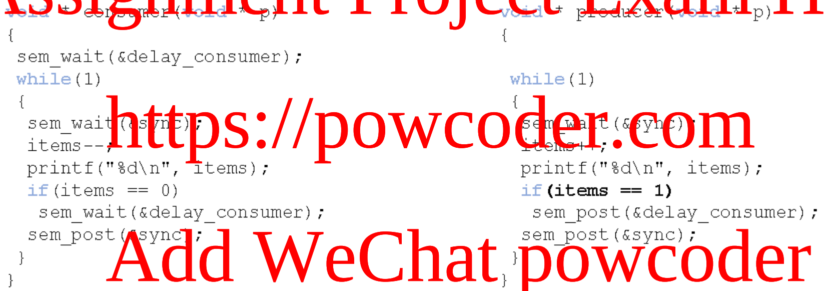


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Deadlocks



```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
 sem_wait(&delay_consumer); (wakeup)
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer); -1 => 0
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync); 0 => 1
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync); 1 => 0
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

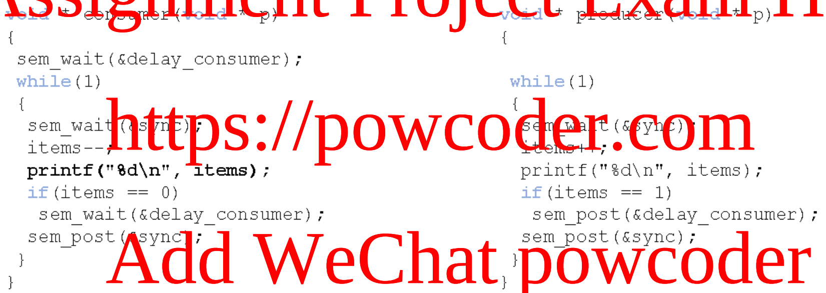
One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--; i --> 0
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: Deadlocks



```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

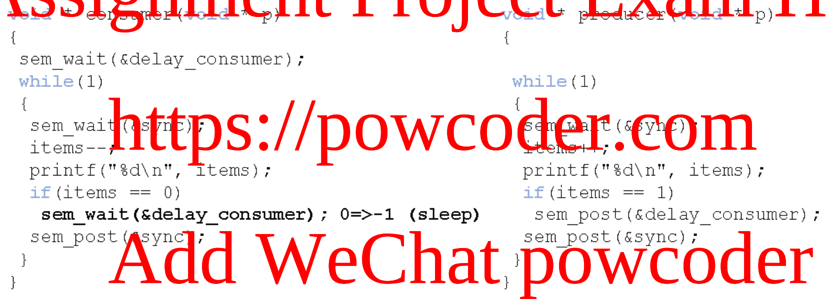
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks



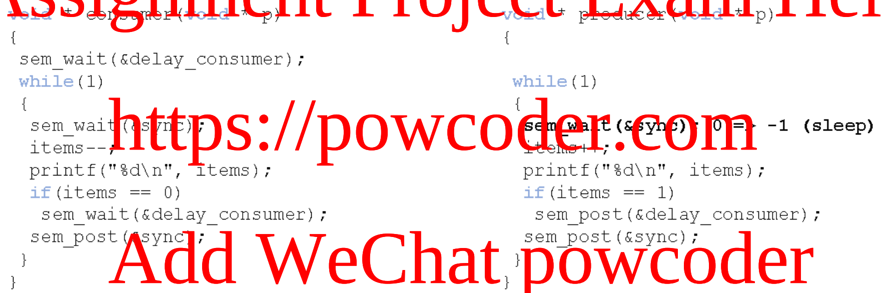
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer); 0=>-1 (sleep)
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks



```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  printf("%d\n", items);
  if(items == 0)
   sem_wait(&delay_consumer);
  sem_post(&sync);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync) : 0 => -1 (sleep)
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

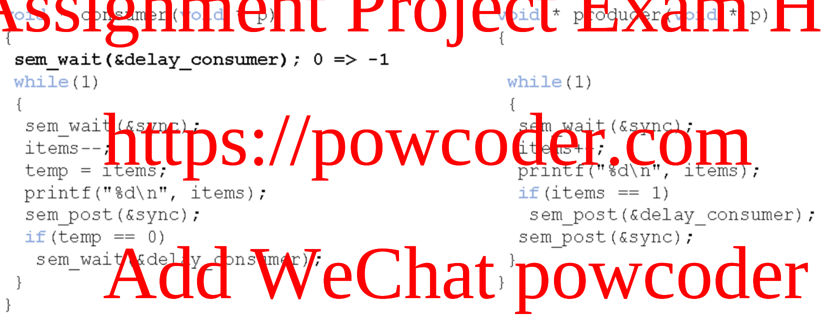
Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

- Use a **temporary variable**.
  - **Copies the value of items** inside the critical section
  - **Decrements** the `delay_consumer` semaphore to make it consistent

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution



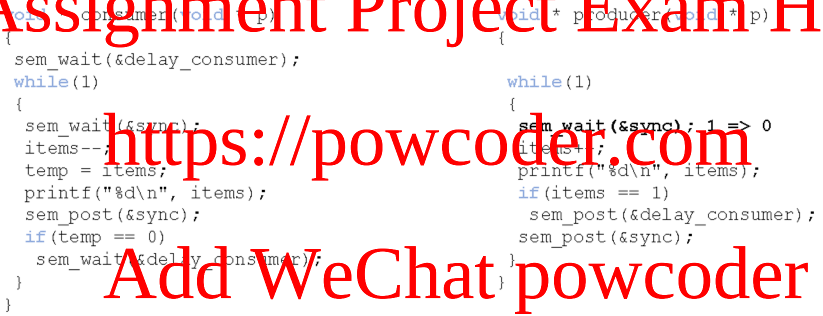
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer); 0 => -1
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



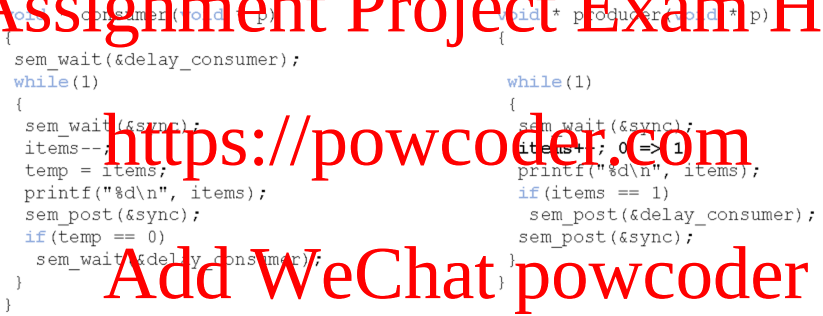
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync); 1 => 0
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

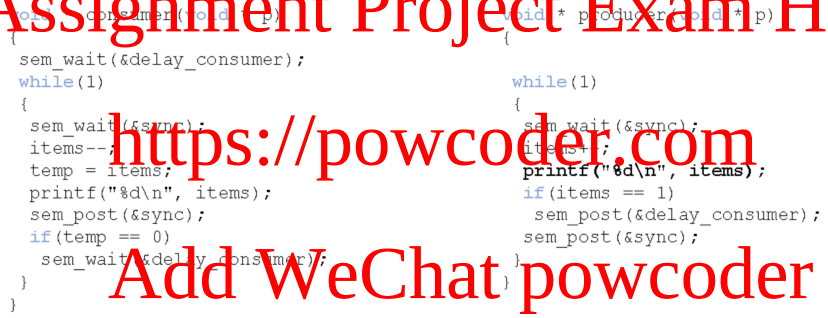


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++; 0 => 1
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution

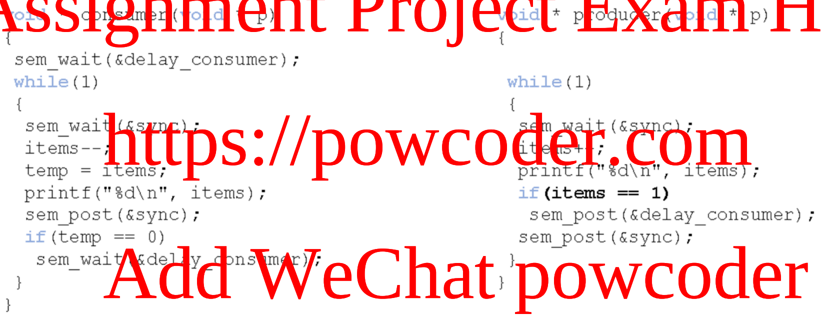


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution



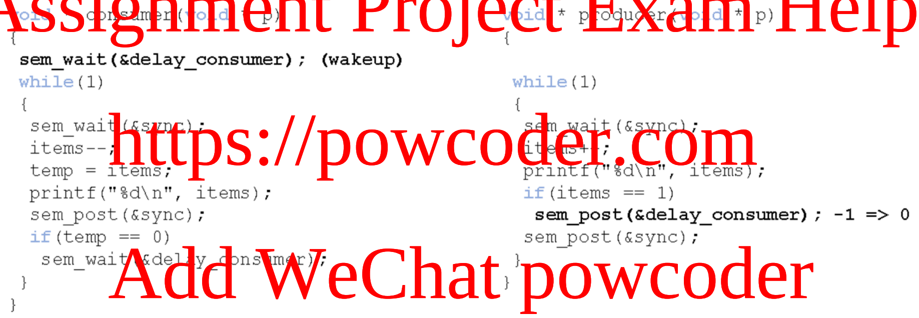
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

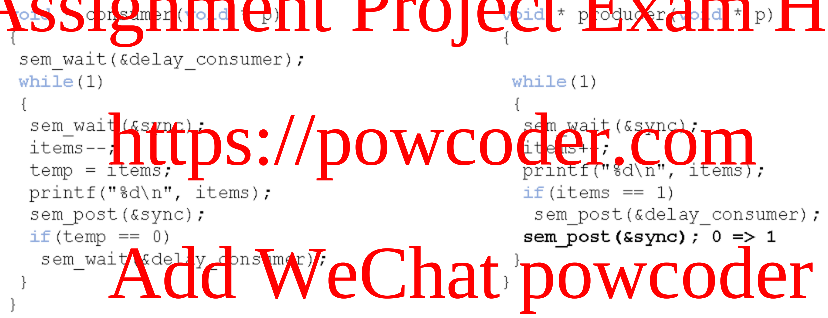


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer); (wakeup)
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer); -1 => 0
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution



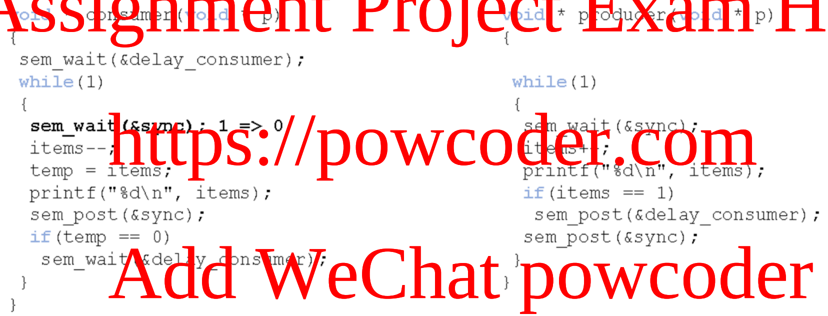
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync); 0 => 1
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



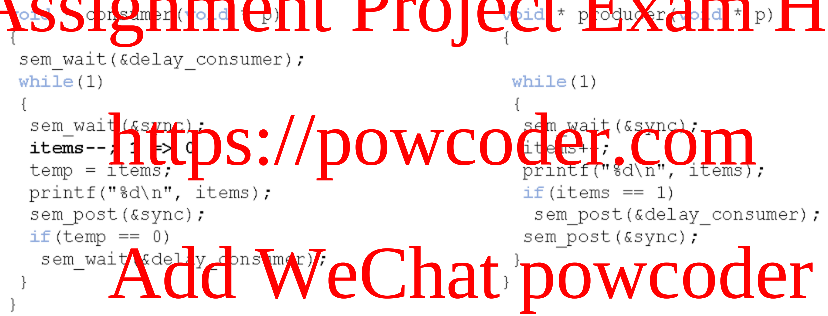
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync); 1 => 0
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



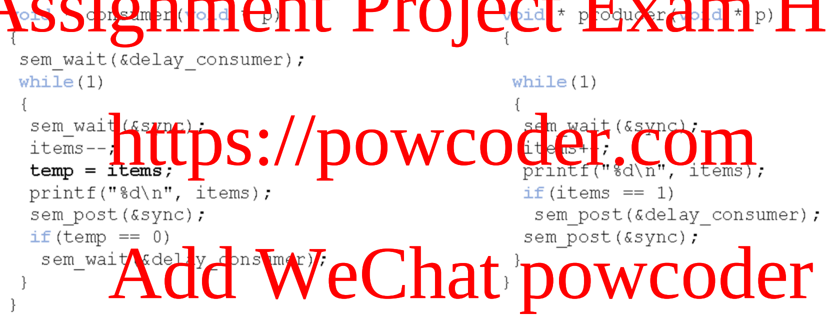
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--; 1 => 0
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



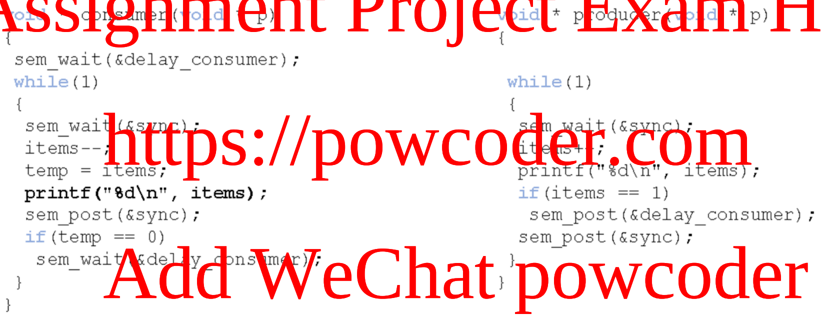
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



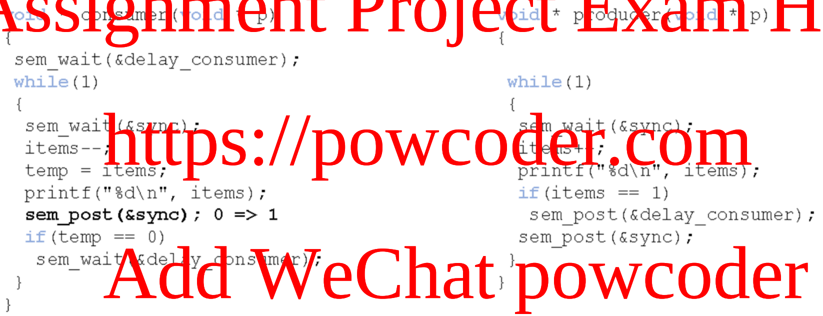
```
void* consumer(void *p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void* producer(void *p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

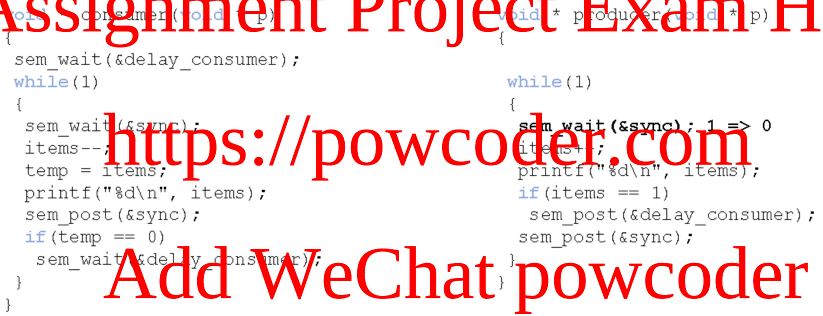


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync); 0 => 1
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution

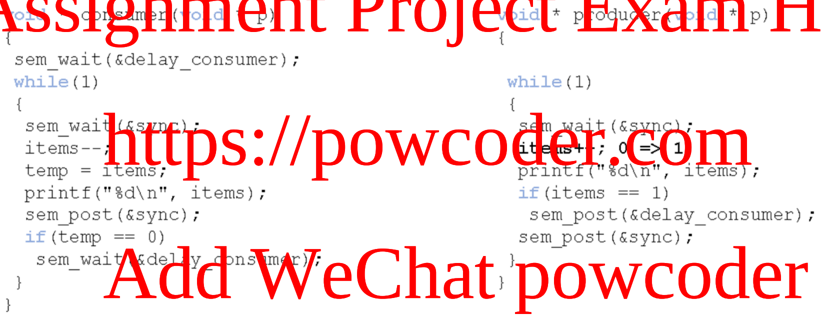

```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void* producer(void * p)
{
 while(1)
 {
  sem_wait(&sync); 1 => 0
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



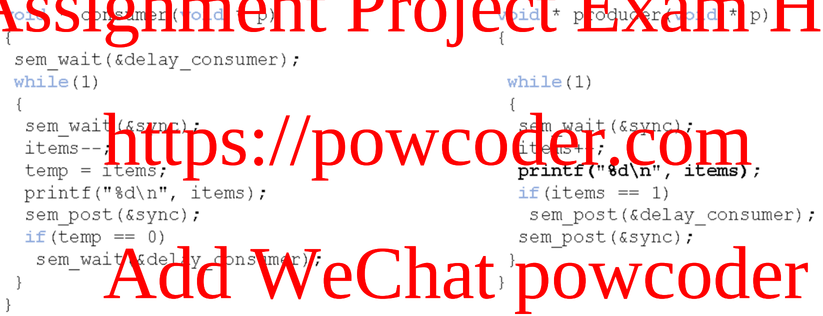
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++; 0 => 1
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



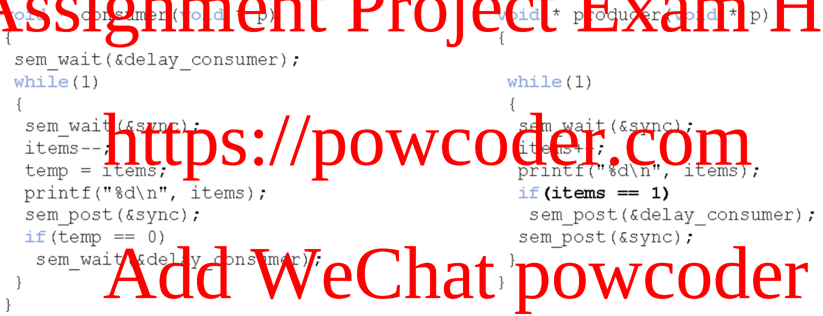
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

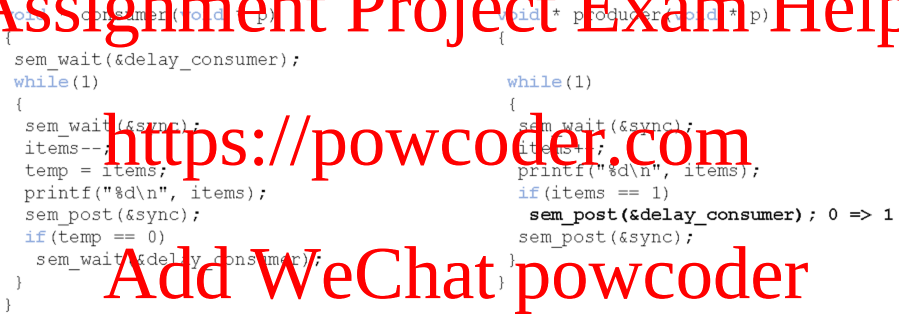


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void* producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution



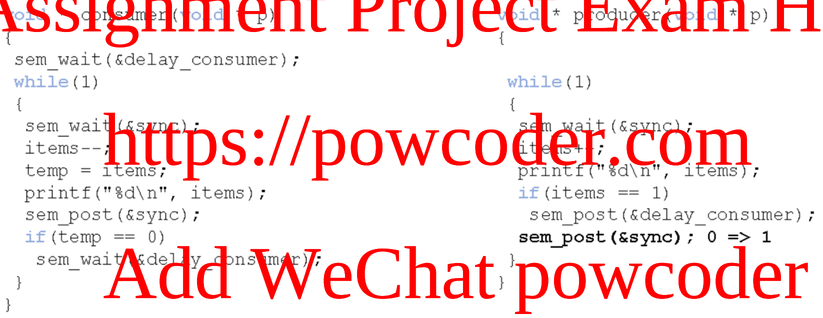
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void* producer(void *p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer); 0 => 1
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

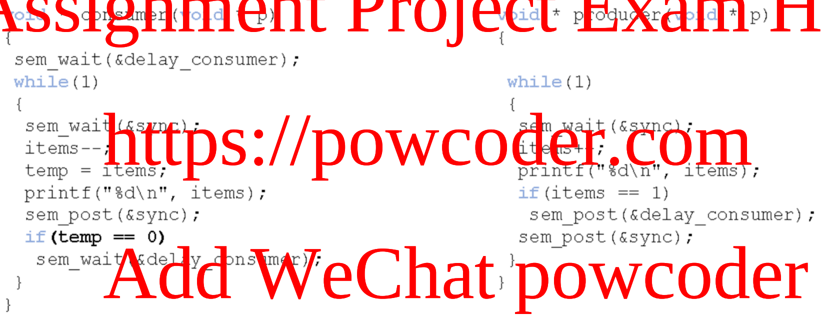


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync); 0 => 1
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution

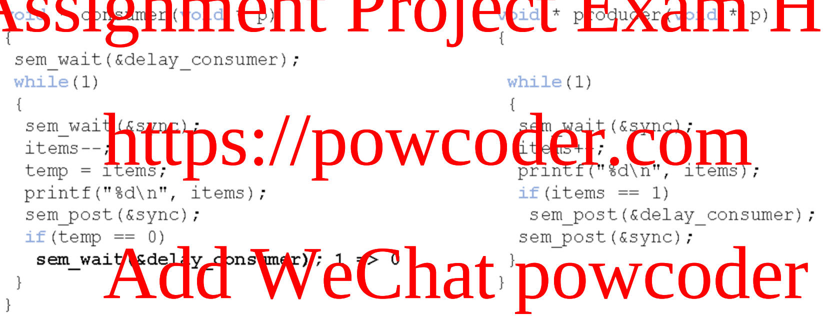


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution

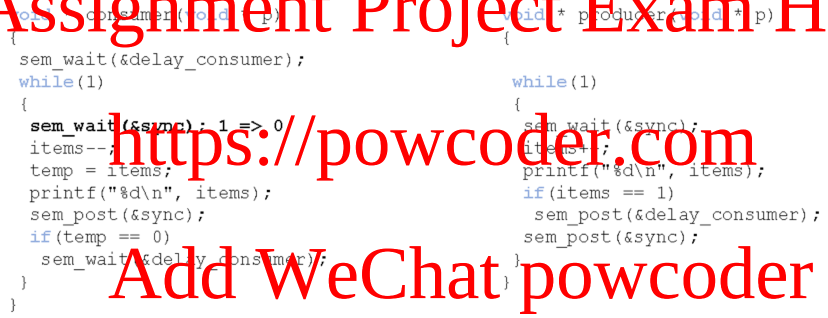


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer); 1 => 0
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution

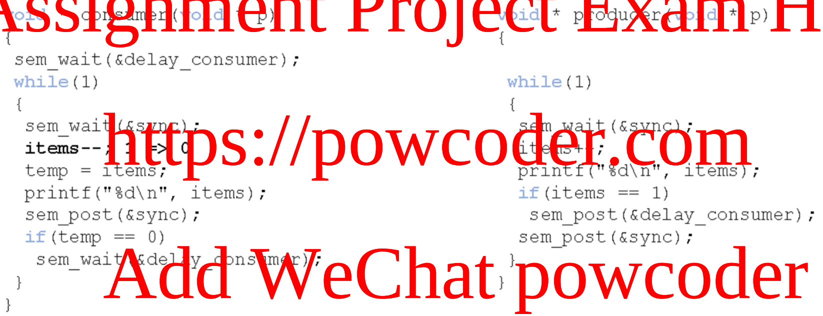


```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync); 1 => 0
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution



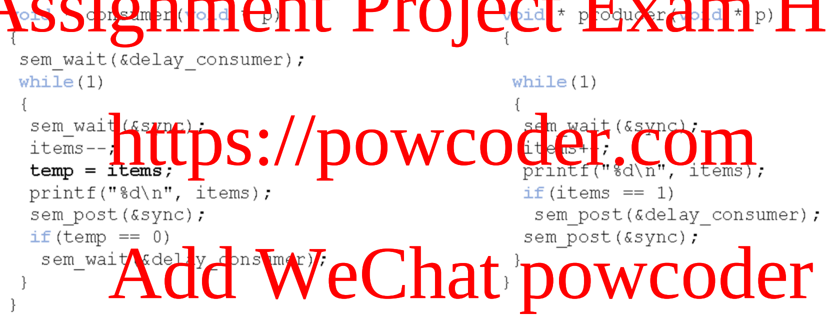
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--; 1 => 0
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{

 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



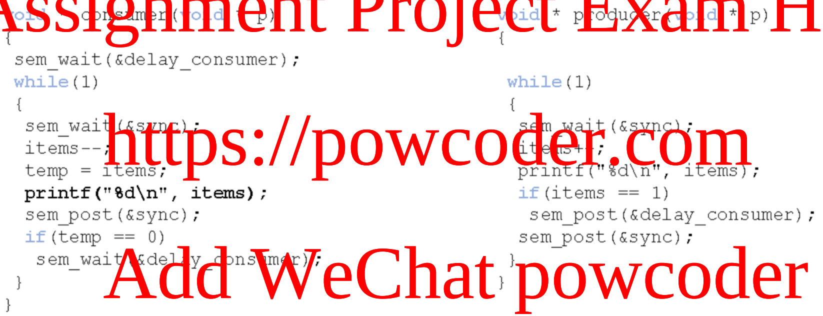
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



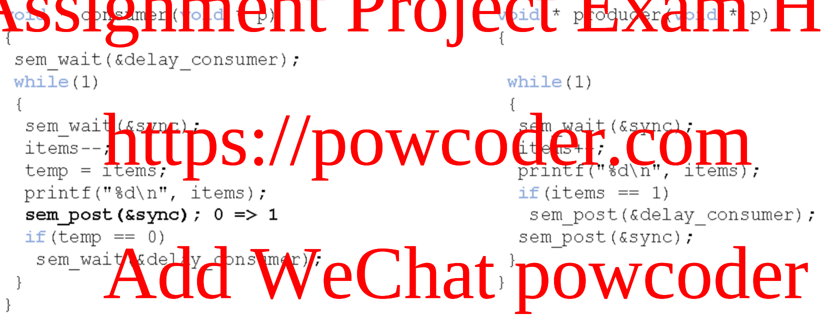
```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

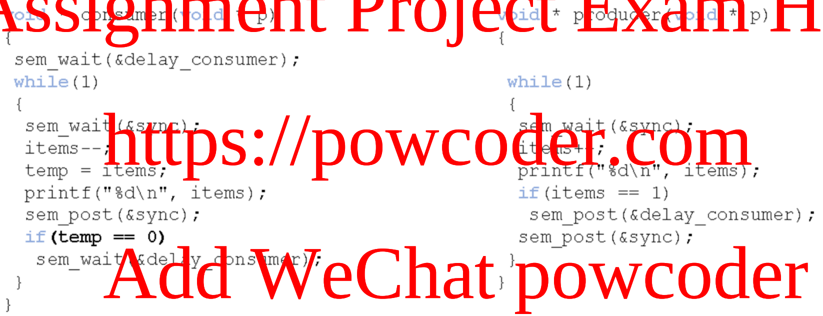


```
void * consumer(void * p)
{
  sem_wait(&delay_consumer);
  while(1)
  {
    sem_wait(&sync);
    items--;
    temp = items;
    printf("%d\n", items);
    sem_post(&sync); 0 => 1
    if(temp == 0)
      sem_wait(&delay_consumer);
  }
}
```

```
void * producer(void * p)
{

  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
      sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: solution



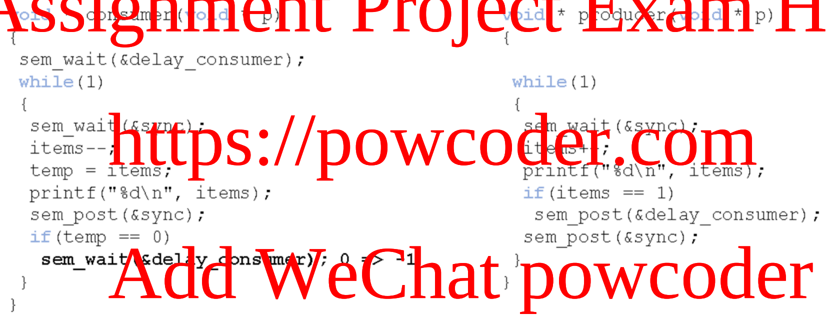
```
void* consumer(void *p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer);
 }
}
```

```
void* producer(void *p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution



```
void * consumer(void * p)
{
 sem_wait(&delay_consumer);
 while(1)
 {
  sem_wait(&sync);
  items--;
  temp = items;
  printf("%d\n", items);
  sem_post(&sync);
  if(temp == 0)
   sem_wait(&delay_consumer); 0 => -1
 }
}
```

```
void * producer(void * p)
{
 while(1)
 {
  sem_wait(&sync);
  items++;
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

Multiple Producers, Multiple Consumers, Bounded Buffer

- The previous code (one consumer, one producer) is made to work by **storing the value of** `items`
- A different variant of the problem has *n* **consumers**, *m* **producers**, and a **fixed buffer size** *N*. The solution is based on **3 semaphores**:
  - `sync`: used to **enforce mutual exclusion** for the buffer
  - `empty`: keeps track of the number of **empty buffers**, initialised to *N*
  - `full`: keeps track of the number of **full buffers**, initialised to 0
- The `empty` and `full` are **counting semaphores** and represent **resources**

The Producer/Consumer Problem

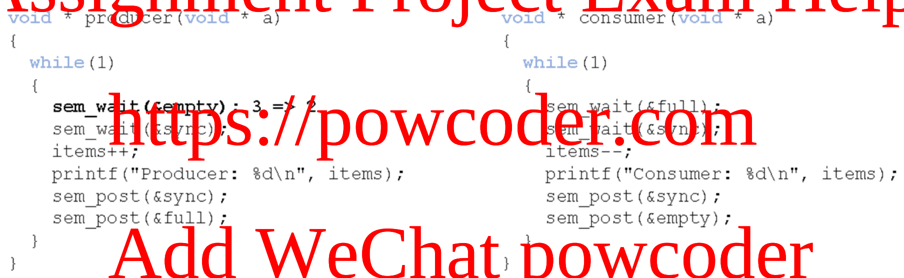
Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

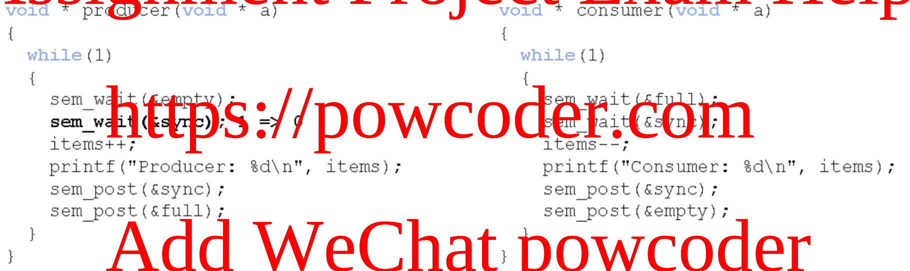
The Producer/Consumer Problem
Multiple Producers & Consumers



```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); 3 => 2
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

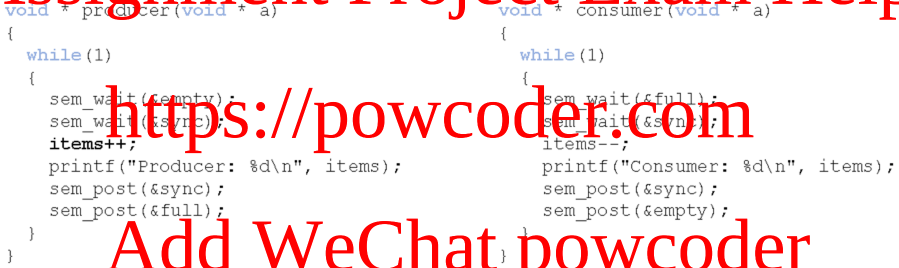


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync); 1 => 0
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers



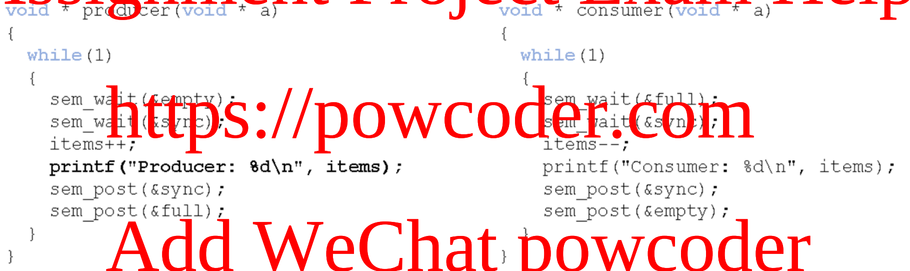
```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers



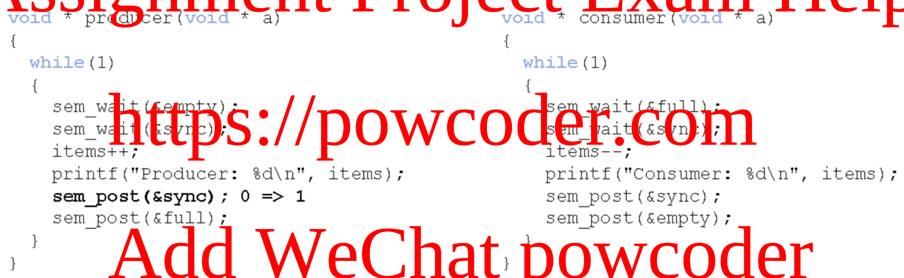
```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers



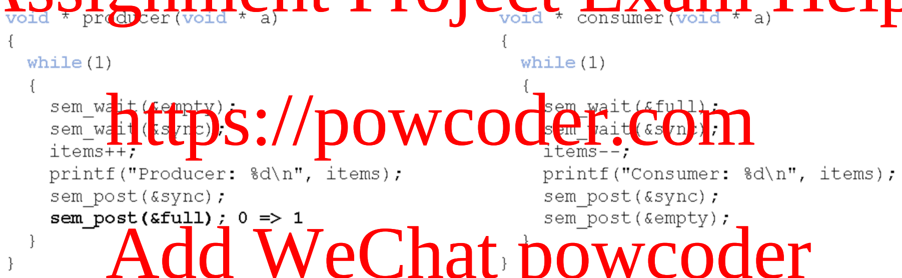
```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync); 0 => 1
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

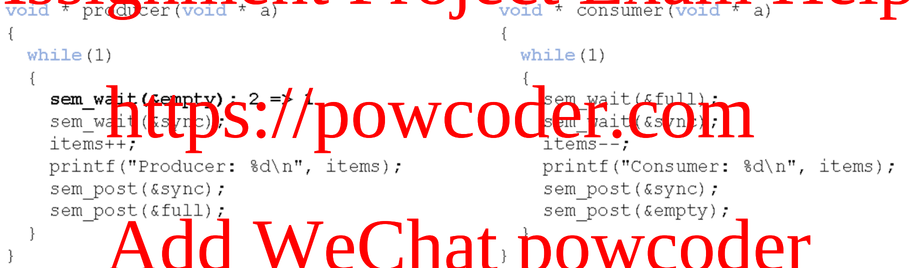


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full); 0 => 1
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers



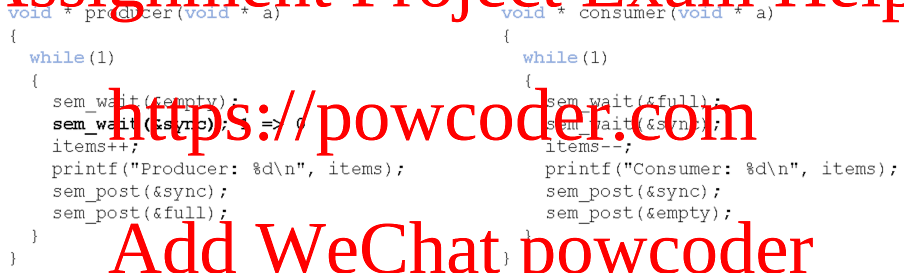
```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); 2 => 1
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

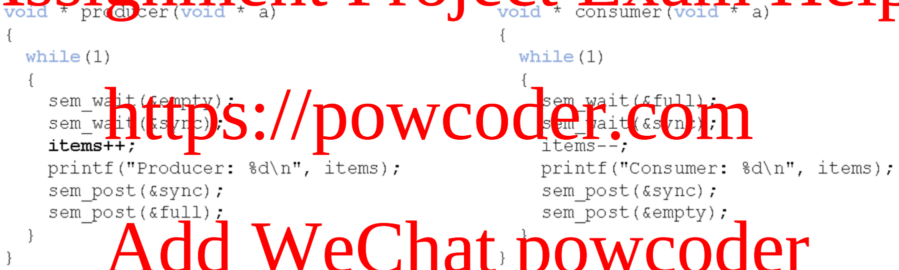


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync); 1 => 0
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers



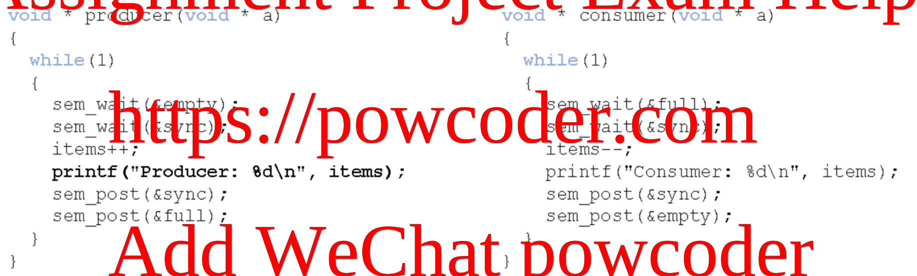
```c
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```c
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

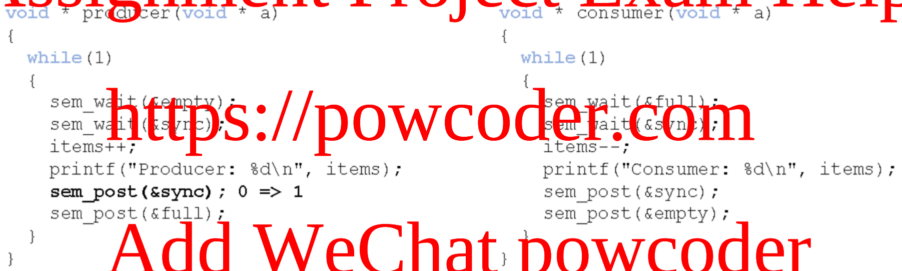


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers



```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync); 0 => 1
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

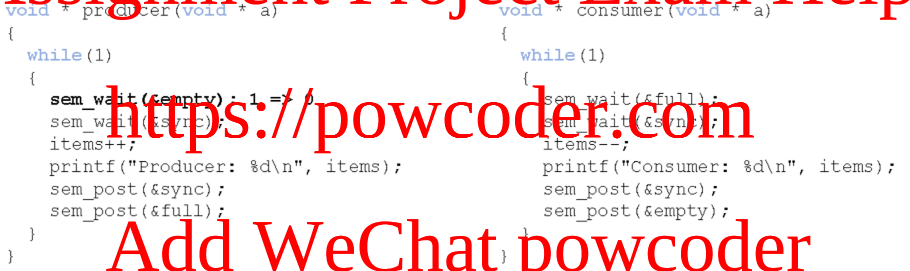
The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full); 1 => 2
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

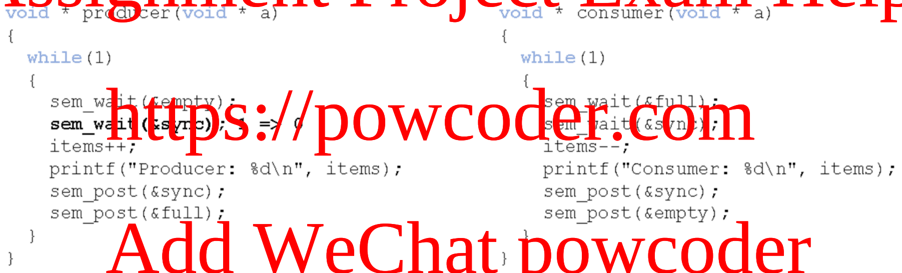


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); 1 => 0
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers

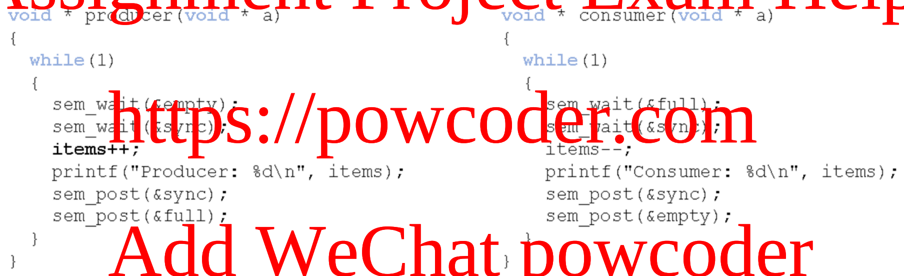


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync); 1 => 0
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers



```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

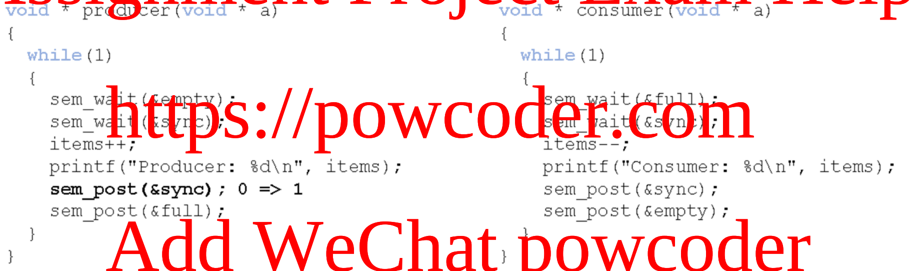
The Producer/Consumer Problem
Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers



```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync); 0 => 1
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers

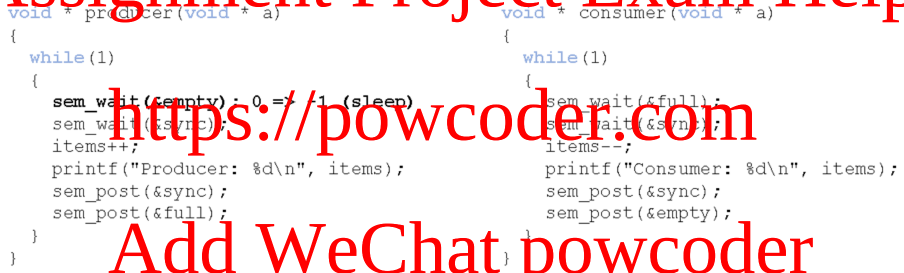
```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full); 2 => 3
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

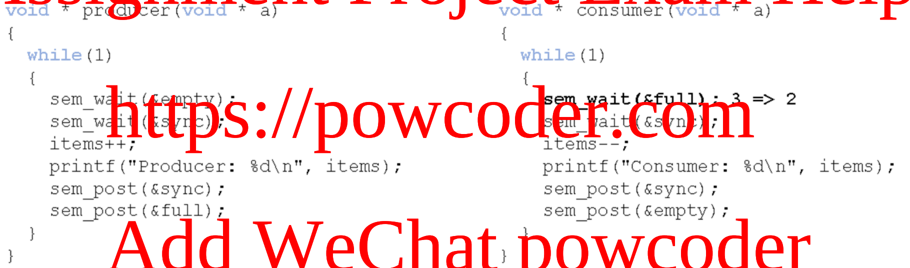


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty) : 0 => -1 (sleep)
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

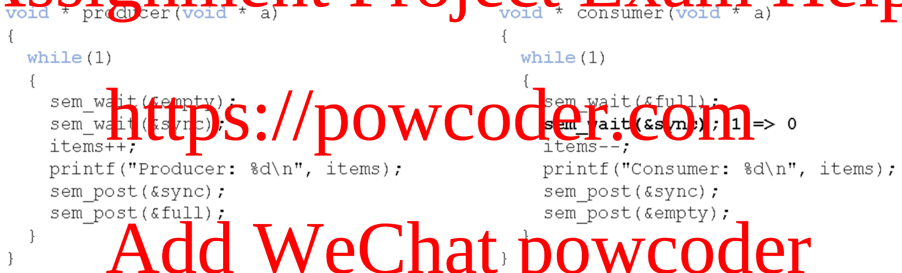
The Producer/Consumer Problem
Multiple Producers & Consumers



```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full); 3 => 2
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

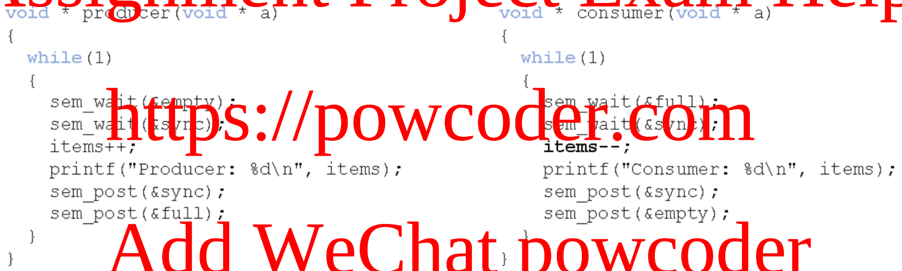


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync); 1 => 0
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers

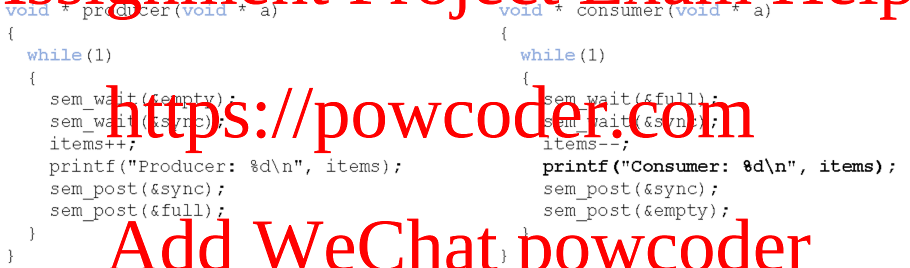


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers



```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

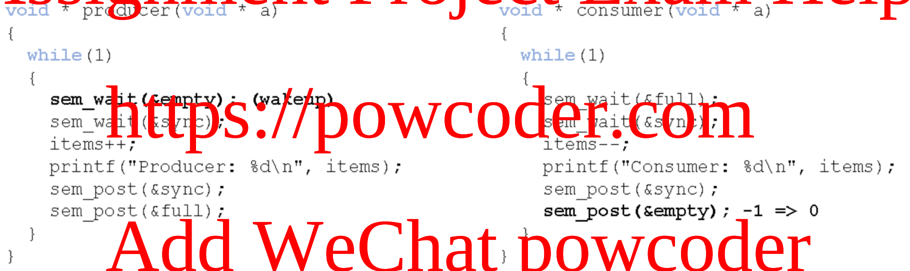
The Producer/Consumer Problem
Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync); 0 => 1
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers

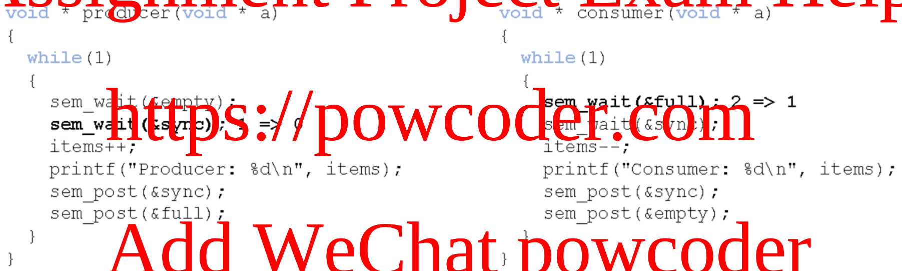


```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); (wakeup)
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty); -1 => 0
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem
Multiple Producers & Consumers



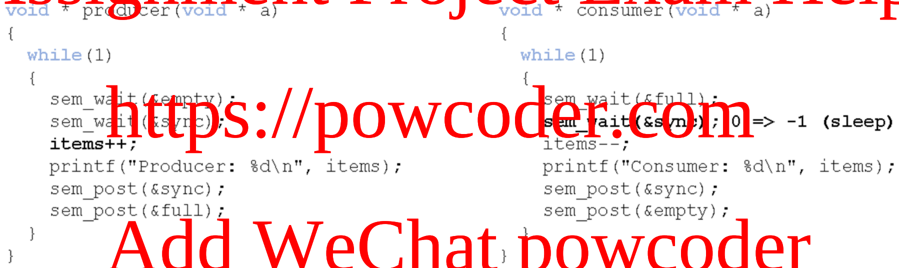
```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync); 1 => 0
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full); 2 => 1
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers



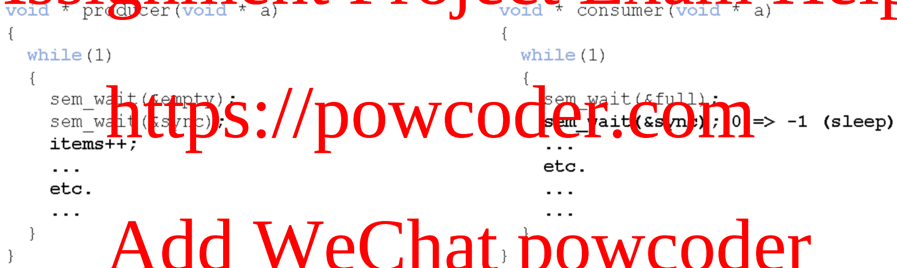
```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync); 0 => -1 (sleep)
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers



```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    ...
    etc.
    ...
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync); 0 => -1 (sleep)
    ...
    etc.
    ...
    ...
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Dining Philosophers Problem
Description

- The problem is defined as:
  - **Five philosophers** are sitting on a round table
  - Each one has one has **a plate** of spaghetti
  - The spaghetti is too slippery, and each philosopher **needs 2 forks** to be able to eat
  - When hungry (in between thinking), the philosopher tries to **acquire the forks on his left and right**
- Note that this reflects the general problem of **sharing a limited set** of resources (forks) between **a number of processes** (philosophers)

The Dining Philosophers Problem
Description

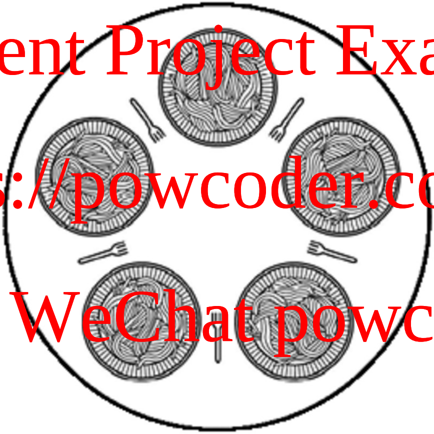


Figure: Tanenbaum, 4th edition

The Dining Philosophers Problem
Solution 1

- ***Forks*** are represented by **semaphores** (initialised to 1)
  - 1 if the **fork is available**: the philosopher can **continue**
  - 0 if the **fork is not available**: the philosopher goes to **sleep** if trying to acquire it
- First approach: Every philosopher **picks up one fork** and waits for the **second one to become available** (without putting the first one down)

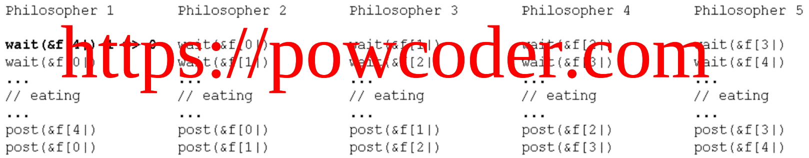
The Dining Philosophers Problem

Solution 1: Naive will Deadlock

```
#define N 5
sem_t forks[N];

void * philosopher(void * id)
{
        int i = *((int *) id);
        int left = (i + N - 1) % N;
        int right = i % N;
        while(1)
        {
                printf("%d is thinking\n", i);
                printf("%d is hungry\n", i);
                sem_wait(&forks[left]);
                sem_wait(&forks[right]);
                printf("%d is eating\n", i);
                sem_post(&forks[left]);
                sem_post(&forks[right]);
        }
}
```

The Dining Philosophers Problem

Solution 1: Illustration



```
Philosopher 1        Philosopher 2        Philosopher 3        Philosopher 4        Philosopher 5

wait(&f[4|) 1 -> 0   wait(&f[0|)          wait(&f[1|)          wait(&f[2|)          wait(&f[3|)
wait(&f[0|)          wait(&f[1|)          wait(&f[2|)          wait(&f[3|)          wait(&f[4|)
...                  ...                  ...                  ...                  ...
// eating            // eating            // eating            // eating            // eating
...                  ...                  ...                  ...                  ...
post(&f[4|)          post(&f[0|)          post(&f[1|)          post(&f[2|)          post(&f[3|)
post(&f[0|)          post(&f[1|)          post(&f[2|)          post(&f[3|)          post(&f[4|)
```

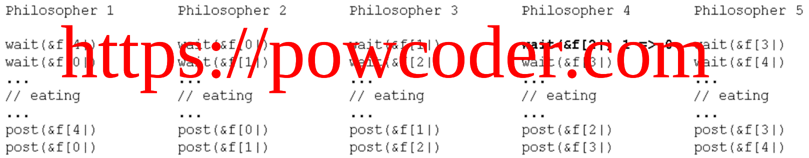
The Dining Philosophers Problem
Solution 1: Illustration

```
Philosopher 1      Philosopher 2          Philosopher 3      Philosopher 4      Philosopher 5

wait(&f[4|)        wait(&f[0|) 1 => 0     wait(&f[1|)        wait(&f[2|)        wait(&f[3|)
wait(&f[0|)        wait(&f[1|)            wait(&f[2|)        wait(&f[3|)        wait(&f[4|)
...                ...                    ...                ...                ...
// eating          // eating              // eating          // eating          // eating
...                ...                    ...                ...                ...
post(&f[4|)        post(&f[0|)            post(&f[1|)        post(&f[2|)        post(&f[3|)
post(&f[0|)        post(&f[1|)            post(&f[2|)        post(&f[3|)        post(&f[4|)
```

```
Philosopher 1     Philosopher 2     Philosopher 3       Philosopher 4     Philosopher 5

wait(&f[4|)       wait(&f[0|)       wait(&f[1|) 1 =>    wait(&f[2|)       wait(&f[3|)
wait(&f[0|)       wait(&f[1|)       wait(&f[2|)         wait(&f[3|)       wait(&f[4|)
...               ...               ...                 ...               ...
// eating         // eating         // eating           // eating         // eating
...               ...               ...                 ...               ...
post(&f[4|)       post(&f[0|)       post(&f[1|)         post(&f[2|)       post(&f[3|)
post(&f[0|)       post(&f[1|)       post(&f[2|)         post(&f[3|)       post(&f[4|)
```

The Dining Philosophers Problem

Solution 1: Illustration



```
Philosopher 1      Philosopher 2      Philosopher 3      Philosopher 4          Philosopher 5

wait(&f[4|)        wait(&f[0|)        wait(&f[1|)        wait(&f[2|) 1 => 0     wait(&f[3|)
wait(&f[0|)        wait(&f[1|)        wait(&f[2|)        wait(&f[3|)            wait(&f[4|)
...                ...                ...                ...                    ...
// eating          // eating          // eating          // eating              // eating
...                ...                ...                ...                    ...
post(&f[4|)        post(&f[0|)        post(&f[1|)        post(&f[2|)            post(&f[3|)
post(&f[0|)        post(&f[1|)        post(&f[2|)        post(&f[3|)            post(&f[4|)
```

The Dining Philosophers Problem
Solution 1: Illustration

```
Philosopher 1      Philosopher 2      Philosopher 3      Philosopher 4      Philosopher 5

wait(&f[4|)        wait(&f[0|)        wait(&f[1|)        wait(&f[2|)        wait(&f[3|) 1 => 0
wait(&f[0|)        wait(&f[1|)        wait(&f[2|)        wait(&f[3|)        wait(&f[4|)
...                ...                ...                ...                ...
// eating          // eating          // eating          // eating          // eating
...                ...                ...                ...                ...
post(&f[4|)        post(&f[0|)        post(&f[1|)        post(&f[2|)        post(&f[3|)
post(&f[0|)        post(&f[1|)        post(&f[2|)        post(&f[3|)        post(&f[4|)
```

The Dining Philosophers Problem

Solution 1: Illustration

```
Philosopher 1       Philosopher 2       Philosopher 3       Philosopher 4       Philosopher 5

wait(&f[4|)         wait(&f[0|)         wait(&f[1|)         wait(&f[2|)         wait(&f[3|)
wait(&f[0|) 0 => -1 wait(&f[1|)         wait(&f[2|)         wait(&f[3|)         wait(&f[4|)
...                 ...                 ...                 ...                 ...
// eating           // eating           // eating           // eating           // eating
...                 ...                 ...                 ...                 ...
post(&f[4|)         post(&f[0|)         post(&f[1|)         post(&f[2|)         post(&f[3|)
post(&f[0|)         post(&f[1|)         post(&f[2|)         post(&f[3|)         post(&f[4|)
```

The Dining Philosophers Problem

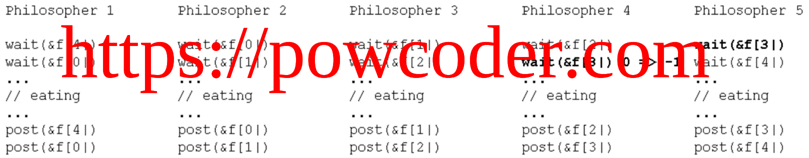
Solution 1: Illustration

```
Philosopher 1      Philosopher 2        Philosopher 3     Philosopher 4     Philosopher 5

wait(&f[4|)        wait(&f[0|)          wait(&f[1|)       wait(&f[2|)       wait(&f[3|)
wait(&f[0|)        wait(&f[1|) 0 => -1  wait(&f[2|)       wait(&f[3|)       wait(&f[4|)
...                ...                  ...               ...               ...
// eating          // eating            // eating         // eating         // eating
...                ...                  ...               ...               ...
post(&f[4|)        post(&f[0|)          post(&f[1|)       post(&f[2|)       post(&f[3|)
post(&f[0|)        post(&f[1|)          post(&f[2|)       post(&f[3|)       post(&f[4|)
```

The Dining Philosophers Problem
Solution 1: Illustration

```
Philosopher 1     Philosopher 2     Philosopher 3         Philosopher 4     Philosopher 5

wait(&f[4|)       wait(&f[0|)       wait(&f[1|)           wait(&f[2|)       wait(&f[3|)
wait(&f[0|)       wait(&f[1|)       wait(&f[2|  0 => -1   wait(&f[3|)       wait(&f[4|)
...               ...               ...                   ...               ...
// eating         // eating         // eating             // eating         // eating
...               ...               ...                   ...               ...
post(&f[4|)       post(&f[0|)       post(&f[1|)           post(&f[2|)       post(&f[3|)
post(&f[0|)       post(&f[1|)       post(&f[2|)           post(&f[3|)       post(&f[4|)
```

The Dining Philosophers Problem
Solution 1: Illustration



```
Philosopher 1      Philosopher 2      Philosopher 3      Philosopher 4              Philosopher 5

wait(&f[4|)        wait(&f[0|)        wait(&f[1|)        wait(&f[2|)                wait(&f[3|)
wait(&f[0|)        wait(&f[1|)        wait(&f[2|)        wait(&f[3|) 0 => -1        wait(&f[4|)
...                ...                ...                ...                        ...
// eating          // eating          // eating          // eating                  // eating
...                ...                ...                ...                        ...
post(&f[4|)        post(&f[0|)        post(&f[1|)        post(&f[2|)                post(&f[3|)
post(&f[0|)        post(&f[1|)        post(&f[2|)        post(&f[3|)                post(&f[4|)
```

The Dining Philosophers Problem
Solution 1: Illustration

```
Philosopher 1       Philosopher 2       Philosopher 3       Philosopher 4       Philosopher 5

wait(&f[4|)         wait(&f[0|)         wait(&f[1|)         wait(&f[2|)         wait(&f[3|)
wait(&f[0|)         wait(&f[1|)         wait(&f[2|)         wait(&f[3|)         wait(&f[4|) 0 => -1
...                 ...                 ...                 ...                 ...
// eating           // eating           // eating           // eating           // eating
...                 ...                 ...                 ...                 ...
post(&f[4|)         post(&f[0|)         post(&f[1|)         post(&f[2|)         post(&f[3|)
post(&f[0|)         post(&f[1|)         post(&f[2|)         post(&f[3|)         post(&f[4|)
```

The Dining Philosophers Problem
Solution 1: Deadlock

- The naive solution can **deadlock**
- Deadlocks can be **prevented by**:
  - Putting the forks down and **waiting a random time** (Ethernet networks)
  - Putting **one additional fork** on the table
  - One **global mutex/lock** set by a philosopher when (s)he wants to eat (only one can eat at a time)
    - Solution does **not result in maximum parallelism** (only one eats at a time)

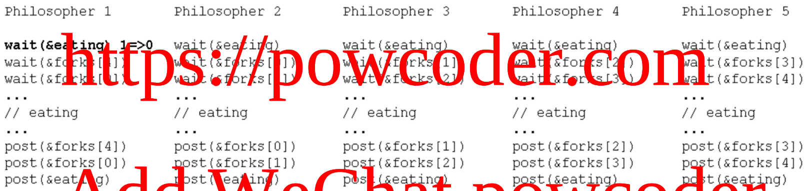
The Dining Philosophers Problem

Solutions 2: Global Mutex/Semaphore

```
sem_t eating;

void * philosopher(void * id)
{
  int i = (int) id;
  int left = (i + N - 1) % N;
  int right = i % N;
  while(1)
  {
    printf("%d is thinking\n", i);
    printf("%d is hungry\n", i);
    sem_wait(&eating);      /**** mutex/semaphore ****/
    sem_wait(&forks[left]);
    sem_wait(&forks[right]);
    printf("%d is eating\n", i);
    sem_post(&forks[left]);
    sem_post(&forks[right]);
    sem_post(&eating);      /**** mutex/semaphore ****/
  }
}
```

The Dining Philosophers Problem

Solutions 2: Illustration



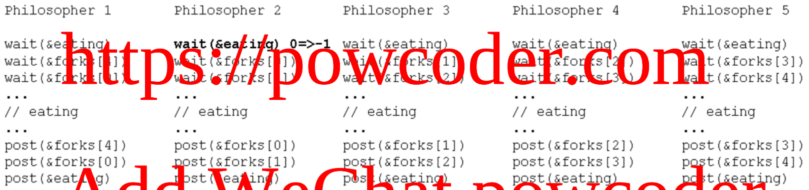
```
Philosopher 1         Philosopher 2         Philosopher 3         Philosopher 4         Philosopher 5

wait(&eating) 1=>0    wait(&eating)         wait(&eating)         wait(&eating)         wait(&eating)
wait(&forks[4])       wait(&forks[0])       wait(&forks[1])       wait(&forks[2])       wait(&forks[3])
wait(&forks[0])       wait(&forks[1])       wait(&forks[2])       wait(&forks[3])       wait(&forks[4])
...                   ...                   ...                   ...                   ...
// eating             // eating             // eating             // eating             // eating
...                   ...                   ...                   ...                   ...
post(&forks[4])       post(&forks[0])       post(&forks[1])       post(&forks[2])       post(&forks[3])
post(&forks[0])       post(&forks[1])       post(&forks[2])       post(&forks[3])       post(&forks[4])
post(&eating)         post(&eating)         post(&eating)         post(&eating)         post(&eating)
```

The Dining Philosophers Problem
Solutions 2: Illustration

```
Philosopher 1        Philosopher 2        Philosopher 3        Philosopher 4        Philosopher 5

wait(&eating)        wait(&eating)        wait(&eating)        wait(&eating)        wait(&eating)
wait(&forks[4]) 1=>0 wait(&forks[0])      wait(&forks[1])      wait(&forks[2])      wait(&forks[3])
wait(&forks[0])      wait(&forks[1])      wait(&forks[2])      wait(&forks[3])      wait(&forks[4])
...                  ...                  ...                  ...                  ...
// eating            // eating            // eating            // eating            // eating
...                  ...                  ...                  ...                  ...
post(&forks[4])      post(&forks[0])      post(&forks[1])      post(&forks[2])      post(&forks[3])
post(&forks[0])      post(&forks[1])      post(&forks[2])      post(&forks[3])      post(&forks[4])
post(&eating)        post(&eating)        post(&eating)        post(&eating)        post(&eating)
```

```
Philosopher 1         Philosopher 2        Philosopher 3        Philosopher 4        Philosopher 5

wait(&eating)         wait(&eating)        wait(&eating)        wait(&eating)        wait(&eating)
wait(&forks[4])       wait(&forks[0])      wait(&forks[1])      wait(&forks[2])      wait(&forks[3])
wait(&forks[0]) 1=>(  wait(&forks[1])      wait(&forks[2])      wait(&forks[3])      wait(&forks[4])
...                   ...                  ...                  ...                  ...
// eating             // eating            // eating            // eating            // eating
...                   ...                  ...                  ...                  ...
post(&forks[4])       post(&forks[0])      post(&forks[1])      post(&forks[2])      post(&forks[3])
post(&forks[0])       post(&forks[1])      post(&forks[2])      post(&forks[3])      post(&forks[4])
post(&eating)         post(&eating)        post(&eating)        post(&eating)        post(&eating)
```

The Dining Philosophers Problem
Solutions 2: Illustration



```
Philosopher 1       Philosopher 2           Philosopher 3       Philosopher 4       Philosopher 5

wait(&eating)       wait(&eating) 0=>-1     wait(&eating)       wait(&eating)       wait(&eating)
wait(&forks[4])     wait(&forks[0])         wait(&forks[1])     wait(&forks[2])     wait(&forks[3])
wait(&forks[0])     wait(&forks[1])         wait(&forks[2])     wait(&forks[3])     wait(&forks[4])
...                 ...                     ...                 ...                 ...
// eating           // eating               // eating           // eating           // eating
...                 ...                     ...                 ...                 ...
post(&forks[4])     post(&forks[0])         post(&forks[1])     post(&forks[2])     post(&forks[3])
post(&forks[0])     post(&forks[1])         post(&forks[2])     post(&forks[3])     post(&forks[4])
post(&eating)       post(&eating)           post(&eating)       post(&eating)       post(&eating)
```

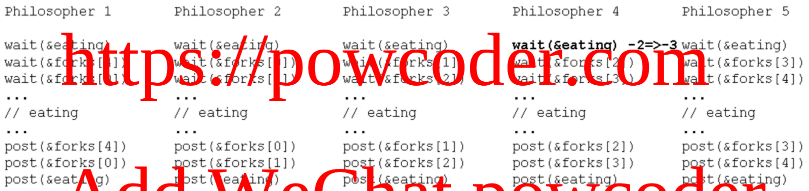
The Dining Philosophers Problem

Solutions 2: Illustration

```
Philosopher 1       Philosopher 2       Philosopher 3            Philosopher 4       Philosopher 5

wait(&eating)       wait(&eating)       wait(&eating) -1=> 2     wait(&eating)       wait(&eating)
wait(&forks[4])     wait(&forks[0])     wait(&forks[1])          wait(&forks[2])     wait(&forks[3])
wait(&forks[0])     wait(&forks[1])     wait(&forks[2])          wait(&forks[3])     wait(&forks[4])
...                 ...                 ...                      ...                 ...
// eating           // eating           // eating                // eating           // eating
...                 ...                 ...                      ...                 ...
post(&forks[4])     post(&forks[0])     post(&forks[1])          post(&forks[2])     post(&forks[3])
post(&forks[0])     post(&forks[1])     post(&forks[2])          post(&forks[3])     post(&forks[4])
post(&eating)       post(&eating)       post(&eating)            post(&eating)       post(&eating)
```

The Dining Philosophers Problem

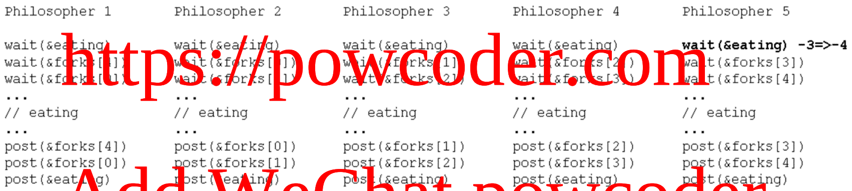
Solutions 2: Illustration



```
Philosopher 1       Philosopher 2       Philosopher 3       Philosopher 4            Philosopher 5

wait(&eating)       wait(&eating)       wait(&eating)       wait(&eating) -2=>-3 wait(&eating)
wait(&forks[4])     wait(&forks[0])     wait(&forks[1])     wait(&forks[2])      wait(&forks[3])
wait(&forks[0])     wait(&forks[1])     wait(&forks[2])     wait(&forks[3])      wait(&forks[4])
...                 ...                 ...                 ...                  ...
// eating           // eating           // eating           // eating            // eating
...                 ...                 ...                 ...                  ...
post(&forks[4])     post(&forks[0])     post(&forks[1])     post(&forks[2])      post(&forks[3])
post(&forks[0])     post(&forks[1])     post(&forks[2])     post(&forks[3])      post(&forks[4])
post(&eating)       post(&eating)       post(&eating)       post(&eating)        post(&eating)
```

The Dining Philosophers Problem

Solutions 2: Illustration



```
Philosopher 1        Philosopher 2        Philosopher 3        Philosopher 4        Philosopher 5

wait(&eating)        wait(&eating)        wait(&eating)        wait(&eating)        wait(&eating) -3=>-4
wait(&forks[4])      wait(&forks[0])      wait(&forks[1])      wait(&forks[2])      wait(&forks[3])
wait(&forks[0])      wait(&forks[1])      wait(&forks[2])      wait(&forks[3])      wait(&forks[4])
...                  ...                  ...                  ...                  ...
// eating            // eating            // eating            // eating            // eating
...                  ...                  ...                  ...                  ...
post(&forks[4])      post(&forks[0])      post(&forks[1])      post(&forks[2])      post(&forks[3])
post(&forks[0])      post(&forks[1])      post(&forks[2])      post(&forks[3])      post(&forks[4])
post(&eating)        post(&eating)        post(&eating)        post(&eating)        post(&eating)
```

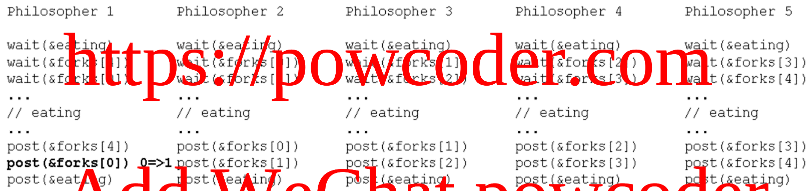
The Dining Philosophers Problem

Solutions 2: Illustration

```
Philosopher 1        Philosopher 2       Philosopher 3       Philosopher 4       Philosopher 5

wait(&eating)        wait(&eating)       wait(&eating)       wait(&eating)       wait(&eating)
wait(&forks[4])      wait(&forks[0])     wait(&forks[1])     wait(&forks[2])     wait(&forks[3])
wait(&forks[0])      wait(&forks[1])     wait(&forks[2])     wait(&forks[3])     wait(&forks[4])
...                  ...                 ...                 ...                 ...
// eating            // eating           // eating           // eating           // eating
...                  ...                 ...                 ...                 ...
post(&forks[4])  0=>1 post(&forks[0])    post(&forks[1])     post(&forks[2])     post(&forks[3])
post(&forks[0])      post(&forks[1])     post(&forks[2])     post(&forks[3])     post(&forks[4])
post(&eating)        post(&eating)       post(&eating)       post(&eating)       post(&eating)
```

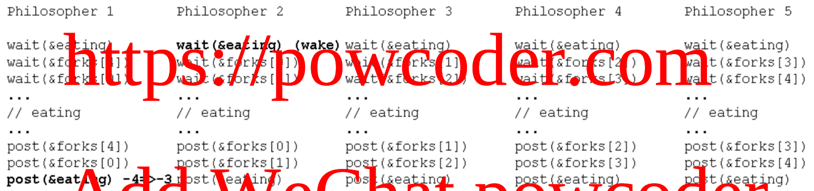
The Dining Philosophers Problem

Solutions 2: Illustration



```
Philosopher 1          Philosopher 2          Philosopher 3          Philosopher 4          Philosopher 5

wait(&eating)          wait(&eating)          wait(&eating)          wait(&eating)          wait(&eating)
wait(&forks[4])        wait(&forks[0])        wait(&forks[1])        wait(&forks[2])        wait(&forks[3])
wait(&forks[0])        wait(&forks[1])        wait(&forks[2])        wait(&forks[3])        wait(&forks[4])
...                    ...                    ...                    ...                    ...
// eating              // eating              // eating              // eating              // eating
...                    ...                    ...                    ...                    ...
post(&forks[4])        post(&forks[0])        post(&forks[1])        post(&forks[2])        post(&forks[3])
post(&forks[0])  0=>1  post(&forks[1])        post(&forks[2])        post(&forks[3])        post(&forks[4])
post(&eating)          post(&eating)          post(&eating)          post(&eating)          post(&eating)
```

The Dining Philosophers Problem
Solutions 2: Illustration



```
Philosopher 1        Philosopher 2              Philosopher 3        Philosopher 4        Philosopher 5

wait(&eating)        wait(&eating)  (wake)      wait(&eating)        wait(&eating)        wait(&eating)
wait(&forks[4])      wait(&forks[0])            wait(&forks[1])      wait(&forks[2])      wait(&forks[3])
wait(&forks[0])      wait(&forks[1])            wait(&forks[2])      wait(&forks[3])      wait(&forks[4])
...                  ...                        ...                  ...                  ...
// eating            // eating                  // eating            // eating            // eating
...                  ...                        ...                  ...                  ...
post(&forks[4])      post(&forks[0])            post(&forks[1])      post(&forks[2])      post(&forks[3])
post(&forks[0])      post(&forks[1])            post(&forks[2])      post(&forks[3])      post(&forks[4])
post(&eating) -4=>-3 post(&eating)              post(&eating)        post(&eating)        post(&eating)
```

- A number of "**real world synchronisation problems**'' (or with a similar structure to real world problems)
- Problems with the **solutions** to them (deadlocks, etc.)
- **Solutions** using semaphores/mutexes