University of Nottingham
UK | CHINA | MALAYSIA

Operating Systems and Concurrency

Lecture 10&11: Concurrency

University of Nottingham, Ningbo China, 2024

Recap
Last Lecture

- Concurrency using semaphores and mutexes
  - Mutex is a spinlock (busy waiting)
  - Semaphore puts the process to sleep
- Practical examples of how to use (code) semaphores
- Solve producer consumer problem using semaphore

Today class

- The Consumer-producer problem
  - Scenarios
  - Solutions

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

- It is obvious that any manipulations of “item” will have to be synchronized.

- 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.

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	1	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	1	1
Enter_CS	0	0	1

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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer


```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	1	1
Enter_CS	0	0	1
	0	0	0

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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer | | | | | | | | | | ....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	1	1
Enter_CS	0	0	1
	0	0	0
	0	0	0

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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

| | | | | | | | | | ....|

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	1	1
Enter_CS	0	0	1
	0	0	0
	0	0	0
Exit_CS	0	1	0

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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer | | | | | | | | | | ....

Consumer interrupted before checking item and then goes to sleep

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

Action	delay_cons=0	Syn=1	Item=0
C_blocked	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	1	1
Enter_CS	0	0	1
	0	0	0
	0	0	0
Exit_CS	0	1	0
Enter_CS	0	0	0

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1

```c
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);
  }
}
```

```c
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1
	1	0	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1
	1	0	1
Exit_CS	1	1	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1
	1	0	1
Exit_CS	1	1	1
	1	1	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1
	1	0	1
Exit_CS	1	1	1
	1	1	1
Enter_CS	1	0	1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

| | | | | | | | | | ....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1
	1	0	1
Exit_CS	1	1	1
	1	1	1
Enter_CS	1	0	1
	1	0	0

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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

| | | | | | | | | | ....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1
	1	0	1
Exit_CS	1	1	1
	1	1	1
Enter_CS	1	0	1
	1	0	0
	1	0	0

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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer


```c
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);
  }
}
```

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

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

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1
	1	0	1
Exit_CS	1	1	1
	1	1	1
Enter_CS	1	0	1
	1	0	0
	1	0	0
Exit_CS	1	1	0

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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer


```c
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);
  }
}
```

```c
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 with unbounded buffer: Race condition (non-existing element => items=-1)

Action	delay_cons=0	Syn=1	Item=0
Enter_CS	0	0	0
	0	0	1
	0	0	1
	0	0	1
	1	0	1
Exit_CS	1	1	1
	1	1	1
Enter_CS	1	0	1
	1	0	0
	1	0	0
Exit_CS	1	1	0
	1	1	0

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0
	1	1	0
	0	1	0

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

| | | | | | | | | | ....

Action	delay_cons=0	Syn=1	Item=0
	1	1	0
	0	1	0
Enter_CS	0	0	0

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

Consumer trying to access non-existing item

Action	delay_cons=0	Syn=1	Item=0
	1	1	0
	0	1	0
Enter_CS	0	0	0
Race_C	0	0	-1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0
	1	1	0
	0	1	0
Enter_CS	0	0	0
Race_C	0	0	-1
	0	0	-1

```
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 with unbounded buffer: Race condition (non-existing element => items=-1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer


```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0
	1	1	0
	0	1	0
Enter_CS	0	0	0
Race_C	0	0	-1
	0	0	-1
Exit_CS	0	1	-1

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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Non-Existing item)

Buffer


```c
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);
  }
}
```

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

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

Action	delay_cons=0	Syn=1	Item=0
	1	1	0
	0	1	0
Enter_CS	0	0	0
Race_C	0	0	-1
	0	0	-1
Exit_CS	0	1	-1
	0	1	-1

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

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

- Deadlock could happen when the consumer interrupted after executing sema_wait(&delay_Consumer) (the consumer putting it self to sleep before exiting the critical section) and the producer tried to enter the critical section.

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0
	-1	0	1

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocsks)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	0	1

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer

A									....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	0	1
Enter_CS	0	0	1

Figure: Single producer/consumer with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer


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

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

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

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	0	1
Enter_CS	0	0	1
	0	0	0

Figure: Single producer/consumer with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer


```c
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);
  }
}
```

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

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

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	0	1
Enter_CS	0	0	1
	0	0	0
	0	0	0

Figure: Single producer/consumer with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer


```c
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);
  }
}
```

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

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

Action	delay_cons=0	Syn=1	Item=0
	-1	1	0
Enter_CS	-1	0	0
	-1	0	1
	-1	0	1
	-1	0	1
Wakeup_C	0	0	1
Exit_CS	0	0	1
Enter_CS	0	0	1
	0	0	0
	0	0	0
	0	0	0

Figure: Single producer/consumer with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer

Interrupt come and processor goes to Producer, before the consumer leave its Critical section

Action	delay_cons=0	Syn=1	Item=0
	0	0	0
Block_C	-1	0	0
	-1	0	0

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Deadlocks)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0
	0	0	0
Block_C	-1	0	0
Deadlock	-1	-1	0

```
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 with unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

| | | | | | | | | | ....

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	0
Enter_CS	-1	0	0	0

```
void * consumer(void * p)
{
  sem_wait(&delay_consumer);  0 => -1 (sleep)
  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 with unbounded buffer: correct solution

Tips:

- Add temp, private variable(not shared variable) &
- Reduces the size of critical section(the probability of deadlock decreases)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	0
	-1	0	0	0
	-1	0	1	0

```c
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);
  }
}
```

```c
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	0
	-1	0	0	0
	-1	0	1	0
	-1	0	1	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	-
	-1	0	0	-
	-1	0	1	-
	-1	0	1	-
	-1	0	1	-

```c
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);
  }
}
```

```c
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	-
	-1	0	0	-
	-1	0	1	-
	-1	0	1	-
	-1	0	1	-
Wakeup_C	0	0	1	-

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

```
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
  }
}
```

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	-
	-1	0	0	-
	-1	0	1	-
	-1	0	1	-
	-1	0	1	-
Wakeup_C	0	0	1	-
Exit_CS	0	1	1	-

Figure: Single producer/consumer with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	-
	-1	0	0	-
	-1	0	1	-
	-1	0	1	-
	-1	0	1	-
Wakeup_C	0	0	1	-
Exit_CS	0	1	1	-
Enter_CS	0	0	1	-

Figure: Single producer/consumer with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

| | | | | | | | | | ....|

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	-
	-1	0	0	-
	-1	0	1	-
	-1	0	1	-
	-1	0	1	-
Wakeup_C	0	0	1	-
Exit_CS	0	1	1	-
Enter_CS	0	0	1	-
	0	0	0	-

Figure: Single producer/consumer with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

| | | | | | | | | | ....|

```
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 with unbounded buffer: correct solution

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	-
	-1	0	0	-
	-1	0	1	-
	-1	0	1	-
	-1	0	1	-
Wakeup_C	0	0	1	-
Exit_CS	0	1	1	-
Enter_CS	0	0	1	-
	0	0	0	-
	0	0	0	0

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

| | | | | | | | | | ....|

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	-
	-1	0	0	-
	-1	0	1	-
	-1	0	1	-
	-1	0	1	-
Wakeup_C	0	0	1	-
Exit_CS	0	1	1	-
Enter_CS	0	0	1	-
	0	0	0	-
	0	0	0	0
	0	0	0	0

Figure: Single producer/consumer with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer


```c
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);
  }
}
```

```c
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 with unbounded buffer: correct solution

Action	delay_cons=0	Syn=1	Item=0	temp
Block_C	-1	1	0	-
	-1	0	0	-
	-1	0	1	-
	-1	0	1	-
	-1	0	1	-
Wakeup_C	0	0	1	-
Exit_CS	0	1	1	-
Enter_CS	0	0	1	-
	0	0	0	-
	0	0	0	0
	0	0	0	0
Exit_CS	0	1	0	0

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

If interrupt happens, before the consumer check the items, it can refer to the old value of item (temp) and avoid race condition.

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0
	1	0	1	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0
	1	0	1	0
Exit_CS	1	1	1	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0
	1	0	1	0
Exit_CS	1	1	1	0
	1	1	1	0

Figure: Single producer/consumer with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0
	1	0	1	0
Exit_CS	1	1	1	0
	1	1	1	0
	0	1	1	0

Figure: Single producer/consumer with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

A									....

```
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);
  }
}
```

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0
	1	0	1	0
Exit_CS	1	1	1	0
	1	1	1	0
	0	1	1	0
Enter_CS	0	0	1	0

Figure: Single producer/consumer with unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer | | | | | | | | | | ....

```
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 with unbounded buffer: correct solution

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0
	1	0	1	0
Exit_CS	1	1	1	0
	1	1	1	0
	0	1	1	0
Enter_CS	0	0	1	0
	0	0	0	0

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer | | | | | | | | | | ....

```
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 with unbounded buffer: correct solution

Action	delay_cons=0	Syn=1	Item=0	temp
Exit_CS	0	1	0	0
Enter_CS	0	0	0	0
	0	0	1	0
	0	0	1	0
	0	0	1	0
	1	0	1	0
Exit_CS	1	1	1	0
	1	1	1	0
	0	1	1	0
Enter_CS	0	0	1	0
	0	0	0	0
	0	0	0	0

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0	temp
	0	0	0	0
	0	0	0	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

| | | | | | | | | | ....

Action	delay_cons=0	Syn=1	Item=0	temp
	0	0	0	0
	0	0	0	0
Exit_CS	0	1	0	0

```
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

| | | | | | | | | | ....|

Action	delay_cons=0	Syn=1	Item=0	temp
	0	0	0	0
	0	0	0	0
Exit_CS	0	1	0	0
	0	1	0	0

```c
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);
  }
}
```

```c
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 with unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer (Solution)

Buffer

| | | | | | | | | | ....

Action	delay_cons=0	Syn=1	Item=0	temp
	0	0	0	0
	0	0	0	0
Exit_CS	0	1	0	0
	0	1	0	0
Block_C	-1	1	0	0

```
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 with unbounded buffer: correct solution

Recap
Take-Home Message

- Modern Operating Systems (Tanenbaum): **Chapter 2(2.3.5, 2.5.1)**
- Operating System Concepts (Silberschatz):**Chapter 6(6.6-7)**
- Operating Systems: Internals and Design Principles (Starlings): **Chapter 5(5.3, 5.6)**