Operating Systems and Concurrency

Lecture 9: Concurrency

G52OSC/COMP2007

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

University Of Nottingham
United Kingdom

2018

- **Software approaches:** Peterson's solution
- **Hardware approaches:**
  - Disabling interrupts
  - `test_and_set()`
  - `compare_and_swap()`
- **Mutexes** as "binary locks"

- **Semaphores** are an approach for **mutual exclusion** (and process synchronisation) provided by **the operating system**
  - They contain an **integer variable**
  - We distinguish between **binary** (0-1) and **counting semaphores** (0-N)

- Two **atomic functions** are used to **manipulate semaphores** (think of the `counter++` example)
  - `wait()` is called when a resource is **acquired**, the counter is **decremented**
  - `signal()/post()` is called when a resource is **released**, the counter is **incremented**

```c
typedef struct {
    int value;
    struct process * list;
} semaphore;
```

Figure: Conceptual definition of a semaphore


```c
wait(semaphore * S) {
    S->value--;
    if(S->value < 0) {
      add process to S->list
      block(); // system call
}
```


Figure: Conceptual implementation of a wait()

```
post(semaphore * S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);  // system call
    }
}
```


Figure: Conceptual implementation of post()

Thread 1	Thread 2	Thread 3
...	...	...
wait(&s) 1 => 0	...	...
...	...	...
...	wait(&s)	...
...	...	wait(&s)
post(&s)	(wakeup)	...
...	...	...
...	...	...
...	post(&s)	(wakeup)
...	...	...
...	...	post(&s)
...	...	...

Figure: Semaphore example

Thread 1	Thread 2	Thread 3
...	...	...
wait(&s)	...	...
...	...	...
...	wait(&s) 0 => -1	...
...	...	wait(&s)
post(&s)	(wakeup)	...
...	...	...
...	...	...
...	post(&s)	(wakeup)
...	...	...
...	...	post(&s)
...	...	...

Figure: Semaphore example

```
Thread 1          Thread 2          Thread 3
...               ...               ...
wait(&s)          ...               ...
...               ...               ...
...               wait(&s)          ...
...               ...               wait(&s) -1 => -2
post(&s)          (wakeup)          ...
...               ...               ...
...               ...               ...
...               post(&s)          (wakeup)
...               ...               ...
...               ...               post(&s)
...               ...               ...
```

Figure: Semaphore example

Thread 1	**Thread 2**	**Thread 3**
...	...	...
wait(&s)	...	...
...	...	...
...	wait(&s)	...
...	...	wait(&s)
post(&s) -2 => -1	**(wakeup)**	...
...	...	...
...	...	...
...	post(&s)	(wakeup)
...	...	...
...	...	post(&s)
...	...	...

Figure: Semaphore example

Thread 1	Thread 2	Thread 3
...	...	...
wait(&s)	...	...
...	...	...
...	wait(&s)	...
...	...	wait(&s)
post(&s)	(wakeup)	...
...	...	...
...	...	...
...	post(&s) -1 => 0	(wakeup)
...	...	...
...	...	post(&s)
...	...	...

Figure: Semaphore example

Thread 1	Thread 2	Thread 3
...	...	...
wait(&s)	...	...
...	...	...
...	wait(&s)	...
...	...	wait(&s)
post(&s)	(wakeup)	...
...	...	...
...	...	...
...	post(&s)	(wakeup)
...	...	...
...	...	**post(&s) 0 => 1**
...	...	...

Figure: Semaphore example

- Calling `wait()` will **block** process when the internal **counter is negative** (**no busy waiting**)
  1. The process **joins the "blocked" queue**
  2. The **process state** is changed from **running** to **blocked**
  3. Control is transferred to the **process scheduler**

- Calling `post()` **removes a process** from the **blocked queue** if the counter is less or equal to 0
  1. The process state is changed from **blocked** to **ready**
  2. Different queueing strategies can be employed to **remove processes** (e.g. FIFO, etc.)

- The **negative value** of a semaphore is the **number of processes waiting** for the resource

- `block()` and `wakeup()` are **system calls** provided by the operating system
- `post()` and `wait()` must be **atomic**
  - Can be achieved through the use of **mutexes** (or disabling interrupts in single CPU systems, hardware instructions)
  - **Busy waiting is moved** form the **critical section** to `wait()` and `post()` (which are short anyway – the original critical sections themselves are usually much longer)

```
post(semaphore * S) {
    // lock mutex here
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);  // system call
    }
    // unlock mutex here
}
```

- Semaphores within the **same process** can be **declared as global variables** of the type `sem_t`
  - `sem_init()` initialises the value of the semaphore
  - `sem_wait()` decrements the value of the semaphore
  - `sem_post()` increments the values of the semaphore
- An **explanation** of any of these functions can be found in the **man pages**, e.g. by typing `man sem_init` on the Linux command line

```c
// includes here, e.g. semaphore.h
sem_t s;
int sum = 0;

void * calc(void * number_of_increments)
{ int i;
  for(i = 0; i < *((int*) number_of_increments);i++)
  { sem_wait(&s);
    sum++;
    sem_post(&s);
  }
}
void main()
{ int iterations = 50000000;
  pthread_t tid1,tid2;
  sem_init(&s,0,1);
  // no error checking for clarity/brevity
  pthread_create(&tid1, NULL, calc, (void *) &iterations);
  pthread_create(&tid2, NULL, calc, (void *) &iterations);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of sum is: %d\n", sum);
}
```

- Synchronising code does result in a **performance penalty**
  - Synchronise **only when necessary**
  - Synchronise **as few instructions** as possible (synchronising unnecessary instructions will delay others from entering their critical section)
- **Carefully consider how** to synchronise!

```
void * calc(void * increments)
{
  int i, temp = 0;
  for(i = 0; i < *((int*) increments);i++)
  {
    temp++;
  }
  sem_wait(&s);
  sum+=temp;
  sem_post(&s);
}
```


Figure: Fast synchronised sums

- **Starvation:** poorly designed **queueing approaches** (e.g. LIFO) may result in fairness violations

- **Deadlocks:** two or more processes are **waiting indefinitely** for an event that can be **caused only by one of the waiting processes**
  - I.e., every process in a set is **waiting for an event** that can only be **caused by another process** in **the same set**
  - E.g., consider the following sequence of **instructions on semaphores**


```
P0          P1
wait(S);    ...
...         wait(Q);
wait(Q);    ...
...         wait(S);
...         ...
```

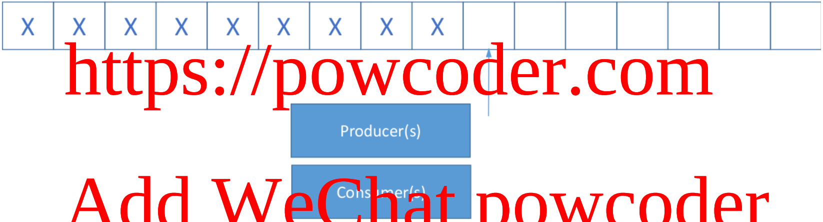
- **Priority inversion** happens when a **high priority process** (`H`) has to wait for a **resource** currently held by **a low priority process** (`L`) – and has to **wait for the lower priority process** to finish
- Priority inversion **can happen in chains**, e.g., a `H` waits for `L` to release a resource, and `L` is interrupted by a medium high priority process (`M`)
  - `H` waits for `L` which is interrupted by `M`
  - **Priority inversion** can be **prevented** by implementing **priority inheritance** to boost `L`'s to the `H`'s priority

Caveats
Potential Difficulties

- **Priority inversion** happens when a **high priority process** (`H`) has to wait for a **resource** currently held by **a low priority process** (`L`) – and has to **wait for the lower priority process** to finish
- Priority inversion **can happen in chains**, e.g., a `H` waits for `L` to release a resource, and `L` is interrupted by a medium high priority process (`M`)
  - `H` waits for `L` which is interrupted by `M`
  - **Priority inversion** can be **prevented** by implementing **priority inheritance** to boost `L`'s to the `H`'s priority
- Programming with mutexes and semaphores remains **prone to errors**

- **Producer(s)** and **consumer(s)** share *n* **buffers** (e.g. an array) that are capable of holding **one item each** (**printer queue**)
  - The buffer can be of **bounded** (size *n*) or **unbounded size**
  - There can be **one or multiple consumers** and/or **producers**

- The **producer(s)** add(s) items and **goes to sleep** if the buffer is **full** (for a bounded buffer)
- The **consumer(s)** remove(s) items and **goes to sleep** if the buffer is **empty**


X X X X X X X X X
Producer(s)
Consumer(s)

- The simplest version of the problem has **one producer**, **one consumer**, and a buffer of **unbounded size**
- A **counter (index)** variable keeps track of the number of **items in the buffer**

- It uses **two binary semaphores**:
  - `sync` **synchronises** access to the **buffer (counter)**, initialised to **1**
  - `delay_consumer` ensures that the **consumer** goes to **sleep** when there are no items available, initialised to **0**

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: First Attempt


```
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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt


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


Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt


```
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

```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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt


```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

```
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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt


```
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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt


```
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

```
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 with unbounded buffer

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: First Attempt


```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

```
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

```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

```
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); 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

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


Figure: Single producer/consumer with unbounded 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++; 0 => 1
  printf("%d\n", items);
  if(items == 1)
   sem_post(&delay_consumer);
  sem_post(&sync);
 }
}
```

Figure: Single producer/consumer with unbounded 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

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt


```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

The Producer/Consumer Problem
One Consumer, One Producer, Unbounded Buffer: First Attempt


```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); (wakeup)
 }
}
```

```c
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

```
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

- It is obvious that any **manipulations of count** 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: 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)

```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); 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)

```
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


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

The Producer/Consumer Problem

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


```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 and an unbounded buffer: Race condition (non-existing element => items = -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 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)

```
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)

```
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)

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

The Producer/Consumer Problem

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


```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);
 }
}
```

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

```
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)

```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 and an unbounded buffer: Race condition (non-existing element => items = -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 and an unbounded buffer: Race condition (non-existing element => items = -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 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)

```
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)

```c
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);
 }
}
```

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

The Producer/Consumer Problem

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


```c
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);
 }
}
```

```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 and an unbounded buffer: Race condition (non-existing element => items = -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 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)

```
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)

```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): 1 => 0
 }
}
```

```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 and an unbounded buffer: Race condition (non-existing element => items = -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 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


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

- **Semaphore** provided by the OS
- Using **semaphores in Linux**
- **Difficulties** in synchronising code