Operating Systems and Concurrency

Concurrency 3

COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture - Approaches to Mutual Exclusion

- **Software approaches:** Peterson's solution
- **Hardware approaches:**
  - `test_and_set()`
  - `compare_and_swap()`
- **Mutexes** as an abstraction providing binary locks

- Recall **Mutexes** are a locking abstraction for providing mutual exclusion.
- Often provided by the operating system, via an API such as pthreads.
- They are **binary** - either a thread has currently acquired the mutex or it hasn't!

```
pthread_mutex_t lock;           // declaration

pthread_mutex_lock(&lock);      // acquire
counter++;
pthread_mutex_unlock(&lock);    // release
```

- **Semaphores** are another abstraction for **mutual exclusion and process synchronisation**, often provided by **the operating system**
  - They have a **capacity**, either a positive number or infinity.
  - We distinguish between **binary** (2 valued) and **counting semaphores** (N-valued or unbounded).
- Two **functions** are used to **manipulate semaphores** (think of the `counter++` example)
  - `wait()` is called when a resource is **acquired**, the capacity is **decremented**
  - `signal()` or `/post()` is called when a resource is **released**, the capacity is **incremented**.
- The semaphore **can only be acquired when its currently capacity is strictly positive**.
- A thread calling `post` **does not** have to have previous called `wait`.

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

Figure: Conceptual definition of a semaphore

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

Figure: Conceptual implementation of a wait()

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

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

Semaphores

Implementation

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

Semaphores

Implementation

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** the process when the internal **counter is not positive**
  1. The process **joins the a queue blocking on the semaphore**
  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 available:
  1. The process state is changed from **blocked** to **ready**
  2. Different queueing strategies can be employed to **remove processes** - so avoid unjustified assumptions in your code.

- The queue length is the number of processes waiting on the semaphore.
- `block()` and `wakeup()` are **system calls** provided by the operating system.
- `post()` and `wait()` must be **atomic**.

Semaphores

OS approaches

```
void post(semaphore* S) {
  lock(&mutex);
  S->count++;
  if(S->count <= 0) {
    // remove process P from queue
    wakeup(P);
  }
  unlock(&mutex);
}
```


Posix Semaphores

Counter++ revisited

- Semaphores within the **same process** can be declared as 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

Posix Semaphores

Example

```c
sem_t s;
int sum = 0;
void* calc(void* arg) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations;i++) {
    sem_wait(&s);
    sum++;
    sem_post(&s);
  }
  return 0;
}
int main() {
  pthread_t tid1,tid2;
  sem_init(&s,0,1);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of sum is: %d\n", sum);
}
```

Question

Does the previous code give the right answer on my Mac?

Question

Does the previous code give the right answer on my Mac?

Answer

Unfortunately, running the code on my Mac gives an answer slightly below 100000000! Details:

- Compiles with compiler warnings that `sem_init` is deprecated.
- `sem_init` is always failing, returning -1!
- Using named semaphores will work - see the lab for these.
- Even then, code using named semaphores must run as root on a Mac to call `sem_unlink` successfully.

Real-world issues

Standards support

Lessons

- **Never ignore compiler warnings**.
- **Always check return values** - slide examples don't for space reasons.
- **Test code thoroughly** - implicit assumption Mac would work like Linux was wrong!
- **Be aware of platform specific issues** such as `sem_unlink` behaviour on Mac.
- **Use the appropriate concurrency primitives** - the example really needed a mutex.

Efficiency

How/when to synchronise

- Synchronising code does result in a **performance penalty**
  - Synchronise **only when necessary**.
  - Synchronise **as few instructions** as possible.
- **Carefully consider how** to synchronise!

Using Semaphores

Counter++ revisited

```
void* calc(void* increments) {
  int number_of_iterations = 50000000;
  int total = 0;
  for(int i = 0; i < number_of_iterations; i++) {
    total++; // Pretend this is non-trivial to work out
  }
  sem_wait(&s);
  sum+=total;
  sem_post(&s);
  return 0;
}
```

Figure: Fast synchronised sums

Caveats

Potential Difficulties

- **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);
...         ...
```

The Producer/Consumer Problem

Problem Description

- **Producer(s)** and **consumer(s)** share a **buffer** of values - this could for example be a printer queue.
  - The buffer can be of **bounded** (maximum size N) or **unbounded size**.
  - There can any number of **producers** or **consumers**.
- A **producer** attempts to add items and **blocks** if the buffer is **full.**
- A **consumer** attempts to remove items and **blocks** if the buffer is **empty**.

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer

- 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 blocks** 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

```
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

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

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);
 }
}
```

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);
 }
}
```

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); (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

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--; 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

```
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

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); 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

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);
 }
}
```

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); 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

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); 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

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

```
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

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); (wakeup)
 }
}
```

```
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

- 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 blocked, 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)

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

```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)

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)

```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); 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

```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); 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

```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);
  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

```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)

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)

Test your understanding

- Is a binary semaphore the same thing as a mutex?
- When should you prefer a mutex rather than a binary semaphore?
- Is there a straightforward way to check concurrent code is correct?