University of Nottingham
UK | CHINA | MALAYSIA

# Operating Systems and Concurrency

## Lecture 9: Concurrency

University of Nottingham, Ningbo China, 2024

## Recap
## Last Lecture

- Software approaches: Peterson’s solution

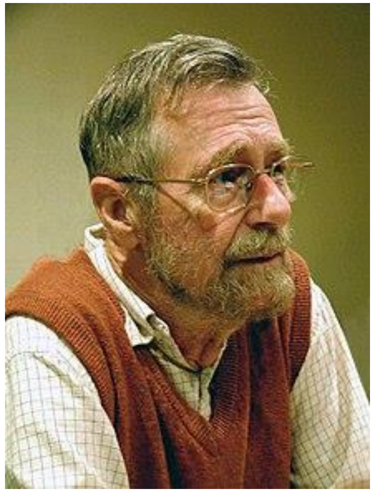
- Hardware approaches:
  - Disabling interrupts
  - test_and_set()
  - compare_and_swap()

- OS Approach: Mutex

# Today Class

- OS approach
  - Semaphores
    - Implementation approaches
    - Examples
  - Difficulties and challenges
  - Producer/consumer problem

# OS approaches
## Semaphores



- Propsed by Edsger Disjktra, is a technique to manage a concurrent process by using a simple integer value, which is know as a semaphore.

- A Semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal()
  - wait() → p [from the Duth word proberen, which means “to test”
    - Is called when a resource is **acquired**, the counter is decremented.
    - This operation is used when a process wants to access a shared resource.

  - Signal() →v from [from the Duth word verhogen, which means “to increment”]
    - Is called when a resource us **released**, the counter is incremented.
    - This operation is used when a process has finished using a resource and releases it, allowing other waiting processes to access it.

# OS approaches (cont.)
## Semaphores

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

Figure: Conceptual definition of a semaphore

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

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

Figure: Conceptual implementation of a acquire() and post()

- **wait() (P or acquire):**
  - S->value--: Decrements the semaphore value.
  - If the value is still ≥ 0, the process can proceed because resources are available.
  - S->value < 0: If the semaphore value goes below 0, it means no more resources are available, so the process must wait.
    - The process is then added to a waiting list, and it is blocked (suspended) until resources become available.

# OS approaches (cont.)
## Semaphores: Process Blocking in wait()

- Decrement the counter (S->value--).
- Check if the counter is negative (S->value < 0):
  - If yes, the process cannot proceed, so:
  - The process is added to the blocked queue (often called the semaphore’s waiting list).
  - The process’s state is changed from running to blocked, indicating it cannot execute further.
  - A system call like block() is invoked to suspend the process, transferring control to the OS scheduler.

- The operating system is responsible for managing blocked processes. When a process is blocked, it’s removed from the CPU, and the scheduler selects another process to run.

- This ensures no busy waiting occurs (i.e., the CPU isn't wasting cycles by constantly checking the semaphore).

# Semaphores
## OS approaches (cont.)

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

Figure: Conceptual definition of a semaphore

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

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

Figure: Conceptual implementation of a acquire() and post()

- signal() (V or release):
  - S->value++: Increments the semaphore value. This indicates that a resource has been released or made available.
  - S->value <= 0: If there are processes waiting (S->value is ≤ 0), one process is removed from the waiting list, and it is woken up to continue execution.

# OS approaches (cont.)

## Semaphores: Semaphore Types

- Binary Semaphore (Mutex): This type of semaphore has a value of either 0 or 1. It is typically used for ensuring mutual exclusion in critical sections where only one process can access a resource at a time.

- Counting Semaphore: This type of semaphore can have any non-negative integer value. It is used to control access to a resource that has multiple instances (e.g., a pool of connections).
  - It is generally used in situations where there are multiple identical resources, such as a pool of database connections, a set of shared memory buffers, or a thread pool.

  - The semaphore's value is initialized to the number of available resources.
    - E.g., if there are 5 instances of a shared resource, the semaphore is initialized with a value of 5.

# OS approaches (cont.)

## Semaphores: counting semaphore wait() Operation (Requesting a Resource)

- Initialization: its value represents the number of available resources.
  - E.g, S = 5 means that 5 resources are available.

- Decrement the counter (S->value--).
- Check if the counter is negative (S->value < 0):
  - If yes, the process cannot proceed, so:
  - The process is added to the blocked queue (often called the semaphore's waiting list).
  - The process's state is changed from running to blocked, indicating it cannot execute further.
  - A system call like block() is invoked to suspend the process, transferring control to the OS scheduler.

# OS approaches (cont.)

## Semaphores: counting semaphore signal() Operation (Releasing a Resource)

- signal() Operation (Releasing a Resource):
    - S->value++: Increments the semaphore value. This indicates that a resource has been released or made available.
    - S->value <= 0: If there are processes waiting (S->value is ≤ 0), one process is removed from the waiting list, and it is woken up to continue execution.
- If the value was non-negative, the resource count is simply incremented without waking up any processes.

# OS approaches (cont.)

## Semaphores: Behavior of Counting Semaphores

- Positive Semaphore Value (S > 0):This indicates the number of resources that are still available for use. A process can proceed immediately when calling wait().

- Zero Semaphore Value (S == 0):All resources are currently in use. Any process that attempts to wait() on the semaphore at this point will block, as no resources are available.

- Negative Semaphore Value (S < 0):A negative semaphore value represents the number of processes that are blocked, waiting for resources.
    - E.g., if S == -3, it means there are 3 processes waiting for resources to become available.

# OS approaches (cont.)
## Semaphores: Use Cases for Counting Semaphores

- Resource Management: Counting semaphores are ideal for managing access to pools of resources where there are multiple instances, such as:
  - Database connections in a connection pool.
  - Shared memory buffers in a buffer pool.
  - Thread pools, where multiple worker threads can be available to handle concurrent tasks.
- Counting semaphores are often used to limit the level of concurrency in tasks. For instance, they can limit the number of concurrent readers or writers in a system.
- Using a counting semaphore to manage a pool of resources allows the operating system to automatically handle synchronization by **blocking** processes when resources are unavailable and **waking** them when resources become available, ensuring processes can access resources as soon as they are free.
- But, does not enforce mutual exclusion.

# Semaphores
## Implementation…

```
Thread 1
...
wait(&s) 1 => 0
...
...
...
post(&s)
...
...
...
...
...
...
```

```
Thread 2
...
...
...
wait(&s)
...
(wakeup)
...
...
post(&s)
...
...
...
```

```
Thread 3
...
...
...
...
wait(&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

# Semaphores
## Implementation…

```
Thread 1
...
wait(&s)
...
...
...
post(&s)
...
...
...
...
...
...
```

```
Thread 2
...
...
...
wait(&s)
...
(wakeup)
...
...
post(&s)
...
...
...
```

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

Figure: Semaphore example

# Semaphores
## Implementation

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

```
Thread 2
...
...
...
wait(&s)
...
(wakeup)
...
...
post(&s)
...
...
...
```

```
Thread 3
...
...
...
...
wait(&s)
...
...
...
(wakeup)
...
post(&s)
...
```

Figure: Semaphore example

# Semaphores
## Implementation

Thread 1
```
...
wait(&s)
...
...
...
post(&s)
...
...
...
...
...
...
```

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

Thread 3
```
...
...
...
...
wait(&s)
...
...
...
(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

# OS Solution
# Semaphores: requirements

- Mutual Exclusion: Semaphores (especially binary semaphores or mutexes) effectively ensure mutual exclusion.
- Progress: Semaphores generally ensure progress, but the system's queueing strategy plays a role in ensuring fairness.
- Bounded Waiting: Semaphores do not inherently ensure bounded waiting. Additional fairness mechanisms are needed to prevent starvation.

# Semaphores in Linux
## Counter++ revisited

- 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 ( https://linux.die.net/man ), e.g. by typing man sem_init on the Linux command line

# Semaphores in Linux
Example

```
// 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);

    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);
}
```

Output:

The value of sum is: 100000000

# OS Solution
# Semaphores: Advantage

- Simple and efficient for basic synchronization tasks.
- No busy waiting (processes can be blocked until they are allowed to proceed).
- Can be used for both mutual exclusion and complex synchronization.

# Caveats
# Potential Difficulties (Disadvantage)

- **Indefinite blocking or starvation**, a situation in which processes wait indefinitely within the semaphore.
  - May occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.
- **Deadlock**: A situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

- E.g. Consider P0 and P1, each accessing two semaphores, S and Q, set the value 1;
- Suppose that **P0 executes wait(S) and then P1 executes wait(Q)**.
- When P0 executes wait(Q), it must wait until P1 executes signal(Q).
- Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S).
- Since these signal() operations cannot be executed, P0 and P1 are deadlocked.

| P0 | P1 |
|---|---|
| Wait(S); | ... |
| ... | Wait(Q); |
| Wait(Q); | ... |
| ... | Wait(S); |
| . | . |
| . | . |
| . | . |
| Signal(S); | ... |
| ... | Signal(Q); |
| Signal(Q); | ... |
| .... | Signal(S); |

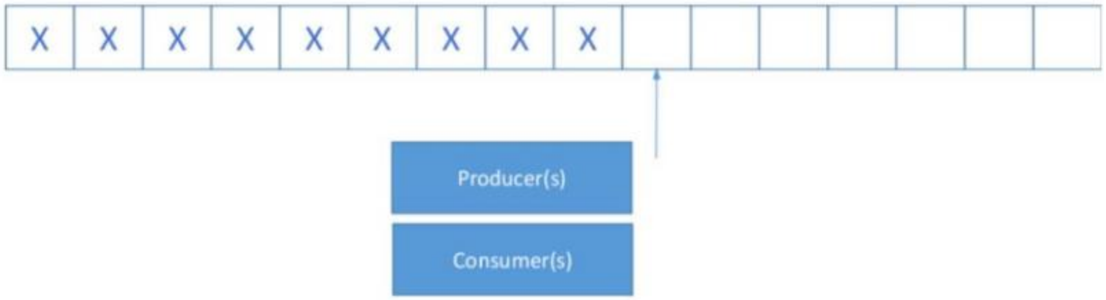
# The Producer/Consumer
## Problem Description

- How Semaphore solve **Producer/Consumer** problem?

- 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
- The **consumer(s)** remove(s) items and **goes to sleep** if the buffer is **empty**

# The Producer/Consumer
## Problem Description




```
while (true) {
    // wait until items in buffer
    while (counter == 0); /* do nothing */

    consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
    /* consume the item in next consumed */
}
```

Consumer

```
while (true) {
    //while buffer is full
    while (counter == BUFFER SIZE) ; /* do nothing */

    // add item when space becomes available
    buffer[in] = new_item;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Producer

# 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.**

- There are two shared variables
  - 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

- Semaphores:
  - sync: This semaphore is used to ensure mutual exclusion when accessing the shared items variable.
  - delay_consumer: This semaphore is used to block the consumer when there are no items available.
- Shared Variable:
  - Counter/items: This represents the number of items in the buffer. The producer increments this value when producing items, and the consumer decrements it when consuming items.

# The Producer/Consumer Problem
## One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

University of Nottingham
UK | CHINA | MALAYSIA

# The Producer/Consumer Problem
## One Consumer, One Producer, Unbounded Buffer (First Attempt)

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 |
| Weakup_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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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 |
| Weakup_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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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 |
| Weakup_C | 0 | 0 | 1 |
| Exit_CS | 0 | 1 | 1 |
| Enter_CS | 0 | 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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

| 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 |
| Weakup_C | 0 | 0 | 1 |
| Exit_CS | 0 | 1 | 1 |
| Enter_CS | 0 | 0 | 1 |
| | 0 | 0 | 0 |

```
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)

Buffer

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | .... |

| 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 |
| Weakup_C | 0 | 0 | 1 |
| Exit_CS | 0 | 1 | 1 |
| Enter_CS | 0 | 0 | 1 |
| | 0 | 0 | 0 |
| | 0 | 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);
    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)

Buffer

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | .... |

| 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 |
| Weakup_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 |

```
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)

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 |
|---|---|---|---|
| C_blocked | -1 | 1 | 0 |
| Enter_CS | -1 | 0 | 0 |
| | -1 | 0 | 1 |
| | -1 | 0 | 1 |
| | -1 | 0 | 1 |
| Weakup_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 |
| | 0 | 1 | 0 |

Figure: Single producer/consumer with unbounded buffer

# The Producer/Consumer Problem
## One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

| 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 |
| Weakup_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 |
| | 0 | 1 | 0 |
| C_blocked | -1 | 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); 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)

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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

University of Nottingham
UK | CHINA | MALAYSIA

# The Producer/Consumer Problem
## One Consumer, One Producer, Unbounded Buffer (First Attempt)

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

# The Producer/Consumer Problem
# One Consumer, One Producer, Unbounded Buffer (First Attempt)

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 |
| Weakup_C | 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);(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)

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 |
| Weakup_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

# Recommended readings

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