Chapter 2

Inter-Process Communication

Part 4

Mutual Exclusion with Busy Waiting - Problems

- Peterson's solution is correct, but it has the problem of busy waiting.
- When a process wants to enter the critical region, it checks to see if the entry is allowed.
  - If it is, the process will enter the critical region
  - Else wait for entry

- Problems with busy waiting
  1. Waste of CPU time
  2. Priority inversion problem

Priority inversion problem (1)

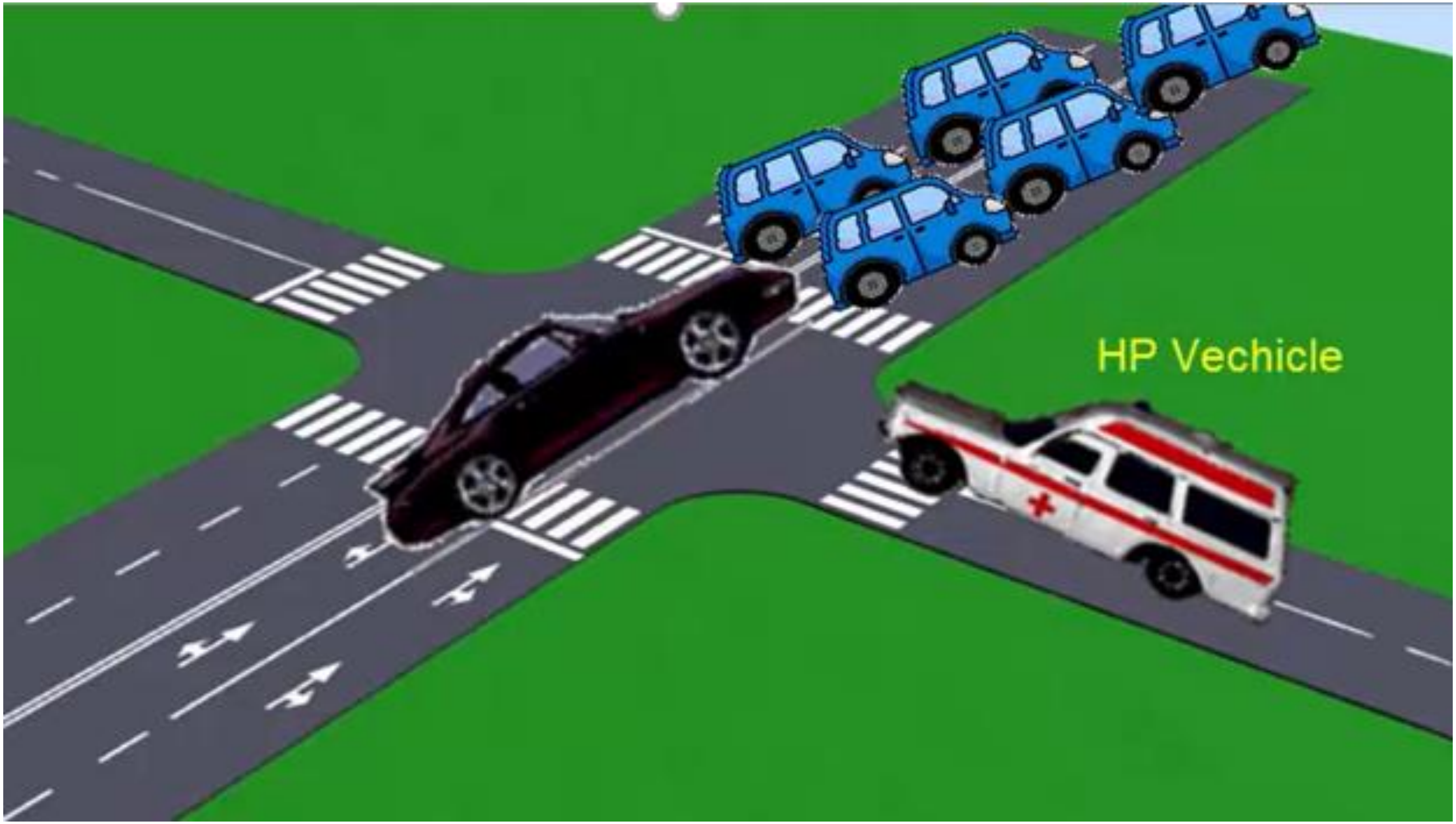
A high priority task can become blocked by a lower priority task – if lower priority task locks access to resources shared by both tasks

H – High priority task

L – Low priority task

- The scenario where a L holds a shared resource, that is required by H.
- Causes the execution of H blocked until L releases the resource → "**inverting**" the relative priorities of the two tasks.

Priority inversion problem (1)



Priority inversion problem (2)

H – High priority task

M – Medium priority task

L – Low priority task

- In the following slide assume that L and H need to access shared memory, but M does not need to access the shared memory.

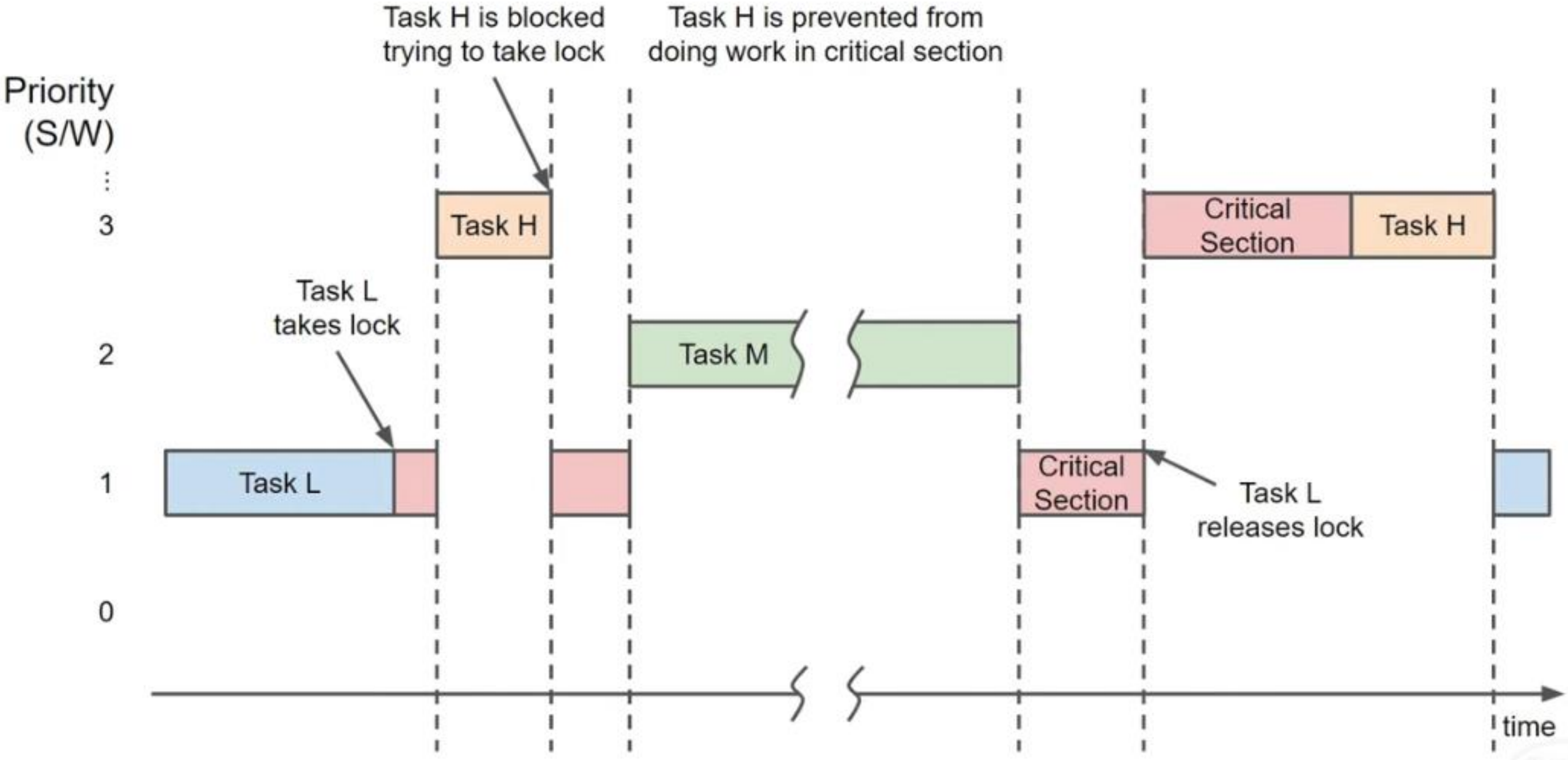
- When L is running M can interrupt L, but H cannot interrupt L as L and H need to access the shared memory

  → It appears to be M has higher priority than H.

Priority inversion problem (2)

Running outside the CS

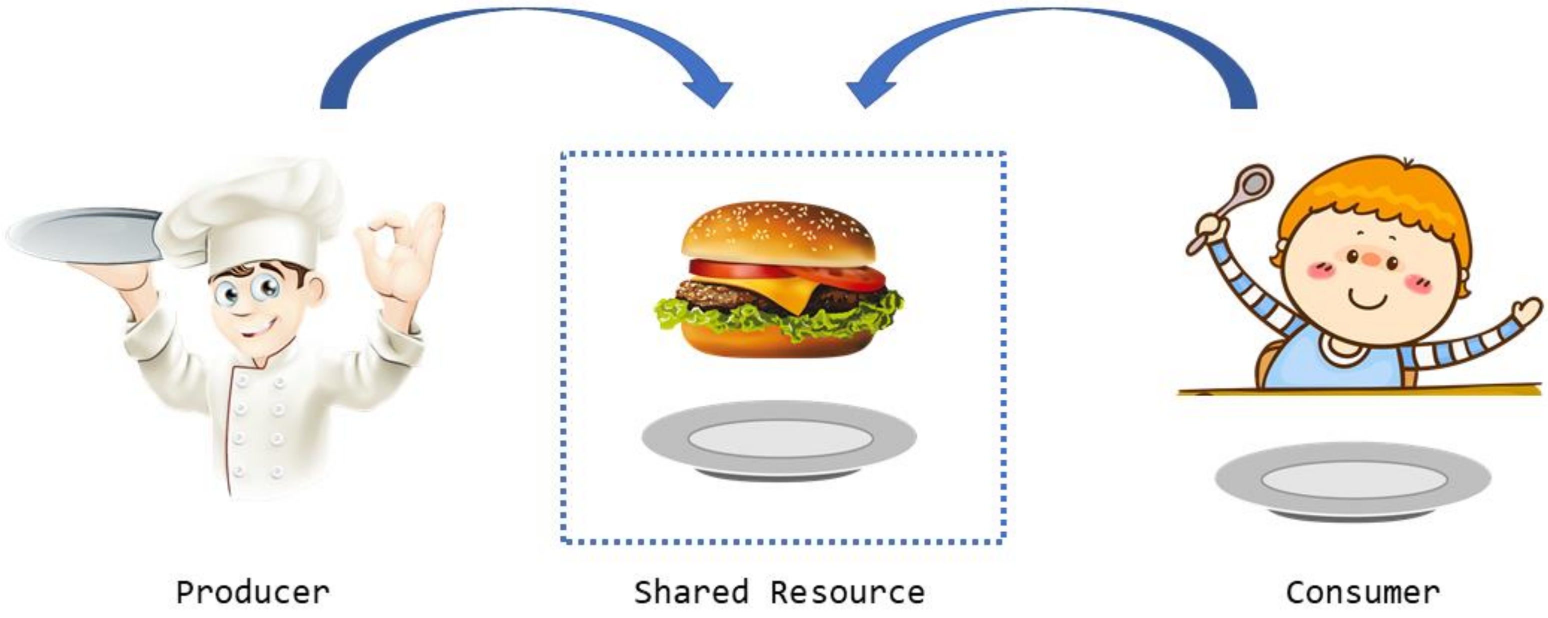
Running inside the CS



Sleep and Wakeup

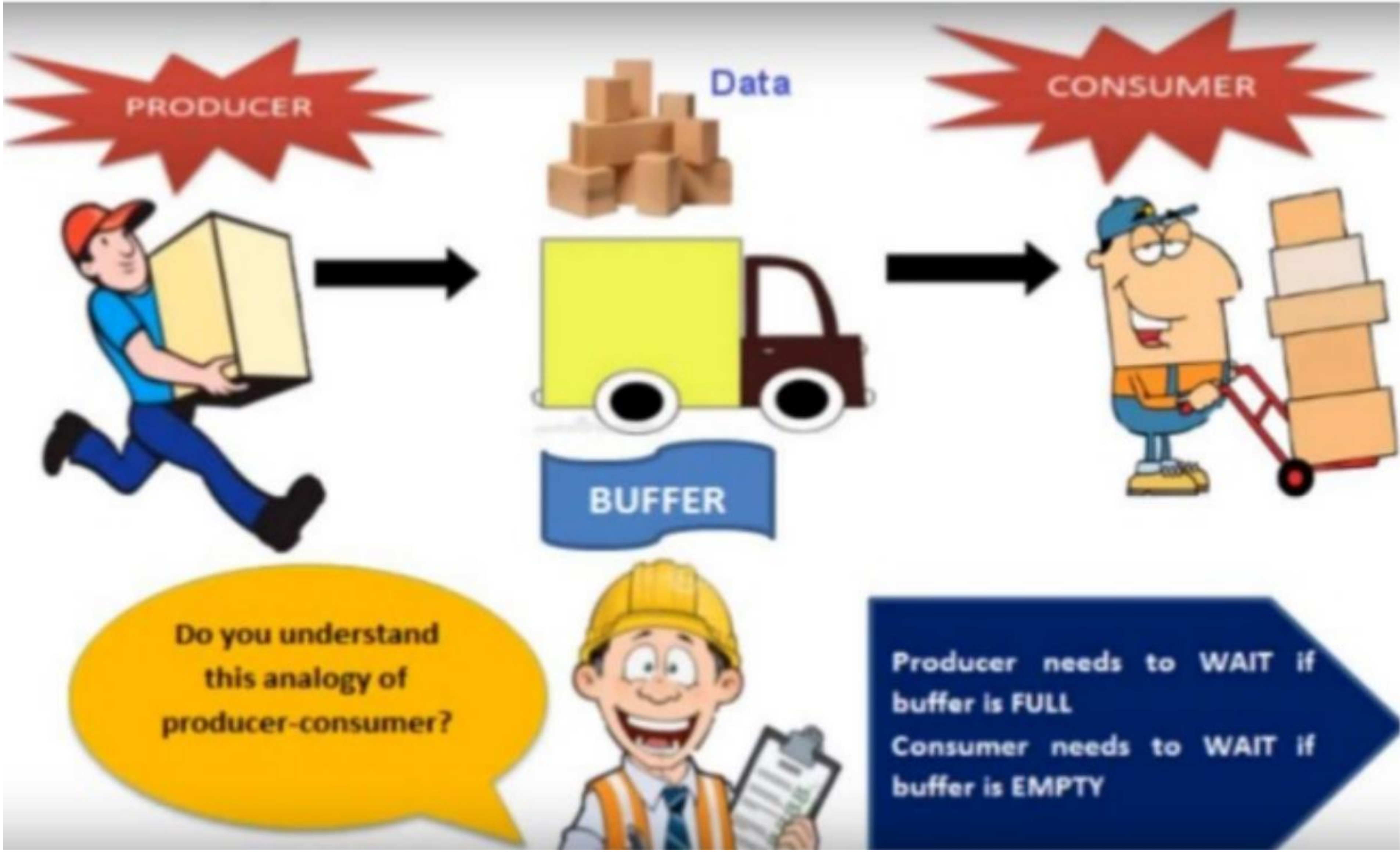
- A solution to busy waiting
- Sleep – system call, that causes the caller to block. i.e. suspend a process until another process wakes it up
- Wakeup – system call, wakes up a sleeping process

Producer-Consumer Problem (Bounded Buffer Problem)



- Consists of two processes, producer and consumer
- They share a common fixed-size buffer
- Producer puts items into the buffer
- Consumer takes items out of buffer

Producer-Consumer



Producer-Consumer Problem (Bounded Buffer Problem)

Problem:

- When producer wants to put some items but the buffer is full

Solution:

- Producer goes to sleep
- Consumer wakes up the producer when he removes an item from the buffer

Producer-Consumer Problem (Bounded Buffer Problem)

Problem:

- When consumer wants to remove an item from the buffer, but the buffer is empty

Solution:

- Consumer goes to sleep
- Producer wakes up the consumer when he puts an item into the buffer

Sleep and Wakeup

Producer module

```
#define N 100                              /* number of slots in the buffer */
int count = 0;                             /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                         /* repeat forever */
        item = produce_item( );            /* generate next item */
        if (count == N) sleep( );          /* if buffer is full, go to sleep */
        insert_item(item);                 /* put item in buffer */
        count = count + 1;                 /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}
```

Producer-consumer problem with fatal race condition

Sleep and Wakeup

Consumer module

```
void consumer(void)
{
      int item;

      while (TRUE) {                                /* repeat forever */
            if (count == 0) sleep( );               /* if buffer is empty, got to sleep */
            item = remove_item( );                  /* take item out of buffer */
            count = count − 1;                      /* decrement count of items in buffer */
            if (count == N − 1) wakeup(producer);   /* was buffer full? */
            consume_item(item);                     /* print item */
      }
}
```

Producer-consumer problem with fatal race condition

Problem? Race condition, since access to count (shared variable) is unconstrained.

Problems with Sleep and Wakeup: Probable race condition?

Assume that the buffer is empty now.

Consumer	Producer	
if(count == 0)		
	produce_item() insert_item() count = 1 wake_up(consumer) ………	Context switching C→P
sleep()		Context switching P→C
	……… ……… sleep()	Context switching C→P

Both P and C will sleep forever

Problems with Sleep and Wakeup Probable race condition?

P – producer, C - consumer

1. C reads count, count = 0
2. C checks whether count == 0 or not
3. The next instruction must be executed by C is sleep()
4. Clock interrupt before C goes to sleep

Problems with Sleep and Wakeup Probable race condition?

Context switching, C → P

3. P starts execution
4. P produces an item
5. P put the item into the buffer
6. P increases the count (count = 1)
8. Since count = 1, P wakes up the C.
9. Since C is not sleeping the wake_up signal will be lost.
10. P produces items until the buffer is full, then goes to sleep.

Context switching, P → C

1. The next instruction executed by C is sleep().
2. C goes to sleep.

Synchronization

- **Synchronization:** Process of coordinating two or more activities, devices, or processes in time.
- **Process Synchronization:** means sharing system resources by processes in such a way that, concurrent access to shared data is handled thereby minimizing the chance of inconsistent data.

Producer

$register_1 = \texttt{counter}$
$register_1 = register_1 + 1$
$\texttt{counter} = register_1$

Consumer

$register_2 = \texttt{counter}$
$register_2 = register_2 - 1$
$\texttt{counter} = register_2$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Atomic operations

- To avoid race conditions
- A sequence of one or more machine instructions that are executed sequentially, **without interruption** or **not performed at all**.
- Other names
  - linearizable
  - indivisible
  - uninterruptible

Atomic operations

Producer

$register_1 = \texttt{counter}$
$register_1 = register_1 + 1$
$\texttt{counter} = register_1$

Consumer

$register_2 = \texttt{counter}$
$register_2 = register_2 - 1$
$\texttt{counter} = register_2$

Situation 1

Producer

$register_1 = \texttt{counter}$
$register_1 = register_1 + 1$
$\texttt{counter} = register_1$

Consumer

$register_2 = \texttt{counter}$
$register_2 = register_2 - 1$
$\texttt{counter} = register_2$

Situation 2

Consumer

$register_2 = \texttt{counter}$
$register_2 = register_2 - 1$
$\texttt{counter} = register_2$

Producer

$register_1 = \texttt{counter}$
$register_1 = register_1 + 1$
$\texttt{counter} = register_1$

Interleaving operations are not possible, if Producer and Consumer are implemented atomically

Backup slides

Priority inversion problem (2)