• Producer-consumer problem
  o Introduction
    • Two processes share a fixed-size buffer
    • One process (the producer) puts information into the buffer
    • The other (the consumer) takes the information out
  o Code using busy waiting

```c
#define N 100
int count = 0;

void producer(void)
{
    int item;

    while(1){
        item = produce_item();
        while( count == N );
        insert_item(item);
        count = count + 1;
    }
}

void consumer(void)
{
    int item;
    while(1){
        while( count == 0 );
        item = remove_item();
        count = count - 1;
        consume_item(item);
    }
}
```

  o Alternative – sleep and wakeup
    • Sleep – causes process to block until another process wakes it up
    • Wakeup – wake up another process
    • Uses as a parameter the process that is to be woken up

```c
#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);
        count = count + 1; /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}
```
Race condition exists

- Buffer is empty and consumer reads count as zero
- Producer is then given the CPU before sleep() is called
- Producer adds an item, increments count, and calls wakeup() on the consumer
- Because the consumer has not called sleep(), the wakeup() call is lost
- When consumer runs next, it will immediately sleep.
- Producer will never call wakeup again, will fill up the buffer, then sleep itself
- Both sleep forever

- Using mutual exclusion on count doesn’t help

```c
#define N 100
int count = 0;

void producer(void)
{
    int item;

    while(1){
        item = produce_item();
        enter_region();
        if( count == N ){
            exit_region();
            sleep();
            enter_region();
        }
        insert_item(item);
        count = count + 1;
        if( count == 1 ) wakeup(consumer);
        exit_region();
    }
}

void consumer(void)
{
    int item;
    while(1){
        enter_region();
        if( count == 0 ){
            exit_region();
        }
    }
}
```
sleep();
    enter_region();
}
item = remove_item();
count = count - 1;
if( count == N - 1 ) wakeup(producer);
exit_region();
consume_item(item);
}

- Race condition now
  - Same situation as before, but now consumer is preempted right after calling `exit_region()` and before `sleep()`.
  - Producer still calls `wakeup()` while consumer is not logically asleep.
  - Both eventually sleep forever
- We can’t remain in the critical region while sleeping or producer will never wake up consumer
- Root issue to address: We need to store wakeups so they aren’t lost

- Semaphores
  - Proposed by Dijsktra in 1965
  - Semaphore: integer variable that count the number of wakeups saved for future use
    - Can be zero (no wakeups) or a positive value representing the number of pending wakeups.
  - Two operations
    - Down (like sleep)
      - If semaphore is zero (none pending), process is put to sleep
      - If non-zero (wakeups pending), semaphore is decremented and process continues
    - Up (like wakeup)
      - Never blocks
      - Increments the semaphore
      - If one or more processes were sleeping in `down`, one is chosen at random to complete its `down` call
  - Key – semaphore operations are done as a single atomic action
    - Checking value
    - Changing value (incrementing/decrementing)
    - Possibly going to sleep or waking up another process
  - Producer-consumer example
mutex is a binary semaphore

- Initialized to one
- Processes call `down` before entering their critical region and `up` right after exiting
- Provides mutual exclusion

empty and full provide synchronization

- Don’t provide mutual exclusion, but are used to prevent infinite sleep

```c
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) { /* TRUE is the constant 1 */
        item = produce_item(); /* generate something to put in buffer */
        down(&empty); /* decrement empty count */
        down(&mutex); /* enter critical region */
        insert_item(item); /* put new item in buffer */
        up(&mutex); /* leave critical region */
        up(&full); /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) { /* infinite loop */
        down(&full); /* decrement full count */
        down(&mutex); /* enter critical region */
        item = remove_item(); /* take item from buffer */
        up(&mutex); /* leave critical region */
        up(&empty); /* increment count of empty slots */
        consume_item(item); /* do something with the item */
    }
}
```