Producer/Consumer with Sleep/Wakeup
(Fatally flawed due to Race Condition)

```c
#define N 100  /* no. of slots in buffer */
int count = 0;  /* no. of items in buffer */
void producer() {
    while (TRUE) {
        int item = produce_item();  /* generate next item */
        if (count == N) sleep();  /* go to sleep if buffer is full */
        insert_item(item);  /* store item in buffer */
        count++;  /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);  /* if buffer was empty, wake consumer */
    }
}

void consumer() {
    while (TRUE) {
        if (count == 0) sleep();  /* sleep if nothing in buffer */
        item = remove_item();  /* take item from buffer */
        count--;  /* decrement count of items in buffer*/
        if (count == N-1) wakeup(producer);  /* if buffer was full, wake producer */
        consume_item(item);
    }
}
```
Semaphores (Dykstra, 1965)

```c
int s;
wait(s) {
    while (s == 0) { /* no-op */ }
    s--;
}

signal(s) {
    s++;
}

- wait must be implemented as an *atomic action*
  - typically by the CPU disabling interrupts, and using a TSL instruction on a multiprocessor machine.
- signal is sometimes known as *up*, or V
```

Using Semaphores

A semaphore can be used to provide Mutual Exclusion:

```c
int mutex = 1;

while (TRUE) {
    wait(mutex);
    critical section
    signal(mutex);
    non-critical section
}
```

This is known as a *mutex lock*
Using Semaphores

A semaphore can be used to ensure that one section of code is run before another section, even when they’re in different processes.

- Process 1:

  ```c
  int signal = 0;

  First Section
  signal(synch);
  
  Second Section
  ```

- Process 2:

  ```c
  wait(sync)
  ```

Implementing Semaphores

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

void wait(semaphore S) {
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        sleep();
    }
}

void signal(semaphore S) {
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
}
```

- wait and signal must be atomic.
Implementing a Mutex using TSL
User-space thread implementation

mutex_lock:
    TSL Register, Mutex // copy mutex to Register, set Mutex to 1
    CMP Register, 0    // test if mutex was zero
    JZE ok             // it is was, mutex wasn’t locked, so return
    CALL thread_yield  // mutex is busy: schedule another thread
    JMP mutex_lock     // loop and try again
ok: RTN            // return to caller, and enter critical section

mutex_unlock:
    MOVE Mutex, 0      // clear the lock
    RTN                // return to caller

---

Producer/Consumer using Semaphores

semaphore mutex = 1; /* controls access to critical section*/
semaphore empty = N; /* counts empty buffer slots */
semaphore full = 0;  /* counts full buffer slots */

void producer() {
    while (TRUE) {
        int item = produce_item();
        wait(&empty);
        wait(&mutex);
        insert_item(item);
        signal(&mutex);
        signal(&full);
    }
}

void consumer() {
    while (TRUE) {
        wait(&full);
        wait(&mutex);
        item = remove_item();
        signal(&mutex);
        signal(&empty);
        consume_item(item);
    }
}

Two different uses of semaphores: mutex is used for mutual exclusion. full and empty are used as blocking counters.
Problems with Semaphores (2)

- It’s really hard to use semaphores correctly.
- Easy to mix up `wait()` and `signal()`.
- Easy to omit `signal()`.
  - Especially when handling errors.

- Such bugs may be hard to find, because they only show up under certain race conditions.

Monitors

- Hoare’s response to Dijkstra’s semaphores
  - Higher-level
  - Structured
- Monitors encapsulate data structures that are not externally accessible
  - Mutual exclusive access to data structure enforced by compiler or language run-time.
Monitors

- Monitors are a high-level synchronization primitive:

```java
monitor monitor-name
{
    shared variable declarations
    procedure P1 ( ... ){
        ...
        ...
        procedure Pn ( ... ){
            ...
            {
                initialization code
            }
        }
    }
}
```

**Key points:**
- Shared variables are only accessible through the monitor’s procedures.
- The language performs locking so that only one process can be running any monitor procedure at a time.

Monitor Condition Variables

- A monitor can have condition variables that can be used to add additional synchronization behaviour beyond basic mutual exclusion.
  - `condition x;`
- The only operations that can be performed on condition variables are `wait()` and `signal()`
  - `x.wait()` behaves like `sleep()`
  - `x.signal()` behaves like `wakeup()`, and wakes up the process sleeping on `x`. 
Producer/Consumer using a Monitor
(pseudo code, not real C/Java)

```java
monitor ProducerConsumer {
    condition full, empty;
    int count=0;
    void insert(int item) {
        if (count == N) full.wait();
        insert_item(item);
        count++;
        if (count == 1) empty.signal();
    }

    int remove() {
        if (count == 0) empty.wait();
        int item = remove_item;
        count--;
        if (count == N-1) full.signal();
    }
}

void Producer() {
    while (TRUE) {
        int item = produce_item();
        ProducerConsumer.insert(item);
    }
}

void Consumer() {
    while(TRUE) {
        int item = ProducerConsumer.remove();
        consume_item(item);
    }
}
```

Monitors in Java

- A monitor can easily be implemented in Java:
  - A class in which all the methods are synchronized.
    - Guarantees mutual exclusion.
  - All instance and class variables need to be private or protected.
- No explicit condition variables, but:
  - Use wait() to put thread to sleep, allowing other threads to enter the monitor.
  - Use notify() to wake up a thread that’s waiting on the monitor’s thread queue.
Condition Synchronisation in Java

- Thread enters monitor when it acquires mutual exclusion lock of monitor
- Thread exits monitor when releasing lock
- Wait causes thread to exit monitor

Semaphore as a Java Monitor

class Semaphore {
    private int value_;  
    Semaphore (int initial) {
        value_ = initial;
    }

    public synchronized down() {
        while (value_==0) wait();
        --value;
    }

    public synchronized up() {
        ++value_;  
        notify();
    }
}
Condition Synchronisation in Java

- Although Java has no condition variables, they can be emulated via a loop:

```java
public synchronized void action() throws InterruptedException {
    while (! condition)
        wait();
    do some stuff
    notifyAll();
}
```

- Loop re-tests condition to make sure that it is valid when the thread re-enters the monitor.

Problems with Semaphores

- Semaphores, Petersons’s algorithm, and TSL all assume processes share memory.
  - But processes are normally protected from each other.

- Three solutions:
  - Semaphores may be stored in the kernel, and only accessed through system calls.
  - Use shared-memory segments (available on Unix and Windows).
  - Use a file.

- None of these are nearly as efficient as synchronization between threads in the same process.
Problems with Monitors

- Monitors are a programming language construct.
  - Not only do they need shared memory, but they’re not even available in many languages.
  - Eg. C, C++.

Message Passing

- If two process cannot share memory, then none of the solutions described can work.
- Then you need to use *Message Passing*.
  - This is the general term for a range of different IPC mechanisms.
  - Two primitives:
    ```
    send(destination, &message);
    receive(source, &message);
    ```
  - These are system calls, not language constructs
### Producer/Consumer with Messages

```c
#define N 100

void producer() {
    int item;
    message m;
    while (TRUE) {
        item = produce_item();
        receive(consumer, &m);
        build_message(&m, item);
        send(consumer, &m);
    }
}

void consumer() {
    int item, i;
    message m;
    for (i=0; i<N; i++)
        send(producer, &m);
    while (TRUE) {
        receive(producer, &m);
        item = extract_item(&m);
        send(producer, &m);
        consume_item(&m);
    }
}
```

### Design Issues with Message Passing

- If processes are on different machines, messages can be lost.
  - Use acknowledgements, and resend if a message is lost.
  - Use sequence numbers to distinguish retransmissions.

- If processes are on the same machine, efficiency is key concern.
  - Message passing involves context switching and copying the messages.

- How to identify the recipient?
  - Directly
  - Use a mailbox.

- Buffered mailbox vs unbuffered rendezvous.
**Barriers**

- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through

**Summary**

Many techniques for inter-process synchronization:

*Semaphores:*
  - low level, efficient, shared memory, hard to use right.

*Monitors:*
  - high level language construct, shared memory.

*Message Passing:*
  - asynchronous, low-level, no shared memory.

- Next: Deadlocks.