Monitors Introduction

- Consider the following pseudo-code – What is wrong with it?

```c
semaphore empty = N;
semaphore full = 0;
mutex the_mutex;

void producer(void){
    ...
    while(1){
        mutex_lock(&mutex);
        sem_wait(&empty);
        insert_item(item);
        sem_post(&full);
        mutex_unlock(&mutex);
    }
}

dvoid consumer(void){
    ...
    while(1){
        mutex_lock(&mutex);
        sem_wait(&full);
        insert_item(item);
        sem_post(&empty);
        mutex_unlock(&mutex);
    }
}
```

- Deadlock – if buffer is full, producer will block waiting for empty to be posted with the mutex locked
  - Consumer will never get access to the buffer to post empty, so both processes will block forever
  - Simply flipping the order of mutex_lock and sem_wait caused catastrophic failure

- Proposed solution – Monitors
  - Higher-level synchronization mechanism than mutexes or semaphores
  - Programming language construct, so compiler takes on the burden of guaranteeing mutual exclusion
  - Programmer can ignore the issues associated with synchronization, so code is more likely to be correct
  - Can be viewed as a special type of class
    - Only one process can be active in the monitor at any instant
    - Processes can call monitor methods at will
      - First few (hidden from programmer) instructions check to see if another process is in the monitor
      - Blocks if another process is active
      - Usually uses mutex or binary semaphore
  - Mutual exclusion is easy, but we need method for blocking when they can’t proceed
• Must be able to block the producer when the buffer is full
• Use condition variables
  o When wait is called, other processes are allowed to enter the monitor
  o Avoids deadlock situation

  ▪ We have to handle signal calls carefully because both processes can’t be in the monitor simultaneously
  • Hoare (1974) – Always allow newly awakened process to run and suspend the process that called signal
    o Hard to implement and not conceptually clear how race conditions are avoided
  • Brinch-Hansen (1973) – Require that the process doing the signal must leave the monitor immediately thereafter
    o signal must be the last statement of a monitor procedure
    o Guarantees that only one process is awakened
  • Third solution – allow signaling process to continue running only restart the waiting process after it exits the monitor

• Example pseudo-code for monitor

```pascal
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end:
  count := 0;
end monitor;
```

  o Looks very similar to code that had race condition before
    ▪ Because mutual exclusion guarantees two processes can’t be in the monitor at the same time, wait is guaranteed to complete after checking count

• Producer and Consumer code is very simple then
procedure producer;
begin
    while true do
    begin
        $item = produce_{item}$;
        ProducerConsumer.insert($item$
    end
end;

procedure consumer;
begin
    while true do
    begin
        $item = ProducerConsumer.remove$;
        consume_{item}($item$
    end
end;

- Need programming language support to allow this – Java

- Sample Java Code
  - Global Code
    ```java
    public class ProducerConsumer {
        static final int $N = 100$; // constant giving the buffer size
        static producer $p = new producer();$ // instantiate a new producer thread
        static consumer $c = new consumer();$ // instantiate a new consumer thread
        static our_mon $mon = new our_mon();$ // instantiate a new monitor

        public static void main(String $args[]$) {
            p.start(); // start the producer thread
            c.start(); // start the consumer thread
        }
    }
    ```

  - Monitor Code
    ```java
    static class our_mon { // this is a monitor
        private int buffer[] = new int[N];
        private int count = 0, lo = 0, hi = 0; // counters and indices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
            buffer[hi] = val; // insert an item into the buffer
            hi = (hi + 1) % N; // slot to place next item in
            count = count + 1; // one more item in the buffer now
            if (count == 1) notify(); // if consumer was sleeping, wake it up
        }

        public synchronized int remove() {
            int val;
            if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
            val = buffer[lo]; // fetch an item from the buffer
            lo = (lo + 1) % N; // slot to fetch next item from
            count = count - 1; // one less item in the buffer
            if (count == N - 1) notify(); // if producer was sleeping, wake it up
            return val;
        }

        private void go_to_sleep() { try(wait()); catch(InterruptedException $exc$) {};
    }
    ```
```
The keyword `synchronized` lets the compiler know that those methods should be treated as monitors.

- `wait` and `notify` are used instead of condition variables
  - Don’t suffer from race conditions when used in synchronized methods
  - `wait` could be interrupted, so the exception must be handled

**Producer and Consumer Code**

```java
static class producer extends Thread {
    public void run() { // run method contains the thread code
        int item;
        while (true) { // producer loop
            item = produce_item();
            mon.insert(item);
        }
    }
    private int produce_item() { ... } // actually produce
}

static class consumer extends Thread {
    public void run() { // run method contains the thread code
        int item;
        while (true) { // consumer loop
            item = mon.remove();
            consume_item(item);
        }
    }
    private void consume_item(int item) { ... } // actually consume
}
```

**Message Passing**

- **Introduction**
  - All primitives already discussed assume the ability to share memory in some way
  - What about distributed systems with nodes that each have their own memory?
    - Connected through LAN, etc.
  - Solution – pass messages back and forth over some medium
    - send(destination, &message)
    - receive(source, &message)
  - Many design issues crop up with this type of method
    - Messages can get lost
      - Need some type of acknowledgment/retransmission mechanism
    - Who are you actually getting messages from?
      - Authentication is a big issue
    - Many more than this
      - Read a book on networking if you are interested
Simple solution to producer-consumer with message passing

- Assume all messages are buffered and cannot be lost
- Each slot in the buffer uses a message
  - Consumer sends empty messages to producer
  - Producer sends full messages to consumer
  - Block when there are no messages buffered

```c
#define N 100 /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m; /* message buffer */

    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        send(consumer, &m); /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&m); /* extract item from message */
        send(producer, &m); /* send back empty reply */
        consume_item(item); /* do something with the item */
    }
}
```

- Many other variants that we won’t study