Synchronization

- Monitors

Semaphore Usage

- Each process must call `wait` and `signal` in correct order
- What if we:
  - Replace `wait` with `signal`?
  - Replace `signal` with `wait`?
  - Switch the order of `wait` and `signal`?
- Must also initialize the semaphore correctly

Bounded-Buffer Problem With Semaphores

```c
item buffer[N];
Semaphore mutex=1; Semaphore items=0, slots=N;

Producer:
while(1) {
    produce(item);
    wait(slots);
    wait(mutex);
    insert(item, buffer);
    signal(mutex);
    signal(items);
}

Consumer:
while(1) {
    wait(items);
    wait(mutex);
    remove(item, buffer);
    signal(mutex);
    signal(slots);
    consume(item);
}
```

- What if we switched `wait(slots)` and `wait(mutex)`?
Problems with Semaphores

- Programming complexity
  - Difficult to use correctly
  - Programmer has to be skilled in their usage, reason about the code
- Debugging difficulty
  - Errors may not be reproducible
  - Depends on order of execution
- Would like: high-level synchronization construct
  - Don't have to worry about explicitly programming in synchronization

Monitors

- High-level synchronization constructs
  - Provided at language level
  - Actual mechanisms hidden from programmer
  - Easier to focus on application-level semantics

Monitor: Definition

- Abstract data type (ADT)
  - Encapsulates private data
  - Public methods provided to operate on data
- Monitor is an ADT:
  - Encapsulates shared variables
  - Ensures mutual exclusion among its procedures
- Only one process can be active inside the monitor at a time

Monitor Syntax

```c
Monitor monitor_name
{
    //shared variable declarations
    ...
    procedure P1(...) {
        ...
    }
    procedure P2(...) {
        ...
    }
    ...
    initialization(...) {
        ...
    }
}
```
Monitor Execution

- At most one process executing inside the monitor
- Monitor has an entry queue of processes
  - Processes waiting to enter the monitor
- When executing process exits:
  - A process is selected from the entry queue

Producer-Consumer Problem With Monitors

```
Producer:    Consumer:
while(1)    while(1)
{          
    produce(item);
    ProducerConsumer.insert(item);

    ProducerConsumer.remove(item);
    consume(item);
}
```

Producer-Consumer Problem With Monitors: Attempt 1

```
Monitor ProducerConsumer
{
    buffer_type buffer;
    integer count;
    procedure insert(item) {
        insert_item(item, buffer);
        count++;
    }
    procedure remove(item) {
        item=remove_item(buffer);
        count--;
    }
    initialization() { count=0; }
}
```

Problem

- We can achieve mutual exclusion
- But how to wait for a specific condition?
  - E.g.: wait on full or empty buffer?
- Need another mechanism
Condition Variables

- Provide a way to wait and signal a condition
- Operations: wait and signal
  - Each cond. var. has a wait queue
- wait: Process is suspended and immediately leaves the monitor
- signal: One suspended process is woken up
  - No-op if no process waiting on cond. var.
- signalAll: Wakes up all processes waiting on cond. var.
  - Broadcast signal

Bounded Buffer Problem With Monitors and Condition Variables

```java
Buffer_type buffer;
integer count;   condition item_avail, slot_empty;

procedure insert(item) {
    if (count==N)
        slot_empty.wait();
    insert_item(item, buffer);
    count++;
    if (count==1)
        item_avail.signal();
}

procedure remove(item) {
    if (count==0)
        item_avail.wait();
    item=remove_item(buffer);
    count--;
    if (count==N-1)
        slot_empty.signal();
}
```

Condition Variable Semantics

- When a process P signals and Q is woken up:
  - Who gets to run in the monitor?
- Signal and wait (Hoare style): P blocks until Q leaves the monitor
- Signal and continue (Mesa style): Q blocks until P leaves the monitor
  - What about the condition Q was waiting on?
- Signal and return: P exits monitor (returns from procedure) immediately after signal

Monitor Implementation

- Language-provided construct
  - Compiler has to enforce mutual exclusion
- Can be implemented using semaphores
- Some languages provide monitors:
  - Concurrent Pascal, C#, Java
- Monitors in Java:
  - Synchronized methods
  - Wait and notify methods