Synchronization

- Monitors
- Other Approaches

Semaphore Usage

- `wait(S)`
- Critical section
- `signal(S)`

- 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

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

Producer:

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

Consumer:

```c
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

```plaintext
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

Monitor ProducerConsumer;

Producer:

```c
while(1) {
    produce(item);
    ProducerConsumer.insert (item);
}
```

Consumer:

```c
while(1) {
    ProducerConsumer.remove (item);
    consume(item);
}
```

Producer-Consumer Problem With Monitors: Attempt 1

```c
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

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
Other Synchronization Approaches

- Transactional Memory
- Language annotations (OpenMP)
- Functional programming languages

Transactional Memory

- Applies idea of transactions to memory reads/writes
  - A set of operations is carried out as a transaction
  - Committed: if no conflicts during execution
  - Aborted: if read/write conflicts

```c
atomic {
  read (x);
  write (y);
  ...
}
```

Transactional Memory: Usage

- Transactions specified using language or library support
  - Compiler/runtime system/hardware implements the underlying concurrency/synchronization
- Lock-free shared data
  - Each thread reads/writes data concurrently
  - Values may have to be rolled back if conflict on memory locations

Transactional Memory: Benefits

- Makes programming easier
- Avoids common problems such as deadlocks, priority inversion
- Better performance than locks (often):
  - Optimistic concurrency control
  - Fine-grained synchronization (memory address/word-level)
**Transactional Memory: Implementation**

- Software Tran. Mem. (STM):
  - Implemented in software
  - Requires compiler/runtime support
  - Support in Java, C, C++
- Hardware Tran. Mem. (HTM):
  - Implemented in the hardware
  - Extended cache coherence protocols

**OpenMP**

- Supports language annotations to specify synchronization constructs such as critical sections
  - E.g.: #pragma omp critical
- Compiler generates code for mutual exclusion
- However:
  - Programmer still responsible for identifying critical sections in the code

**Functional Programming Languages**

- Do not have mutable state
  - Variables cannot be overwritten (or modified in place)
  - Every operation depends only on arguments, has no side effects
  - E.g.: add(list, node) will create and return a new list (not modify list in place)
- Do not suffer from synchronization problems
- E.g.: Lisp, Scheme, Erlang, Scala