Indirect Communication

Arnav Gupta

November 16, 2024

Contents

1 Indirect Communication

`P` and `V` are low-level primitives for protecting critical sections and establishing synchronization between tasks.

Higher level facilities can perform info communication and mutual exclusion automatically using help from the programming language and compiler.

2 Critical Regions

Declare which variables are shared with syntax like `VAR v : SHARED INTEGER`.

Access to shared variables is restricted to within a region statement, and within the region, mutual exclusion is guaranteed:

```
REGION v DO
// critical section
END REGION
```

This does not allow simultaneous reads.

Can modify to allow reading of shared variables outside the critical region, but this could bring issues with reading partially updated information that another task is updating inside the critical region.

Using nesting to attempt to fix this can result in deadlock.

3 Conditional Critical Regions

Can fix issues with critical regions using a condition that must be true as well as having mutual exclusion:

```
REGION v DO
    AWAIT conditional-expresson
// critical section
END REGION
```

If the condition is false, the region lock is released and entry is started again (busy waiting). To prevent busy waiting, block on the queue for the shared variable, and on region exit, search for true conditional-expression and unblock.

4 Monitor

Monitor: abstract data type that combines shared data with serialization of its modification

```
_Monitor name {
    // shared data
    // members that see and modify the data
};
```

Mutex Member: a member that does not begin execution if there is another active mutex member

A call to a mutex member may become blocked while awaiting entry, and queues of waiting tasks may form. Public member routines of a monitor are implicitly mutex. Others can be made mutex with `_Mutex`.

This works like if the monitor had a lock `P`'d on entry to a monitor member and `V`'d on exit.

Recursive entry is allowed, one mutex member can call another or itself.

Unhandled exceptions raised within a monitor should always release the implicit monitor locks so the monitor can continue to function.

The destructor must be mutex so that closing a monitor blocks if a thread is in the monitor.

5 Scheduling (Synchronization)

A monitor may want to schedule tasks in an order different than arrival.

The two techniques are:

- **external**: scheduling tasks outside the monitor and is accomplished with the accept statement
- **internal**: scheduling tasks inside the monitor and is accomplished using condition variables with signal and wait

External scheduling is easier to specify and explain over internal with condition variables.

External scheduling cannot be used if:

- scheduling depends on member param values

- scheduling must block in the monitor but cannot guarantee the next call fulfills cooperation

5.1 External Scheduling

`_Accept` controls which mutex members can accept calls. By preventing members from accepting calls at times, it is possible to control task scheduling.

Each `_Accept` defines what cooperation must occur for the accepting task to proceed. Queues of tasks form outside the monitor, waiting to be accepted.

An acceptor blocks all calls except a call to the specified mutex member(s). The accepted call is executed like the usual member call. When the accepted task exits the mutex member, the acceptor continues.

Can satisfy acceptor's requirements with multiple calls.

With external scheduling, unblocking is implicit.

5.2 Internal Scheduling

Scheduling among tasks inside the monitor.

Condition: external synchronization lock

- `empty` returns false if there are tasks blocked on the queue and true otherwise
- `front` returns an integer value stored with the waiting task at the front of the condition queue
- `wait` blocks the current thread and restarts a signalled task or implicitly releases the monitor lock
- `signal` unblocks the thread on the front of the condition queue after the signaller thread blocks or exits
- `signalBlock` unblocks the thread on the front of the condition queue and blocks the signaller thread

A task waits by placing itself on a condition, which atomically places the executing task at the back of the condition queue and allows another task into the monitor by releasing the monitor lock.

A task is made ready by signalling the condition, which removes and makes ready the blocked task at the front of the condition queue.

The signaller does not block, so the signalled task must continue waiting until the signaller exits or waits.

Entry queue: FIFO lst of calling tasks to the monitor

6 Readers/Writers

Can write similar to `P` and `V`, but can also simplify protocol to have a single read and a single write action.

Can simplify further by having writers use `_Mutex` in monitor (let monitor handle mutual exclusion).

Can solve with condition shadow queue or with external scheduling.

7 Exceptions

An exception raised in a monitor member propagates to the caller's thread.

On exiting a method that raised an exception, a caller implicitly raises a non-local `RendezvousFailure` exception at monitor acceptor's thread to identify failed cooperation.

For multiple `_Accept` clauses, a flag variable is required to know which member failed.

8 Nested Monitor Calls

Nested monitor problem: acquire monitor (lock), call to another monitor, and wait on condition in the other monitor, so potential deadlock

Releasing all locks can inadvertently release a lock. This is called the **lock composition** problem.

9 Intrusive Lists

Non-contiguous variable-length data structures normally require dynamic allocation as the structure increases/decreases as the structure increases/decreases when adding/deleting nodes.

The types of collections are copy data, copy pointer, and intrusive pointers.

Copy creates a collection node with link fields, with dynamic allocation for links and possibly data, copying data and/or data pointer into nodes, and then linking the node into collection.

Intrusive assumes a node with data and link fields, so no dynamic allocation for collection links or copying.

A programmer usually manages node lifetime for copy pointer and intrusive.

μC++ provides **intrusive data structures** allowing global/stack/heap nodes and no copying. This implementation uses private intrusive links for non-copyable objects like a coroutine or task.

Intrusive links can have:

- one link field, `uColable`, for a collection
- two link fields, `uSeqable`, for a sequence

Template classes `uStack` and `uQueue` are singly linked, so collections. Template class `uSequence` is doubly linked, so sequence.

Each intrusive list has associated iterators, `uStackIter`, `uQueueIter`, and `uSeqIter`.

Lifetime of node is duration of blocked thread.

10 Counting Semaphore vs Condition

`P` only blocks if semaphore is 0, `wait` always blocks.

`V` before `P` affects `P`, `signal` before `wait` is lost.

Multiple `V` can start multiple tasks simultaneously, while multiple signals only start one task at a time because each task must exit serially through the monitor.

Can simulate `P` and `V` using a monitor.

11 Monitor Types

Explicit scheduling occurs when:

- an accept statement blocks the active task on the acceptor stack and makes a task ready from the specified mutex member queue
- a signal moves a task from the specified condition to the specified stack

Implicit scheduling occurs when a task waits in or exits from a mutex member, and a new task is selected first from the acceptor/signalled stack, then the entry queue.

Monitors are classified by the implicit scheduling (who gets control) of the monitor when a task waits or signals or exits.

Implicit scheduling can select from the calling (C), signalled (W), and signaller (S) queues.

Assigning different relative priorities to these queues creates different monitors.

For $C < W < S$ and $C < S < W$, this creates a useful monitor with prevention and no barging.

For $C = W < S$ and $C = S < W$, this creates a useable monitor that needs avoidance and barging, starvation without avoidance.

The remaining monitors are rejected, either for being confusing or unsound.

Monitors either have an explicit signal or an implicit signal (automatic). The implicit signal monitor has no condition variables or explicit signal statement. Instead, there is a `waitUntil` statement.

Immediate-return signal: additional restricted monitor-type requiring the signaller exit immediately from the monitor (not powerful enough for all cases, but optimizes common cases)

A no-priority blocking monitor requires the signaller task to recheck the waiting condition in case of a barging task (use while loop around signal).

A no-priority non-blocking monitor requires the signalled task to recheck the waiting condition in case of a barging task (use while loop around wait).

A monitor with implicit signalling is good for prototyping but has poor performance.

A priority non-blocking monitor has no barging and optimizes signal before return (supply cooperation).

A priority blocking monitor has no barging and handles internal cooperation within the monitor (wait for cooperation).

A `_Cormonitor` is a coroutine with implicit mutual exclusion on calls to specified member routines. This means the coroutine can be used my multiple threads.

12 Java Monitor

Java has `synchronized` class members and a `synchronized` statement.

All classes have one implicit condition variable and `wait`, `notify`, and `notifyAll` to manipulate.

Internal scheduling is no-priority unblocking, so barging: wait statements must be in while loops to recheck conditions.

One condition queue makes certain solutions difficult or impossible.

Spurious wakeup: wakeup when a task is not supposed to wake up

Problems related to spurious wakeup can be solved using generations.

Cannot build condition variables in Java with nested monitors since issues with deadlocks arise from monitor locks not being released.