Direct Communication

Arnav Gupta

November 25, 2024

Contents

1 Direct Communication

Monitors work well for passive objects that require mutual exclusion because of sharing, but have indirect communication.

Tasks can communicate directly by calling each other's member routines.

2 Task

A task is like a coroutine since it has a distinguished member, `main`, which has its own execution state.

Tasks are unique from their thread of control, which begins execution when the task is created. Tasks provide mutual exclusion and synchronization so only one thread is active in the object:

- public members of a task are implicitly mutex and other members can be made mutex
- external scheduling allows direct calls to mutex members (task's thread blocks while caller executes)
- without external scheduling, tasks must call out to communicate (third party or emulate external scheduling with internal)

In general, basic execution properties produce different abstractions:

- no thread or stack is either a class or a monitor
- no thread but has stack is either coroutine or coroutine-monitor
- thread but no stack does not occur
- thread and stack can only be a task

When the thread or stack are missing, comes from calling object.

Abstractions are derived from basic properties and each abstraction has a set of problems it can solve.

3 Scheduling

A task may wants to schedule access to itself by other tasks in an order different from the order in which requests arrive.

3.1 External Scheduling

The `_Accept` statement can be used to control which mutex members of a task can accept calls.

`_When` clause is like the condition of conditional critical region: the condition must be true and a call to the specified member must exist before a member is accepted.

If all accepts are conditional and false, the statement does nothing.

If some conditionals are true, but no outstanding calls, the acceptor is blocked until a call to an appropriate member is made.

If several members are accepted and outstanding calls exist to them, a call is selected based on the order of the `_Accept` clauses (so order indicates priority). There is potential starvation here.

Must ensure that for every true conditional, only corresponding members are accepted.

The acceptor is pushed on top of the acceptor/signaled stack and normal implicit scheduling occurs ($C < W < S$).

Once the accepted call completes or the caller `wait`, the statement after the accepting `_Accept` clause is executed and the accept statement completes.

If there is a terminating `_Else` clause and no `_Accept` can be executed immediately, the terminating `_Else` clause is executed, so it allows the call to accepted without the acceptor blocking.

3.2 Internal Scheduling

Scheduling among tasks inside the monitor, condition with `signal` and `wait` are used.

Requires a combination of internal and external scheduling.

Rendezvous is logically pending when `wait` restarts `_Accept` task, but post `_Accept` statement is still executed (no RendezvousFailure). The acceptor must eventually complete rendezvous for waiting caller.

Signalled tasks cannot leave because the task continues in the monitor. Signal-blocked tasks leave immediately because the task blocks.

3.3 Accepting the Destructor

A common way to terminate a task is to have a stop member that is called when the task must stop.

```
for ( ;; ) {
    _Accept( stop ) { break; } // terminate
}
```

However, if termination and deallocaton follow one another, it may be better just to accept the destructor.

The semantics for accepting a destructor are different from accepting a normal mutex member:

- when the call ot the destructor occurs, the caller blocks immediately if there is a thread active in the task, since a task's storage cannot be deallocated while in use
- when the destructor is accepted, the caller is blocked and pushed onto the A/S stack instead of the acceptor
  - control restarts at the accept statement without executing the destructor member to allow the mutex object to clean up before termination.
- the task now behaves like a monitor since its thread is halted
- only when the caller to the destructor is popped off the A/S stack by the implicit scheduling is the destructor executed
- the destructor can then reactivate any blocked tasks on condition variables and/or the A/S stack

4 Increasing Concurrency

Two tasks are involved in direct communication: the client (caller) and server (callee). Concurrency can be increased on both sides.

4.1 Server Side

The server manages a resource, and the server thread should introduce additional concurrency (assuming no return value).

```
// NO CONCURRENCY
_Task server {
    public:
        void mem1( ... ) { S1 }
        void mem2( ... ) { S2 }
        void main() {
            ...
            _Accept( mem1 );
            or _Accept( mem2 );
            // server blocked while
            // client does work
        }
}

// SOME CONCURRENCY
_Task server {
    public:
        void mem1( ... ) { S1.copy_in }
        void mem2( ... ) { S2.copy_out }
        void main() {
            ...
            _Accept( mem1 ) { S1.work }
            or _Accept( mem2 ) { S2.work };
            // client blocks in member, then
            // server does work and unblocks
            // client
        }
}
```

Concurrency is possible here if the service can be factored into administrative and work code, so moving code from the member to the statement executed after member is accepted.

There is a small overlap between the client and server since the client gets away earlier, increasing concurrency.

4.1.1 Internal Buffer

Use a larger internal buffer to allow clients to get in and out of the server faster, where the internal buffer is used to store the arguments of multiple clients until the server processes them.

Issues:

- unless the average time for production and consumption is approx. equal with only small variance, the buffer is either always full or empty
- because of the mutex property of a task, no calls can occur while the server is working, so clients cannot drop off their arguments
  - server could periodically accept calls while processing requests (but this is awkward)

- clients may need to wait for replies, in which case a buffer does not help unless there is some advantage to processing requests in non-FIFO order

Only way to free server's thread to receive new requests and return finished results to clients is by adding another thread $\to$ **worker thread**.

The worker thread calls the server to get work from the buffer and return results from the buffer.

Number of workers has to balance with the number of clients to maximize concurrency (bounded-buffer problem).

4.1.2 Administrator

A server managing multiple clients and worker tasks. Administrator only manages: delegating work to others, receiving and checking completed work, passing completed work on.

Administrator only called by others, so it is always accepting calls.

Administrator makes no call to another task (otherwise, this will block administrator).

Administrator maintains a list of work to pass to worker tasks.

Some typical workers include:

- **timer**: prompt administrator at time intervals
- **notifier**: perform a potentially blocking wait for an external event
- **simple worker**: do work and return result to administrator
- **complex worker**: do work and interact directly with client
- **courier**: perform blocking call on behalf of administrator

4.2 Client Side

Not all servers attempt to shorten client delay.

In some cases, a client may not have to wait for a server to process a request, since it can be accomplished with an asynchronous call.

Asynchronous call requires implicit buffering between client and server to store the client's arguments from the call.

It is possible to build asynchronous facilities out of the synchronous ones and vice versa.

4.2.1 Returning Values

If a client only drops off data to be processed by the server, the asynchronous call is simple.

If a result is returned from the call, the call must be divided into two calls:

```
callee.start( arg );
// callee performs other
// work simultaneously
result = callee.wait(); // obtain result
```

Not the same as START/WAIT since server thread exists (many-to-one vs one-to-one).

Time between calls allows calling task to execute asynchronously with task performing operation on the caller's behalf.

If the result is not ready in time, the caller either blocks or must poll. This requires a protocol so when the client makes the second call, the correct result can be found and returned.

4.2.2 Tickets

First part of the protocol transmits the arguments specifying the desired work and a ticket is returned immediately.

The second call pulls the result by passing the ticket. The ticket is matched with a result that is returned if available or the caller is blocked or polls until the result is available.

Error prone since the caller may not obey protocol (never retrieve result, use same ticket twice, forged ticket).

4.2.3 Call-Back Routine

Transmit (register) a routine on the initial call.

When the result is ready, the routine is called by the task generating the result, passing it the result.

The call-back routine cannot block the server, only store the result and set an indicator known to the client (like a semaphore).

The original client must poll th indicator or block until the indicator is set.

Allows the server to push the result back to the client faster (nagging client to pickup).

Client can write the call-back routine, so they can decide to poll or block or do both.

4.2.4 Futures

Provides the same asynchrony, but without an explicit protocol. This removes the problem of when the caller should try to retrieve the result.

A future is an object that is a subtype of the result type expected by the caller. A single call is made that returns the future, which is empty.

The caller continues execution and at some time in the future, the result is calculated and the future is filled.

If the caller tries to use the future before its value is filled in, the caller is implicitly blocked.

```
class Future : public ResultType {
        friend _Task server; // allow server to access
                             // internal state
        ResultType result;   // place result here
        uSemaphore avail;    // wait here if no result
        Future * link;       // intrusive data structure
    public:
        Future() : avail( 0 ) {}

        ResultType get() {
            avail.P();       // wait for result
            return result;
        }
};
```

In languages without garbage collection, the future must be explicitly deleted.

In μC++, two types of template futures are:

- `Future_ESM<T>` which must be allocated and deallocated explicitly by the client
- `Future_ISM<T>` which automatically allocates and frees storage (GC), simpler to use but less efficient
  - `available` returns true if asynchronous call is completed (result available, server raise exception, or call canceled) and false otherwise
  - `operator()` returns read-only copy of future result, blocking if the future is unavailable and raising an exception if one was returned by the server (can be re-retrieved)
  - `reset` marks the future as empty for reuse
  - `cancel` attempts to cancel the asynchronous call the future refers to (waiting clients are unblocked)
  - `cancelled` returns true if the future is canceled and false otherwise
  - `delivery(T result)` copies the client result into the future, unblocking clients waiting for the result
  - `delivery( uBaseEvent * cause )` copies an exception into the future and the exception is thrown at waiting clients (exception must be dynamically allocated)

On the client:

- after the future result is retrieved, it can be retrieved again cheaply (no blocking)
- be careful of deadlocks like with `osacquire`

The **select statement** waits for 1+ heterogeneous futures based on logical selection criteria. The selector expression must be satisfied before execution continues.

A `_Select` clause may be guarded with a logical expression and have code executed after a future received a value. Each select clause action is executed when its sub-selector expression is satisfied, but control does not continue until the selector expression associated with the entire statement is satisfied.

An action statement is triggered only once for its selector expression, even if the selector expression is compound.

A select statement can be non-blocking using a terminating `_Else` clause.