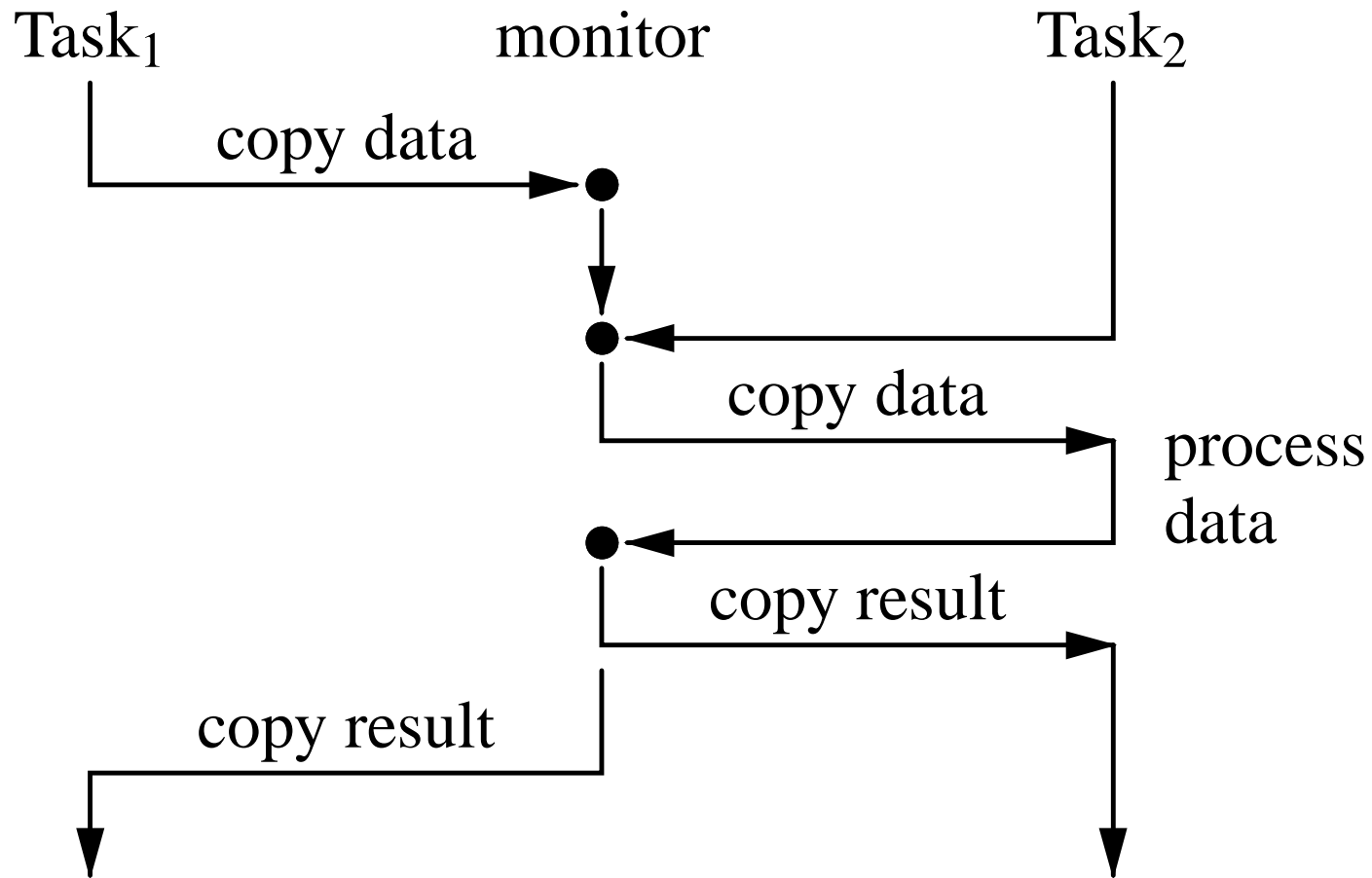


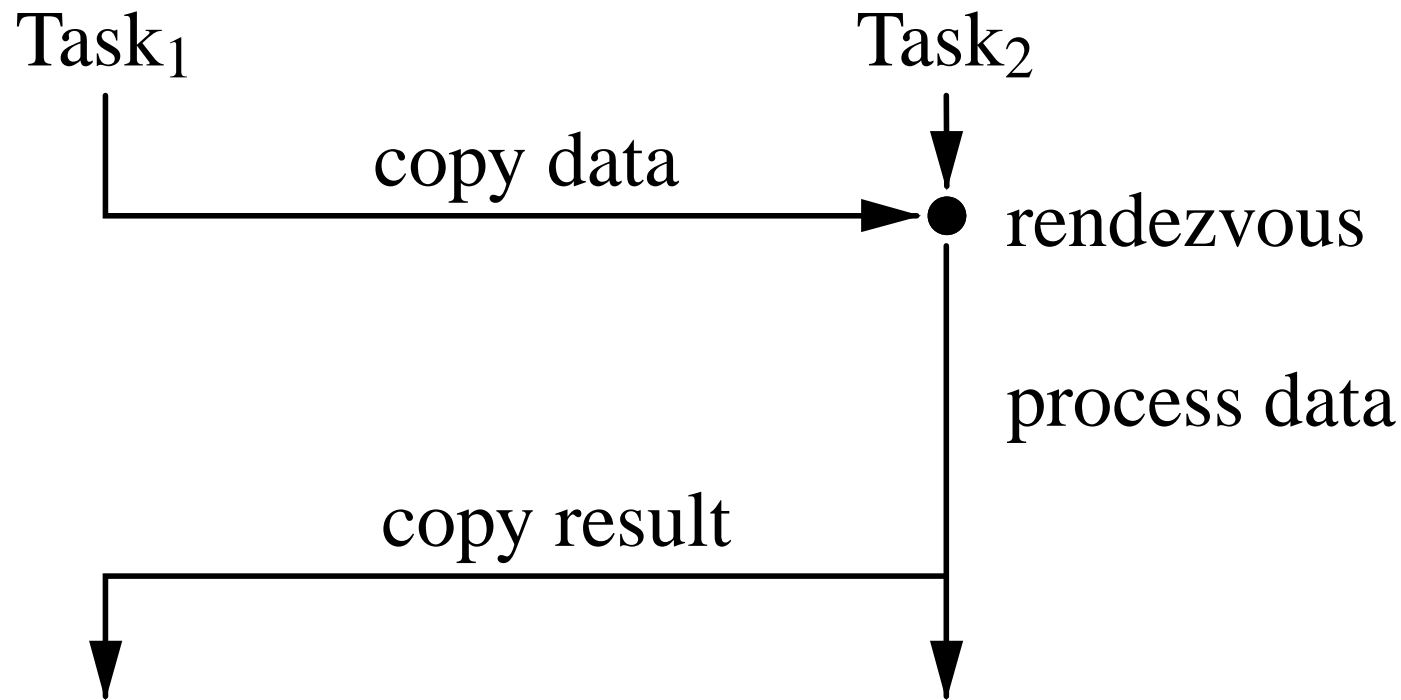
7 Direct Communication

- Monitors work well for passive objects that require mutual exclusion because of sharing.
- However, communication among tasks with a monitor is indirect.
- Problem: point-to-point with reply indirect communication:

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.



- Point-to-point with reply direct communication:



- Therefore, tasks need to communicate directly.

7.1 Task

- A task is like a monitor, because it provides mutual exclusion and can also perform synchronization.
 - Public members of a task are implicitly mutex and other kinds of members can be made explicitly mutex.

- A task is also like a coroutine, because it has a distinguished member, (task main), which has its own execution state.
- A task is unique because it has a thread of control, which begins execution in the task main when the task is created.
- In general, basic execution properties produce different abstractions:

object properties		member routine properties	
thread	stack	No ME/S	ME/S
No	No	1 class	2 monitor
No	Yes	3 coroutine	4 coroutine-monitor
Yes	No	5 reject	6 reject
Yes	Yes	7 reject?	8 task

- When thread or stack is missing it comes from calling object.
- Abstractions are not ad-hoc, rather derived from basic properties.
- Each of these abstractions has a particular set of problems it can solve, and therefore, each has a place in a programming language.

7.2 Scheduling

- A task may want to schedule access to itself by other tasks in an order different from the order in which requests arrive.
- As for monitors, there are two techniques: external and internal scheduling.

7.2.1 External Scheduling

- As for a monitor, the accept statement can be used to control which mutex members of a task can accept calls.

```

_Task BoundedBuffer {
    int front, back, count;
    int Elements[20];
public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }
    void insert( int elem ) {
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
    }
    int remove() {
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        return elem;
    }
protected:
    void main() {
        for ( ;; ) {
            _When (count != 20) _Accept(insert) { // after call
            } or _When (count != 0) _Accept(remove) { // after call
            } // _Accept
        } // for
    }
};

```

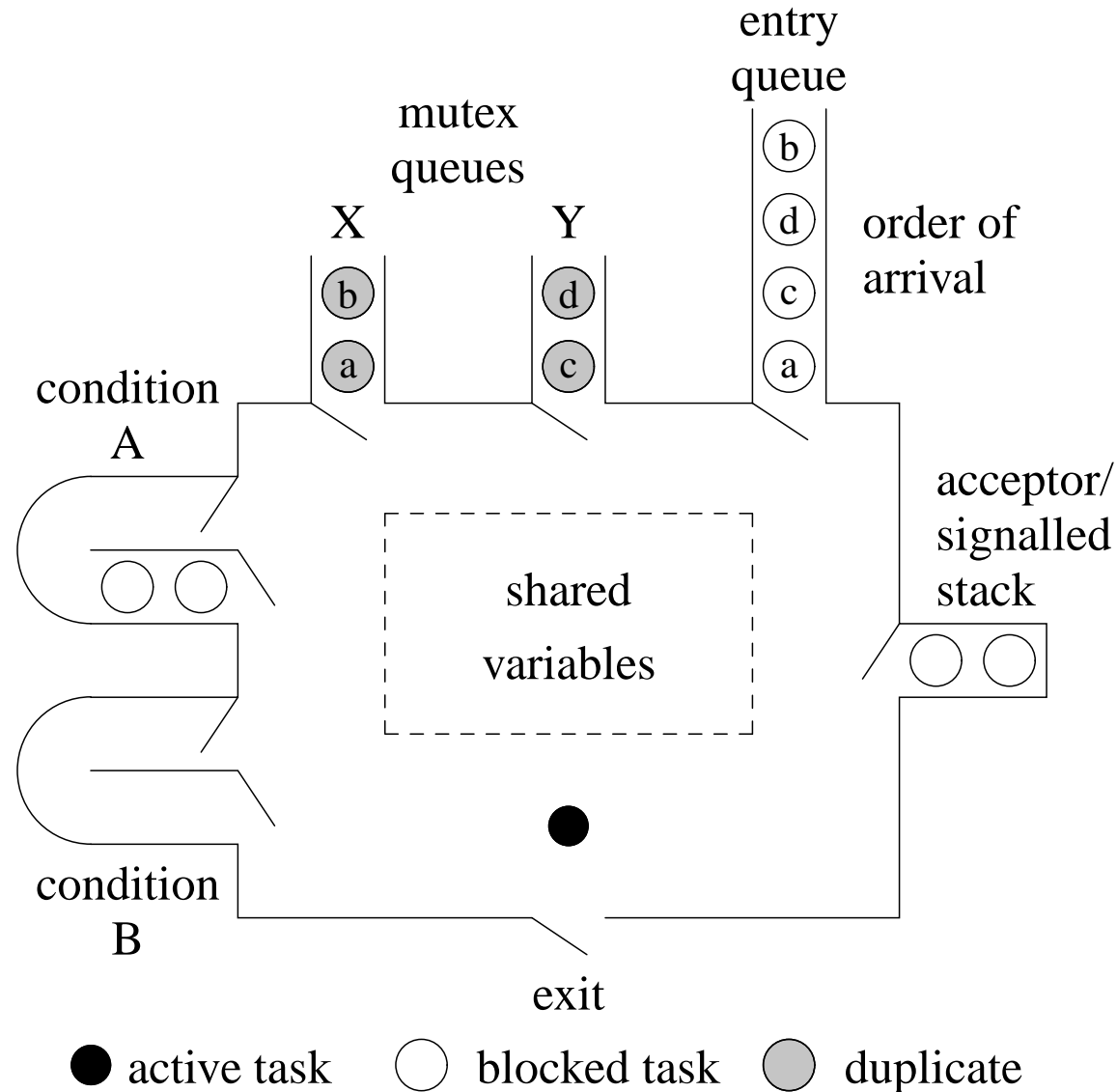
- **_Accept**(m1, m2) S1 \equiv **_Accept**(m1) S1; **or** **_Accept**(m2) S1;
if (C1 | C2) S1 \equiv **if** (C1) S1; **else if** (C2) S1;
- Extended version allows different **_When** and code after call for each accept.
- Equivalence using **if** statements:


```

if ( 0 < count && count < 20 ) _Accept( insert, remove );
else if ( count < 20 ) _Accept( insert );
else /* if ( 0 < count ) */ _Accept( remove );

```
- $2^N - 1$ **if** statements needed to simulate N accept clauses.
- Why is BoundedBuffer::main defined at the end of the task?
- The **_When** clause is like the condition of conditional critical region:
 - The condition must be true (or omitted) *and* a call to the specified member must exist before a member is accepted.
- If all the accepts are conditional and false, the statement does nothing (like **switch** with no matching **case**).
- If some conditionals are true, but there are no outstanding calls, the acceptor is blocked until a call to an appropriate member is made.

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



- If several members are accepted and outstanding calls exist to them, a call is selected based on the order of the **_Accepts**.
 - Hence, the order of the **_Accepts** indicates their relative priority for selection if there are several outstanding calls.
- Once the accepted call has completed, the statement after the accepting **_Accept** clause is executed.
 - To achieve greater concurrency in the bounded buffer, change to:

```

void insert( int elem ) {
    Elements[back] = elem;
}
int remove() {
    return Elements[front];
}
protected:
void main() {
    for ( ;; ) {
        _When ( count != 20 ) _Accept( insert ) {
            back = (back + 1) % 20;
            count += 1;
        } or _When ( count != 0 ) _Accept( remove ) {
            front = (front + 1) % 20;
            count -= 1;
        } // _Accept
    } // for
}

```

- If there is a terminating **else** clause and no **_Accept** can be executed immediately, the terminating **else** clause is executed.

```

_Accept( ... ) {
} or _Accept( ... ) {
} else { ... } // executed if no callers

```

- Hence, the terminating **else** clause allows a conditional attempt to accept a call without the acceptor blocking.
- An exception raised in a task member propagates to the caller, and a special exception is raised at the task's thread to identify a problem.

```

_Task T {
public:
    void mem() {
        ... _Throw E; ... // E goes to caller
    } // uRendezvousFailure goes to "this"
private:
    void main() {
        try {
            _Enable {
                ... _Accept( mem ); ... // assume call completed
            }
        } catch ( uSerial::RendezvousFailure ) {
            // deal with rendezvous failure
        } // try
    }
};

```

7.2.2 Accepting the Destructor

- Common way to terminate a task is to have a done (join) member:

```

_Task BoundedBuffer {
public:
    ...
    void done() { /* empty */ }
private:
    void main() {
        // start up
        for ( ;; ) {
            _Accept( done ) {           // terminate ?
                break;
            } or _When ( count != 20 ) _Accept( insert ) {
                ...
            } or _When ( count != 0 ) _Accept( remove ) {
                ...
            } // _Accept
        } // for
        // close down
    }
}

```

- Call done when task is to stop:

```
void uMain::main() {
    BoundedBuffer buf;
    // create producer & consumer tasks
    // delete producer & consumer tasks
    buf.done();           // no outstanding calls to buffer
    // maybe do something else
} // delete buf
```

- Alternatively, throw a concurrent exception, but delayed deliver.
- If termination and deallocation follow one another, accept destructor:

```
void main() {
    for ( ;; ) {
        _Accept( ~BoundedBuffer ) {
            break;
        } or _When ( count != 20 ) _Accept( insert ) { ...
        } or _When ( count != 0 ) _Accept( remove ) { ...
        } // _Accept
    } // for
}
```

- However, the semantics for accepting a destructor are different from accepting a normal mutex member.
- When the call to the destructor occurs, the caller blocks immediately if there is thread active in the task because 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*.
- Therefore, control restarts at the accept statement *without* executing the destructor member.
- This allows a mutex object to clean up before it terminates (monitor or task).
- At this point, the task behaves like a monitor because 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 reactivate any blocked tasks on condition variables and/or the acceptor/signalled stack.

7.2.3 Internal Scheduling

- Scheduling among tasks inside the monitor.
- As for monitors, condition, signal and wait are used.

```
_Task BoundedBuffer {  
    uCondition NonEmpty, NonFull;  
    int front, back, count;  
    int Elements[20];  
public:  
    BoundedBuffer() : front(0), back(0), count(0) {}  
  
    _Nomutex int query() { return count; }  
  
    void insert( int elem ) {  
        if ( count == 20 ) NonFull.wait();  
        Elements[back] = elem;  
        back = ( back + 1 ) % 20;  
        count += 1;  
        NonEmpty.signal();  
    }  
}
```

```

int remove() {
    if ( count == 0 ) NonEmpty.wait();
    int elem = Elements[front];
    front = ( front + 1 ) % 20;
    count -= 1;
    NonFull.signal();
    return elem;
}
protected:
void main() {
    for ( ;; ) {
        _Accept( ~BoundedBuffer )
            break;
        or _Accept( insert, remove );
        // do other work
    } // for
}
};

```

- Is there a potential starvation problem?

7.3 Increasing Concurrency

- 2 task involved in direct communication: client (caller) & server (callee)
- possible to increase concurrency on both the client and server side

7.3.1 Server Side

- Use server thread to do administrative work so client can continue concurrently (assuming no return value).
- E.g., move administrative code from the member to the statement executed after the member is accepted:

```

_Task server1 {
  public:
    void xxx(...) { S1 }
    void yyy(...) { S2 }
    void main() {
      ...
      _Accept( xxx );
      or _Accept( yyy );
    }
}

```

```

_Task server2 {
  public:
    void xxx(...) { S1.copy }
    void yyy(...) { S2.copy }
    void main() {
      ...
      _Accept( xxx ) { S1.admin }
      or _Accept( yyy ) { S2.admin };
    }
}

```

- overlap between client and server increases potential for concurrency

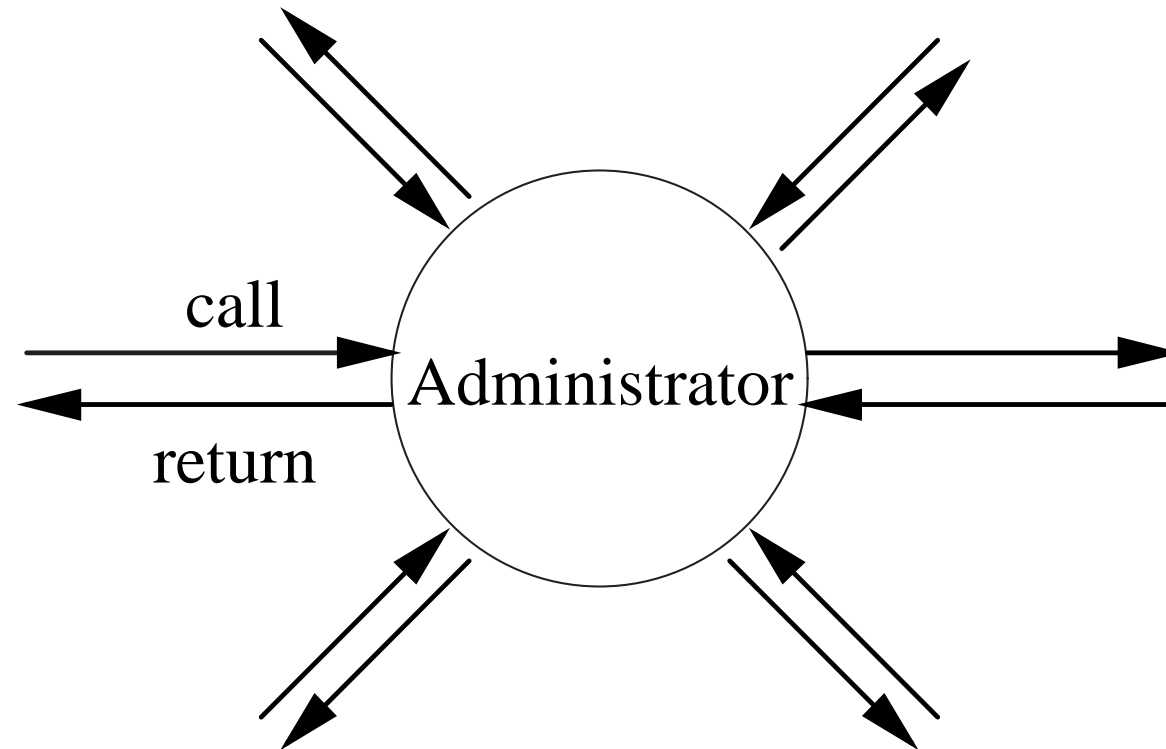
7.3.1.1 Internal Buffer

- The previous technique provides buffering of size 1 between the client and server.
- Use a larger internal buffer to allow clients to get in and out of the server faster?
- I.e., an internal buffer can be used to store the arguments of multiple clients until the server processes them.
- However, there are several issues:
 - Unless the average time for production and consumption is approximately equal with only a 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. The server could periodically accept calls while processing requests from the buffer (awkward).
 - Clients may need to wait for replies, in which case a buffer does not help unless there is an advantage to processing requests in non-FIFO order.

- These problems can be handled by changing the server into an administrator.

7.3.1.2 Administrator

- An **administrator** is used to manage a complex interaction or complex work or both.
- The key is that an administrator does little or no “real” work; its job is to manage.
- Management means delegating work to others, receiving and checking completed work, and passing completed work on.
- An administrator is called by others; hence, an administrator is always accepting calls.

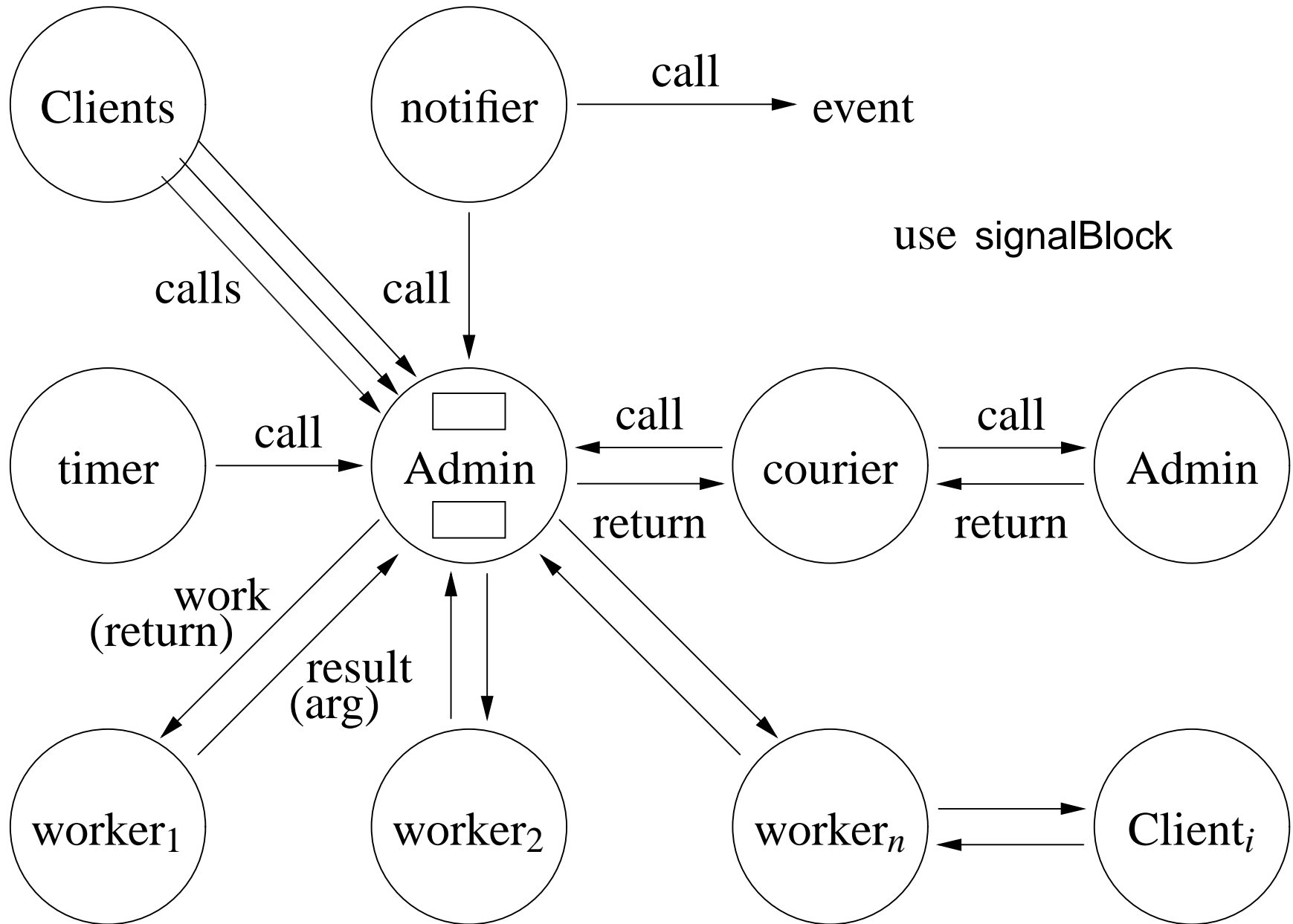


- An administrator makes no call to another task because such calls may block the administrator.
- An administrator usually maintains a list of work to pass to **worker tasks**.
- Typical workers are:
 - timer** - prompt the administrator at specified time intervals
 - notifier** - perform a potentially blocking wait for an external event (key press)

simple worker - do work given to them by and return the result to the administrator

complex worker - do work given to them by administrator and interact directly with client of the work

courier - perform a potentially blocking call on behalf of the administrator



7.3.2 Client Side

- While a server can attempt to make a client's delay as short as possible, not all servers do it.
- In some cases, a client may not have to wait for the server to process a request (producer/consumer problem)
- This can be accomplished by an asynchronous call from the client to the server, where the caller does not wait for the call to complete.
- Asynchronous call requires implicit buffering between client and server to store the client's arguments from the call.
- $\mu\text{C++}$ provides only synchronous call, i.e., the caller is delayed from the time the arguments are delivered to the time the result is returned (like a procedure call).
- It is possible to build asynchronous facilities out of the synchronous ones and vice versa.

7.3.2.1 Returning Values

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

- However, if a result is returned from the call, i.e., from the server to the client, the asynchronous call is significantly more complex.
- To achieve asynchrony in this case, a call must be divided into two calls:
 1. transmit the arguments
 2. retrieve the results
- The time between the two calls allows the calling task to execute asynchronously with the task performing the operation on the caller's behalf.
- If the result is not ready when the second call is made, the caller blocks or the caller has to call again (poll).
- However, this requires a protocol so that when the client makes the second call, the correct result can be found and returned.

7.3.2.2 Tickets

- One form of protocol is the use of a token or ticket.
- The first part of the protocol transmits the arguments specifying the desired work and a ticket (like a laundry ticket) is returned immediately.
- The second call passes the ticket to retrieve the result.

- The ticket is matched with a result, and the result is returned if available or the caller is blocks or polls until the result is available.
- However, protocols are error prone because the caller may not obey the protocol (e.g., never retrieve a result, use the same ticket twice, forged ticket).

7.3.2.3 Call-Back Routine

- Another protocol is to 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; it can only store the result and set an indicator (e.g., V a semaphore) known to the original client.
- The original client must *poll* the indicator or block until the indicator is set.
- The advantage is that the server does not have to store the result, but can drop it off immediately.
- Also, the client can write the call-back routine, so they can decide to poll or block or do both.

7.3.2.4 Futures

- A **future** provides the same asynchrony as above but without an explicit protocol.
- The protocol becomes implicit between the future and the task generating the result.
- Further, it removes the difficult problem of when the caller should try to retrieve the result.
- In detail, a future is an object that is a subtype of the result type expected by the caller.
- Instead of two calls as before, a single call is made, passing the appropriate arguments, and a future is returned.
- The future is returned immediately and it is empty.
- The caller “believes” the call completed and continues execution with an empty result value.
- The future is filled in at some time in the “future”, when the result is calculated.
- If the caller tries to use the future before its value is filled in, the caller is implicitly blocked.

- A simple future can be constructed out of a semaphore and link field, as in:

```

class future {
    friend _Task server;      // can access internal state

    uSemaphore resultAvailable;
    future *link;
    ResultType result;
public:
    future() : resultAvailable( 0 ) {}

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

```

- the semaphore is used to block the caller if the future is empty
- the link field is used to chain the future onto a server work-list.
- Unfortunately, the syntax for retrieving the value of the future is awkward as it requires a call to the get routine.

- Also, in languages without garbage collection, the future must be explicitly deleted.