

8 Other Approaches

8.1 Languages with Concurrency Constructs

8.1.1 Ada 95

- Like $\mu\text{C++}$, but not as general.
- E.g., monitor bounded-buffer, restricted implicit (automatic) signal:

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

```

protected type buffer is -- _Monitor
  entry insert( elem : in ElemType ) when count < Size is
  begin
    -- add to buffer
    count := count + 1;
  end insert;
  entry remove( elem : out ElemType ) when count > 0 is
  begin
    -- remove from buffer, return via parameter
    count := count - 1;
  end remove;
private:
  ... // buffer declarations
  count : Integer := 0;
end buffer;

```

- The **when** clause is external scheduling because it can only be used at start of entry routine not within.
- The **when** expression can contain only global object variables; parameter or local variables are disallowed \Rightarrow no dating-service.
- E.g., task bounded-buffer:

```
task type buffer is -- _Task
  ... -- buffer declarations
  count : integer := 0;
begin -- thread starts here (task main)
  loop
    select -- _Accept
      when count < Size => -- guard
      accept insert(elem : in ElemType) do -- mutex member
        -- add to buffer
        count := count + 1;
      end;
      -- executed if this accept called
    or
      when count > 0 => -- guard
      accept remove(elem : out ElemType) do -- mutex member
        -- remove from buffer, return via parameter
        count := count - 1;
      end;
    end select;
  end loop;
end buffer;

var b : buffer      -- create a task
```

- **select** is external scheduling and can only appear in a **task**, not any of its subprograms
- Hence, Ada has no direct internal-scheduling mechanism, i.e., no condition variables.
- Instead a **requeue** statement can send a request to be postponed to another (usually non-public) mutex member of the object.
- The request is re-blocked on that mutex member's entry queue, which can be subsequently accepted when the request can be restarted.
- However, all **requeue** techniques suffer the problem of dealing with accumulated temporary results:
 - If a request must be postponed, its temporary results must be returned and bundled with the initial request before forwarding to the mutex member handling the next step.
 - Alternatively, the temporary results can be re-computed at the next step if possible.
- In contrast, waiting on a condition variable automatically saves the execution location and any partially computed state.

8.1.2 Modula-3/Java/C#

- Java's concurrency constructs are largely derived from Modula-3.

```
class Thread implements Runnable {  
    public Thread();  
    public Thread(String name);  
    public String getName();  
    public void setName(String name);  
    public void run();  
    public synchronized void start();  
    public static Thread currentThread();  
    public static void yield();  
    public final void join();  
}
```

- Thread is like uBaseTask in $\mu\text{C++}$, and all tasks must explicitly inherit from it:

```
class myTask : Thread {           // inheritance
    private int arg;              // communication variables
    private int result;
    public mytask() {...}         // task constructors
    public void run() {...}       // task main
    public int result() {...}    // return result
    // unusual to have more members
}
```

- Thread starts in member run.
- Java requires explicit starting of a thread by calling start after the thread's declaration.
⇒ coding convention to start the thread or inheritance is precluded (can only start a thread once)
- Termination synchronization is accomplished by calling join.
- Returning a result on thread termination is accomplished by member(s) returning values from the task's global variables.

```

mytask th = new myTask(...); // create and initialized task
th.start();           // start thread
// concurrency
th.join();           // wait for thread termination
a2 = th.result();    // retrieve answer from task object

```

- Like $\mu\text{C++}$, when the task's thread terminates, it becomes an object, hence allowing the call to result to retrieve a result.
- Java has **synchronized** class members (i.e., `_Mutex` members but incorrectly named), and a **synchronized** statement.
- Modula-3 has a lock statement that can be used to simulate `_Mutex` members if certain coding conventions are followed.
- Neither language has very useful external scheduling.
- While it is possible to have public **synchronized** members of a task, there is no mechanism to manage direct calls, i.e., no accept statement.
⇒ no useful direct communication.
- Internal scheduling is no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck conditions
- All classes have one implicit condition variable and these routines to manipulate it:

```

public wait();
public notify();
public notifyall()

```

- Bounded buffer:

```

class buffer {
    // buffer declarations
    private int count = 0;
    public synchronized void insert( int elem ) {
        while ( count == Size ) wait(); // busy-waiting
        // add to buffer
        count += 1;
        notify();
    }
    public synchronized int remove() {
        while ( count == 0 ) wait(); // busy-waiting
        // remove from buffer
        count -= 1;
        notify();
        return elem;
    }
}

```

- Because there is only one condition queue \Rightarrow certain solutions are

difficult or impossible.

8.2 Threads & Locks Library

- All concurrency libraries are unsafe or inefficient because the compiler thinks the program is sequential.
- unsafe:
 - Valid sequential optimizations can invalidate concurrent programs:

```
P(lock);  
// critical section  
V(lock);
```

compiler moves P or V after/before critical section
 - Turn off sequential optimizations to prevent concurrent error but now entire program is unnecessarily slower.
- Several libraries exist for C (pthreads) and C++ (μ C++).
- C libraries built around simple model of starting a thread in a routine and mutex/condition locks (“attribute” parameters not shown).

```
int pthread_create( pthread_t *new_thread_ID,  
                  void * (*start_func)(void *), void *arg );  
int pthread_join( pthread_t target_thread, void **status );  
pthread_t pthread_self( void );
```

```
int pthread_mutex_init( pthread_mutex_t *mp );  
int pthread_mutex_lock( pthread_mutex_t *mp );  
int pthread_mutex_unlock( pthread_mutex_t *mp );  
int pthread_mutex_destroy( pthread_mutex_t *mp );
```

```
int pthread_cond_init( pthread_cond_t *cp );  
int pthread_cond_wait( pthread_cond_t *cp, pthread_mutex_t *mutex );  
int pthread_cond_signal( pthread_cond_t *cp );  
int pthread_cond_broadcast( pthread_cond_t *cp );  
int pthread_cond_destroy( pthread_cond_t *cp );
```

- Thread starts in routine `start_func` via `pthread_create`. Initialization data is single **void *** value.
- Termination synchronization is performed by calling `pthread_join`.
- Return a result on thread termination by passing back a single **void *** value from `pthread_join`.

```
void *rtn( void *arg ) { ... }  
int i = 3, r, rc;  
pthread_t t;           // thread id  
rc = pthread_create(&t,rtn,(void *)i); // create and initialized task  
if ( rc != 0 ) ...    // check for error  
// concurrency  
rc = pthread_join(t,&r); // wait for thread termination and result  
if ( rc != 0 ) ...    // check for error
```

- All C library approaches have type-unsafe communication with tasks.
- No external scheduling \Rightarrow no direct communication.
- Internal scheduling is no-priority nonblocking \Rightarrow barging \Rightarrow wait statements must be in while loops to recheck conditions

```
typedef struct {  
    // buffer declarations  
    pthread_mutex_t mutex;        // mutual exclusion  
    pthread_cond_t Full, Empty;  // synchronization  
} buffer;  
void ctor( buffer *buf ) {      // constructor  
    ...  
    pthread_mutex_init( &buf->mutex );  
    pthread_cond_init( &buf->Full );  
    pthread_cond_init( &buf->Empty );  
}  
void dtor( buffer *buf ) {     // destructor  
    pthread_mutex_lock( &buf->mutex );  
    ...  
    pthread_cond_destroy( &buf->Empty );  
    pthread_cond_destroy( &buf->Full );  
    pthread_mutex_destroy( &buf->mutex );  
}
```

```
void insert( buffer *buf, int elem ) {  
    pthread_mutex_lock( &buf->mutex );  
    while ( buf->count == Size  
            pthread_cond_wait( &buf->Empty, &buf->mutex );  
    // add to buffer  
    buf->count += 1;  
    pthread_cond_signal( &buf->Full );  
    pthread_mutex_unlock( &buf->mutex );  
}  
int remove( buffer *buf ) {  
    pthread_mutex_lock( &buf->mutex );  
    while ( buf->count == 0 )  
        pthread_cond_wait( &buf->Full, &buf->mutex );  
    // remove from buffer  
    buf->count -= 1;  
    pthread_cond_signal( &buf->Empty );  
    pthread_mutex_unlock( &buf->mutex );  
    return elem;  
}
```

- Since there are no constructors/destructors in C, explicit calls are necessary to ctor/dtor before/after use.

- All locks must be initialized and finalized.
- Mutual exclusion must be explicitly defined where needed.
- Condition locks should only be accessed with mutual exclusion.
- `pthread_cond_wait` atomically blocks thread and releases mutex lock, which is necessary to close race condition on baton passing.

8.3 Threads & Message Passing

- Message passing is an alternative mechanism to parameter passing.
- In message passing, all information transmitted is grouped into a single data area and (usually) passed by value.
- Hence, all pointers must be dereferenced before being put into the message (i.e., no pointers are passed in a message).
- This makes a message independent of the context of the message sender (i.e., no shared memory is required).
- Hence, the receiver can be on the same or different machines, it makes no difference (distributed systems).
- On shared memory machines, pointers can still be passed.

- Message passing is usually direct communication.
- Messages are directed to a specific task and received from a specific task (receive specific):

task₁

send(tid₂, msg)

task₂

receive(tid₁, msg)

- Or messages are directed to a specific task and received from any task (receive any):

task₁

send(tid₂, msg)

task₂

tid = receive(msg)

8.3.1 Nonblocking Send

- Send does not block, and receive gets the messages in the order they are sent (SendNonBlock).
- \Rightarrow an infinite length buffer between the sender and receiver
- since the buffer is bounded, the sender occasionally blocks until the receiver catches up.
- the receiver blocks if there is no message

```

Producer() {
    ...
    for (;;) {
        produce an item
        SendNonBlock(CId,msg);
    }
}

Consumer() {
    ...
    for (;;) {
        Receive(PId,msg);
        consume the item
    }
}

```

8.3.2 Blocking Send

- Send or receive blocks until a corresponding receive or send is performed (SendBlock).
- I.e., information is transferred when a **rendezvous** occurs between the sender and the receiver

8.3.3 Send-Receive-Reply

- Send blocks until a reply is sent from the receiver, and the receiver blocks if there is no message (SendReply).

```

Producer() {
    ...
    for (;;) {
        produce an item
        SendReply(CId,ans,msg);
        ...
    }
}

Consumer() {
    ...
    for (;;) {
        prod = ReceiveReply(msg);
        consume the item
        Reply(prod,ans) never
                        blocks
    }
}

```

- Why use receive any instead of receive specific?

- E.g., Producer/Consumer

– Producer

```

for ( i = 1; i <= NoOfItems; i += 1 ) {
    msg = rand() % 100 + 1;
    cout << " Producer: " << msg << endl;
    SendReply(Cons, rmsg, msg);
}
msg = -1;
SendReply(Cons, rmsg, msg);

```

– Consumer

```
for (;;) {  
    prod = ReceiveReply(msg);  
    // check msg  
    Reply(prod, rmsg);  
    if ( msg < 0 ) break;  
    cout << "Consumer : " << msg << endl;  
}
```

8.4 Message Format

- variable-size messages
 - complex implementation, easy to use
- fixed-size message
 - simple/fast implementation
 - requires long messages to be broken up and transmitted in pieces and reconstructed by the receiver
- typed message
 - only messages of a certain type can be sent and received among tasks
 - *requires dynamic type checking*