

3 Concurrency

- A **thread** is an independent sequential execution path through a program. Each thread is scheduled for execution separately and independently from other threads.
- A **process** is a program component (like a routine) that has its own thread and has the same state information as a coroutine.
- A **task** is similar to a process except that it is reduced along some particular dimension (like the difference between a boat and a ship, one is physically smaller than the other). It is often the case that a process has its own memory, while tasks share a common memory. A task is sometimes called a light-weight process (LWP).
- **Parallel execution** is when 2 or more operations occur simultaneously, which can only occur when multiple processors (CPUs) are present.
- **Concurrent execution** is any situation in which execution of multiple threads *appears* to be performed in parallel. It is the threads of control associated with processes and tasks that results in concurrent execution.

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

3.1 Why Write Concurrent Programs

- Dividing a problem into multiple executing threads is an important programming technique just like dividing a problem into multiple routines.
- Expressing a problem with multiple executing threads may be the natural (best) way of describing it.
- Multiple executing threads can enhance the execution-time efficiency, by taking advantage of inherent concurrency in an algorithm and of any parallelism available in the computer system.

3.2 Why Concurrency is Difficult

- to understand:
 - While people can do several things concurrently, the number is small because of the difficulty in managing and coordinating them.
 - Especially when the things interact with one another.
- to specify:
 - How can/should a problem be broken up so that parts of it can be solved at the same time as other parts?

- How and when do these parts interact or are they independent?
- If interaction is necessary, what information must be communicated during the interaction?
- to debug:
 - Concurrent operations proceed at varying speeds and in non-deterministic order, hence execution is not repeatable (Heisenbug).
 - Reasoning about multiple streams or threads of execution and their interactions is much more complex than for a single thread.
- E.g. Moving furniture out of a room; can't do it alone, but how many helpers and how to do it quickly to minimize the cost.
- How many helpers?
 - 1,2,3, ... N, where N is the number of items of furniture
 - more than N?
- Where are the bottlenecks?
 - the door out of the room, items in front of other items, large items
- What communication is necessary between the helpers?
 - which item to take next

- some are fragile and need special care
- big items need several helpers working together

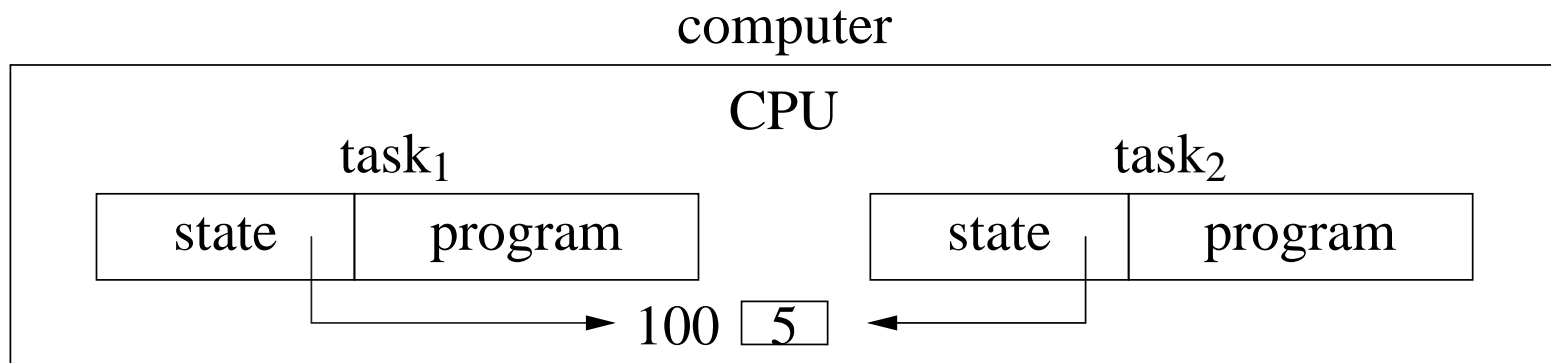
3.3 Structure of Concurrent Systems

- Concurrent systems can be divided into 3 major types:
 1. those that attempt to *discover* concurrency in an otherwise sequential program, e.g., parallelizing loops and access to data structures
 2. those that provide concurrency through *implicit* constructs, which a programmer uses to build a concurrent program
 3. those that provide concurrency through *explicit* constructs, which a programmer uses to build a concurrent program
- In case 1, there is a fundamental limit to how much parallelism can be found and current techniques only work on certain kinds of programs.
- In case 2, threads are accessed indirectly via specialized mechanisms (e.g., pragmas or parallel **for**) and implicitly managed.
- In case 3, threads are directly access and explicitly managed.
- Cases 1 & 2 are always built from case 3.
- To solve all concurrency problems, threads needs to be explicit.

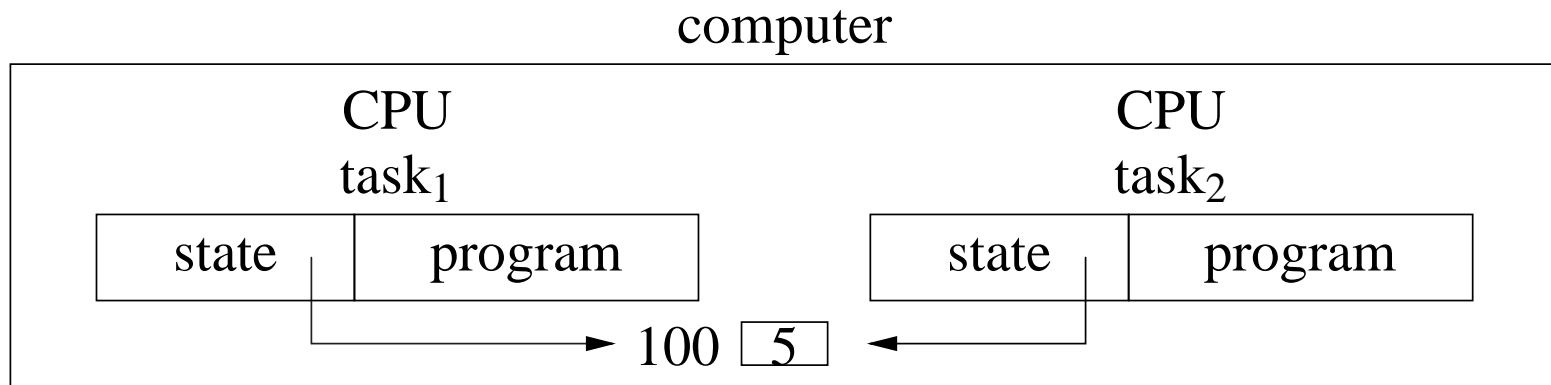
- Both implicit and explicit mechanisms are complementary, and hence, can appear together in a single programming language.
- However, the limitations of implicit mechanisms require that explicit mechanisms always be available to achieve maximum concurrency.
- $\mu\text{C++}$ only supports explicit mechanisms, but nothing in its design precludes implicit mechanisms.
- Some concurrent systems provide a single technique or paradigm that must be used to solve all concurrent problems.
- While a particular paradigm may be very good for solving certain kinds of problems, it may be awkward or preclude other kinds of solutions.
- Therefore, a good concurrent system must support a variety of different concurrent approaches, while at the same time not requiring the programmer to work at too low a level.
- Fundamentally, as the amount of concurrency increases, so does the complexity to express and manage it.

3.4 Structure of Concurrent Hardware

- Concurrent execution of threads is possible on a computer which has only one CPU (**uniprocessor**); **multitasking** for multiple tasks or **multiprocessing** for multiple processes.

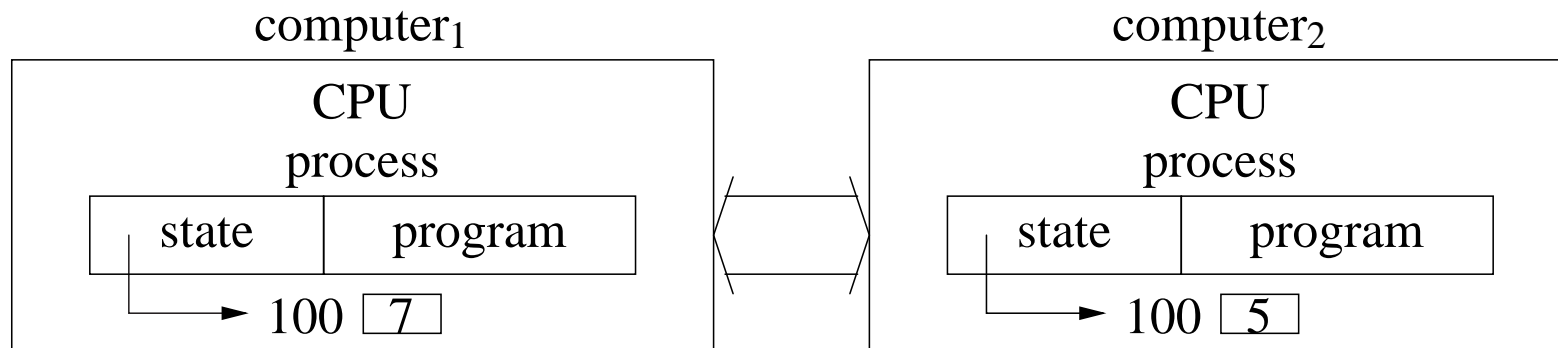


- Parallelism is simulated by rapidly context switching the CPU back and forth between threads.
 - Unlike coroutines, task switching may occur at non-deterministic program locations, i.e., between any two *machine* instructions.
 - Switching is usually based on a timer interrupt that is independent of program execution.
- Or on the same computer, which has multiple CPUs, using separate CPUs but sharing the same memory (**multiprocessor**):



These tasks run in parallel with each other.

- Processes may be on different computers using separate CPUs and separate memories (**distributed system**):

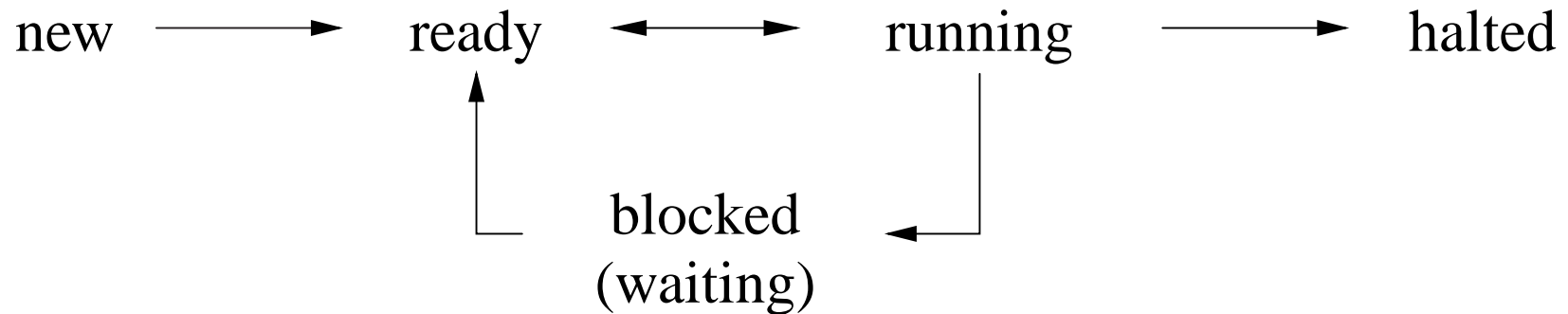


These processes run in parallel with each other.

- By examining the first case, which is the simplest, all of the problems that occur with parallelism can be illustrated.

3.5 Execution States

- A thread may go through the following states during its execution.



- **state transitions** are initiated in response to events:
 - timer alarm (running → ready)
 - completion of I/O operation (blocked → ready)
 - exceeding some limit (CPU time, etc.) (running → halted)
 - exceptions (running → halted)

3.6 Thread Creation

- Concurrency requires the ability to specify the following 3 mechanisms in a programming language.

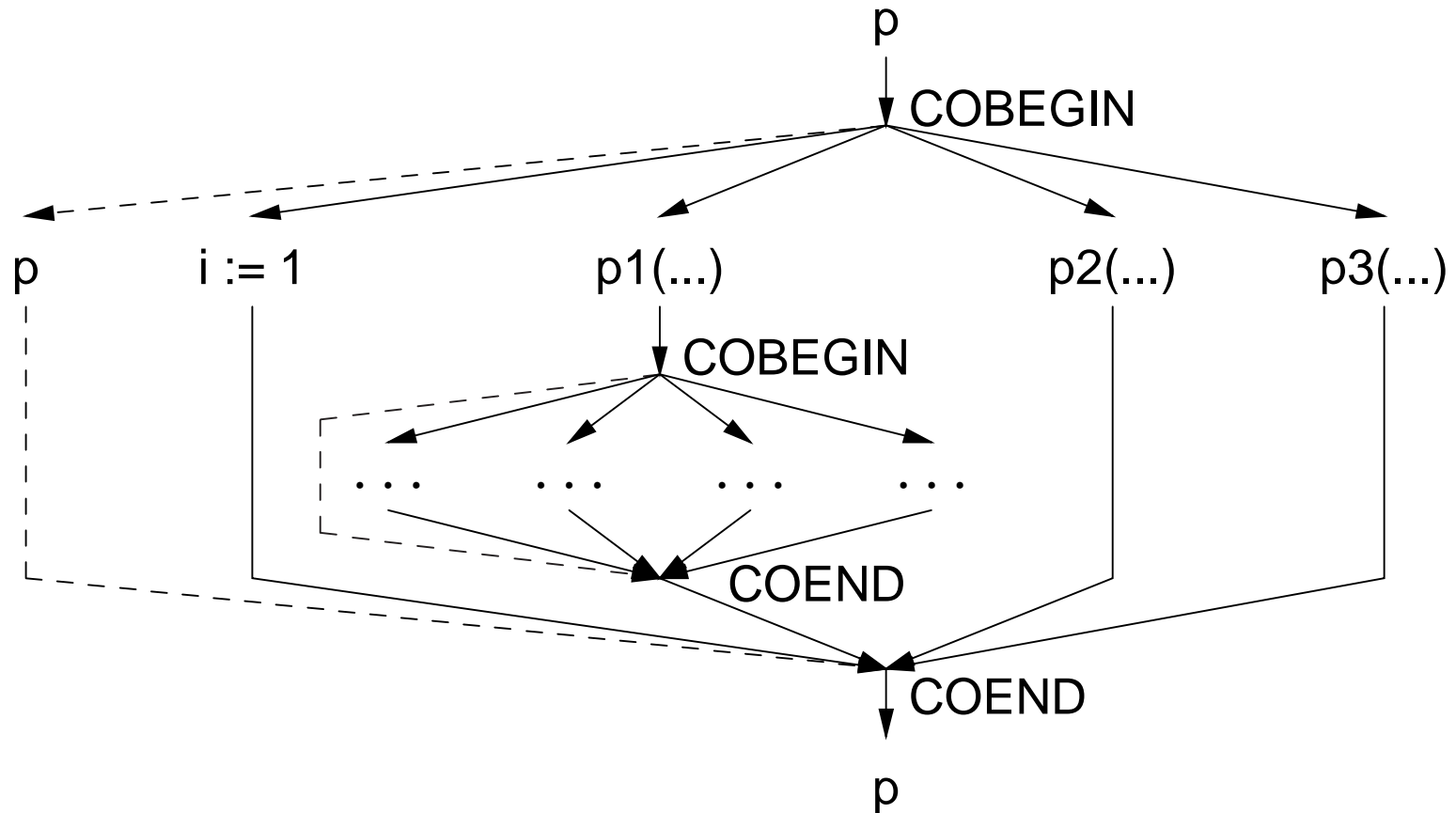
1. thread creation – the ability to cause another thread of control to come into existence.
 2. thread synchronization – the ability to establish timing relationships among threads, e.g., same time, same rate, happens before/after.
 3. thread communication – the ability to correctly transmit data among threads.
- Thread creation must be a primitive operation; cannot be built from other operations in a language.
 - ⇒ need new construct to create a thread and define where the thread starts execution, e.g., COBEGIN/COEND:

```

BEGIN      initial thread creates internal threads,
  COBEGIN one for each statement in this block
    BEGIN i := 1; ... END;
    p1(5);  order and speed of execution
    p2(7);  of internal threads is unknown
    p3(9);
  COEND initial thread waits for all internal threads to
END;      finish (synchronize) before control continues

```

- A **thread graph** represents thread creations:



- Restricted to creating trees (lattice) of threads.
- In $\mu\text{C++}$, a task must be created for each statement of a COBEGIN using a **_Task** object:

```

_Task T1 {
    void main() { i = 1; }
};
_Task T2 {
    void main() { p1(5); }
};
_Task T3 {
    void main() { p2(7); }
};
_Task T4 {
    void main() { p3(9); }
};

```

```

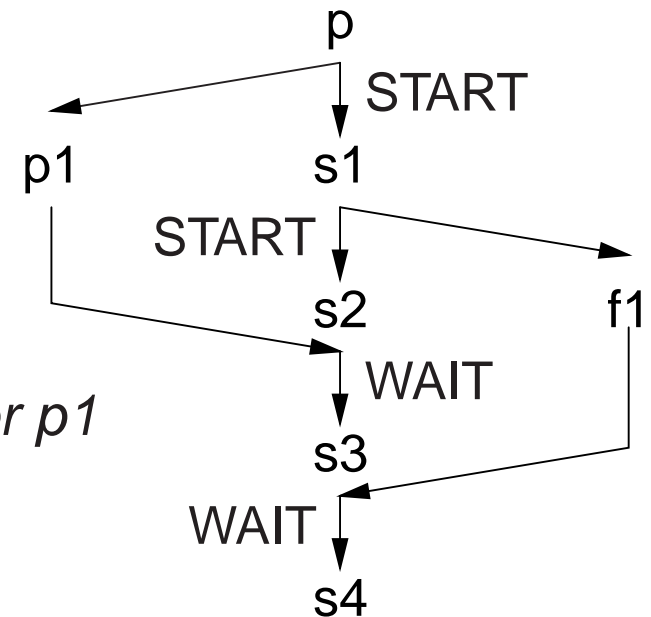
void uMain::main() {
    // { int i, j, k; } ???
    { // COBEGIN
        T1 t1; T2 t2; T3 t3; T4 t4;
    } // COEND
}
void p1(...) {
    { // COBEGIN
        T5 t5; T6 t6; T7 t7; T8 t8;
    } // COEND
}

```

- Unusual to create objects in a block and not use them.
- For task objects, the block waits for each task's thread to finish.
- Alternative approach for thread creation is START/WAIT, which can create arbitrary thread graph:

```

PROGRAM p
  PROC p1(...) ...
  FUNCTION f1(...) ...
  INT i;
  BEGIN
    (fork) START p1(5); thread starts in p1
      s1      continue execution, do not wait for p1
    (fork) START f1(8); thread starts in f1
      s2
    (join) WAIT p1; wait for p1 to finish
      s3
    (join) WAIT i := f1; wait for f1 to finish
      s4
  
```



- COBEGIN/COEND can only approximate this thread graph:

```

COBEGIN
  p1(5);
  BEGIN s1; COBEGIN f1(8); s2; COEND END // wait for f1!
COEND
s3; s4;

```

- START/WAIT can simulate COBEGIN/COEND:

```

COBEGIN      START p1(...)
  p1(...)    START p2(...)
  p2(...)    WAIT p2
COEND        WAIT p1

```

- In $\mu\text{C++}$:

```

_Task T1 {
  void main() { p1(5); }
};
_Task T2 {
  int temp;
  void main() { temp = f1(8); }
public:
  ~T2() { i = temp; }
};

```

```

void uMain::main() {
  T1 *p1p = new T1; // start a T1
  ... s1 ...
  T2 *f1p = new T2; // start a T2
  ... s2 ...
  delete p1p; // wait for p1
  ... s3 ...
  delete f1p; // wait for f1
  ... s4 ...
}

```

- Variable i cannot be assigned until the delete of $f1p$, otherwise the value could change in $s2/s3$.
- Allows same routine to be started multiple times with different arguments.

3.7 Termination Synchronization

- A thread terminates when:
 - it finishes normally
 - it finishes with an error
 - it is killed by its parent (or sibling) (not supported in $\mu\text{C++}$)
 - because the parent terminates (not supported in $\mu\text{C++}$)
- Children can continue to exist even after the parent terminates (although this is rare).
 - E.g. sign off and leave child process(s) running
- Synchronizing at termination is possible for independent threads.
- Termination synchronization may be used to perform a final communication.
- E.g., sum the rows of a matrix concurrently:

```

_Task Adder {
    int *row, size, &subtotal;
    void main() {
        subtotal = 0;
        for ( int r = 0; r < size; r += 1 ) {
            subtotal += row[r];
        }
    }
public:
    Adder( int row[], int size, int &subtotal ) :
        row( row ), size( size ), subtotal( subtotal ) {}
};

void uMain::main() {
    int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], total = 0, r;
    Adder *adders[rows];
    // read in matrix
    for ( r = 0; r < rows; r += 1 ) { // start threads to sum rows
        adders[r] = new Adder( matrix[r], cols, subtotals[r] );
    }
    for ( r = 0; r < rows; r += 1 ) { // wait for threads to finish
        delete adders[r];
        total += subtotals[r];
    }
    cout << total << endl;
}

```

	matrix				subtotals
$T_0 \Sigma$	23	10	5	7	0
$T_1 \Sigma$	-1	6	11	20	0
$T_2 \Sigma$	56	-13	6	0	0
$T_3 \Sigma$	-2	8	-5	1	0
	total				Σ

3.8 Synchronization and Communication during Execution

- Synchronization occurs when one thread waits until another thread has reached a certain point in its code.
- One place synchronization is needed is in transmitting data between threads.
 - One thread has to be ready to transmit the information and the other has to be ready to receive it, simultaneously.
 - Otherwise one might transmit when no one is receiving, or one might receive when nothing is transmitted.

```

bool Insert = false, Remove = false;
int Data;

_Task Prod {
    int N;
    void main() {
        for ( int i = 1; i <= N; i += 1 ) {
            Data = i;          // transfer data
            Insert = true;
            while ( ! Remove ) {} // busy wait
            Remove = false;
        }
    }
public:
    Prod( int N ) : N( N ) {}
};

```

```

_Task Cons {
    int N;
    void main() {
        int data;
        for ( int i = 1; i <= N; i += 1 ) {
            while ( ! Insert ) {} // busy wait
            Insert = false;
            data = Data; // remove data
            Remove = true;
        }
    }
public:
    Cons( int N ) : N( N ) {}
};
void uMain::main() {
    Prod prod( 5 ); Cons cons( 5 );
}

```

- 2 infinite loops! No, because of implicit switching of threads.
- cons synchronizes (waits) until prod transfers some data, then prod waits for cons to remove the data.
- Are 2 synchronization flags necessary?

3.9 Communication

- Once threads are synchronized there are many ways that information can be transferred from one thread to the other.
- If the threads are in the same memory, then information can be transferred by value or address (VAR parameters).
- If the threads are not in the same memory (distributed), then transferring information by value is straightforward but by address is difficult.

3.10 Exceptions

- Exceptions can be handled locally within a task, or nonlocally among coroutines, or concurrently among tasks.
 - All concurrent exceptions are nonlocal, but nonlocal exceptions can also be sequential.
- Local task exceptions are the same as for a class.
 - An unhandled exception raised by a task terminates the program.
- Nonlocal exceptions are possible because each task has its own stack (execution state)

- Nonlocal exceptions between a task and a coroutine are the same as between coroutines (single thread).
- Concurrent exceptions among tasks are more complex due to the multiple threads.
- A concurrent exception provides an additional kind of communication among tasks.
- For example, two tasks may begin searching for a key in different sets:

```

_Task searcher {
    searcher &partner;
    void main() {
        try {
            ...
            if ( key == ... )
                _Throw stopEvent() _At partner;
        } catch( stopEvent ) { ... }
    }
}

```

When one task finds the key, it informs the other task to stop searching.

- For a concurrent raise, the source execution may only block while queueing the event for delivery at the faulting execution.
- After the event is delivered, the faulting execution propagates it at the

soonest possible opportunity (next context switch); i.e., the faulting task is not interrupted.

- **Nonlocal delivery is initially disabled for a task**, so handlers can be set up before any exception can be delivered.

```
void main() {  
    // initialization, no nonlocal delivery  
    try { // setup handlers  
        _Enable { // enable delivery of exceptions  
            // rest of the code  
        }  
    } catch( nonlocal-exception ) {  
        // handle nonlocal exception  
    }  
    // finalization, no nonlocal delivery  
}
```

3.11 Critical Section

- Threads may access non-concurrent objects, like a file or linked-list.
- There is a potential problem if there are multiple threads attempting to

operate on the same object simultaneously.

- Not a problem if the operation on the object is **atomic** (not divisible).
- This means no other thread can modify any partial results during the operation on the object (but the thread can be interrupted).
- Where an operation is composed of many instructions, it is often necessary to make the operation atomic.
- A group of instructions on an associated object (data) that must be performed atomically is called a **critical section**.
- Preventing simultaneous execution of a critical section by multiple thread is called **mutual exclusion**.
- Must determine when concurrent access is allowed and when it must be prevented.
- One way to handle this is to detect any sharing and serialize all access; wasteful if threads are only reading.
- Improve by differentiating between reading and writing
 - allow multiple readers or a single writer; still wasteful as a writer may only write at the end of its usage.

- Need to minimize the amount of mutual exclusion (i.e., make critical sections as small as possible) to maximize concurrency.

3.12 Static Variables

- Static variables in a class are shared among all objects generated by that class.
- However, shared variables may need mutual exclusion for correct usage.
- There are a few special cases where **static** variables can be used safely, e.g., task constructor.
- If task objects are generated serially, **static** variables can be used in the constructor.
- E.g., assigning each task its own name:

```

_Task T {
    static int tid;
    string name; // must supply storage

    ...
public:
    T() {
        name = "T" + itostr(tid); // shared read
        setName( name.c_str() );
        tid += 1; // shared write
    }
    ...
};
int T::tid = 0;    // initialize static variable in .C file
T t[10];          // 10 tasks with individual names

```

- Instead of **static** variables, pass a task identifier to the constructor:

```

T::T( int tid ) { ... } // create name
T *t[10];           // 10 pointers to tasks
for ( int i = 0; i < 10; i += 1 ) {
    t[i] = new T(i); // with individual names
}

```

- These approaches only work if one task creates all the objects so creation is performed serially.
- In general, it is best to avoid using shared **static** variables in a concurrent program.

3.13 Mutual Exclusion Game

- Is it possible to write (in your favourite programming language) some code that guarantees that a statement (or group of statements) is always serially executed by 2 threads?
- Rules of the Game:
 1. Only one thread can be in its critical section at a time.
 2. Threads run at arbitrary speed and in arbitrary order, and the underlying system guarantees each thread makes progress (i.e., threads get some CPU time).
 3. If a thread is not in its critical section or the entry or exit code that controls access to the critical section, it may not prevent other threads from entering their critical section.
 4. In selecting a thread for entry to the critical section, the selection

cannot be postponed indefinitely. Not satisfying this rule is called **indefinite postponement**.

5. There must exist a bound on the number of other threads that are allowed to enter the critical section after a thread has made a request to enter it. Not satisfying this rule is called **starvation**.

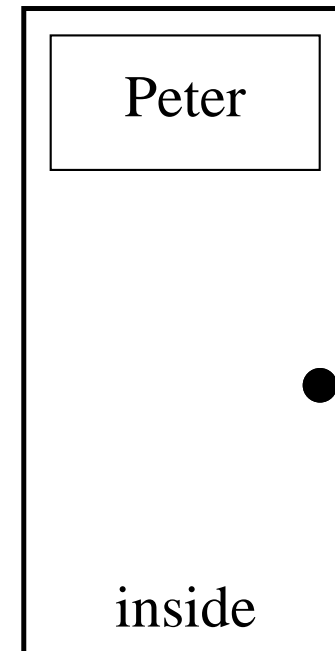
3.14 Self-Testing Critical Section

```

uBaseTask *CurrTid;           // current task id

void CriticalSection() {
    ::CurrTid = &uThisTask();
    for ( int i = 1; i <= 100; i += 1 ) { // work
        if ( ::CurrTid != &uThisTask() ) {
            uAbort( "interference" );
        }
    }
}

```



- What is the minimum number of interference tests and where?

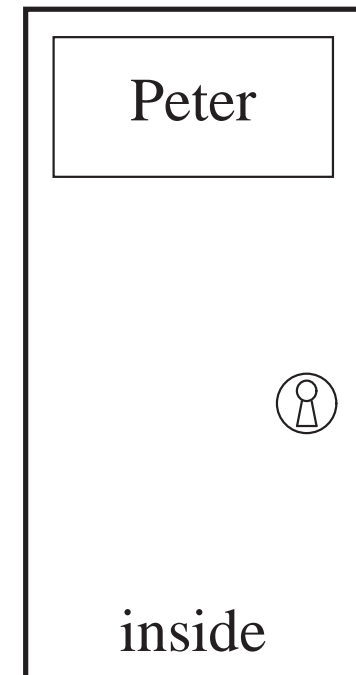
3.15 Software Solutions

3.15.1 Lock

```
enum Yale {CLOSED, OPEN} Lock = OPEN; // shared
```

```
_Task PermissionLock {
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            while (::Lock == CLOSED) {} // entry protocol
            ::Lock = CLOSED;
            CriticalSection();           // critical section
            ::Lock = OPEN;               // exit protocol
        }
    }
public:
    PermissionLock() {}
};
void uMain::main() {
    PermissionLock t0, t1;
}
```

Breaks rule 1



3.15.2 Alternation

```

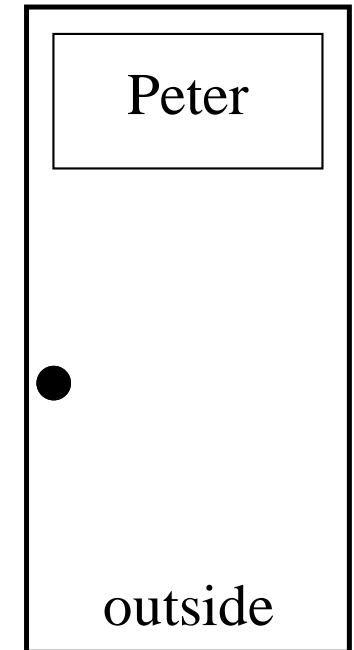
int Last = 0;                                // shared

_Task Alternation {
    int me;

    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            while (::Last == me) {           // entry protocol
                CriticalSection();          // critical section
                ::Last = me;                // exit protocol
            }
        }
    }
public:
    Alternation(int me) : me(me) {}
};

void uMain::main() {
    Alternation t0( 0 ), t1( 1 );
}

```



Breaks rule 3

3.15.3 Declare Intent

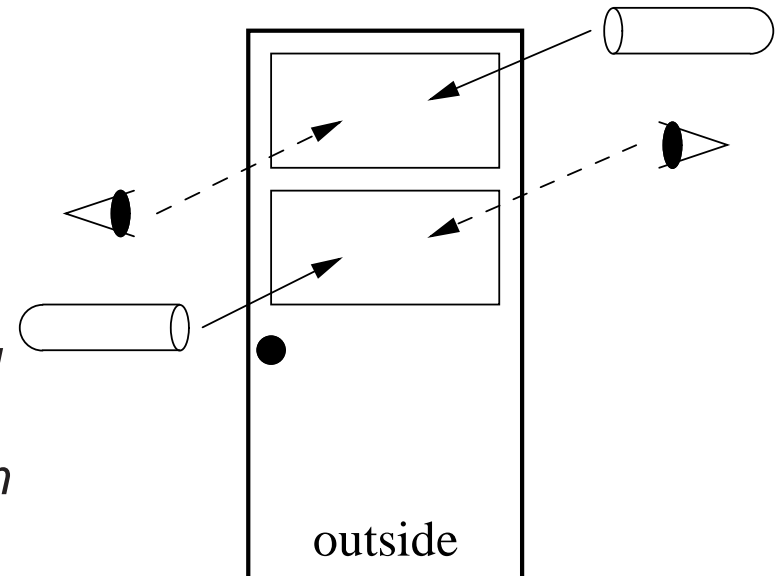
```

enum Intent {WantIn, DontWantIn};

_Task DeclIntent {
  Intent &me, &you;
  void main() {
    for ( int i = 1; i <= 1000; i += 1 ) {
      me = WantIn;          // entry protocol
      while ( you == WantIn ) {
        CriticalSection(); // critical section
        me = DontWantIn;  // exit protocol
      }
    }
  }
public:
  DeclIntent( Intent &me, Intent &you ) :
    me(me), you(you) {}
};

void uMain::main() {
  Intent me = DontWantIn, you = DontWantIn;
  DeclIntent t0( me, you ), t1( you, me );
}

```



Breaks rule 4

3.15.4 Retract Intent

```

enum Intent {WantIn, DontWantIn};
_Task RetractIntent {
    Intent &me, &you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {                // entry protocol
                me = WantIn;
                if (you == DontWantIn) break;
                me = DontWantIn;
                while ( you == WantIn ) {}
            }
            CriticalSection();        // critical section
            me = DontWantIn;        // exit protocol
        }
    }
}
public:
    RetractIntent( Intent &me, Intent &you ) : me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    RetractIntent t0( me, you ), t1( you, me );
}

```

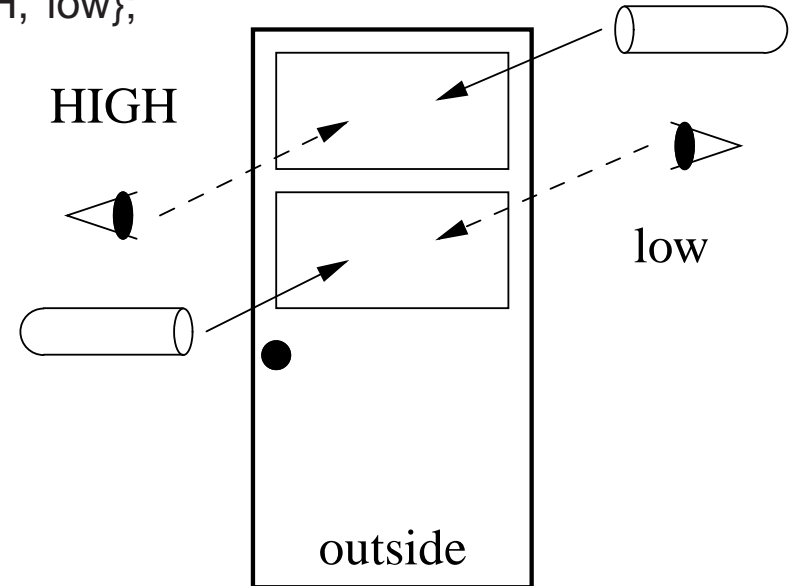
Breaks rule 4

3.15.5 Prioritize Entry

```

enum Intent {WantIn, DontWantIn}; enum Priority {HIGH, low};
_Task PriorityEntry {
    Intent &me, &you; Priority priority;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            if ( priority == HIGH ) { // entry protocol
                me = WantIn;
                while ( you == WantIn ) {}
            } else {
                for ( ;; ) {
                    me = WantIn;
                    if ( you == DontWantIn ) break;
                    me = DontWantIn;
                    while ( you == WantIn ) {}
                }
            }
            CriticalSection(); // critical section
            me = DontWantIn; // exit protocol
        }
    }
};
public:
    PriorityEntry( Priority p, Intent &me, Intent &you ) : priority(p), me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    PriorityEntry t0( HIGH, me, you ), t1( low, you, me );
} // main

```



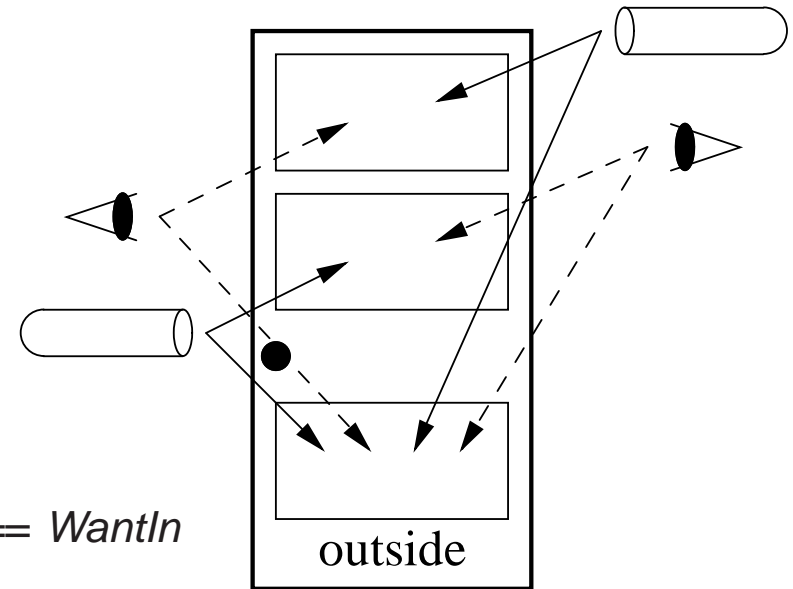
Breaks rule 5

3.15.6 Dekker

```

enum Intent {WantIn, DontWantIn};
Intent *Last;
_Task Dekker {
    Intent &me, &you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {                // entry protocol
                me = WantIn;
                if ( you == DontWantIn ) break;
                if ( ::Last == &me ) {
                    me = DontWantIn;
                    while ( ::Last == &me ) {} // you == WantIn
                }
            }
            CriticalSection();          // critical section
            ::Last = &me;              // exit protocol
            me = DontWantIn;
        }
    }
};
public:
    Dekker( Intent &me, Intent &you ) : me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    ::Last = &me;
    Dekker t0( me, you ), t1( you, me );
}

```



3.15.7 Peterson

```
enum Intent {WantIn, DontWantIn};
Intent *Last;
```

```
_Task Peterson {
    Intent &me, &you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            me = WantIn;                // entry protocol
            ::Last = &me;
            while ( you == WantIn && ::Last == &me ) {}
            CriticalSection();          // critical section
            me = DontWantIn;           // exit protocol
        }
    }
public:
    Peterson(Intent &me, Intent &you) : me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    Peterson t0(me, you), t1(you, me);
}
```

- Differences between Dekker and Peterson
 - Dekker's algorithm makes no assumptions about atomicity, while Peterson's algorithm assumes assignment is an atomic operation.
 - Dekker's algorithm works on a machine where bits are scrambled during simultaneous assignment; Peterson's algorithm does not.
- Prove Dekker's algorithm has no simultaneous assignments.

3.15.8 N-Thread Prioritized Entry

```

enum Intent { WantIn, DontWantIn };

_Task NTask {
    Intent *intents;           // position & priority
    int N, priority, i, j;
    void main() {
        for ( i = 1; i <= 1000; i += 1 ) {
            // step 1, wait for tasks with higher priority
            do {                // entry protocol
                intents[priority] = WantIn;
                // check if task with higher priority wants in
                for ( j = priority-1; j >= 0; j -= 1 ) {
                    if ( intents[j] == WantIn ) {
                        intents[priority] = DontWantIn;
                        while ( intents[j] == WantIn ) {}
                        break;
                    }
                }
            } while ( intents[priority] == DontWantIn );
        }
    }
}

```

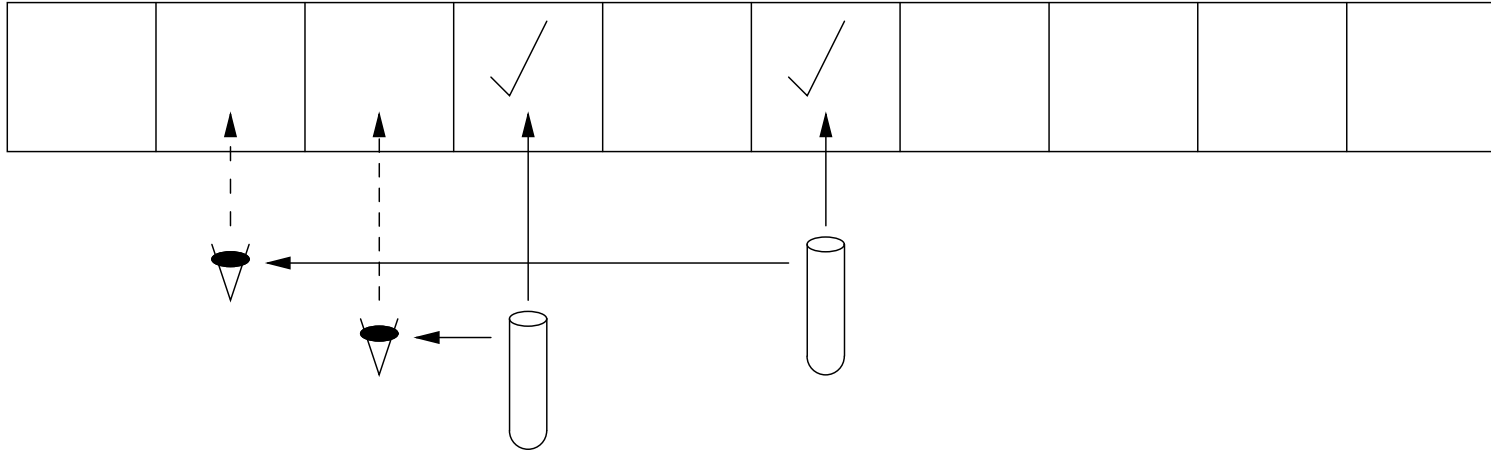
```

// step 2, wait for tasks with lower priority
for ( j = priority+1; j < N; j += 1 ) {
    while ( intents[j] == WantIn ) {}
}
CriticalSection();           // critical section
intents[priority] = DontWantIn; // exit protocol
}
}
public:
    NTask( Intent i[], int N, int p ) : intents(i), N(N), priority(p) {}
};

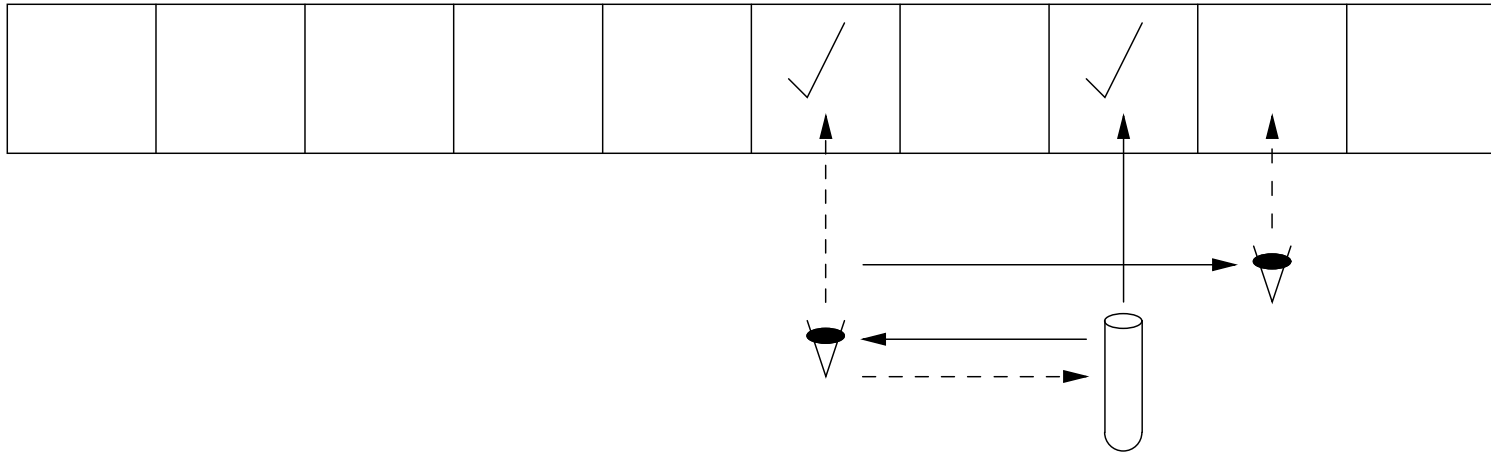
```

Breaks rule 5

HIGH priority 0 1 2 3 4 5 6 7 8 9 low priority



HIGH priority 0 1 2 3 4 5 6 7 8 9 low priority



3.15.9 N-Thread Bakery (Tickets)

```

_Task Bakery { // (Lamport) Hehner/Shyamasundar
    int *ticket, N, priority;
    void main() {
        for ( int i = 0; i < 1000; i += 1 ) {
            // step 1, select a ticket
            ticket[priority] = 0;           // highest priority
            int max = 0;                    // O(N) search
            for ( int j = 0; j < N; j += 1 ) // for largest ticket
                if ( max < ticket[j] && ticket[j] < INT_MAX )
                    max = ticket[j];
            ticket[priority] = max + 1;     // advance ticket
            // step 2, wait for ticket to be selected
            for ( int j = 0; j < N; j += 1 ) { // check tickets
                while ( ticket[j] < ticket[priority] ||
                    (ticket[j] == ticket[priority] && j < priority) ) {}
            }
            CriticalSection();
            ticket[priority] = INT_MAX;     // exit protocol
        }
    }
}
public:
    Bakery( int t[], int N, int p ) : ticket(t), N(N), priority(p) {}
};

```

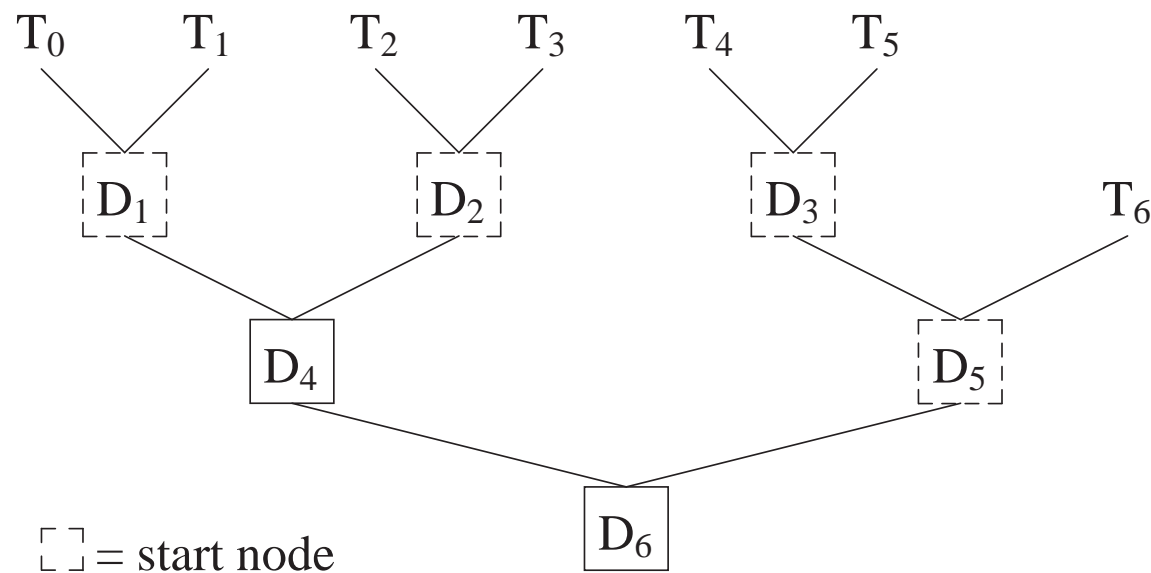
HIGH priority	0	1	2	3	4	5	6	7	8	9	low priority
	∞	∞	17	∞	∞	18	18	0	20	19	

- ticket value of ∞ (INT_MAX) \Rightarrow don't want in
- low ticket and position value \Rightarrow high priority
- ticket selection is unusual
- tickets are not unique \Rightarrow use position as secondary priority
- ticket values cannot increase indefinitely \Rightarrow could fail

3.15.10 Tournament

- N -thread Prioritized Entry uses N bits.
- However, no known solution for all 5 rules using only N bits.
- N-Thread Bakery uses NM bits, where M is the ticket size (e.g., 32 bits), but is only probabilistically correct (limited ticket size).
- Other N -thread solutions are possible using more memory.

- The tournament approach uses a minimal binary tree with $\lceil N/2 \rceil$ start nodes (i.e., full tree with $\lceil \lg N \rceil$ levels).
- Each node is a Dekker or Peterson 2-thread algorithm.
- Each thread is assigned to a particular start node, where it begins the mutual exclusion process.



- At each node, one pair of threads is guaranteed to make progress; therefore, each thread eventually reaches the root of the tree.
- With a minimal binary tree, the tournament approach uses $(N - 1)M$ bits, where $(N - 1)$ is the number of tree nodes and M is the node size (e.g., Last, me, you, next node).

3.15.11 Arbiter

- Create full-time arbitrator task to control entry to critical section.

```

bool intent[5]; // initialize to false
bool serving[5]; // initialize to false

_Task Client {
    int me;
    void main() {
        for ( int i = 0; i < 100; i += 1 ) {
            intent[me] = true;           // entry protocol
            while ( ! serving[me] ) {}
            CriticalSection();
            intent[me] = false;        // exit protocol
            while ( serving[me] ) {}
        }
    }
public:
    Client( int me ) : me( me ) {}
};

```

```

_Task Arbiter {
    void main() {
        int i = 0;
        for ( ;; ) {
            // cycle for request => no starvation
            for ( ; ! intent[i]; i = (i + 1) % 5 ) {}
            serving[i] = true;
            while ( intent[i] ) {}
            serving[i] = false;
        }
    }
};

```

- Mutual exclusion becomes a synchronization between arbiter and each waiting client.
- Arbiter cycles through waiting clients \Rightarrow no starvation.
- Does not require atomic assignment \Rightarrow no simultaneous assignments.
- Cost is creation, management, and execution (continuous spinning) of the arbiter task.

3.16 Hardware Solutions

- Software solutions to the critical section problem rely on nothing other than shared information and communication between threads.
- Hardware solutions introduce level below software level.
- At this level, it is possible to make assumptions about execution that are impossible at the software level. E.g., that certain instructions are executed atomically.
- This allows elimination of much of the shared information and the checking of this information required in the software solution.
- Certain special instructions are defined to perform an atomic read and write operation.
- This is sufficient for multitasking on a single CPU.
- Simple lock of critical section failed:

```

int Lock = OPEN;           // shared
// each task does
while ( Lock == CLOSED ); // fails to achieve
Lock = CLOSED;             // mutual exclusion
// critical section
Lock = OPEN;

```

- Imagine *if* the C conditional operator ? is executed atomically.

```

while((Lock==CLOSED? CLOSED : (Lock=CLOSED),OPEN)
      ==CLOSED);
// critical section
Lock = OPEN;

```

- Works for N threads attempting entry to critical section and only depend on one shared datum (lock).
- However, rule 5 is broken, as there is no bound on service.
- *Unfortunately, there is no such atomic construct in C.*
- Atomic hardware instructions can be used to achieve this effect.

3.16.1 Test/Set Instruction

- The test-and-set instruction performs an atomic read and fixed assignment.

```

int Lock = OPEN; // shared
int TestSet( int &b ) {
    /* begin atomic */
    int temp = b;
    b = CLOSED;
    /* end atomic */
    return temp;
}

void Task::main() { // each task does
    while( TestSet( Lock ) == CLOSED );
    /* critical section */
    Lock = OPEN;
}

```

- if test/set returns open \Rightarrow loop stops and lock is set to closed
- if test/set returns closed \Rightarrow loop executes until the other thread sets lock to open
- In the multiple CPU case, memory must also guarantee that multiple CPUs cannot interleave these special R/W instructions on the same memory location.

3.16.2 Swap Instruction

- The swap instruction performs an atomic interchange of two separate values.

```

int Lock = OPEN; // shared
Swap( int &a, &b ) {
    int temp;
    /* begin atomic */
    temp = a;
    a = b;
    b = temp;
    /* end atomic */
}

void Task::main() { // each task does
    int dummy = CLOSED;
    do {
        Swap( Lock,dummy );
    } while( dummy == CLOSED );
    /* critical section */
    Lock = OPEN;
}

```

- if swap returns open \Rightarrow loop stops and lock is set to closed
- if swap returns closed \Rightarrow loop executes until the other thread sets lock to open

3.16.3 Compare/Assign Instruction

- The compare-and-assign instruction performs an atomic compare and conditional assignment (erronously called compare-and-swap).

```

int Lock = OPEN; // shared
bool CAssn( int &val,      void Task::main() { // each task does
             int comp, int nval ) {           while ( ! CAssn(Lock,OPEN,CLOSED));
             /* begin atomic */                /* critical section */
             if (val == comp) {                Lock = OPEN;
                 val = nval;                    }
                 return true;
             }
             return false;
             /* end atomic */
         }
     }

```

- if compare/assign returns open \Rightarrow loop stops and lock is set to closed
- if compare/assign returns closed \Rightarrow loop executes until the other thread sets lock to open
- compare/assign can build other solutions (stack data structure) with a bound on service but with short busy waits.
- However, these solutions are complex.