Concurrency

Arnav Gupta

October 27, 2024

Contents

1 Concurrency

Thread: independent sequential execution path through a program, scheduled for execution separately and independently of other threads

Process: program component that has its own thread and has the same state information as a coroutine

Task: similar to process but is reduced along some particular dimension and shares common memory (lightweight process)

Parallel Execution: when 2+ operations occur simultaneously (requires multiple processors)

Concurrent Execution: any situation in which execution of multiple threads appears to be performed in parallel, caused by threads of control rather than processors

2 Why Write Concurrent Programs

Dividing a problem into threads is a programming technique and may be natural to expressing a problem. Can also enhance execution time efficiency by taking advantage of concurrency in an algorithm.

3 Why Concurrency is Difficult

Difficult to:

- coordinate
- specify how a problem should be broken up
- specify if and how parts interact
- specify how info is communicated during interaction
- debug due to non-deterministic order
- reason about

4 Concurrent Hardware

Concurrent execution is possible with one CPU, but **multitasking** requires **multiprocessing**.

Parallelism occurs by context switching between threads on the CPU. Most issues in concurrency can be illustrated without parallelism.

Task switching may occur at non-deterministic program locations, between any two machine instructions.

Switching happens explicitly but conditionally when calling routines, since they may not context switch depending on hidden internal state.

Switching can happen implicitly because of an external **interrupt** independent of program execution.

If an interrupt affects **scheduling**, it is **preemptive**, otherwise it is **non-preemptive**. Context switching is instruction level for preemptive and routine level for non-preemptive.

Pointers among tasks work because memory is shared. Thought, concurrent execution of threads is possible with 1+ CPUs on different computers with

separate memories, in a **distributed system**. For a distributed system, pointers among tasks do not work.

5 Execution State

Thread states can be new, ready, running, blocked, and halted.

*State transitions are initiated in response to events:

- entering the system (new → ready)
- assigning thread to computing resource (ready → running)
- timer alarm for preemption (running → ready)
- long-term delay vs spinning (running → blocked)
- completion of delay (blocked → ready)
- normal completion or error (running → halted)

Thread cannot bypass the ready state during a transition so the scheduler maintains complete control of the system.

Sequential operations, however small, are unsafe in a concurrent program.

6 Threading Model

For multiprocessor systems, a **threading model** defines the relationship between threads and CPUs.

OS manages CPUs by providing logical access via **kernel threads** (virtual processors) scheduled across the CPUs.

Can have more kernel threads than CPUs to provide multiprocessing.

A process may have multiple kernel threads to provide parallelism. A program may have 1+ user threads scheduled on each of its process's kernel threads.

User threads are low cost, kernel threads are high cost.

Kernel threading uses exactly the same number of user and kernel threads, **user threading** uses different numbers of user and kernel threads.

Can recursively add stackless **nano threads** on top of stackful user threads and a virtual machine below the OS.

7 Concurrent Systems

Major types of concurrent systems:

1. attempt to discover implicit concurrency in an otherwise sequential program
   (a) limit to how much concurrency can be found
   (b) only works on certain problems
2. provide concurrency through implicit constructs
   (a) concurrency accessed indirectly via specialized mechanisms
   (b) threads implicitly managed
3. provide concurrency through explicit constructs
   (a) concurrency accessed directly
   (b) threads explicitly managed

Discovery and implicit are built from explicit, since explicit threads are required at some level.

Implicit and explicit mechanisms are complementary. Implicit is limited, so maximum concurrency requires explicit.

As concurrency increases, so does the complexity to express and manage it.

8 Speedup

Speedup is defined as

$$S_C = T_1/T_C$$

where C is the number of CPUs and T_1 is sequential execution.

Types of speedup:

- **non-linear** is the most common
- **sub-linear** is less common, defined as $S_C < C$

- **linear** is ideal, defined as $S_C = C$
- **super linear** is unlikely, defined as $S_C > C$

Aspects affecting speedup (assuming sufficient parallelism):

1. amount of concurrency
2. critical path among concurrency
3. scheduler efficiency

An algorithm/program is composed on sequential and concurrent sections.

Amdahl's Law: concurrent section of a program P, gives that the sequential section is $1 - P$, so the maximum achievable speedup using C CPUs is

$$S_C = \frac{1}{(1-P) + P/C}$$

As $C \to \infty$, $P/C \to 0$, so the maximum speedup is $\frac{1}{1-P}$, which is the time for the sequential section.

Concurrent programming consists of minimizing sequential section $1 - P$.

This law ignores any administrative costs of concurrency.

While sequential sections bound speedup, concurrent sections bound speedup by the **critical path** of computation:

- **independent execution**: all threads created together and do not interact
- **dependent execution**: threads created at different times and interact (critical path exists)

Longest path bounds speedup. Speedup can also be affected by scheduler efficiency/ordering, but no control over this. Greedy scheduling is less concurrent and LIFO scheduling gives priority to newly waiting tasks (starvation).

In general, benefit comes when many programs achieve some speedup so there is an overall improvement on a multiprocessor computer.

9 Thread Creation

Concurrency requires:

1. creation: cause another thread of control to come into existence
2. synchronization: establish timing relationships among threads
3. communication: transmit data among threads

Thread creation must be a primitive operation.

9.1 COBEGIN/COEND

Compound statement with statements run by multiple threads.

```
#include <uCobegin.h>
COBEGIN // threads execute statement in block
    BEGIN p0( 0 ); END
    BEGIN p1( 1 ); END
    BEGIN p2( 2 ); END
    BEGIN p3( 3 ); END
COEND // initial thread waits for all internal threads to finish (synchronize) before c
```

Order and speed of internal thread execution is unknown, so it is implicit concurrency.

Thread graph represents thread creation. This construct is restricted to creating trees of threads.

9.2 START/WAIT

Start thread in routine and wait (join) at thread termination, allowing arbitrary thread graph.

```
#include <uCobegin.h>
decltype(START( p, 5 )) tp = START( p, 5 );
{
    s1; // continue execution, without waiting for tp
}
decltype(START( f, 8 )) tf = START( f, 8 );
{
    s2; // continue execution, without waiting for tf
}
WAIT( tp ); // wait for tp to finish
{
    s3; // continue execution, without waiting for tf
}
```

```
WAIT( tf ); // wait for tf to finish
```

Allows same routine to be started multiple times with different arguments. Explicit concurrency since it is specified where threads should be started and terminated explicitly.

START/WAIT can simulate COBEGIN/COEND.

9.3 Actor

Unit of work without a thread, like BEGIN/END.

An executor thread matches an actor with a message and runs the actor's behaviour for the message, like COBEGIN/COEND.

Communication is via a polymorphic queue of messages with dynamic type checking. Usually no shared info among actors and no blocking is allowed.

Must declare messages and actors.

```
#include <uActor.h>

struct StrMsg : public uActor::Message {
    string val;
    StrMsg( string val ) : Message( uActor::Delete ), val( val ) {} // delete after use
};

_Actor Hello { // derived from public uActor
    Allocation receive( Message & msg ) {
        Case( StrMsg, msg ) {
            ...;
            msg_d->val; // access derived message
            ...;
        } else Case( StopMsg, msg ) return Delete; // delete actor
        return Nodelete; // reuse actor
    }
};

int main() {
    uActor::start(); // start actor system
    // create actors and pass messages with | operator
    uActor::stop(); // wait for all actors to terminate
}
```

Implicit concurrency for actors since no defined threads of control.

Must start actor system to create thread pool and wait for actors to complete. Actors must receive at least 1 message to start. Messages are received and executed sequentially in FIFO order.

```
class uActor {
    public:
        enum Allocation {
            Nodelete, // actor/message persists after return from receive
            Delete, // actor/message is deleted after return from receive
            Destroy, // actor/message's dtor is called after return from receive (stora
            Finished, // actor/message marked finished but neither dtor not delete call
        };
        struct Message {
            Allocation allocation;
            ...;
        }
        static struct StartMsg : public uActor::SenderMsg {} startMsg;
        static struct StopMsg : public uActor::SenderMsg {} stopMsg;
        static void start();
        static bool stop();
    private:
        Allocation allocation;
};
```

Executor finds an actor with messages and passes the first message to the actor to process. After actor returns, the executor checks what to do with the message and actor.

9.4 Thread Object

Thread is an object to leverage class features and uses allocation/deallocation to define thread lifetime.

```
_Task T { ... }
{
    T t; // like COBEGIN/COEND
}
T * t = new T; // like START
delete T; // like WAIT
```

Explicit concurrency since the thread object is explicitly started and stopped.

10 Termination Synchronization

A thread terminates when:

- it finishes normally
- it finishes with an error
- it is killed by its parent or sibling
- parent terminates

Children can continue to exist after the parent terminates (though rare). Synchronizing at termination possible for terminating threads and can be used to perform final communication.

11 Divide and Conquer

Ability to subdivide work across data, so the work performed on each data group is identical to the work performed on data as a whole.

Must ensure that administration of concurrency does not exceed cost of work.

`COFOR` logically creates end − start threads, indexed from start to end - 1. This is implicit concurrency since these threads are logically created in an undefined manner.

12 Exceptions

Can be handled locally within a task, nonlocally among coroutines, or concurrently among tasks.

For coroutines, `UnhandledException` goes to the last resumer. For tasks, `UnhandledException` goes to the task's joiner.

Nonlocal exceptions between a task and a coroutine are the same as between coroutines. A concurrent exception provides an additional kind of communication among tasks.

For a concurrent raise, the source execution may only block while queuing the event for delivery at the faulting execution.

After an event is delivered, the faulting execution is not interrupted and it instead polls:

- when an `_Enable` statement begins and ends
- after a call to `suspend` or `resume`
- after a call to `yield`
- after a call to `_Accept` unblocks for `RendezvousFailure`

13 Synchronization and Communication During Execution

Synchronization occurs when one thread waits until another thread has reached a certain execution point.

Synchronization is needed in transmitting data between threads, where one thread must be ready to transmit and another must be ready to receive simultaneously.

Busy wait: a loop waiting for an event among threads

14 Communication

If threads are in the same memory, info can be transferred by value or address (reference).

If threads are not in the same memory (distributed), transferring info by value is straightforward (but not by address).

15 Critical Section

Threads may access non-concurrent objects, so a problem occurs if multiple threads attempt to operate on the same object simultaneously.

Atomic operations are fine, so it is often necessary to make an operation atomic.

Critical Section: group of instructions on associated data that must be performed atomically

Mutual Exclusion: preventing simultaneous execution of a critical section by multiple threads

Must determine when concurrent access is allowed/prevented.

Can detect any sharing and serialize all access, but this is wasteful for reads. Better to allow multiple readers or a single writer.

Must minimize the amount of mutual exclusion to maximize concurrency.

16 Static Variables

Shared among all objects generated by that class, so they may require mutual exclusion.

Static variables can be safely used in a task constructor if task objects are generated serially. This approach only works if one task creates all objects and initialization data is internal.

Best to avoid shared static variables in a concurrent program.

17 Mutual Exclusion Game

Consider mutual exclusion as a game with the following rules:

1. **safety**: only one thread can be in a critical section at a time with respect to a particular object
2. threads may run at arbitrary speed and in arbitrary order, while the underlying system guarantees that all threads make some progress
3. if a thread is not in the entry or exit code controlling access to the critical section, it may not prevent other threads from entering the critical section
4. in selecting a thread for entry to a critical section, a selection cannot be postponed indefinitely, which is **liveness**, and not satisfying this rule is **indefinite postponement** or **livelock**
5. after a thread starts entry to the critical section, it must eventually enter, and not satisfying this rule is **starvation**

Indefinite postponement and starvation are related by busy waiting. Looping for an event in mutual exclusion must ensure eventual progress.

If threads are not serviced in first-come first-serve order, there is a notion of **unfairness**. Unfairness implies waiting threads are overtaken by arriving threads, called **barging**.

18 Software Solutions

Can have a self-testing critical section.

Using a basic permission lock with an enum breaks rule 1.

Using alternation breaks rule 3.

Using basic declare intent causes livelock, breaking rule 4.

Using basic retract intent can also cause livelock, breaking rule 4.

Using prioritized retract intent causes starvation, breaking rule 5.

18.1 Dekker's Algorithm

```
enum Intent { WantIn, DontWantIn };
Intent * Last;
_Task Dekker {
    Intent & me, & you;
    void main() {
        for (int i = 1; i <= 1000; i += 1) {
            for ( ;; ) { // entry protocol, high priority
                me = WantIn;
                if ( you == DontWantIn ) break; // does not want in
                if ( ::Last == &me ) { // low priority task
                    me = DontWantIn; // retract intent
                    while ( ::Last == &me && // this line by Hesselink
                            you == WantIn ) {}
                }
            }
            CriticalSection();
            // exit protocol
            if ( ::Last != &me ) { // this line by Hesselink
                ::Last = &me;
            }
            me = DontWantIn;
        }
```

```
    }
};
```

Dekker's Algorithm appears to be **read/write-safe** (RW-safe):

- on cheap multi-core computers, read/write is not atomic
- **RW safe**: a mutual exclusion algorithm works for non-atomic read/write
- Dekker has no simultaneous write/write since intent reset is after alternation in exit protocol
- Dekker has simultaneous read/write but all are equality so it works if final value never flickers (bits changing during write)

In 2015, Hesselink found 2 failure cases if the values flicker, and added the two lines.

Dekker has **unbounded overtaking** (not starvation) since race loser retracts intent, but this is allowed.

18.2 Peterson's Algorithm

```
enum Intent { WantIn, DontWantIn };
Intent * Last;
_Task Peterson {
    Intent & me, & you;
    void main() {
        for (int i = 1; i <= 1000; i += 1) {
            me = WantIn;
            ::Last = &me;
            while ( ::Last == &me && you == WantIn ) {}
            CriticalSection();
            me = DontWantIn;
        }
    }
};
```

Peterson's Algorithm is RW-unsafe so it requires atomic read/write operations. Peterson also has **bounded overtaking** since the race loser does not retract intent, since the thread exiting critical section excludes itself for reentry.

18.3 N-Thread Prioritized Entry

```
enum Intent { WantIn, DontWantIn };
_Task NTask {
    Intent * intents;
    int N, priority, i, j;
    void main() {
        for ( i = 1; i <= 1000; i += 1 ) {
            do {
                intents[priority] = WantIn;
                for ( j = priority - 1; j <= 0; j -= 1 ) {
                    if ( intents[j] == WantIn ) {
                        intents[priority] = DontWantIn;
                        while ( intents[j] == WantIn ) {}
                        break;
                    }
                }
            } while ( intents[priority] == DontWantIn );

            for ( j = priority + 1; j < N; j += 1 ) {
                while ( intents[j] == WantIn ) {}
            }
            CriticalSection();
            intents[priority] = DontWantIn;
        }
    }
};
```

This algorithm can cause starvation for lower priority tasks.

Only N bits are needed since only the want in status is needed for each task. No known solution exists for all 5 rules using only N bits. Other N thread solutions use more memory. The best solutions are 3-bit RW-unsafe and 4-bit RW-safe.

18.4 N-Thread Bakery (Tickets)

```
_Task Bakery {
    int * ticket, N, priority;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
```

```
            ticket[priority] = 0;
            int max = 0;
            for ( int j = 0; j < N; j += 1 ) {
                int v = ticket[j];
                if ( v != INT_MAX && max < v ) max = v;
            }
            max += 1;
            ticket[priority] = max;

            for ( int j = 0; j < N; j += 1 ) {
                while ( ticket[j] < max ||
                    ( ticket[j] == max && j < priority ) ) {}
            }
            CriticalSection();
            ticket[priority] = INT_MAX;
        }
    }
};
```

Ticket value of INT_{MAX} means they don't want in. Ticket value of 0 means selecting ticket.

Tickets are not unique, position gives secondary priority. Low ticket and position means high priority.

This uses NM bits where M is the ticket size.

Lamport version is RW-safe. Hehner/Shyamasundar is RW-unsafe since the `ticket[priority] = max` line can flicker to INT_{MAX} which would allow other tasks to proceed.

18.5 Tournament

Consider a binary tree with $\lceil N/2 \rceil$ start nodes and $\lceil \log N \rceil$ levels. A thread is assigned to a start node, where it begins mutual exclusion.

Each node is like a Dekker or Peterson 2 thread algorithm. The tree structure tries to find a compromise between fairness and performance.

Exit protocol must retract intents in reverse order. Otherwise a race between retracting/released threads along the same tree path.

No overall livelock or starvation since each node ensures neither occurs at

the node level.

Tournament algorithm RW safetyy depends on MX algorithm since tree traversal is local to each thread. Tournament algorithms have unbounded overtaking as no synchronization exists among the nodes of the tree.

For 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.

```
_Task TournamentMax {
    struct Token { int intents[2], turn };
    static Token ** t;
    int depth, id;

    void main() {
        unsigned int lid;
        for ( int i = 0; i < 1000; i += 1 ) {
            lid = d;
            for ( int lv = 0; lv < depth; lv += 1 ) {
                binary_prologue( lid & 1, &t[lv][lid >> 1] );
                lid >>= 1;
            }
            CriticalSection( id );
            for ( int lv = depth - 1; lv >= 0; lv -= 1 ) {
                lid = id >> lv;
                binary_prologue( lid & 1, &t[lv][lid >> 1] );
            }
        }
    }
}
```

This can be optimized to 3 shifts and XOR using Peterson 2-thread for binary.

Path from leaf to root is fixed per thread, so table lookup is possible using max or min tree.

18.6 Arbiter

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

```
bool intents[N], serving[N];
```

```
_Task Client {
    int me;
    void main() {
        for ( int i = 0; i < 100; i += 1 ) {
            intents[me] = true;
            while ( !serving[me] ) {}
            CriticalSection();
            serving[me] = false;
        }
    }
}

_Task Arbiter {
    void main() {
        int i = N;
        for ( ;; ) {
            do {
                i = ( i + 1 ) % n;
            } while ( !intents[i] );
            intents[i] = false;
            serving[i] = false;
            while ( serving[i] ) {}
        }
    }
}
```

Mutual exclusion becomes synchronization between arbiter and clients. Arbiter never uses the critical section, so no indefinite postponement (livelock). Arbiter cycles through waiting clients so no starvation.

This is RW unsafe due to read flicker, like when modifying `intents` and `serving`.

Cost comes from the creation, management, and execution (busy waiting) of the arbiter task.

19 Hardware Solutions

Hardware solutions occur below the software level.

This allows the elimination of the shared info and checking of this info re-

quired in the software solution.

This approach uses special instructions for atomic read/write.

19.1 Test/Set Instruction

Performs an atomic read and fixed assignment.

```
int TestSet( int & b ) {
    // begin atomic
    int temp = b;
    b = CLOSED;
    // end atomic
    return temp;
}
```

If instruction returns open, lock must have been open before, so can enter critical section.

No guarantee of eventual progress, since a task can keep locking and opening.

In the case of multiple CPUs, the bus must ensure multiple CPUs cannot interleave special RW instructions on the same memory location.

19.2 Swap Instruction

Performs an atomic interchange of two values.

```
void Swap( int & a, & b ) {
    int temp;
    // begin atomic
    temp = a;
    a = b;
    b = temp;
    // end atomic
}
```

If variable swapped with lock becomes OPEN, then lock must have ben open, so can enter critical section.

19.3 Fetch and Increment Instruction

Performs an atomic increment between the read and write.

```
int FetchInc( int & val ) {
    // begin atomic
    int temp = val;
    val += 1;
    // end atomic
    return temp;
}
```

Can be generalized to add or subtract any value.

Lock counter can overflow during busy waiting and starvation, which occurs if a task resets the counter and then immediately re-enters. This can be fixed with a ticket counter, where `acquire` atomically sets the ticket and `release` increments the next ticket to be served.