20
Concurrency

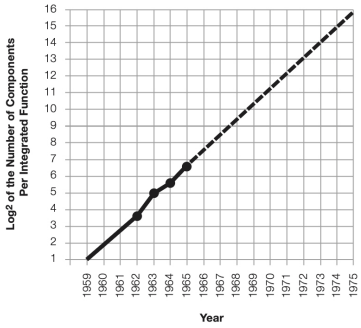

Figure 20.1: Gordon Moore's chart on the basis of which his 1965 eponymous "law" was extrapolated.

In 1965, Gordon Moore, one of the founders of the pioneering electronics company Fairchild Semiconductor, noted the exponential growth in the number of components that were being placed on integrated circuit chips, the building blocks of all kinds of electronics but especially of computers. Extrapolating from just four points on a chart (Figure 20.1), Moore saw that the number of integrated circuit components had been growing at "a rate of roughly a factor of two per year", and he expected that rate to continue for the foreseeable future. Ten years later, he revised his estimate to a doubling per two years. This prediction became "Moore's law", and has been generalized to many other aspects of computer technology and performance.

The generalized form of Moore's law would have it that computer performance, measured, say, in total number of instructions executed per second, should grow exponentially as well, as indeed it has. Empirical data on the number of standardized operations performed per second, charted as squares in Figure 20.2, shows that Moore's law as applied to the performance of microprocessor chips has held up remarkably well for many decades. Data on clock speed, the rate at which individual instructions can be executed, charted as circles, shows a different story. The clock speed of the processors showed the same exponential growth through the mid to late 2000's, but flattened after that. What could account for this differential? If the processors weren't executing instructions faster, how could they be executing more instructions in the same amount of time. The answer is given by the third series, shown as triangles in Figure 20.2, which graphs the number of processors per chip. Over the last decade or so, we've seen a regular rise in the number of processors per chip, making up the difference in Moore's law by having *multiple instructions executed in parallel*.

These days, specialized architectures like graphics processing units (GPUs) and AI accelerators take advantage of even larger scale paral-

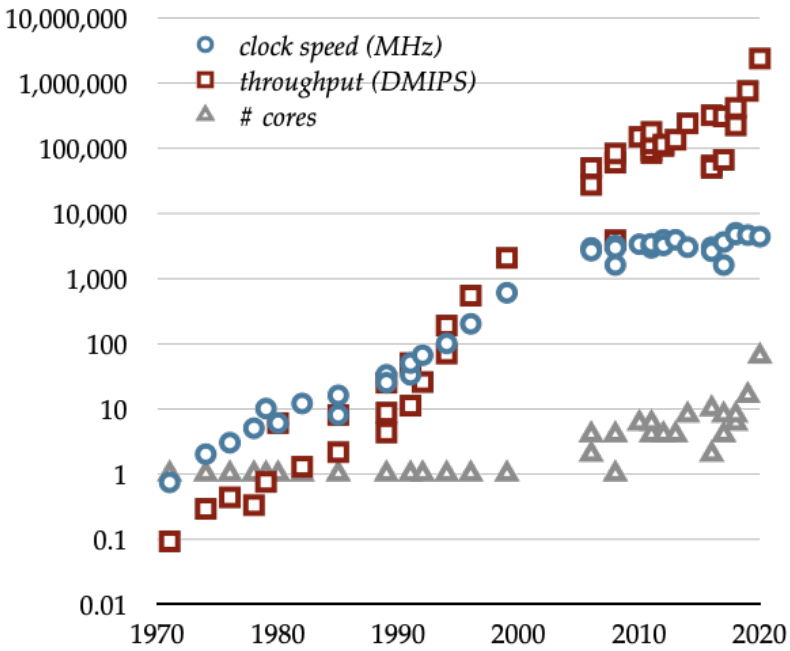

Figure 20.2: Chart showing growth in clock speed (in megaHertz (MHZ), as circles), throughput (in Dhrystone millions of instructions per second (DMIPS), as squares), and number of cores per chip for Intel and recent AMD microcomputer chips. Note the logarithmic vertical scale.

lelism to speed up complex highly structured computations for graphics or machine learning. In fact, parallel computing is responsible for the recent breakthroughs in machine learning performance, and is in large part the future of maintaining the tremendous performance improvements that Moore's law has captured.

There's no free lunch. Programming computations that happen concurrently introduces new challenges, requiring new programming abstractions to manage them. In this chapter, we'll explore some of the promise, difficulty, and tools of concurrent programming. As usual, in the effort to simplify the management of the daunting problems of concurrency, new abstractions will be crucial.

20.1 Sequential, concurrent, and parallel computation

It will be helpful, in thinking about these issues, to imagine computation as proceeding sequentially in a series of small atomic steps. Indeed, computation *does* proceed that way, down at the level of abstraction at which the hardware processors operate. The role of a compiler is to translate programs written using higher-level abstractions down to a sequence of atomic instructions directly runnable on (possibly virtual) hardware.[1]

Suppose we have two tasks (A and B) to complete, each requiring

[1] Exactly what constitutes an atomic step depends on the particularities of the hardware; we needn't concern ourselves with the details here. We'll just assume that operations like reading a value from memory (as, for instance, accessing a variable's value), modifying a value in memory (instantiating a variable or updating a mutable variable, for instance), performing a simple operation on retrieved values (arithmetic operations, for instance), and the like are atomic. In the examples we'll use, we'll write the atomic steps on separate lines, so that any line of code will be assumed to execute atomically.

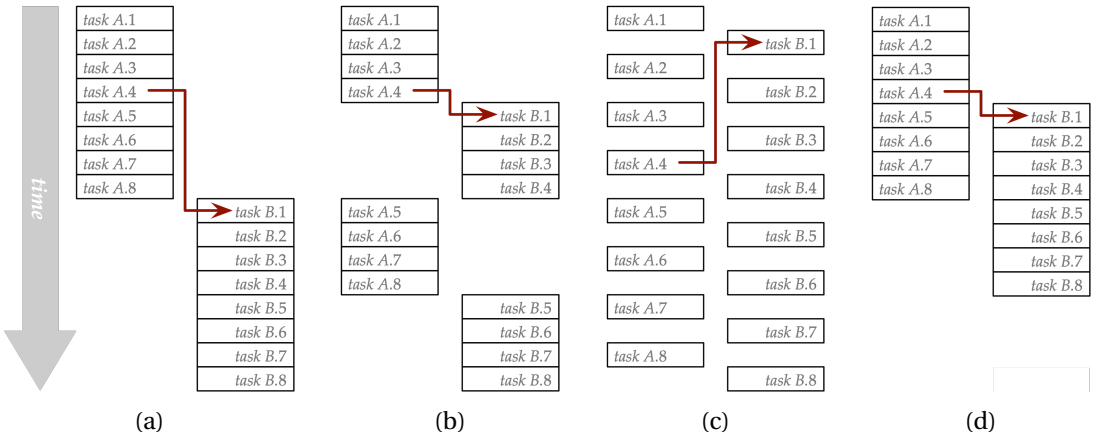

Figure 20.3: Two tasks running in various forms of sequential and concurrent computation. Each task is depicted as atomic steps (the individual boxes) executing through time (running from top to bottom). (a) Task A runs sequentially to completion before task B. (b) Coarsely concurrent execution of the two tasks, with some steps of task B first running after four steps of task A. (c) Finer concurrent execution, interleaving at each atomic step. (d) Parallel computation of the tasks, with task B beginning execution after the fourth step of task A and running simultaneously. The arrows denote a dependency requiring task B.1 to run after task A.4. Note that that dependency is violated in (c).

execution of a sequence of atomic steps. One way of completing the tasks is to execute the two tasks SEQUENTIALLY, all of the steps of Task A before any of the steps of Task B, as depicted in Figure 20.3(a). Alternatively, we might execute some of the steps in Task A, then some from Task B, then the remainder of Task A and the remainder of Task B (Figure 20.3(b)). The tasks are said to be running CONCURRENTLY. Even more fine-grained concurrency is possible of course (Figure 20.3(c)).

Why might such concurrency be useful? Through concurrent execution, Task B might be able to generate some useful behavior earlier than having to wait for Task A to complete. Perhaps Task A part way through its execution computes some value that is needed by Task B, or vice versa. Waiting for Task A to complete may postpone Task B for a long time. Indeed, some computations are intended never to complete. Think of the process that runs a bank ATM, which is always running a single program to handle requests from users as they walk up to and interact with it. Although the ATM process never completes, other processes may want to interact with it and intersperse their computations on the same processor, perhaps to report changes to a central database. In sum, concurrency allows multiple separate processes to interact and communicate without requiring one of them to complete before the other begins.

Where such concurrency is possible, a further benefit can accrue – carrying out the steps of the tasks IN PARALLEL (Figure 20.3(d)) by making use of separate hardware for processing the sequences of instructions. Parallelism allows both tasks to complete in fewer time steps, effectively trading time for "space" (hardware).

20.2 Dependencies

In taking advantage of concurrency or parallelism, delicate issues quickly arise when there are dependencies between the two sequences

of instructions. For instance, Task B might read a value at one of its steps (its first step, say) that Task A computes at its, say, fourth step. We've indicated such a dependency with the arrows in Figure 20.3.

If Task A and B run sequentially in that order, then of course the value generated by Task A will be available to Task B at the proper time. But concurrent computation is also possible, say if Task A completes its first four steps before Task B begins. But other interleavings can be problematic, for instance, if Tasks A and B interleave after each step. Task B will then attempt to make use of the value that Task A will calculate before it has actually been calculated. This kind of dependency, where one task must read a value only after another task writes it, is sometimes referred to as a READ-AFTER-WRITE DEPENDENCY. What happens when concurrent execution violates the read-after-write dependency may not be well defined, but it certainly is not a good situation.

In addition to read-after-write dependencies, other kinds of dependencies (WRITE-AFTER-READ, WRITE-AFTER-WRITE) are also important. The details are beyond the scope of this discussion. At this point, we're merely concerned with how to allow concurrency while avoiding violations of ordering dependencies whatever they might be.

In summary, if we just allow tasks to interleave however they happen to, with no control over which parts of which task run when, dependencies introduce a kind of race between the tasks. Will Task A's write step run faster and execute, as it should, before Task B's read? Or will Task B win the race, performing its read before task A has a chance to write? This kind of RACE CONDITION leads to the possibility of a new kind of error. Gaining the benefits of concurrency and parallelism, while avoiding race conditions and other new classes of errors, is the challenge of concurrent and parallel programming.

20.3 Threads

In order to demonstrate these issues and experiment with abstractions that can help avoid these new classes of errors, we need a way to implement concurrency. OCaml provides a programming abstraction that allows us to experiment with concurrency, the THREAD. A thread can be thought of as providing a separate virtual processor.[2]

Suppose we need to do two tasks – call them Task A and B as before – implemented as OCaml functions named accordingly. Perhaps we want to sum the results returned by these two tasks. We can easily execute them *sequentially*, task A before B:

```
let resultA = taskA () in
let resultB = taskB () in
```

[2] OCaml thread's provide concurrency, not true parallelism, but the issues they raise apply equally well to parallel processing, so they're all we'll need to demonstrate the problems. Other OCaml modules, and aspects of many other programming languages, provide concurrency and parallelism constructs that introduce just the same issues.

```
(* log thread_name msg -- Prints a message recording that a
   thread with the given `thread_name` has generated a log
   message `msg`. Also prints an indication of time in seconds
   since an initialization time. Used for tracking concurrent
   executions. *)
let log =
  (* store a fixed marker time for comparison *)
  let init_time = Unix.gettimeofday () in
  fun thread_name msg ->
    Printf.printf "[%3.4f %s: %s]\n%!"
                  ((Unix.gettimeofday ()) -. init_time)
                  thread_name
                  msg ;;

(* task_delayed name delay value -- Prints a message
   recording that a thread with the given `thread_name` has
   generated a log message `msg`. Also prints an indication of
   time in seconds since an initialization time. Used for
   tracking concurrent executions. *)
let task_delayed (name : string)
                 (delay : float)
                 (value : 'a)
               : 'a =
  log name "starts";
  Thread.delay delay;
  log name "ends";
  value ;;

(* Two sample tasks taking differing lengths of time and
   returning different values. *)
let task_short () = task_delayed "task_short" 0.1 1 ;;
let task_long  () = task_delayed "task_long " 0.2 2 ;;
```

Figure 20.4: For reference, some logistical code used in the concurrency demonstrations.

```
resultA + resultB ;;
```

We can think of the two tasks (along with the computation of their sum) as being executed in a single thread of computation. The semantics of the `let` construct ensures that `taskA ()` will be evaluated, generating `resultA`, before `taskB ()` begins its evaluation.

In order to demonstrate the idea, and prepare for the significantly more subtle examples to come, we define a test function that takes two functions as its argument, which play the roles of tasks A and B.

```
# let test_sequential taskA taskB =
#   let resultA = taskA () in
#   let resultB = taskB () in
#   resultA + resultB ;;
val test_sequential : (unit -> int) -> (unit -> int) -> int = <fun>
```

We can test this sequential computation using some simulated tasks. The unit function `task_short` simulates a task that engages in a shorter computation (0.1 seconds) returning the value 1. The corresponding function `task_long` takes longer (0.2 seconds) and returns the value 2. (The details of how they're implemented aren't important, but for completeness, they're provided in Figure 20.4.) Let's test it out.

```
# test_sequential task_short task_long ;;
[1.1166 task_short: starts]
[1.2168 task_short: ends]
[1.2169 task_long : starts]
[1.4171 task_long : ends]
- : int = 3
```

The test returns the summed results 3. Along the way, various key events are logged. We see the start of the short task and its ending, followed by the start and end of the long task, indicating their sequentiality. The logged start and end times indicate that the short and long tasks required about 0.1 and 0.2 seconds, respectively, together requiring 0.3 seconds, as expected.

If we'd like the two tasks to execute *concurrently*, we can establish a separate thread (that is, a separate virtual processor) corresponding to `taskA`. We refer to this process as FORKING a new thread. OCaml provides for creating and manipulating threads in its `Thread` module.[3] To fork a new thread, we use the `Thread.create` function, which takes a function and its argument and returns a *separate new thread of computation* (a value of type `Thread.t`) in which the function is applied to its argument. Its type is thus `('a -> 'b) -> 'a -> Thread.t`. So we can evaluate tasks A and B in separate threads, concurrently, as follows:

```
let threadA = Thread.create taskA () in
let resultB = taskB () in
...
```

[3] The `Thread` module is part of OCaml's threads library, which allows for creating multiple concurrent threads. To make use of the library in the REPL, you'll need to make it available with

```
#use "topfind" ;;
#thread ;;
```

The evaluation of the `Thread.create` expression returns immediately, without waiting for the result of the application of `taskA` to `()` to finish in the new thread. Thus when `taskB ()` is evaluated, it doesn't wait until `taskA` completes.[4]

[4] The `Thread` library allows for concurrent execution of the various threads forked in the process, not parallel execution. An exception is that the `Thread.delay` function, which we use to simulate computations that take a long time, allows other threads to continue to run during the delay period.

20.4 Interthread communication

We've enabled two tasks to operate concurrently using threads. But we have no way as of yet for threads to communicate with each other. For instance, in the example above, how can `taskA`, isolated in its own thread, inform the thread running `taskB` about its return value? Similarly, how can `taskB` communicate information to `taskA` if it needs to?

A simple mechanism for this interthread communication is for the threads to share mutable values, which serve as channels of communication between the threads. Let's start with how the created thread executing `taskA` can communicate its return value to the main thread that needs to calculate the sum.

We'll define another test function called `test_communication` to test the communication between two tasks executed in separate threads as above.

```
let test_communication taskA taskB =
  ...
```

We'll use a shared mutable value called `shared_result` of type `int option`, initially `None` since no result is yet available.

```
  ...
  let shared_result = ref None in
  ...
```

Now we can create a new thread for executing task A, saving its return value in the shared result.

```
  ...
  let _thread =
    Thread.create
      (fun () -> shared_result := Some (taskA ())) () in
  ...
```

In the original thread, we perform task B, saving its result.

```
  ...
  let resultB = taskB () in
  ...
```

Finally, we can extract the result from task A from the shared value, and compute with the two results, by adding them as before.

```
...
match !shared_result with
| Some resultA ->
   (* compute with the two results *)
   resultA + resultB
| None ->
   (* Oops, taskA hasn't completed! *)
   failwith "shouldn't happen!" ;;
```

Putting it all together, we have

```
# let test_communication taskA taskB =
#   let shared_result = ref None in
#   let _thread =
#     Thread.create
#       (fun () -> shared_result := Some (taskA ())) () in
#   let resultB = taskB () in
#   match !shared_result with
#   | Some resultA ->
#      (* compute with the two results *)
#      resultA + resultB
#   | None ->
#      (* Oops, taskA hasn't completed! *)
#      failwith "shouldn't happen!" ;;
val test_communication : (unit -> int) -> (unit -> int) -> int =
  <fun>
```

Again, we can test using the simulated tasks. To start, we fork the new thread running the shorter task, with the longer task in the main thread.

```
# test_communication task_short task_long ;;
[2.1291 task_long : starts]
[2.1291 task_short: starts]
[2.2294 task_short: ends]
[2.3293 task_long : ends]
- : int = 3
```

the communication works as expected. The short task returns 1 – passed through and retrievable from the shared variable – and the long task returns 2. The test as a whole computes their sum, 3 as expected.

The logged events show the starting of the long task in the main thread, followed immediately by the short task starting in the newly created thread. The latter short thread completes quickly (it's shorter, after all), ending before the long task does. The main thread can extract the completed value for the short task and add it to the result from the long task.[5]

But what if the task in the forked thread takes longer than that in the main thread? We can simulate this by swapping the long and short tasks in the test function.

```
# test_communication task_long task_short ;;
[2.3809 task_short: starts]
```

[5] As before, the logged times indicate that the short and long tasks required about 0.1 and 0.2 seconds. This time, however, the overall computation requires only 0.2 seconds, since the `delay` function allows for some parallelism between the two threads (as noted in footnote 4. The simulation thus gives a hint of the potential for parallel processing to speed computation.

```
[2.3810 task_long : starts]
[2.4812 task_short: ends]
Exception: Failure "shouldn't happen!".
```

In this version of the test, the short task in the main thread completes before the forked thread has time to complete the long task and update the shared variable, leading to a run-time exception. The code has a race condition with respect to a read-after-write dependency. These two executions of the test demonstrate that, depending on which task "wins the race", the value to be read may or may not be written in time as it needs to be.

In general, one doesn't have the kind of detailed information about run times of various tasks as we have for `task_short` and `task_long`. This kind of concurrent computation, without careful controls, thus leads to indeterminacy at runtime. And debugging these intermittent bugs that can come and go, perhaps appearing only rarely, can be especially confounding. More tools are needed.

The lesson here is that the main thread shouldn't attempt to use the shared variable until the forked thread has completed. We thus need a way of guaranteeing that a thread has completed. One solution you may have thought of is to have the main thread test if the shared value has not been properly set, and if not, to just "try again later". We can implement this with a simple loop,

```
while !shared_result == None do
  Thread.delay 0.01
done;
```

which continually waits for a fraction of a second so long as the shared result has not been properly set, a technique called BUSY WAITING.

```
# let test_communication taskA taskB =
#   let shared_result = ref None in
#   let _thread =
#     Thread.create
#       (fun () -> shared_result := Some (taskA ())) () in
#   let resultB = taskB () in
#    while !shared_result == None do
#       Thread.delay 0.01
#    done;
#   match !shared_result with
#   | Some resultA ->
#      (* compute with the two results *)
#      resultA + resultB
#   | None ->
#      (* Oops, taskA hasn't completed! *)
#      failwith "shouldn't happen!" ;;
val test_communication : (unit -> int) -> (unit -> int) -> int =
  <fun>
```

The errant race condition is now handled properly.

```
# test_communication task_long task_short ;;
[3.5939 task_short: starts]
[3.5940 task_long : starts]
[3.6941 task_short: ends]
[3.7942 task_long : ends]
- : int = 3
```

But this kind of brute force trick of repeatedly polling the shared variable until it is ready is profligate and inelegant. It can waste computation that would be better allocated to other threads, and can waste time if the delay is longer than needed.

Instead, we'd like to be able to directly specify to wait *until the forked thread completes*. The companion to the *fork* function `Thread.create` is the *join* function `Thread.join`. `Thread.join` takes a thread as its argument and returns *only once that thread has completed*. By requiring the join before accessing the shared variable, we are guaranteed that the variable will have been updated at the time that we need it.

```
# let test_communication taskA taskB =
#   let shared_result = ref None in
#   let thread =
#     Thread.create
#       (fun () -> shared_result := Some (taskA ())) () in
#   let resultB = taskB () in
#    Thread.join thread;
#   match !shared_result with
#   | Some resultA ->
#      (* compute with the two results *)
#      resultA + resultB
#   | None ->
#      (* Oops, taskA hasn't completed! *)
#      failwith "shouldn't happen!" ;;
val test_communication : (unit -> int) -> (unit -> int) -> int =
  <fun>
```

Using this version of the test, the race condition is avoided, and the calculation completes properly.

```
# test_communication task_long task_short ;;
[4.5559 task_short: starts]
[4.5560 task_long : starts]
[4.6563 task_short: ends]
[4.7563 task_long : ends]
- : int = 3
```

20.5 Futures

The structure of this small example, in which a thread is forked to allow it to compute a return value that is needed in the future, is so common that it deserves its own abstraction, a kind of value dubbed

a FUTURE. This abstraction is implemented via two functions: The `future` function takes a task to be carried out for its return value, and returns a "future value". We can later use the `force` function to force the future value to be extracted when available. A module signature can help clarify the needed functionality:

```
# module type FUTURE =
#   sig
#     type 'result future
#
#     (* future fn x -- Forks a new thread within which `fn`
#        is applied to `x`. Immediately returns a `future`
#        which can be used to synchronize with the thread
#        and extract the result. *)
#     val future : ('arg -> 'result) -> 'arg -> 'result future
#
#     (* force fut -- Causes the calling thread to wait until the
#        thread computing the future value `fut` is done and then
#        returns its value. *)
#     val force : 'result future -> 'result
#   end ;;
module type FUTURE =
  sig
    type 'result future
    val future : ('arg -> 'result) -> 'arg -> 'result future
    val force : 'result future -> 'result
  end
```

There are multiple ways to implement this functionality, but we'll use the shared value method from the previous example. In this implementation, a future value (an element of the `future` type) is a record that contains the thread identifier in which the future task is being carried out and the mutable variable for communicating the result back to the calling thread.

```
# module Future : FUTURE =
#   struct
#     type 'result future = {tid : Thread.t;
#                            value : 'result option ref}
#
#     let future (f : 'arg -> 'result) (x : 'arg) : 'result future =
#       let r : 'result option ref = ref None in
#       let t = Thread.create (fun () -> r := Some (f x)) ()
#       in {tid = t; value = r}
#
#     let force (f : 'result future) : 'result =
#       Thread.join f.tid;
#       match !(f.value) with
#       | Some v -> v
#       | None -> failwith "impossible!"
#   end ;;
module Future : FUTURE
```

With the future abstraction in hand, the `test_communication` example above can be greatly simplified.

```
# let test_future taskA taskB =
#   let futureA =  Future.future  taskA () in
#   let resultB = taskB () in
#   Future.force futureA + resultB ;;
val test_future : (unit -> int) -> (unit -> int) -> int = <fun>

# test_future task_long task_short ;;
[6.3759 task_short: starts]
[6.3760 task_long : starts]
[6.4762 task_short: ends]
[6.5763 task_long : ends]
- : int = 3
```

This is hardly more complicated than the sequential version (`test_sequential`) that we started with above, requiring only the simple addition of the highlighted `future` call.

Exercise 198

Exercise 98 concerned implementing a fold operation over binary trees defined by

```
# type 'a bintree =
# | Empty
# | Node of 'a * 'a bintree * 'a bintree ;;
type 'a bintree = Empty | Node of 'a * 'a bintree * 'a bintree
```

Define a version of the fold operation, `foldbt_conc`, that performs the recursive folds of the left and right subtrees concurrently, making use of futures to ensure that results are available when needed.

20.6 Futures are not enough

The sharing of mutable data across two concurrent threads is a valuable ability. It implements a kind of communication channel between the threads. But managing this communication is complex. We've already seen this in the context of a thread's "return value". The calling thread mustn't read the shared variable that will be storing the called thread's return value until the latter has completed its computation and updated the return value. Managing this ordering is the whole point of the future/force abstraction.

Sharing mutable data across threads is a useful technique well beyond just allowing for return values to be communicated.

1. Threads may have need for coordinating data manipulation beyond the mere passing of a return value. For instance, think of multiple threads manipulating a shared database.

2. In the case of threads that are not intended to terminate, the whole notion of a return value is inapplicable. Importantly, not all concurrent computations are intended to terminate. Indeed, one of the

benefits of concurrency as a programming concept is that it allows multiple threads of nonterminating computation to interact. We still need to manage the interaction so that the concurrent computations satisfy the various dependencies among them without dangerous race conditions.

A standard example of this kind of concurrent nonterminating computation is the ATM. ATMs are computers that run a program that interacts with bank patrons to allow them to manipulate their bank accounts in various ways. The bank accounts constitute a shared database of mutable data. And because banks have multiple geographically distributed ATMs, multiple instances of the program are running concurrently, and potentially transforming the same shared data, the balances of the various accounts.

To demonstrate the problem, let's think of a bank as having multiple accounts each of which is an instance of an `account` class defined as follows:

```
# class account (initial_balance : int) =
#   object
#     val mutable balance = initial_balance
#
#     method balance = balance
#
#     method deposit (amt : int) : unit =
#       balance <- balance + amt
#
#     method withdraw (amt : int) : int =
#       if balance >= amt then begin
#         balance <- balance - amt;
#         amt
#       end else 0
#   end ;;
class account :
  int ->
  object
    val mutable balance : int
    method balance : int
    method deposit : int -> unit
    method withdraw : int -> int
  end
```

The `deposit` and `withdraw` methods both potentially affect the value of the mutable `balance` variable. The `withdraw` function, in particular, verifies that the balance is sufficient to cover the withdrawal amount, updates the balance accordingly, and returns the amount to be dispensed (0 if the balance is insufficient).

Now what happens when we try multiple concurrent withdrawals from the same account? To simulate such an occurrence, the following `test_wds` function carries out withdrawals of $75 and $50 in separate

threads (call them "thread A" and "thread B" for ease of reference) from a single account with initial balance of $100, using a `future` for the larger withdrawal. To track what goes on, the test function returns the amount dispensed in thread A and thread B, along with the final balance in the account.

```
# let test_wds () =
#   let acct = new account 100 in
#   let threadA_ftr = Future.future acct#withdraw 75 in
#   let threadB = acct#withdraw 50 in
#   let threadA = (Future.force threadA_ftr) in
#   threadA, threadB, acct#balance ;;
val test_wds : unit -> int * int * int = <fun>
```

What behavior would we like to see in this case? One or the other of the two withdrawals, whichever comes first, should see a sufficient balance, dispense the requested amount, and update the balance accordingly. The other attempted withdrawal should see a reduced and insufficient balance and dispense no funds. In particular, if task A completes first, the two accounts should see withdrawals of $75 and $0 respectively, leaving a balance of $25, that is, the simulation function should return the triple `(75, 0, 25)`. If task B completes first, the two accounts should see withdrawals of $0 and $50 respectively, leaving a balance of $50, that is, the simulation function should return the triple `(0, 50, 50)`. Let's try it.

```
# test_wds () ;;
- : int * int * int = (0, 50, 50)
```

In order to experiment with the possibility of interleavings of the various components of the withdrawals, we make two changes to the withdrawal simulation. First, we divide the balance update

```
balance <- balance - amt
```

into two parts: the computation of the updated balance and the update of the `balance` variable itself:

```
let diff = balance - amt in balance <- diff
```

Doing so separates the *reading* of the shared balance from its *writing*, allowing interposition of other threads in between. Then, we introduce some random delays at various points in the computation: before the withdrawal first executes, immediately after the balance check, and after computing the updated balance just before carrying out the update. For this purpose, we use a function `random_delay`, which pauses a thread for a randomly selected time interval.

```
# let random_delay (max_delay : float) : unit =
#   Thread.delay (Random.float max_delay) ;;
val random_delay : float -> unit = <fun>
```

Updating the withdrawal function to insert these delays, we have

```
method withdraw (amt : int) : int =
  random_delay 0.004;
  if balance >= amt then begin
    random_delay 0.001;
    let diff = balance - amt in
    random_delay 0.001;
    balance <- diff;
    amt
  end else 0 ;;
```

Here is a typical outcome from this simulation.

```
# test_wds () ;;
- : int * int * int = (75, 50, -25)
```

If we run the simulation many times, we see (Figure 20.5) that the result is quite variable. Certainly, there are many occurrences (about half) showing the desired behavior, with either $75 or $50 dispensed and a final balance of $25 or $50, respectively. But we also see plenty of instances where both withdrawals go through, dispensing both $75 *and* $50, leaving a final balance of $−25. Or $25. Or $50. The use of `future` ensures that the return value dependency is properly obeyed, but the various dependencies having to do with the updates to and uses of the account's balance are uncontrolled. Different interleavings of these operations can yield different results. Let's examine a few of the many possible interleavings.

valid?	*first*	*second*	*balance*	*count*
	75	50	−25	31
	75	50	25	30
	75	50	50	21
✓	75	0	25	12
✓	0	50	50	6

Figure 20.5: Table of outcomes from multiple runs of simultaneous withdrawals. Each row represents a possible outcome, with columns showing the amount dispensed for the first withdrawal, the amount dispensed for the second withdrawal, the final balance, and the number of times this outcome occurred in 100 trials. Only the check-marked trials are valid in respecting dependencies.

First, thread A (the $75 withdrawal) may execute fully before thread B (the $50 withdrawal) begins. That is, they may execute sequentially. This interleaving is depicted in Figure 20.6. In this representation of the two threads executing, the executed lines of thread A are on the left, thread B on the right. We assume that each line of code executes atomically, with the order of the numbered lines indicating the order in which they are executed in the concurrent computation. The ellipses (⋯) indicate code lines that were not executed since they fell in the non-chosen branch of a conditional. In line 1, the balance test in thread A is evaluated. Since the balance is initially 100, and the withdrawal amount is 75, the condition holds and lines 2-4 in the `then` branch are executed. Line 3 in particular updates the shared balance to 25, so that in line 5 when thread B tests the balance, the test fails

	thread A ($75 withdrawal)	*thread B ($50 withdrawal)*
1.	`if balance >= amt then begin`	
2.	`  let diff = balance - amt in`	
3.	`  balance <- diff;`	
4.	`  amt`	
5.	⋯	`if balance >= amt then begin` ⋯
6.		`end else 0`

Figure 20.6: An unproblematic (essentially sequential) interleaving of the threads.

	thread A ($75 withdrawal)	*thread B ($50 withdrawal)*
1.	`if balance >= amt then begin`	
2.	`  let diff = balance - amt in`	
3.		`if balance >= amt then begin`
4.		`  let diff = balance - amt in`
5.	`  balance <- diff;`	
6.	`  amt`	
7.	…	`  balance <- diff;`
8.		`  amt`
		…

Figure 20.7: A problematic interleaving of the threads.

and the second withdrawal does not complete (line 6). In summary, the $75 withdrawal attempt succeeds, dispensing the $75, and the $50 withdrawal attempt fails, leaving a balance of $25.

Of course, if thread B had executed fully before thread A, the corresponding result would have occurred, dispensing only the $50 and leaving a balance of $50.

But other results are also possible. For instance, consider the interleaving in Figure 20.7. Each thread verifies the balance as being adequate and computes its updated value before the other performs the balance update. Both threads go on to update the balance (lines 5 and 7); since thread B updates the balance later, its balance value, $50, overwrites thread A's $25 balance, so the final balance is $50. In summary, both attempted withdrawals succeed, dispensing both $75 and $50, leaving a surprising $50 balance. Sure enough, Figure 20.5 indicates that such outcomes were actually attested in the simulations.

Exercise 199

Construct an interleaving in which both withdrawals succeed, leaving a balance of $25.

Exercise 200

Construct an interleaving in which both withdrawals succeed, leaving a balance of $ − 25.

As Figure 20.5 shows, and these possible interleavings explain, there are important dependencies that are not being respected in the concurrent implementation of the account operations. A solution to this problem of controlling data dependencies requires further tools.

20.7 Locks

To gain better control over the interleavings, we introduce a new abstraction, the LOCK. The underlying idea is that while a thread is executing the withdrawal method, it ought to be the only thread with access to the balance it is manipulating. Just as you might lock your door to prevent others from using your car, you might want to lock some data to prevent others from manipulating it. OCaml provides a simple interface to a locking mechanism called MUTEX LOCKS in its `Mutex` library. The name comes from the idea of *mutual exclusion*;

other threads should be excluded from certain regions of code when a lock is in force.

To create a mutex lock for a given datum, the mutable balance, say, in the ATM example, we use `Mutex.create`.

```
# let balance_lock = Mutex.create () ;;
val balance_lock : Mutex.t = <abstr>
```

As shown, this creates a lock of type `Mutex.t`. We can then lock and unlock the lock as needed with the functions `Mutex.lock` and `Mutex.unlock`.

The mutex locks work as follows. When `Mutex.lock` is called on a lock, the lock is first verified to be in its unlocked state. If so, the lock switches to the locked state and computation proceeds.[6] But if not, the thread in which the call was made is suspended until such time as the lock becomes unlocked, presumably by a call to `Mutex.unlock` in another thread.

[6] Crucially, the testing for unlocked status and subsequent locking occur atomically, so that other threads can't interleave between them. How this is accomplished, the subject of fundamental research in concurrent computation, is well beyond the scope of this text.

Inserting the locks in the ATM example, we would have a withdraw method like this:

```
method withdraw (amt : int) : int =
   Mutex.lock balance_lock;
  if balance >= amt then begin
    balance <- balance - amt;
    amt
  end else 0;
   Mutex.unlock balance_lock  ;;
```

The code between the locking and unlocking is the CRITICAL REGION, a computation that must be carried out atomically from the point of view of the resource that is being locked. In this case, the entire body of the `withdraw` method is a critical region.

Now consider the previous problematic case of Figure 20.7 – in which thread B's withdrawal code begins executing partway through thread A's withdrawal code – except now with the locking implementation above. As seen in Figure 20.8, thread A now begins by establishing the balance lock in step 1. When the first step of thread B executes at the intermediate point within thread A's execution (after step 3) and attempts to itself acquire the balance lock, the lock causes thread B to suspend until such time as the lock becomes available, which is not until thread A releases the lock at step 6. The delay in thread B changes the interleaving to a safe one, like that of Figure 20.6.

20.7.1 Abstracting lock usage

This idiom – wrapping a critical region with a lock at the beginning and an unlock at the end – captures the stereotypical use of locks.

	thread A ($75 withdrawal)	*thread B ($50 withdrawal)*
1.	`Mutex.lock balance_lock;`	
2.	`if balance >= amt then begin`	
3.	`  let diff = balance - amt in`	
		`Mutex.lock balance_lock;`
4.	`  balance <- diff;`	
5.	`  amt`	
	…	*thread B suspended*
6.	`Mutex.unlock balance_lock;`	
7.		`Mutex.lock balance_lock;`
8.		`if balance >= amt then begin`
9.		`  let diff = balance - amt in`
10.		`  balance <- diff;`
11.		`  amt`
		…
12.		`Mutex.unlock balance_lock;`

Figure 20.8: The problematic interleaving, corrected by the use of locks.

In this idiom, the lock is explicitly unlocked after the need for the lock is over. The unlocking is crucial; without it, other threads would be *permanently* prevented from carrying out their own computations requiring the lock. We can codify the importance of matching the locks and unlocks by way of an abstracted function that wraps a computation with the lock and its corresponding unlock. We call the function `with_lock`:

```
# (* with_lock l f -- Run thunk `f` in context of acquired lock `l`,
#    unlocking on return *)
# let with_lock (l : Mutex.t) (f : unit -> 'a) : 'a =
#   Mutex.lock l;
#   let result = f () in
#   Mutex.unlock l;
#   result ;;
val with_lock : Mutex.t -> (unit -> 'a) -> 'a = <fun>
```

If we stick with using `with_lock`, we never need to worry that we will perform a lock without the matching unlock, in keeping with the edict of prevention.

Or will we? What would happen if the computation of `f ()` raised an exception of some sort? The body of the `let` will never be performed, and the lock will not be unlocked! (Of course, that possibility also held for the `withdraw` method just above.) We'll want to fix that by adjusting `with_lock` to make sure to handle exceptions properly, further manifesting the edict of prevention. We leave that for an exercise.

Exercise 201

Define a version of `with_lock` that handles exceptions by making sure to unlock the lock.

Using `with_lock`, the `withdraw` method becomes

```
method withdraw (amt : int) : int =
   with_lock balance_lock (fun () ->
    if balance >= amt then begin
      balance <- balance - amt;
```

```
  amt
end else 0) ;;
```

With this modified implementation of accounts, the simulation of many trials of simultaneous deposits performs much better, with only valid results, as depicted in Figure 20.9.

valid?	*first*	*second*	*balance*	*count*
✓	75	0	25	50
✓	0	50	50	50

Figure 20.9: Rerunning the test of simultaneous withdrawals, with locking in place, all trials now respect the dependencies, though the results can still vary depending on which of the two withdrawals in each trial happens to occur first.

20.8 Deadlock