Threads

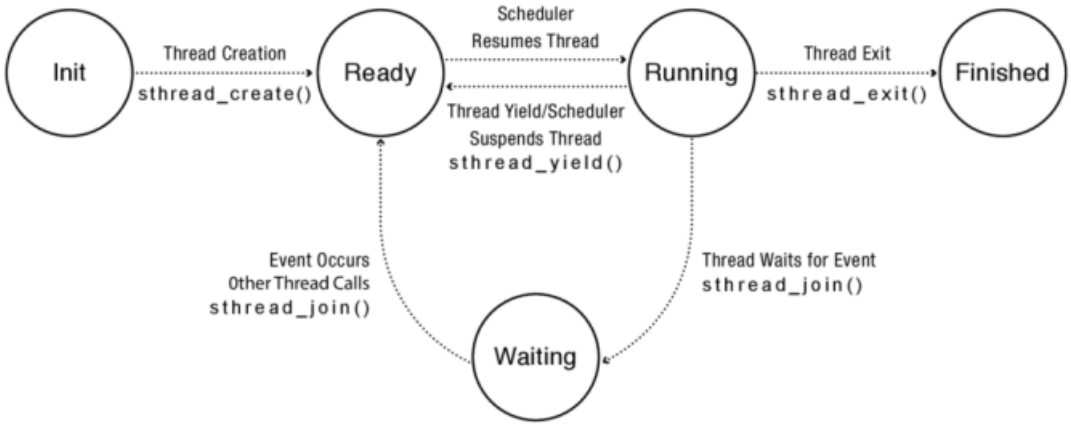
One of the largest priorities for a computer in this day and age is to be able to perform multiple tasks at the same time in order to function smoothly and efficiently for the user. We can define that characteristic as concurrency, which is having multiple activities happening at the same time. A necessity for a modern computer is to have the capacity for handling these concurrent activities, or tasks. Modern computers obtain those capabilities from their operating systems like Windows or Mac which are invisible, yet vital, softwares that manage a computer's resources. Operating systems are capable of keeping up with concurrent tasks through the use of a tool called a thread.

A thread is a single stream of execution, let us break down that statement into components. Beginning with the term stream, a stream is a sequence of data to be processed. A common analogy for a stream is items on a conveyor belt being processed one at a time. So when we say a thread is a single stream of execution, we are meaning it is committing to a set of operations or tasks in a specific order. An example of one single thread execution in a program can be one function call.

```
int Sum(int x, int y) {
    return x + y;
}
```

Using this simple C++ example of a sum function, if we call 'Sum(1,2);' that can be interpreted as one thread in terms of our initial definition. Now if we take an entire traditional program, it can be viewed as one single-threaded program, starting top to bottom executing in that order. Now imagine having multiple of these processes, or threads, executing all concurrently throughout the program. This is what is known as a multi-threaded program. These concepts are what support operating systems to use threads to manage the systems resources concurrently.

Threads work in this fashion but all also follow a certain cycle for working and executing. The chart of the life cycle of a thread below allows for deeper understanding of how threads are able to communicate with each other as well as complete their tasks.



First we have the Init state, creating a thread puts it into its init state and allocates any necessary data to any structures within the threads creation. After that the thread gets put on the ready list. The ready list is a priority queue of threads that are ready to be run through their stream of execution. The ready state is where threads go to be run but that aren't currently running, it's basically just a red light for threads that are just waiting for their turn. The running state is for threads that are running on a processor, this state can be transitioned to ready if they need to be suspended or yielded for whatever reason as well as the finished state if the thread completes its stream of execution. The waiting state is for threads that are needing to wait for certain events to occur in order to continue, where it is moved back to ready. The finished state is where a thread goes to exit its stream, it completes all executions assigned to it and it has nothing left to do.

Threads, like everything else in coding, are not perfect. There are scenarios that occur quite often with multi-threaded programs where all the different threads can use too many resources to accomplish their jobs and causing a problem called starvation. Starvation occurs when threads are in the middle of utilizing resources for their streams of execution and end up having to wait for more resources from the system when there is none left to give. Another more intense form of starvation can occur when one thread is waiting for another to take some sort of action so that it can continue, but that thread can not continue either creating what is known as a deadlock. There are four conditions that result in a deadlock as shown here:

There are four necessary conditions for deadlock to occur. Knowing these conditions is useful for designing solutions: if you can prevent any one of these conditions, then you can eliminate the possibility of deadlock.

1. **Bounded resources.** There are a finite number of threads that can simultaneously use a resource.
2. **No preemption.** Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it.
3. **Wait while holding.** A thread holds one resource while waiting for another. This condition is sometimes called *multiple independent requests* because it occurs when a thread first acquires one resource and then tries to acquire another.
4. **Circular waiting.** There is a set of waiting threads such that each thread is waiting for a resource held by another.

A classic example of deadlock is called 'Dining Philosophers' as shown here:

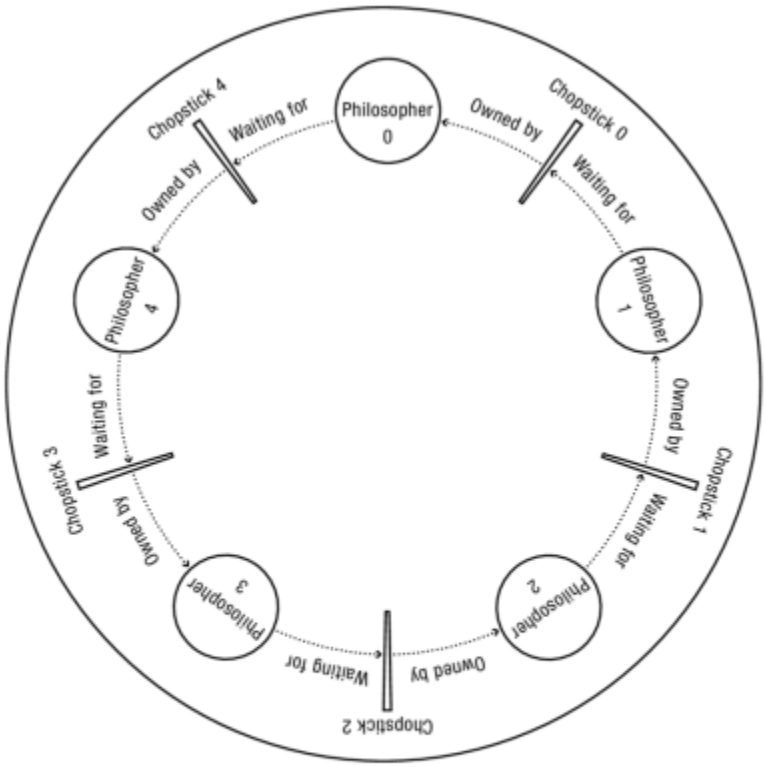


Figure 6.16: Graph representation of the state of a deadlocked Dining Philosophers system. Circles represent threads, boxes represent resources, an arrow from a box/resource to a circle/thread represents an *owned by* relationship, and an arrow from a circle/thread to a box/resource represents a *waiting for* relationship.

This example includes all four conditions for deadlock. For ‘bounded resources’, each chopstick can only be held by one philosopher at a time. For ‘no preemption’, once a philosopher picks up their chopstick, they do not release it until they are done eating, even if that means the rest can’t eat. For ‘wait while holding’, when a philosopher needs to wait for a chopstick, they continue to hold onto any chopsticks they have already picked up. Finally ‘circular waiting’ since there is a cycle of waiting that can not be broken.

Lastly, threads are non-deterministic in nature. The term non-deterministic means that the outcome or result of a process can not be determined or predicted. This means that threads will complete in different orders with varying results every execution of a multi-threaded program.

Most of this information on threads comes from two texts, sections 4.1 and 4.2 of “Concepts, Techniques, and Models of Computer Programming” by Peter Van Roy et al. As well as Chapter 4 of “Operating Systems Principles and Practice” 2nd edition by Thoman Anderson and Micheal Dahlin.

Oz syntactic sugar

Oz is a multiparadigm programming language made for programming language education. A multiparadigm programming language basically means that it has flexibility in format to allow the best programming style to accomplish a task. For example C++ is an object oriented programming language (OOP) and haskell is a functional programming language (FP). So being a multiparadigm programming language allows any of these styles of approach to tasks.

Oz has two syntaxes for coding. One is a sort of ‘pseudocode’ esq syntax called syntactic sugar, the other is the official product called kernel syntax. The key difference is that this syntactic sugar is near exact in syntax to kernel but formatted to be more readable removing the clutter from local environments as well as their closing statements. For example, when you are creating a new environment with variables that looks like this:

```
local A B C in
....
end
```

Calling a ‘local’ needs a corresponding ‘end’, the same goes for if statements, functions, procedures, as well as case statements. This need for ‘end’ statements can be referred to as pattern matching. This pattern matching leads to much confusion when preparing a decently sized program, if there is a missing ‘end’ somewhere it can take a lot of time and energy in order to discover where it needs to be placed. For that reason we prefer syntactic sugar when coding more complex problems.

An example of the difference using syntactic sugar over kernel syntax can be seen by the case statement in Oz.

```
⟨statement⟩   ::= case ⟨expression⟩
                  of ⟨pattern⟩ [ andthen ⟨expression⟩ ] then ⟨inStatement⟩
                  { '[]' ⟨pattern⟩ [ andthen ⟨expression⟩ ] then ⟨inStatement⟩ }
                  [ else ⟨inStatement⟩ ] end
                | ...
⟨pattern⟩     ::= ⟨variable⟩ | ⟨atom⟩ | ⟨int⟩ | ⟨float⟩
                | ⟨string⟩ | unit | true | false
                | ⟨label⟩ '(' { [ ⟨feature⟩ ':' ] ⟨pattern⟩ } [ '...' ] ')'
                | ⟨pattern⟩ ⟨consBinOp⟩ ⟨pattern⟩
                | '[' { ⟨pattern⟩ }+ ']'
⟨consBinOp⟩ ::= '#' | '|'
```

Breaking down this example, we have a case statement with some form of an expression. An ‘expression ‘is a sequence of operations that return some sort of value, it's similar to a ‘statement’ which is the same thing but with no return of a value. From that ‘case (expression)’ we then proceed with an ‘of (pattern)’ which works similarly to an if statement, branching into this case if it's true. That of statement is then preceded by an ‘andthen’ which works like an ‘if else’ in C++ for multiple options, finished with a ‘then’ for that ‘of’ case. After we are done using all of the ‘of’ we finish with an else case. A syntactic sugar example of a case statement can be shown as:

```
case Xs#Ys
of nil#Ys then ⟨s⟩1
[] Xs#nil then ⟨s⟩2
[] (X|Xr)#(Y|Yr) andthen X=<Y then ⟨s⟩3
else ⟨s⟩4 end
```

This case statement is saying we take the pair of lists Xs and Ys, if Ys is nil then we perform whatever $(s)_1$ is. '[]' is basically a repeat of 'of', so an additional case. We then see if Xs is nil and if it is we perform whatever $(s)_2$ is. For our final case we pull the head out of Xs and Ys in the syntax (X|Xr) and (Y|Yr). What this syntax means is that we take the list imputed from the case call, separate the head of the list as 'X' and the rest of the list as 'Xr' same for 'Y' and 'Yr'. After we do that we compare the newly separated heads, 'X' and 'Y' and if 'X' is less than or equal to 'Y' we perform $(s)_3$. If none of these cases are branched into we then just execute $(s)_4$ instead.

Now we take this syntactic sugar of the case statement and convert it to kernel syntax.

```
case Xs of nil then ⟨s⟩1
else
   case Ys of nil then ⟨s⟩2
   else
      case Xs of X|Xr then
         case Ys of Y|Yr then
            if X=<Y then ⟨s⟩3 else ⟨s⟩4 end
         else ⟨s⟩4 end
      else ⟨s⟩4 end
   end
end
```

It performs the same function, but do you notice how cluttered the necessary pattern matching can make our code? That reason is why syntactic sugar is useful and utilized when formulating your code and what you're needing to do as a form of pseudo code, then converting to kernel syntax by adding the necessary pattern matching once the logic is sound.

So having Oz syntactic sugar being able to closely represent the official kernel syntax is a more useful version of pseudocode. Being able to read and understand the logic of your code and easily convert to kernel syntax is one of the uses and popularity of Oz syntactic sugar.

The information and examples are taken from sections 2.1 and 2.5 of "Concepts, Techniques, and Models of Computer Programming" by Peter Van Roy et al.