The Windows Win32 Threads API Specification

Leslie Lamport

Mon 13 May 1996 [17:21]

Contents

1 Description of the Specification

1.1 Introduction

We can view a sequential computer as a collection of registers and a memory containing data and instructions. Starting from some initial state, the computer performs a sequence of steps, each of which changes the contents of the registers and/or the memory as a function of their previous contents.

We can view a concurrent computer as a set of memories and a set of such collections of registers. Each of these collections of registers is called a *thread of control*, or simply a *thread*. Threads are partitioned into *processes*, where threads in the same process share the same memory, and threads in different processes have disjoint memories. A step of the concurrent computer consists of selecting one thread and executing one of its steps.

A multithreaded operating system provides the programmer with a virtual concurrent computer, in which threads can be created and destroyed. It does this by providing *kernel procedures* for creating and destroying threads and processes, and for synchronizing the threads. The part of the operating system devoted to implementing such kernel procedures is called the *kernel*. The Win32 threads Application Programming Interface (API) is the collection of all such procedures provided by a family of operating systems that includes Windows95 and WindowsNT. I present here a specification of a simplified version of that interface.

One of the problems in writing a formal specification of the Win32 threads API is isolating it from the rest of the Win32 API. The threads API procedures interact with procedures not generally considered part of the threads API—for example, procedures that read and write data files. A complete specification of the threads API would have to specify this interaction—for example, by specifying that aspect of the file-manipulating procedures. For simplicity, I ignore this kind of interaction, so the specification is incomplete. It would be straightforward to remedy this incompleteness by enlarging the specification.

Another major problem in specifying the API is choosing the appropriate level of detail for modeling the interaction between the user's program and the kernel. A programmer writing in a high-level language wants a model in which a procedure call is an atomic step that transfers all its arguments to the operating system. A machine-language programmer or a compiler writer needs to know the bit-level details of the procedure-calling interface. I have chosen a high level model. Later, I plan to produce a lower-level specification from this one by using an interface refinement.

In such a high-level specification, one would expect the only observable actions to be calls to kernel procedures and returns from those calls. However, the Win32 kernel keeps some data structures in user memory, and the existence of those data structures should be visible in the specification. This specification therefore includes atomic actions of creating and destroying those structures, which we call *user objects*.

For simplicity, the specification given here does not model the following aspects of the real Win32 threads API:

- Procedures for creating and deleting threads or processes. These include *CreateThread*, *ExitThread*, *TerminateThread*, *CreateProcess*, and *ExitProcess*. Including them would require enhancing the interface parameters to permit one to describe the externally visible effects of these procedures. In the specification, the procedures' action methods would have to be generalized to allow the specification of (externally visible) changes to the interface variables. An additional complication would be added because threads returning from a wait on a mutex would have an additional possibility of returning the value "WAIT_ABANDONED". This would require making every object's *Unwait* method yield an object-structure, return-value pair instead of just an object structure.

- Protection. Describing the protection aspects would add a lot of details, but would require no changes to the specification's structure.

- The following procedures:

ResumeThread	*SetThreadContext*
SuspendThread	*Read/WriteProcessMemory*

  The first two would be easy to add. The second two would require a more elaborate interface to permit the relevant information to be described.

- The ability to wait on other kinds of objects besides the synchronization objects described here—in particular, files, console input/output, threads, and processes. It would be straightforward to add files and console i/o. If the specification were extended to allow creation and destruction of processes and threads, it would be easy to allow waiting on them.

- Resource bounds. Errors that occur when there are insufficient resources to successfully execute a kernel call are not modeled. It would be easy always to allow the possibility of such an error. The more accurately one attempted to model when such errors should occur, the more complicated the specification would become.

- Interrupts.

- The fact that the user's calling stack, which contains the arguments to the procedure call, is in the process's memory and can be modified by other threads during a call. Modeling or forbidding this would require a more detailed interface.

- The fact that part of the Win32 kernel's code resides in the process's memory and can be destroyed by the user program, causing arbitrary behavior by the process (and crashing the entire operating system in Windows95). Modeling or forbidding this would require a more detailed interface.

- The following relevant but uninteresting procedures:

GetCurrentThread	*GetCurrentProcess*
GetCurrentThreadId	*GetCurrentProcessId*

  Adding them would either be trivial or would require some minor modifications to the existing specification.

1.2 The Structure of the Specification

The specification's modules are presented in "descending order", meaning that a module imports or includes only modules that come after it. Reading the modules in this order provides a top-down view of the specification. The following standard modules are not given:

Naturals Defines the set *Nat* of natural numbers and operators on them, including numerals like 1 and 31459.

Reals Defines the set *Real* of real numbers, and its popular subsets such as *Nat*, as well as operators on real numbers and decimal numbers like 3.14159.

Sequences Defines the operators *Len* (length) and $\circ$ (concatenation) on sequences, and the operator $..$, where $i..j$ is the set of integers from i through j.

FiniteSets Defines a couple of operators for describing finite sets.

RealTime Declares a variable *now* that represents the current time, and defines the temporal formula *RT* which essentially asserts that *now* is nondecreasing and that other variables remain unchanged when *now* changes.

I have tried to include enough comments to make the specification self-contained. However, familiarity with TLA$^+$ is assumed. In the absence of an index, I have added in comments to all `IMPORT` statements the names of all the imported operators that are used by the importing module. So, to find the definition of an operator that is not defined in the current module, look for the operator in the comments of the `IMPORT` statements, and then look up the module in the table of contents.

The Win32 API is described by informal documentation that talks about kernel objects, which are data items that cannot be directly observed or modified by the user program. The specification is written with an object-oriented style. Procedures are described by procedure methods, and objects are described by object methods. This specification style is nothing more than a style—a particular way of writing the specification. It requires nothing new in TLA$^+$.

The specification is still missing some details because I don't know them. These omissions are marked by `???`.

1.3 Object Orientation—an Aside

There is much debate about what "object oriented" means. However, it seems to be generally agreed that one (and perhaps the) key aspect of object oriented programming is that operations are defined along with the data on which they operate. For example, consider an operation *Delete*(o) that deletes an arbitrary object o. Suppose that in addition to removing o from the object database, the operation decrements a reference count in each object that o refers to. Since objects of different types refer to different types of objects, what the operation *Delete*(o) does depends on the type of o.

In a conventional programming style, one might define the operation as a procedure by writing something like

```
PROCEDURE Delete(o : OBJECT) :
  IF o.type = cell
    THEN ...
    ELSE IF (o.type = file) AND (o.status = active)
           THEN  ...
           ELSE  ...
```

In an object-oriented style, one would let each deletable object have a *delete method* and would define $Delete(o)$ simply as $o.delete$.

In object oriented programming languages, objects usually have fields and methods. An object o might contain a reference-count field $o.rcnt$, whose value is a natural number, as well as a delete method $o.delete$. For the Win32 threads specification, all objects of the same type have exactly the same methods. Thus, we do not have to make an object's method m part of the object itself; instead, we define a function *ObjMethod* and write $ObjMethod[o.type].m$ rather than $o.m$.

Just as each object has its own reference-count field, each object could have its own delete method. If different objects of the same type t could have different m methods, then methods as well as fields would have to be represented as record components. The m method of object o would then be represented by $o.m$. However, this poses a problem that we now discuss.

Any sufficiently general way of writing methods must allow them to be functions on the set *Object* of all objects. For example, suppose object o has a method $o.m$ that increments the *count* field of an arbitrary object. In other words, $o.m[ob]$ is the object that is identical to ob except with its *count* field incremented by 1—or else equals ob if ob has no *count* field. Then $o.m$ is a function that maps objects to objects. Let OO be the set $[Object \rightarrow Object]$ of all functions from objects to objects. Since methods can be quite arbitrary, we expect that for any function f in OO, there is an object o such that $o.m$ equals f. This would imply that there are at least as many elements in *Object* as there are in OO. However, a theorem of set theory asserts that for any set S, the set $[S \rightarrow S]$ has more elements than S. So, it is impossible for an object to have a field that can equal an arbitrary function from objects to objects.

This dilemma forces us to let $o.m$ not equal the actual function f in OO, but rather some representation of that function. For example, we could let

$o.m$ be a string that is the TLA$^+$ definition of the function f. The dilemma is resolved because only a small subset of OO consists of representable functions. In general, we need to define a set R of representations and a function M with domain R such that $M[o.m]$ is the method represented by the element $o.m$ of R. If R is the set of strings that are TLA$^+$ definitions of functions, then M essentially defines the semantics of TLA$^+$ function definitions. Defining M would be a rather long, if amusing, exercise. In most specifications, there is a small finite set of methods, each of which can be explicitly defined. We can then just let R be a finite set of strings (the method names) and define M to be a record. (Remember that in TLA$^+$, $r.c$ is an abbreviation for $r[\text{“c”}]$.)

Another aspect of object-oriented programming is inheritance: defining one object type to be a subtype of another object type (or of several other object types). In the TLA$^+$ representation, an object is a record, and a record is a function. It is therefore trivial to describe inheritance in TLA$^+$.

2 The High-Level Specification

```
MODULE W32ThreadsSpec

IMPORT Reals
       RealTime   Declares now, a variable representing the current time,
                  and defines RT.
PARAMETERS

procCalls : VARIABLE
```

Used to describe kernel procedure calls. The actual values assumed by `procCalls` *are not specified; they are determined by the parameters* `IsCall` *and* `IsReturn` *of module* `W32TParameters`.

```
userObjects : VARIABLE
```

The set of objects kept in user memory.

```
MODULE ThreadsInner

PARAMETERS
  oStruct : VARIABLE
```

Represents the kernel's internal state. It will be an object structure, which is a function from a set of object ids to objects. (See module `W32TObjects` *for further details.)*

```
IMPORT W32TActions      Defines PgmAction KernelProcAction HighestPriority
                                OStructInit OStructInvariant.
       W32DataTypes     Defines W32Type, Pointer.
       W32TParameters   Declares ProcessIdent.
       W32UserObjects   Defines UserObjects.
```

$\texttt{threadIds} \triangleq$

The set of all object ids of threads.

$\{\texttt{id} \in \texttt{DOMAIN oStruct} : \texttt{oStruct[id].type} = \texttt{"Thread"}\}$

$\texttt{Init} \triangleq$

This predicate describes the initial state, which represents the system after the machine has been booted but before user programs are started. (A fixed set of user processes and threads is assumed.) Note that no initial conditions are assumed of `procCalls`*, and no separate initial condition is needed for* `userObjects` *.*

$\land\ \texttt{OStructInit}$
$\land\ \texttt{now} \in \texttt{Real}$

$\texttt{Invariant} \triangleq$

```
    Asserts a correctness condition on the state during execution, including that
    userObjects is a set of process identifier, pointer, type triples. The value of
    procCalls is not described.
    ∧ OStructInvariant    Describes oStruct.
    ∧ userObjects ⊆ ProcessIdent × Pointer × W32Type
    ∧ now ∈ Real

  vars ≜ <procCalls, oStruct>
    Note the absence of userObjects, which is just specified to be a function of oStruct.

  ISpec ≜
     ∧ ∧ Init
       ∧ □[ ∃ tid ∈ threadIds :
                   ∨ A user-program step executed by the thread.
                     PgmAction(tid)
                   ∨ A kernel step executed by (or for) the thread.
                     KernelProcAction(tid)
          ]_vars
       ∧ ∀ tid ∈ threadIds :
            WF_vars( ∧ KernelProcAction(tid)
                     ∧ This process has highest priority.
                       HighestPriority(tid)   )

     ∧ Defines userObjects as a function of oStruct.
       □(userObjects = UserObjects(oStruct))

     ∧ RT(vars)   Describes how now changes.

   THEOREM Invariance ≜ ISpec ⇒ □Invariant

Spec ≜ ∃ oStruct : ThreadsInner.ISpec
```

```
─────────────────────── MODULE W32TActions ───────────────────────
IMPORT Naturals
       W32DataTypes          Defines False ArgPreToType.
       W32TKernelCalls       Defines KernelCall KernelReturn.
       W32TProcs             Defines ThreadsProcedureName ProcMethod.
                                     ExecuteProc.
       W32ObjectStructures   Defines Object.
       W32TObjects           Defines ObjectStructureOK
                                     InitialObjectStructure.

PARAMETERS
  now, procCalls, oStruct : VARIABLE
──────────────────────────────────────────────────────────────────
```

PgmAction(tid) $\triangleq$

A user-program action performed by the thread with object id `tid`*—that is, an action of the thread when it is not executing a kernel call. The action specifies changes to* `procCalls` *and* `oStruct`*.*

```
∧ oStruct[tid].callState = "None"
∧ KernelCall(tid)
```

KernelProcAction(tid) $\triangleq$

This action describes a step performed during the execution of a kernel call for the thread with object id `tid`*.*

```
∨ An internal step of the kernel.
   ∧ oStruct[tid].callState = "Ready"
   ∧ Change oStruct appropriately.
     ExecuteProc(tid, oStruct, oStruct′, now)
   ∧ UNCHANGED <procCalls>
∨ A return from the kernel call.
   ∧ oStruct[tid].callState  = "Done"
   ∧ KernelReturn(tid)
```

HighestPriority(tid) $\triangleq$

True iff thread `tid` *has the highest priority among all active, nonwaiting threads, where priorities 0 through 15 are considered equivalent. A thread's* `priority` *field corresponds to the "static" priority in the Win32 documentation. The documentation describes scheduling in terms of a "dynamic" priority. If a thread with static priority less than 15 remains unscheduled long enough, its dynamic priority will rise to 15. For specifying liveness properties, we can just pretend that all threads with static priority less than 15 have dynamic priority 15.*

LET schedThreads $\triangleq$

The set of all threads that can be scheduled for execution.

```
        { t ∈ DOMAIN oStruct :
            ∧ oStruct[t].type      = "Thread"
            ∧ oStruct[t].suspended = False  }
  IN  ∀ t ∈ schedThreads :
        oStruct[t].priority ≤ IF oStruct[tid].priority ≤ 15
                                THEN 15
                                ELSE oStruct[tid].priority
```

PTInv $\triangleq$

Asserts that every thread and process ident has a corresponding object. (These might have been made part of the thread and process `Initial` *and* `IsOK` *methods, but were not because they aren't correctness conditions on individual objects.*

```
  ∧ ∀ th ∈ ThreadIdent :
        ∃ id ∈ DOMAIN os : ∧ os[id].type = "Thread"
                           ∧ os[id].thread = th
  ∧ ∀ pr ∈ ProcessIdent :
         ∃ id ∈ DOMAIN os : ∧ os[id].type = "Process"
                            ∧ os[id].process = pr
```

OStructInit $\triangleq$

The initial condition for `oStruct`

```
  ∧ InitialObjectStructure(oStruct)
  ∧ PTInv
```

OStructInvariant $\triangleq$

The invariant for `oStruct`

```
  ∧ ObjectStructureOK(oStruct)
  ∧ PTInv
```

∧ *If a thread is calling a threads procedure, then it has the right* `argSpec` *field for that procedure. It would be nice to make this part of the* `IsOK` *method for thread objects. However, this would be circular, since it depends on a procedure's* `argSpec` *method, and procedure methods are defined in terms of object methods. Permitting object methods to be defined in terms of procedure methods would require restructuring the specification and doesn't seem worth the trouble.*

```
    ∀ id ∈ DOMAIN os :
      ∧ os[id].type = "Thread"
      ∧ os[id].callState ≠ "None"
      ⇒ os[id].argSpec =
           ArgPreToType(ProcMethod[os[id].kernelProc].argPreSpec,
                        os[id].args)
```

MODULE W32TKernelCalls

```
IMPORT Naturals  Sequences
       W32TParameters        Declares IsCall IsReturn.
       W32TProcs             Defines ThreadsProcedureName ProcMethod.
       W32DataTypes          Defines ArgPreToType False.

PARAMETERS
  now, procCalls, oStruct : VARIABLE

KernelCall(tid)  ≜
  This action describes the changes to procCalls and oStruct that occur when executing
  a kernel call for the thread with object id tid.
  ∃ args, argPreSpec:
    ∧ IsCall(procCalls, procCalls′, oStruct[tid].thread,
             oStruct[oStruct[tid].process].process, args,
             argPreSpec)
    ∧ This is a threads procedure.
      proc ∈ ThreadsProcedureName
    ∧ The value of argSpec is the argument specifier for the procedure.
      argSpec = ArgPreToType(ProcMethod[proc].argPreSpec, args)
    ∧ Set the thread object fields for the new kernel call.
      oStruct′ = [oStruct EXCEPT
                   ![tid].kernelProc = proc    ,
                   ![tid].callState  = "Ready" ,
                   ![tid].callTime   = now     ,
                   ![tid].suspended  = False   ,
                   ![tid].argSpec    = argSpec ,
                   ![tid].argVals    = args    ,
                   ![tid].callData   = { }      ] Arbitrary initial value

KernelReturn(tid)  ≜
  This action describes the changes to procCalls and oStruct that occur when executing
  a return from a kernel call for the thread with object id tid.
  ∧ IsReturn(procCalls, procCalls′, oStruct[tid].thread,
             oStruct[oStruct[tid].process].process,
             oStruct[tid].returnVal,
             oStruct[tid].argVals, oStruct[tid].argSpec)
  ∧ oStruct′ = [oStruct EXCEPT ![tid].callState  = "None" ]
```

3 Procedure Specifications

MODULE W32TProcs

Every kernel thread procedure is specified by procedure methods. (Procedure methods are distinct from object methods.) Each procedure has the following methods.

`argPreSpec` *Specifies the number, class, and types of arguments. It is a sequence of argument descriptors. An argument descriptor is a pair* `<cl, tp>` *where* `cl` *is an argument class and* `tp` *is an argument pre-type. Possible argument classes are:*

`Value`: *a call-by-value argument.*
`Input`: *an input argument.*
`Out`: *an output argument.*
`UserObj`: *a user object.*

For all classes except `Value`, *it indicates an argument that is a pointer to a region containing a value of the indicated type. A pre-type is the same as a type, except for array types. Types and pre-types are specified in module* `W32DataTypes`, *but the meanings of the type specifications should be self-evident for most of the arguments.*

`action` *The set of* `<tid, oldOS, newOS, t>` *tuples such that* `newOs` *is a possible new object structure resulting from executing the procedure at real time* `t` *for a thread with objectid* `tid` *starting with an object structure* `oldOS`. *Here, executing the procedure means making the appropriate change to the object structure, but not making any externally visible changes. Those changes—in particular, the return from the procedure—are controlled by the contents of the object structure. See the description of thread objects in module* `W32TThreadObjects` *for details.*

For each synchronization object of type O, there is a module `W32T`*O*`Procs` *that defines the procedure methods for procedures associated with that object type. There is also a module for the* `WaitFor...` *procedures and one for various other procedures not associated with a particular object type. The current module combines all those definitions into the definition of* `ProcMethod`, *where* `ProcMethod[p].m` *is the* `m` *method of the procedure whose name (a string) is* `p`. *Each of the modules* `W32T...Procs` *defines for each procedure* `p` *a record called* `p` *with components* `name`, `argPreSpec`, *and* `action`, *which give the procedure's name and its methods. Module* `W32T...Procs` *defines* `Spec` *to be a set of such records.*

```
INCLUDE W32TWaitForProcs          AS Event
INCLUDE W32TEventProcs            AS Event
INCLUDE W32TMutexProcs            AS Mutex
INCLUDE W32TSemaphoreProcs        AS Semaphore
INCLUDE W32TCriticalSectionProcs  AS CriticalSection
INCLUDE W32TMiscProcs             AS Misc
```

`AllProcSpecs` $\triangleq$

```
    WaitFor.Spec
  ∪ Event.Spec
  ∪ Mutex.Spec
  ∪ Semaphore.Spec
  ∪ CriticalSection.Spec
  ∪ Misc.Spec
```

`ThreadsProcedureName ≜`

The set of names of all threads kernel procedures. It is a set of strings.

```
{ps.name : ps ∈ AllProcSpecs}
```

`ProcMethod[pn : ThreadsProcedureName] ≜`

A mapping from procedure names to records such that `ProcMethod[pn]` *is the record containing all but the name field of the procedure-specification record for procedure* `pn`*.*

```
LET pspec ≜ CHOOSE ps : ∧ ps ∈ AllProcSpecs
                        ∧ ps.name = pn
IN  [c ∈ (DOMAIN pspec) \ {"name"} ↦ pspec[c]]
```

`ExecuteProc(tid, oldOs, newOs, t) ≜`

If, in object structure `oldOs`*, the thread with object id* `tid` *is ready to perform an internal kernel call step (its* `callState` *component equals "*`Ready`*"), then this equals true iff* `newOs` *is a possible object structure obtained by taking such a step at real time* `t`*.*

```
∧ oldOs[tid].kernelProc ∈ ThreadsProcedureName
∧ <tid, oldOs, newOs, t> ∈
        ProcMethod[oldOs[tid].kernelProc].action
```

```
─────────────────── MODULE W32TEventProcs ───────────────────
IMPORT Naturals
       W32DataTypes          Defines SecurityAttributesType True
                                     False NilHandle.
       W32TProcOps           Defines ActionDescriptor NormalReturn
                                     ErrorReturn Argument HandleToId
                                     InvalidObjectError.
       W32ObjectStructures   Defines NotAnObjectId.
       W32TObjects           Defines NamedObject.
       W32TUnwaiting         Defines UnwaitAll.
├────────────────────────────────────────────────────────────┤
CreateEvent ≜
  Create a new event object, and return its handle.
  [name       ↦ "CreateEvent",
   argPreSpec ↦ < <"Input", SecurityAttributesType>, Ignored
                  <"Value", <"Boolean">>,   Manual reset.
                  <"Value", <"Boolean">>,   Initial state.
                  <"Input", <"String">> >   New object's name.
   action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   LET name ≜ Argument(4, tid, oldOS)
         The name argument. (Equals NotAW32Value if no name is specified.)
    IN ∨ ∃ e : ∧ BadNewNameError(e, name, oldOS)
               ∧ newOS = ErrorReturn(tid, e, NilHandle, oldOS)
                   A failed call returns the nil handle???
       ∨ ∧ ¬∃ e : BadNewNameError(e, name, oldOS)
         ∧ ∃ h, id, os1, os2 :
              ∧ <id, os1> =
                  AddObject(
                    [type   ↦ "Event",
                     name   ↦  name ,
                     status ↦ IF Argument(3, tid, oldOS) = True
                                 THEN "Signaled"
                                 ELSE "NotSignaled",
                     manualReset ↦ IF Argument(2, tid, oldOS)
                                           = True
                                        THEN True
                                        ELSE False ],
```

```
                    oldOS)
               ∧ <h, os2> ∈ CreateHandle(tid, id, os1)
               ∧ newOS = NormalReturn(tid, h, os2) } ]
```

`OpenEvent` ≜

Returns a new handle to the event object of the specified name.

```
[name       ↦ "OpenEvent",
 argPreSpec ↦ < <"Value", <"DWord">>,     Ignored
                <"Value", <"Boolean">>,   Ignored
                <"Input", <"String">> >,  Event object's name.
 action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET eid ≜ NamedObject(Argument(3, tid, oldOS), oldOS)
        The object id of the object named, or NotAnObjectId if there is no such object.
  IN ∨ ∃ e : ∧ InvalidObjectError(e, eid, oldOS, "Event")
                  Handle does not correspond to an event object.
             ∧ newOS = ErrorReturn(tid, e, NilHandle, oldOS)
     ∨ ∧ ¬∃ e : InvalidObjectError(e, eid, oldOS, "Event")
       ∧ ∃ h, os :
            ∧ <h, os> = CreateHandle(tid, eid, oldOS)
            ∧ newOS = NormalReturn(tid, h, os) }]
```

`SetEvent` ≜

Sets the event with the indicated handle to the signaled state, and releases the appropriate waiting thread(s), if necessary. Returns a boolean indicating whether or not an error occurred.

```
[name       ↦ "SetEvent",
 argPreSpec ↦ < <"Value", <"Handle">> >,  A handle to the object.
 action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET eid ≜ HandleToId(Argument(1, tid, oldOS), tid, oldOS)

  IN  ∨ ∃ e : ∧ InvalidObjectError(e, eid, oldOS, "Event")
              ∧ newOS = ErrorReturn(tid, e, NilHandle, oldOS)

      ∨ ∧ ¬∃ e : InvalidObjectError(e, eid, oldOS, "Event")
        ∧ ∃ os ∈ UnwaitAll([oldOS EXCEPT
                              ![eid].status = "Signaled"]) :
```

```
                    UnwaitAll(...) is the set of possible object structures obtained by
                    setting eid's status to signalled and then unwaiting a maximal set
                    of threads.
                 newOS = NormalReturn(tid, True, os) } ]

ResetEvent ≜
  Sets the event object with the indicated handle to the not-signalled state. Returns True
  iff there is no error.
  [name      ↦ "ResetEvent",
   argPreSpec ↦ < <"Value", <"Handle">> >,     A handle to the object.
   action    ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET eid ≜ HandleToId(Argument(1, tid, oldOS), tid, oldOS)
        The object id of the event object.

      PossibleError(e) ≜
        True iff the call can fail with an error value e.

        ∨ Not a handle to an event object.

          InvalidObjectError(e, eid, oldOS, "Event")
        ∨ It makes no sense to reset an auto-reset event, so this is an error???

          ∧ oldOS[eid].manualReset = False
          ∧ e ≜ ???
  IN  ∨ ∃ e : ∧ PossibleError(e)
              ∧ newOS = ErrorReturn(tid, e, False, oldOS)
      ∨ ∧ ¬∃ e : PossibleError(e)
        ∧ newOS =
             NormalReturn(tid,
                          True,
                          [oldOS EXCEPT
                            ![eid].status = "NotSignaled"])} ]

PulseEvent ≜
  Transiently sets the event object with the indicated handle to the signalled state, unwaits
  any appropriate waiting threads, and then resets the object to the unsignalled state.
  Returns True iff there was no error.
  [name      ↦ "PulseEvent",
   argPreSpec ↦ < <"Value", <"Handle">> >,     A handle to the object.
   action    ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET eid ≜ HandleToId(Argument(1, tid, oldOS), tid, oldOS)
```

```
IN  ∨ ∃ e : ∧ InvalidObjectError(e, eid, oldOS, "Event")
            ∧ newOS = ErrorReturn(tid, e, False, oldOS)
    ∨ ∧ ¬∃ e : InvalidObjectError(e, eid, oldOS, "Event")
      ∧ ∃ os ∈
             UnwaitALL([oldOS
                          EXCEPT ![eid].status = "Signaled"]) :
```

The set of possible object structures obtained by setting `eid`*'s status to signalled and then unwaiting a maximal set of threads.*

```
           newOS = [NormalReturn(tid, True, os)
                        EXCEPT ![eid].status = "NotSignaled"] } ]
```

The object structure obtained from `os` *by issuing a return for* `tid` *and resetting the event object's status.*

```
Spec ≜ {CreateEvent, OpenEvent, SetEvent, ResetEvent, PulseEvent}
```

```
MODULE W32TMutexProcs
IMPORT Naturals               Defines numerals and -.
       W32DataTypes           Defines SecurityAttributesType True
                                      False NilHandle.
       W32TProcOps            Defines ActionDescriptor NormalReturn
                                      ErrorReturn Argument HandleToId
                                      InvalidObjectError.
       W32ObjectStructures    Defines NotAnObjectId.
       W32TObjects            Defines NamedObject.
```

```
CreateMutex ≜
  Create a new mutex object with the given name. Returns a handle to the object.

  [name       ↦ "CreateMutex",
   argPreSpec ↦ < <"Input", SecurityAttributesType>,   Ignored.
                  <"Value", <"Boolean">>               Immediate ownership flag.
                  <"Input", <"String">> >,             Mutex's name.
   action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET name ≜ Argument(3, tid, oldOS)
  IN ∨ ∃ e : ∧ BadNewNameError(e, name, oldOS)
             ∧ newOS = ErrorReturn(tid, e, NilHandle, oldOS)
     ∨ ∧ ¬∃ e : BadNewNameError(e, name, oldOS)
       ∧ ∃ h, id, os1, os2 :
            ∧ <id, os1> =
                 Asserts that id is the new object's id and os1 is the object struc-
                 ture obtained by adding the object to oldOS.
                 AddObject(
                   [type     ↦ "Mutex",
                    name     ↦  name ,
                    owner    ↦  IF Argument(2, tid, oldOS) = True
                                   THEN tid
                                   ELSE NotAnObjectId
                    recCount ↦  IF Argument(2, tid, oldOS) = True
                                   THEN 1
                                   ELSE 0 ],
                   oldOS)
            ∧ <h, os2> ∈ CreateHandle(tid, id, os1)
                 Asserts that os2 is the object structure obtained from os1 by
                 creating a new handle h for the object.
```

```
             ∧ newOS = NormalReturn(tid, h, os2) }]

ReleaseMutex ≜
  Releasing a mutex reduces the count of the number of times the current owner has
  acquired it (through a call to WaitFor...). If that number becomes 0, this puts the
  mutex object into the signaled state, allowing a waiting thread to acquire it.
  [name       ↦ "ReleaseMutex",
   argPreSpec ↦ <<"Value", <"Handle">>>,
   action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   LET sid ≜ HandleToId(Argument(1, tid, oldOS), tid, oldOS)
         Id of the mutex object.

       PossibleError(e) ≜
           ∨ InvalidObjectError(e, sid, oldOS, "Mutex")
           ∨ Mutex not owned by calling thread.
              ∧ oldOS[sid].owner ≠ tid
              ∧ e = ???
   IN  ∨ ∃ e : ∧ PossibleError(e)
               ∧ newOS = ErrorReturn(tid, e, False, os1)
       ∨ ∧ ¬∃ e : PossibleError(e)
         ∧ LET os1 ≜
                    The object structure obtained from oldOS by updating the mutex
                    object.
                    [oldOS EXCEPT ![sid].owner
                                    = IF ![sid].recCount = 1
                                        THEN NotAnObjectId
                                        ELSE @ ,
                                 ![sid].recCount = @ - 1    ]
            IN  ∃ os2 :
                    ∧ If the mutex has been released, then os2 is obtained from
                      os1 by unwaiting an appropriate waiting thread. Otherwise,
                      it equals os2.
                      ∨ ∧ os1[sid].owner = NotAnObjectId
                        ∧ os2 ∈ UnwaitAll(os1)
                      ∨ ∧ os1[sid].owner ≠ NotAnObjectId
                        ∧ os2 = os1
                    ∧ newOS = NormalReturn(tid, True, os2)

Spec ≜ {CreateMutex, ReleaseMutex}
```

```
┌──────────────────── MODULE W32TSemaphoreProcs ────────────────────┐
IMPORT W32DataTypes         Defines SecurityAttributesType True.
       W32TProcOps          Defines ActionDescriptor NormalReturn
                                    ErrorReturn Argument HandleToId
                                    InvalidObjectError.
       W32ObjectStructures  Defines NotAnObjectId.
       W32TObjects          Defines NamedObject.
├───────────────────────────────────────────────────────────────────┤
CreateSemaphore ≜
  Creates a new semaphore object and returns a handle to it.
  [name      ↦ "CreateSemaphore",
   argPreSpec ↦ <<"Input", SecurityAttributesType>,   Ignored
                  <"Value", <"Long">>,              Initial Count.
                  <"Value", <"Long">>,              Maximum Count.
                  <"Input", <"String">> >,          Semaphore's name.
   action    ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET name ≜ IF Argument(4, tid, oldOS)
      PossibleError(e) ≜
         ∨ BadNewNameError(e, name, oldOS)
         ∨ Illegal maximum count specified.
            ∧ Argument(3, tid, oldOS) < 1
            ∧ e = ???
         ∨ Illegal initial count specified.
            ∧ ∨ Argument(2, tid, oldOS) < 0
              ∨ Argument(3, tid, oldOS) < Argument(2, tid, oldOS)
            ∧ e = ???
  IN ∨ ∃ e : ∧ PossibleError(e)
             ∧ newOS = ErrorReturn(tid, e, NilHandle, oldOS)
     ∨ ∧ ¬∃ e : PossibleError(e)
       ∧ ∃ h, id, os1, os2 :
             ∧ <id, os1> =
                 AddObject(
                  [ type     ↦ "Semaphore",
                    name     ↦  name ,
                    count    ↦  Argument(2, tid, oldOS),
                    maxCount ↦  Argument(3, tid, oldOS)],
                  oldOS)
             ∧ <h, os2> ∈ CreateHandle(tid, id, os1)
```

```
             ∧ newOS = NormalReturn(tid, h, os2)}

ReleaseSemaphore ≜
```
Release the semaphore, incrementing its count. Returns the `True` *iff the release is performed. Returns the old value of the semaphore as an output argument, even if the release fails.*

```
  [name       ↦ "ReleaseSemaphore",
   argPreSpec ↦ < <"Input" <"Handle">>,      Semaphore's handle.
                  <"Input" <"Long">>,        Increment value.
                  <"Output" <"Long">> >,     Previous value of sem.
   action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   LET sid ≜ HandleToId(Argument(1, tid, oldOS), tid, oldOS)
          Id of semaphore object.

       newcount ≜ Argument(2, tid, oldOS) + oldOS[sid].count
          New value requested for semaphore.

       PossibleError(e) ≜
          ∨ InvalidObjectError(e, sid, oldOS, "Semaphore")
          ∨ New value for semaphore too big.
             ∧ oldOS[sid].maxCount < newcount
             ∧ e = ???
       os1 ≜
          Object structure oldOS with output value set.

          [oldOS EXCEPT
            ![tid].argVals[3] =
              IF InvalidObjectError(e, sid, oldOS, "Semaphore")
                THEN ???
                ELSE ![sid].count ]
   IN   ∨ ∃ e : ∧ PossibleError(e)
                ∧ newOS = ErrorReturn(tid, e, False, os1)
        ∨ ∧ ¬∃ e : PossibleError(e)
          ∧ ∃ os2 :
               ∧ os2 ∈ UnwaitAll([os1 EXCEPT
                                   ![sid].count = newcount])
               ∧ newOS NormalReturn(tid, True, os2) } ]

Spec ≜ {CreateSemaphore, ReleaseSemaphore}
```

```
---------------------- MODULE W32TCriticalSectionProcs ----------------------
IMPORT Naturals
       W32DataTypes         Defines CriticalSectionType Void
                                    NilPointer True False.
       W32TProcOps          Defines ActionDescriptor NormalReturn
                                    ErrorReturn Argument
                                    InvalidObjectError.
       W32ObjectStructures  Defines NotAnObjectId.
       W32UserObjects       Defines UserObjId.
-----------------------------------------------------------------------------
                         AUXILIARY DEFINITIONS

CSArgObjectId(tid, os)
  The object id of the critical section user object that is the first argument of the call by
  thread tid in object structure os.
  UserObjId(Argument(1, tid, os), os[tid].process, os)

BadCSPointerError(e, tid, os) ≜
  True iff e is a possible error return caused by the first argument of the call by thread
  tid not being a pointer to a user's critical section object.
  LET cid ≜ CSArgObjectId(tid, os)
        The object id of the critical section object.
  IN  ∧ ∨ Argument(1, tid, os) = NilPointer
        ∨ cid = NotAnObjectId
        ∨ os[cid].type ≠ "CriticalSection"
              Possible if we add other user object types.
      ∧ e = ???
-----------------------------------------------------------------------------
                         PROCEDURE SPECIFICATIONS

InitializeCriticalSection ≜
  Creates a new critical section object, which is a user object, at the indicated memory
  location.
  [name       ↦ "InitializeCriticalSection",
   argPreSpec ↦ <<"UserObj", CriticalSectionType>>, The User object.
   action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   LET PossibleError(e) ≜
         ∧ Argument(1, tid, oldOS) = NilPointer Called with nil pointer.
         ∧ e = ???
   IN  ∨ ∧ ∃ e : PossibleError(e)
```

```
        ∧ newOS = ErrorReturn(tid, e, Void, oldOS)
     ∨  ∧ newOS =
            NormalReturn(
              tid,
              Void,
              AddObject([type        ↦ "CriticalSection",
                         process     ↦ oldOS[tid].process,
                         pointer     ↦ Argument(1, tid, oldOS),
                         userObjType ↦ CriticalSectionType,
                         owner       ↦ NotAnObjectId ,
                         recCount    ↦ 0],
                        oldOS)) } ]
```

DeleteCriticalSection ≜

Delete a critical section object. The object must be unowned.

```
[name       ↦ "DeleteCriticalSection",
 argPreSpec ↦ <<"UserObj", CriticalSectionType>>,
 action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   LET cid ≜ CSArgObjectId(tid, oldOS)
         The object id of the critical section object.

       PossibleError(e) ≜
         ∨ BadCSPointerError(e, tid, oldOS)
         ∨ ∧ oldOS[cid].owner ≠ NotAnObjectId
           ∧ e = ???          Critical section still owned.
   IN  ∨ ∃ e : ∧ PossibleError(e)
               ∧ newOS = ErrorReturn(tid, e, Void, oldOS)
       ∨ ∧ ¬∃ e : PossibleError(e)
         ∧ newOS = NormalReturn(tid,
                                Void,
                                [id ∈ (DOMAIN oldOS) \ {cid}
                                  ↦ oldOS[id]] ) }
```

EnterCriticalSection ≜

Acquire critical section, or wait if critical section is busy. (The thread is unwaited by another process's `LeaveCriticalSection` *action.)*

```
[name       ↦ "EnterCriticalSection",
 argPreSpec ↦ <<"UserObj", CriticalSectionType>>,
 action     ↦
```

```
{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   LET cid ≜ CSArgObjectId(tid, oldOS)
   IN  ∧ ![tid].suspended = False          Not already waiting.
       ∧ ∨ ∃ e : ∧ BadCSPointerError(e, tid, oldOS)
                 ∧ newOS = ErrorReturn(tid, e, Void, oldOS)
         ∨ ∧ ¬∃ e : BadCSPointerError(e, tid, oldOS)
           ∧ ∨ Critical section is free or owned by caller.
                 ∧ oldOS[cid].owner ∈ {NotAnObjectId, tid}
                 ∧ newOS =
                     NormalReturn(
                       tid,
                       Void,
                       [oldOS EXCEPT ![cid].owner = tid,
                                     ![cid].recCount = @ + 1])
             ∨ Critical section is owned by another thread.
                 ∧ oldOS[cid].owner ∉ {NotAnObjectId, tid}
                 ∧ newOS =
                     [oldOS EXCEPT ![tid].suspended = True,
                                   ![cid].waiting = @ ∪ {tid}] } ]

LeaveCriticalSection ≜
  [name       ↦ "LeaveCriticalSection",
   argPreSpec ↦  <<"UserObj", CriticalSectionType>>,,
   action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   LET cid ≜ CSArgObjectId(tid, oldOS)
       PossibleError(e) ≜
         ∨ BadCSPointerError(e, tid, oldOS)
         ∨ Thread doesn't own the critical section
            ∧ oldOS[cid].owner ≠ tid
            ∧ e = ???
   IN  ∨ ∃ e : ∧ PossibleError(e)
               ∧ newOS = ErrorReturn(tid, e, Void, oldOS)
       ∨ ∧ ¬∃ e : PossibleError(e)
         ∧ LET os1 ≜
                 Object structure obtained from oldOS when by changing critical
                 section object to indicate that tid has released it.
                 [oldOS EXCEPT ![cid].owner =
                                IF ![cid].recCount = 1
```

```
                                     THEN NotAnObjectId
                                     ELSE @ ,
                                 ![cid].recCount = @ - 1 ]
          os2(id) ≜
            Object structure obtained from os1 when waiting thread id en-
            ters the critical section (and prepares to return to caller).
            NormalReturn(id,
                         Void,
                         [os1 EXCEPT
                           ![cid].owner = id,
                           ![cid].recCount = 1,
                           ![cid].waiting = @ \ {id}])
      IN  ∃ os3 :
            ∧ Asserts that os3 equals os2(id) for some waiting thread id,
              or equals os1 if there are no waiting threads.
              ∨ ∃ id ∈ os1[cid].waiting : os3 = os2(id)
              ∨  ∧ os1[cid].waiting = { }
                 ∧ os3 = os1
            ∧ newOS = NormalReturn(tid, Void, os3) } ]

Spec ≜
  {InitializeCriticalSection, DeleteCriticalSection,
   EnterCriticalSection, LeaveCriticalSection}
```

MODULE W32TWaitForProcs

This module specifies the procedures "`WaitForSingleObject`" and "`WaitForMultipleObjects`".

```
IMPORT Reals Sequences
       W32TProcOps           Defines ActionDescriptor NormalReturn
                                     ErrorReturn Argument HandleToId
       W32ObjectStructures   Defines NotAnObjectId.
       W32Values             Defines WAIT_TIMEOUT ERROR_INVALID_HANDLE
                                     STG_E_INVALIDHANDLE
                                     MAXIMUM_WAIT_OBJECTS DWord.
       W32TObjects           Defines ObjMethod and TObjectType.
       W32TUnwaiting         Defines CanUnwait Unwait.
       W32DataTypes          Defines True False.
```

WaitForSingleObject $\triangleq$

Wait on a single synchronization object. Returns 0 for a normal return, `WAIT_TIMEOUT` *if it times out, or a negative value for an error.*

```
[name       ↦ "WaitForSingleObject",
 argPreSpec ↦ <<"Value", <"Handle">> ,   Handle of object.
               <"Value", <"DWord"> >,   Time-out interval value (ms).
 action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   LET sid ≜ HandleToId(Argument(1, tid, oldOS), tid, oldOS)
          The id of the synchronization object thread tid is waiting for.
       os ≜
          If the thread is not already suspended, then os equals the object structure
          obtained from oldOS by setting the thread's callData component to be the
          one-element sequence consisting of the synchronization object's id. Otherwise,
          it equals oldOS.
          IF oldOS[tid].suspended = False
            THEN [oldOS EXCEPT ![tid].callData = <sid>]
            ELSE oldOS
       PossibleError(e) ≜
          True iff the call can produce error value e.
            ∨ Argument not an object handle.
               ∧ sid = NotAnObjectId
               ∧ e = ERROR_INVALID_HANDLE
            ∨ Handle does not refer to a waitable object.
```

```
          ∧ oldOS[sid].type ∉ WaitObjectType
          ∧ e = STG_E_INVALIDHANDLE
       ∨ Illegal timeout-value argument.
          ∧ Argument(2, tid, oldOS) < -1
          ∧ e = ???
    CanTimeOut ≜
      True iff the call can time out.
      ∧ Argument(2, tid, oldOS) ≥ 0
      ∧ (Argument(2, tid, oldOS) * .001) + oldOS[tid].callTime
           ≤ t
IN  ∨ Error return
      ∧ oldOS[tid].suspended = False   Not already waiting.
      ∧ ∃ e, dw : ∧ PossibleError(e)
                  ∧ (dw ∈ DWord) ∧ (dw < 0)
                  ∧ newOS = ErrorReturn(tid, e, dw, os)
    ∨ Unwait
      ∧ ¬∃ e : PossibleError(e)
      ∧ CanUnwait(tid, os)
      ∧ oldOS[tid].suspended = False
      ∧ newOS ∈ Unwait(tid, os)
    ∨ Timeout
      ∧ ∨ ¬∃ e : PossibleError(e)        Just called.
        ∨ oldOS[tid].suspended = True      Already waiting.
      ∧ ¬CanUnwait(tid, os)
      ∧ CanTimeOut
      ∧ newOS = NormalReturn(tid, WAIT_TIMEOUT, os)
    ∨ Suspend thread.
      ∧ ¬∃ e : PossibleError(e)
      ∧ ¬CanUnwait(tid, os)
      ∧ ¬CanTimeOut
      ∧ oldOS[tid].suspended = False
      ∧ newOS = [os EXCEPT ![tid].suspended = True] } ]

WaitForMultipleObjects ≜
  Wait for either one or all of an array of multiple synchronization objects.
  [name        ↦ "WaitForMultipleObjects",
   argPreSpec ↦ <<"Value", <"DWord">>,    Number of synchron. objects
                  <"Input", <"Array",
                              [s ∈ Seq(W32Value) ↦ s[1]],
                              "Handle">>,
```

```
                    Specifies that the second argument is an array of handles
                    whose length is equal by the first argument.
                  <"Value" <"Boolean">>,     True means wait for all.
                                             False means wait for any.
                  <"Value", <"DWord"> >>,    Time-out interval (in ms).
   action    ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET os ≜
        If thread tid not already suspended in action structure oldOS, then os equals
        the structure obtained from oldOS by setting the thread's callData field to
        the sequence of ids of the synchronization objects being waited for. Otherwise,
        os equals oldOS.
        IF oldOS[tid].suspended = False
          THEN [oldOS EXCEPT
                  ![tid].callData =
                    [n ∈ DOMAIN Argument(2, tid, !) ↦
                      HandleToId(Argument(2, tid, !)[n], tid, !)]]
          ELSE oldOS
      PossibleError(e) ≜
        True iff the call can produce error value e.
        ∨ Argument 1 (number of handles) bad.
           ∧ ∨ Argument(1, tid, oldOS) < 1
             ∨ Argument(1, tid, oldOS) > MAXIMUM_WAIT_OBJECTS
           ∧ e = ???
        ∨ ∃ n ∈ DOMAIN os[tid].callData :
             ∨ Some handle doesn't correspond to any object.
                ∧ os[tid].callData[n] = NotAnObjectId
                ∧ e = ERROR_INVALID_HANDLE
             ∨ Some handle refers to a non-waitable object.
                ∧ os[os[tid].callData[n]].type ∉ WaitObjectType
                ∧ e = STG_E_INVALIDHANDLE
             ∨ The synchronization objects are not distinct.
                ∧ os[tid].callData[n] ≠ NotAnObjectId
                ∧ ∃ m ∈ 1..(n-1) :
                     os[tid].callData[n] = os[tid].callData[m]
        ∨ Illegal timeout value
           ∧ Argument(4, tid, oldOS) < -1
           ∧ e = ???
      CanTimeOut ≜
```

```
        True iff the call can time out.
        ∧ Argument(4, tid, oldOS) ≥ 0
        ∧ (Argument(4, tid, oldOS) * .001) + os[tid].callTime ≤ t
  IN  ∨ Error return
         ∧ oldOS[tid].suspended = False    Not already waiting.
         ∧ ∃ e, dw : ∧ PossibleError(e)
                     ∧ (dw ∈ DWord) ∧ (dw < 0)
                     ∧ newOS = ErrorReturn(tid, e, dw, os)
      ∨ Unwait
         ∧ ¬∃ e : PossibleError(e)
         ∧ CanUnwait(tid, os)
         ∧ newOS ∈ Unwait(tid, os)
      ∨ Timeout
         ∧ ∨ ¬∃ e : PossibleError(e)
           ∨ os[tid].suspended = True    Already waiting.
         ∧ ¬CanUnwait(tid, os)
         ∧ CanTimeOut
         ∧ newOS = NormalReturn(tid, WAIT_TIMEOUT, os)
      ∨ Suspend thread.
         ∧ os[tid].suspended = False    Not already waiting.
         ∧ ¬∃ e : PossibleError(e)
         ∧ ¬CanUnwait
         ∧ ¬CanTimeOut
         ∧ newOS = [os EXCEPT ![tid].suspended = True] } ]

Spec ≜ {WaitForSingleObject, WaitForMultipleObjects}
```

```
───────────────────── MODULE W32TMiscProcs ─────────────────────
```

Defines procedures for closing and duplicating handles, and the `GetLastError` *procedure.*

```
IMPORT W32Values Defines DUPLICATE_CLOSE_SOURCE OptionSpecified
       W32TProcOps          Defines ActionDescriptor NormalReturn
                                    ErrorReturn Argument HandleToId
                                    CreateHandle InvalidObjectError.
       W32ObjectStructures Defines NotAnObjectId.
       W32DataTypes         True False.
```

AUXILIARY DEFINITIONS

`CloseHandleOS(hdl, pid, os)`

The object structure obtained from `os` *by closing a handle* `hdl` *belonging to the process with object id* `pid`*. If there is no such process, or the process has no such open handle, then this equals* `os`*.*

```
  IF (pid = NotAnObjectId) ∨ (os[pid].type ≠ "Process")
    THEN os
    ELSE [os EXCEPT ![pid].handleToObjId =
                       [h ∈ (DOMAIN @) \ {hdl} ↦ @[h]]]
```

PROCEDURE SPECIFICATIONS

```
CloseHandle ≜
  [name       ↦ "CloseHandle",
   argPreSpec ↦ <<"Value", <"Handle">>>,
   action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET oid ≜ HandleToId(Argument(1, tid, oldOS), tid, oldOS)
      PossibleError(e) ≜
        ∨ Handle doesn't specify any object.
           ∧ soid = NotAnObjectId
           ∧ e = ???
        ∨ Handle specifies a non-waitable object that isn't a process or thread.
           ∧ oldOS[oid].type ∉
               WaitObjectType ∪ {"Process", "Thread"}
           ∧ e = ???
  IN  ∨ ∃ e : ∧ PossibleError(e)
              ∧ newOS = ErrorReturn(tid, e, False, oldOS)
      ∨ ∧ ¬∃ e : PossibleError(e)
        ∧ newOS = NormalReturn(
                    tid,
```

```
                        True,
                        CloseHandleOS(HandleToId(
                                          Argument(1, tid, oldOS),
                                          oldOS[tid].process,
                                          oldOS),
                                      oldOS[tid].process,
                                      oldOS) ) } ]

DuplicateHandle  ≜
  [name       ↦ "DuplicateHandle",
   argPreSpec ↦ <<"Value", <"Handle">>,     Source process handle.
                 <"Value", <"Handle">>,     Source object handle.
                 <"Value", <"Handle">>,     Target process handle.
                 <"Output", <"Handle">>,    Target object handle.
                 <"Value", <"DWord">>,      Ignored. (Access.)
                 <"Value", <"Boolean">>,    Ignored. (Inheritance.)
                 <"Value", <"DWord">> >,    Options.
   action  ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
  LET spid ≜ HandleToId(Argument(1, tid, oldOS), tid, oldOS)
      soid ≜ HandleToId(Argument(2, tid, oldOS), tid, oldOS)
      tpid ≜ HandleToId(Argument(3, tid, oldOS), tid, oldOS)
      os1   ≜
```

Equals the object structure obtained from `oldOS` *by closing the old object handle if* `DUPLICATE_CLOSE_SOURCE` *option is specified. Otherwise, it equals* `oldOS`.

```
        IF OptionSpecified(DUPLICATE_CLOSE_SOURCE,
                           Argument(7, tid, oldOS) )
          THEN CloseHandleOS(Argument(2, tid, oldOS), spid, oldOS)
          ELSE os
      PossibleError(e) ≜
        ∨ Bad source process object.
          InvalidObjectError(e, spid, oldOS, "Process")
        ∨ Bad target process object.
          InvalidObjectError(e, tpid, oldOS, "Process")
        ∨ Source object handle does not reference an object.
          ∧ soid = NotAnObjectId
          ∧ e = ???
        ∨ Source object handle does not refers to a waitable object, or a process or
          thread.
```

```
          ∧  ∨  tpid = NotAnObjectId
             ∨  oldOS[soid].type ∉
                  WaitObjectType ∪ {"Process", "Thread"}
          ∧  e = ???
  IN   ∨ ∃ e : ∧ PossibleError(e)
                 ∧ newOS = ErrorReturn(tid, e, False, os1)
       ∨  ∧ ¬∃ e : PossibleError(e)
          ∧ ∃ h, os2 :
               ∧ <h, os2> ∈ CreateHandle(tid, soid, os1)
               ∧ newOS =
                    NormalReturn(
                      tid,
                      True,
                      [os2 EXCEPT ![tid].argVals[4] = h]) } ]
```

```
GetLastError ≜
```

Returns the most recent error value.

```
  [name       ↦ "DuplicateHandle",
   argPreSpec ↦ < >,
   action     ↦

{<tid, oldOS, newOS, t> ∈ ActionDescriptor :
   newOS = NormalReturn(tid, oldOS[tid].errorVal, oldOS)} ]

Spec ≜ {CloseHandle, DuplicateHandle, GetLastError}
```

MODULE W32TUnwaiting

Threads suspended while in a `WaitFor...` *kernel call are unwaited (and their* `callStatus` *set to "*`Done`*") by calls to procedures that signal an object. The* `action` *method of those procedures is defined in terms of the operator* `UnwaitAll` *defined here. Some of the operators used to define* `UnwaitAll` *are also used to define the* `action` *method of the* `WaitFor...` *procedures. In the explanations of these operators, a thread object that is "calling* `WaitFor...`*" is one whose* `callData` *field is a list of ids of the objects it is waiting for.*

```
IMPORT Sequences
       W32TProcOps          Defines ActionDescriptor NormalReturn
                                    ErrorReturn Argument HandleToId.
       W32ObjectStructures  Defines ObjectStructure.
       W32TObjects          Defines ObjMethod.
       W32TDataTypes        Defines W32BoolToBool True.
```

```
ObjNums(tid, os) ≜
```
If the thread with object id `tid` *in object structure* `os` *is executing a* `WaitFor...` *procedure waiting for* `k` *objects, then this is the set* `1..k`.
```
  DOMAIN os[tid].callData
```

```
CanUnwaitOne(n, tid, os) ≜
```
True iff thread with object id `tid` *in object structure* `os`*, which is calling* `WaitFor...`*, can be unwaited from the* `n`*th object it is waiting for—as determined by that object's* `CanUnwait` *method.*
```
  W32BoolToBool(ObjMethod[os[os[tid].callData[n]].type].CanUnwait[
                              os[tid].callData[n], os, tid])
```

```
CanUnwait(tid, os) ≜
```
True iff `tid` *is the object id of a thread in object structure* `os` *that is calling* `WaitFor...` *and can be unwaited.*
```
  ∧ tid ∈ DOMAIN os
  ∧ os[tid].type = "Thread"
  ∧ os[tid].callState  = "Ready"
  ∧ ∨ Multiple-object wait for any object
        ∧ os[tid].kernelProc = "WaitForMultipleObjects"
        ∧ Argument(3, tid, os) = False
        ∧ ∃ n ∈ ObjNums(tid, os) : CanUnwaitOne(n, tid, os)

    ∨ Multiple-object wait for all objects, or single-object wait.
        ∧ os[tid].kernelProc ∈
              {"WaitForSingleObject", "WaitForMultipleObjects"}
        ∧ ∀ n ∈ ObjNums(tid, os) : CanUnwaitOne(n, tid, os)
```

```
Unwait(tid, os) ≜
  If CanUnwait(tid, os) is true, then this is the set of possible object structures that
  can be obtained by unwaiting the thread.

  LET UnwaitFrom(n, os1) ≜
        The object structure obtained from os1 by applying the Unwait method of the
        nth object that the thread is waiting for.

        ObjMethod[os1[os1[tid].callData[n].type].Unwait[
                              os1[tid].callData[n], os1, tid]

      UnwaitThru[n : Nat, os1 : ObjectStructure] ≜
        The object structure os2 obtained from object structure os1 by unwaiting the
        thread from the first n of the objects it is waiting for. (It doesn't matter in
        which order the unwaiting occurs. It is easier to define the recursion to do the
        unwaiting from n through 1.)

        IF n = 0 THEN os1
                 ELSE UnwaitThru[n-1, UnwaitFrom(n, os1)]

  IN IF ∧ os[tid].kernelProc = "WaitForMultipleObjects"
        ∧ Argument(3, tid, os) = False
       THEN Multiple wait for any object
            {os2 ∈ ObjectStructure :
              ∃ n ∈ ObjNums(tid, os) :
                 ∧ CanUnwaitOne(n, tid, os)
                 ∧ os2 =
                     NormalReturn(tid, n - 1, UnwaitFrom(n, os))}

       ELSE Multiple-object wait for all objects, or single-object wait.
            { NormalReturn(
                tid,
                0,    Does this always return 0 ???
                UnwaitThru[Len(os[tid].callData), os]) }

UnwaitAll(os) ≜
  maximal set of threads suspended executing WaitFor....

  LET Unwaitable(os1) ≜
        The set of threads suspended on a WaitFor... call that can be unwaited in
        object structure os1.

        {tid ∈ DOMAIN oss : ∧ CanUnwait(tid, oss)
                            ∧ oss[tid].suspended = True }

      UWA[S : SUBSET Unwaitable(os), os1 : ObjectStructure] ≜
```

The set of all possible object structures obtainable from `os1` *by unwaiting a maximal number of threads whose object ids are in* `S`*. It is defined recursively to be the set of all object structures* `os3` *obtained by unwaiting some thread* `tid` *in* `S` *to obtain* `os2`*, and then applying* `UWA` *to* `os3` *and its unwaitable threads.*

```
IF S = { }
  THEN {os1}
  ELSE {os3 ∈ ObjectStructure :
         ∃ tid ∈ S :
           ∃ os2 ∈ Unwait(tid, os1) :
             os3 ∈ UWA[Unwaitable(os3), os3]}
```

MODULE W32TProcOps

This module defines various operators that are used in the procedure-specifying modules.

```
IMPORT Reals
       W32ObjectStructures   Defines NotAnObjectId ObjectId
                                     ObjectStructure.
       W32Values             Defines ERROR_INVALID_HANDLE.
       W32TObjects           Defines NamedObject TObjectType ObjMethod.
       W32DataTypes          Defines False.
```

InvalidObjectError(e, id, os, tp)

True iff id *is not the object id of an object of type* tp*, for a reason that generates an error return value of* e*. It is used when* id *is the object id obtained from a handle argument. (Do all calls return the same error value if called with an invalid handle???)*

```
∨ ∧ eid = NotAnObjectId              Not an object.
  ∧ e = ERROR_INVALID_HANDLE
∨ ∧ os[eid].type ≠ tp                Not an object of type tp.
  ∧ e = ???
```

BadNewNameError(e, nm, os)

True iff nm *is a string, rather than* NotAW32Value*, and trying to create a new object name* nm *in object structure* os *produces an error* e*. (If string argument* n *has the value* NilPointer*, then* Argument(n, tid, os) *equals* NotAW32Value*.)*

```
∧ name ≠ NotAW32Value
∧ ∨ The name already exists.
      ∧ NamedObject(name, oldOS) ≠ NotAnObjectId
      ∧ e = ???
  ∨ The name has a backslash in it.
      ∧ ∃ c ∈ DOMAIN name : name[c] = "\"[1]
      ∧ e = ???
```

Argument(n, tid, os) $\triangleq$

The value of the nth *argument of the current procedure call for thread* tid *in object structure* os

os[tid].argVals[n]

ActionDescriptor $\triangleq$

The set of all possible values for the action *method of a procedure. (See module* W32TProcs*.)*

ObjectId × ObjectStructure × ObjectStructure × Real

AddObject(obj, os) $\triangleq$

The pair `<id, ns>` *where* `ns` *is the object structure obtained from* `os` *by adding the new object* `obj` *and giving it the new object id* `id`. *(Since object id's are not externally visible, it never matters what id is assigned to the new object.)*

```
LET id ≜ CHOOSE oid : oid ∈ ObjectId \ (DOMAIN os)
IN  <id, [oid ∈ (DOMAIN os) ∪ {id} ↦
           IF oid = id THEN obj
                       ELSE os[oid] ] >
```

`NormalReturn(tid, rval, os) ≜`

The object structure obtained from object structure `os` *by changing the thread object with id* `tid` *to cause the thread to return from the kernel call with the with return value* `rval`. *This enables the externally visible return action.*

```
[os EXCEPT ![tid].callState  = "Done",
           ![tid].returnVal  = rval,
           ![tid].suspended  = False ]
```

`ErrorReturn(tid, eval, rval, os) ≜`

Like `NormalReturn`, *but it also sets the value returned by* `GetLastError` *to be* `eval`. *is never*

```
[os EXCEPT ![tid].callState = "Done",
           ![tid].returnVal = rval,
           ![tid].suspended  = False,
           ![tid].errorVal   = e ]
```

`WaitObjectType ≜`

The set of types of objects a thread can wait on, which are the types of objects that have an `Unwait` *method.*

```
{tp ∈ TObjectType : "Unwait" ∈ DOMAIN (ObjMethod[tp])}
```

Handles are data values used by the application program to refer to objects in kernel procedure calls. A process object's `handleToObjId` *field is a mapping from handles to object id's for the handles belonging to (threads of) the process. Handles are meaningful only in the context of a particular process. Thus, two different threads in the same process can refer to an object by the same handle.*

`HandleToId(h, tid, os) ≜`

The object id in object structure `os` *of the object known to the thread with object id* `tid` *as the object with handle* `h`. *If there is no such object, then it equals* `NotAnObjectId`.

```
LET handleToObjId ≜
      The handleToObjId field of thread tid's process.

      os[os[tid].process].handleToObjId
IN  IF h ∈ DOMAIN handleToObjId
      THEN handleToObjId[h]
```

```
        ELSE NotAnObjectId

CreateHandle(tid, oid, os) ≜
```

The set of all pairs `<h, ns>` *such that* `ns` *is the object structure obtained by creating, for the process containing thread* `tid`*, a new handle* `h` *for the object with id* `oid`*.*

```
  LET  pid = os[tid].process
  IN { <h, ns> ∈ Handle × ObjectStructure :
       ∧ h ∉ DOMAIN os[pid].handleToObjId
       ∧ ns = [os EXCEPT
                 ![pid].handleToObj =
                   [hdl ∈ DOMAIN @ ∪ {h} ↦
                     IF hdl = h THEN oid
                                ELSE @[hdl]]] }
```

4 Object Specifications

MODULE W32TObjects

Every object of type O is specified in a module W32T*O*Objects*, which defines* ObjectSpec *to be the pair* <t, m>*, where* t *is the set of all objects of type O, and* m *is a record of methods for that type. The following methods are defined. All objects have the first two; only waitable objects have the last two.*

IsOK[id, os]
: *Specifies correctness conditions on an object with object id* id *of the given type in object structure* os*. This condition is an additional constraint beyond those implied by type correctness (that the object is an element of the set of all objects of that type). Its value is either* True *or* False*.*

Initial[id, os]
: *Equals* True *iff object* id *of object structure* os *is a legal object for the initial state. It should imply* IsOK*.*

CanUnwait[id, os, tid]
: *If thread* tid *is calling* WaitFor... *to wait for the object* id *in object structure* os*, then this method equals* True *iff the thread can be unwaited from that object.*

Unwait[id, os, tid]
: *If* CanUnwait[id, os, tid] *equals* True*, then this method yields the object structure obtained from* os *by changing the object with id* id *to reflect the unwaiting of thread* tid *from the object.*

Each module W32T*O*Objects *first defines* Type *to be the set of all objects of that type, then defines* InitialMethod*, etc. to be the methods. Finally, it puts the definitions together to define* ObjectSpec *to be the pair* <t, r> *where* t *is the object's type (a string) and* r *is the record whose components are its methods.*

The current module combines all those definitions into the definition of the function ObjMethod *such that* ObjMethod[t].m *is the* m *method of objects of type* t*. (The domain of* ObjMethod *is the set* TObjectType *of all thread object types, which is a set of strings.)*

```
IMPORT Naturals
       W32ObjectStructures  Defines W32BoolToBool NotAnObjectId
       W32DataTypes         Defines W32BoolToBool
INCLUDE W32TThreadObjects          AS Thread
INCLUDE W32TProcessObjects         AS Process
INCLUDE W32TEventObjects           AS Event
INCLUDE W32TMutexObjects           AS Mutex
INCLUDE W32TSemaphoreObjects       AS Semaphore
INCLUDE W32TCriticalSectionObjects AS CriticalSection
```

ObjectSpecs $\triangleq$

```
  { Thread.ObjectSpec,
    Process.ObjectSpec,
    Event.ObjectSpec,
    Mutex.ObjectSpec,
    Semaphore.ObjectSpec,
    CriticalSection.ObjectSpec  }

TObject ≜
  The set of all objects of types defined in the specification.
  UNION {r[1] : r ∈ ObjectSpecs}

TObjectType ≜ {t.type : t ∈ TObject}
  The set of all object-type names. (It is a set of strings.)

ObjMethod[t : TObjectType] ≜
  Defines ObjMethod[t].m to be the m method for object type t.
  LET objspec ≜ CHOOSE o ∈ ObjectSpecs : o[1].type = t
  IN  objspec[2]

ObjectStructureOK(os) ≜
  A condition characterizing a "good" object structure.
  ∧ Every object in object structure os is a TObject whose IsOK method equals True.
     ∀ id ∈ DOMAIN os :
        ∧ os[id] ∈ TObject
        ∧ W32BoolToBool(ObjMethod[os[id].type].IsOK[id, os])
  ∧ Named objects have unique names.
     ∀ id1, id2 ∈ DOMAIN os :
           ∧ "name" ∈ DOMAIN ObjMethod[os[id1].type]
           ∧ "name" ∈ DOMAIN ObjMethod[os[id2].type]
           ∧ ObjMethod[os[id1].type].name =
                 ObjMethod[os[id2].type].name
           ∧ ObjMethod[os[id1].type].name ≠ NotAW32Value
           ⇒ (id1 = id2)

InitialObjectStructure(os)
  Is true iff every object in object structure os is a TObject whose Initial method
  equals True.
  ∀ id ∈ DOMAIN os :
     ∧ os[id] ∈ TObject
     ∧ W32BoolToBool(ObjMethod[os[id].type].IsOK[id, os])
```

```
NamedObject(nm, os) ≜
```

The object id of the named object in object structure `os` *that has the name* `nm`*, or* `NotAnObjectId` *if there is none.*

```
LET P(id) ≜ ∧ id ∈ DOMAIN os
            ∧ "name" ∈ DOMAIN ObjMethod[os[id].type]
            ∧ ObjMethod[os[id].type] = nm
IN  IF ∃ id : P(id) THEN CHOOSE id : P(id)
                    ELSE NotAnObjectId
```

```
MODULE W32TThreadObjects
IMPORT Reals  Sequences
       W32ObjectStructures Defines ObjectId ObjValue ObjectStructure
                                   BoolToW32Bool
       W32DataTypes        Defines W32Value W32ArgSpec Boolean False.
       W32Values           Defines W32ErrorValue
       W32TParameters      Declares ThreadIdent.

Type ≜
  [ type       : {"Thread"} ,
    thread     : ThreadIdent,   The ident of the thread.
    process    : ObjectId   ,   The process to which the thread belongs.
    kernelProc : STRING     ,
      The name of the kernel procedure now being called, if any.
    callState : {"None"  ,  Not engaged in a kernel call.
                 "Ready",   Ready to do the real work of a kernel procedure.
                 "Done"} ,  Ready to issue a return from a kernel procedure.
    callTime  : Real,
      The time at which the call was issued, and from which timeouts are timed.
    suspended : Boolean
      Equals True iff the thread is in the middle of a call to a waiting procedure and is
      actually waiting on a synchronization object.
    argVals : Seq(W32Value),
      The sequence of argument values of the current call, with non-value input pointer
      arguments replaced by their values. When callState equals "Ready", input argu-
      ments have their input values. When callState equals "Done", output arguments
      have their return values.
    argSpec : W32ArgSpec ,
      The sequence of argument specifiers for the current kernel call, if there is one. See
      the comments in module W32TProcs for a description of argument specifiers.
    returnVal : W32Value ,
      When callState is "Done", equals the value to be returned.
    errorVal : W32ErrorValue
      The value to be returned by a call to GetLastError.
    callData : ObjValue ,
      Any data that is relevant to processing the call. For a call to the WaitFor...
      procedures, it is the sequence of ids of the synchronization objects being waited
      for.
    priority : 0..31
```

This is a scheduling priority, used to specify fairness properties. It is called the base priority in the Win32 documentation. That documentation also describes a dynamic priority, which is irrelevant to the formal specification.

```
  ]

IsOKMethod[id : ObjectId, os : ObjectStructure] ≜
  BoolToW32Bool(
      ∧ Different thread objects have different thread fields.
        ∀ tid ∈ DOMAIN os :
            ∧ os[tid].type = "Thread"
            ∧ os[tid].thread = os[id].thread
            ⇒ (tid = id)

      ∧ The process field is the id of a process object.
        os[os[id].process].type = "Process"

      ∧ (os[id].callState ≠ "None") ⇒
          When executing a kernel call:

          ∧ the argVals, and argSpec fields are sequences of the same lengths.
            Len(os[id].argVals) = Len(os[id].argSpec)
          ∧ Input and value variables are of the right type.
            ∀ n ∈ DOMAIN os[id].argVals :
              os[id].argSpec[n] ∈ {"Value", "In"}
                ⇒ os[id].argVals[n] ∈
                      ElementsOfType(os[id].argSpec[n][2])
     ∧ Output variables are of the right type when ready to return.
       (os[id].callState = "Done") ⇒
          ∀ n ∈ DOMAIN os[id].argVals :
              os[id].argSpec[n] = "Out"
                ⇒ os[id].argVals[n] ∈
                      ElementsOfType(os[id].argSpec[n][2])
      ∧ (os[id].suspended = True)   If thread is suspended, Then:
          ⇒ ∧ (os[id].callState = "Ready") The thread is Ready.
            ∧ ∨ Waiting for critical section.
                os[id].kernelProc = "EnterCriticalSection"
              ∨ Waiting for WaitFor... object.
                ∧ os[id].kernelProc ∈
                      {"WaitForSingleObject",
                          "WaitForMultipleObject"}
                ∧ ∀ n ∈ DOMAIN os[id].callData
```

```
                        ∧ os[id].callData[n] ∈ DOMAIN os
                        ∧ "Unwait" ∈
                            DOMAIN
                              ObjMethod[os[os[id].callData[n]].type]
    )

InitialMethod[id : ObjectId, os : ObjectStructure] ≜
  BoolToW32Bool(
     ∧ os[os[id].process].type = "Process"
     ∧ os[id].callState = "None"
     ∧ os[id].suspended = False
     ∧ Different thread objects have different thread fields.
       ∀ tid ∈ DOMAIN os :
          ∧ os[tid].type = "Thread"
          ∧ os[tid].thread = os[id].thread
          ⇒ (tid = id)
     ∧ os[id].argVals = < >
     ∧ os[id].argSpec = < >
     ∧ os[id].errorVal = ???
          The value returned by GetLastError before there have been any errors.

ObjectSpec ≜ <Type, [ IsOK      ↦ IsOKMethod,
                      Initial   ↦ InitialMethod  ] >
```

```
---------------- MODULE W32TProcessObjects ----------------
IMPORT W32DataTypes
       W32ObjectStructures   Defines ObjectId ObjectStructure.
       W32TParameters        Declares ProcessIdent.
-----------------------------------------------------------
Type ≜
  [ type          : {"Process"}  ,
    process       : ProcessIdent ,   The process's ident.
    handleToObjId :
      UNION {[S → ObjectId] : S ∈ SUBSET Handle} ,
      The process's mapping from handles to object id's. The domain of handleToObjId
      is the set of all the process's open handles.
  ]

InitialMethod[id : ObjectId, os : ObjectStructure] ≜
  ∧ The handleToObjId component is a mapping from handles to exiting objects.
     ∀ h ∈ DOMAIN os[id].handleToObjId :
        os[id].handleToObjId[h] ∈ DOMAIN os
  ∧ Different process objects have different process fields.
     ∀ tid ∈ DOMAIN os :
        ∧ os[tid].type = "Process"
        ∧ os[tid].process = os[id].process
        ⇒ (tid = id)

IsOKMethod[id : ObjectId, os : ObjectStructure] ≜
   InitialMethod[id, os]

ObjectSpec ≜ <Type, [ IsOK    ↦ IsOKMethod,
                       Initial ↦ InitialMethod ] >
===========================================================
```

```
MODULE W32TEventObjects
```

An event object is a synchronization object with two states: signaled, and nonsignaled. A waiting thread can be unwaited when the event object is in the signaled state. A manual-reset event object remains in the signaled state until it is explicitly reset to the nonsignaled state. An auto-reset event object returns to the nonsignaled state when an object is unwaited from it.

```
IMPORT W32DataTypes         Defines String NotAW32Value Boolean
                                    True False.
       W32ObjectStructures  Defines ObjectId ObjectStructure.

Type ≜
  [ type        : {"Event"} ,
    name        : String ∪ {NotAW32Value} ,
    status      : {"Signaled", "NotSignaled"} ,
    manualReset : Boolean      Equals True for a manual-reset event.
  ]

InitialMethod[id : ObjectId, os : ObjectStructure] ≜ False
  Initially, there are no event objects.

IsOKMethod[id : ObjectId, os : ObjectStructure] ≜
  A manual-reset event can never be in the signaled state except transiently.
  BoolToW32Bool((os[id].manualReset = False)
    ⇒ (status = "NotSignaled"))

CanUnwaitMethod[
    id : ObjectId, os : ObjectStructure, th : ObjectId]  ≜
  A thread waiting on an event object can be unwaited iff the object is in the signaled
  state.
  BoolToW32Bool(os[id].status = "Signaled")

UnwaitMethod[id : ObjectId, os : ObjectStructure, th : ObjectId]  ≜
  An automatic (non-manual) reset event is reset to the nonsignaled state when a waiting
  thread is unwaited.
  IF os[id].manualReset = True
    THEN os
    ELSE [os EXCEPT ![id].status = "NotSignaled"] .

ObjectSpec ≜ <Type, [IsOK      ↦ IsOKMethod,
                     Initial   ↦ InitialMethod,
                     CanUnwait ↦ CanUnwaitMethod,
                     Unwait    ↦ UnwaitMethod    ] >
```

MODULE W32TMutexObjects

A mutex is a synchronization object that can be acquired by at most one thread. A thread acquires a mutex by calling a `WaitFor...` *procedure, and releases it by calling* `ReleaseMutex`.

```
IMPORT Naturals
       W32ObjectStructures  Defines ObjectId NotAnObjectId
                                    ObjectStructure.
       W32DataTypes         Defines String NotAW32Value False.

Type ≜
  [ type     : {"Mutex"},
    name     : String ∪ {NotAW32Value} ,
    owner    : ObjectId ∪ {NotAnObjectId},    Current owner.
    recCount : Nat
      A thread can repeatedly acquire the same mutex, but must release the mutex as
      many times before another thread can acquire it. This field is the number of times
      the current owner has acquired it.
  ]

InitialMethod[id : ObjectId, os : ObjectStructure] ≜ False
  Initially, there are no mutex objects.

IsOKMethod[id : ObjectId, os : ObjectStructure] ≜
  BoolToW32Bool(
    ∧ The mutex's owner (if any) is a thread object.
      (os[id].owner ≠ NotAnObjectId) ⇒
         (HToObj(os[id].owner).type = "Thread")
    ∧ The mutex has an owner iff its recCount is positive.
      (os[id].recCount > 0) ≡ (os[id].owner ≠ NotAnObjectId) )

CanUnwaitMethod[
    id : ObjectId, os : ObjectStructure, tid : ObjectId]  ≜
  A thread can acquire the mutex iff the mutex is not owned or is owned by that thread.

  BoolToW32Bool(∨ os[id].owner = NotAnObjectId
                ∨ os[id].owner = tid)

UnwaitMethod[id : ObjectId, os : ObjectStructure, tid : ObjectId] ≜
  Unwaiting a thread sets that thread to be the mutex's owner (redundant if the thread
  already owns the mutex) and increments the recCount field.

  [os EXCEPT ![id].owner = tid,
             ![id].recCount = @ + 1]
```

```
ObjectSpec ≜ <Type, [IsOK      ↦ IsOKMethod,
                     Initial   ↦ InitialMethod,
                     CanUnwait ↦ CanUnwaitMethod,
                     Unwait    ↦ UnwaitMethod    ] >
```

MODULE W32TSemaphoreObjects

A semaphore is a synchronization object that has a count. A thread can be unwaited from the semaphore iff the semaphore's count is positive. Unwaiting the thread decrements the count. The count is incremented by calling `ReleaseSemaphore`.

```
IMPORT Naturals
       W32ObjectStructures  Defines ObjectId ObjectStructure.
       W32DataTypes         Defines String NotAW32Value False.
```

```
Type ≜
  The set of semaphore objects.

  [ type     : {"Semaphore"} ,
    name     : String ∪ {NotAW32Value} ,
    count    : Nat ,                      The semaphore's current value.
    maxCount : {i ∈ Nat : i > 0}  ]       The semaphore's maximum value.
  ]

InitialMethod[id : ObjectId, os : ObjectStructure] ≜ False
  Initially, there are no semaphore objects.

IsOKMethod[id : ObjectId, os : ObjectStructure] ≜
  The semaphore's value is always less than its maximum value.

  BoolToW32Bool(os[id].count ≤ os[id].maxCount)

CanUnwaitMethod[
    id : ObjectId, os : ObjectStructure, th : ObjectId]  ≜
  An object can be unwaited from a semaphore iff its count is positive.

  BoolToW32Bool(os[id].count > 0)

UnwaitMethod[id : ObjectId, os : ObjectStructure, th : ObjectId] ≜
  Unwaiting from a semaphore decreases its count by one.

  [os EXCEPT ![id].count = @ - 1]

ObjectSpec ≜ <Type, [IsOK      ↦ IsOKMethod,
                     Initial   ↦ InitialMethod,
                     CanUnwait ↦ CanUnwaitMethod,
                     Unwait    ↦ UnwaitMethod    ] >
```

MODULE W32TCriticalSectionObjects

A critical section is a synchronization object that is like a mutex except that (i) it exists within a process and can be used only by threads of that process, (ii) it is acquired with the `EnterCriticalSection` *procedure rather than* `WaitFor...`, *and (iii) it is a user object. (See module* `W32UserObjects` *for an explanation of user objects.) Critical sections are not waitable objects (ones used with* `WaitFor...`*), they have no* `CanUnwait` *or* `Unwait` *method.*

```
IMPORT W32ObjectStructures  Defines ObjectId ObjectStructure.
       W32DataTypes         Defines Pointer NilPointer False
                                    CriticalSectionType.
```

```
Type ≜
  [ type : {"CriticalSection"} ,
    process   : ObjectId ,
```
The process containing the critical section object.

```
    pointer   : Pointer \ {NilPointer},
```
Pointer to the memory region holding the critical section object.

```
    userObjType : CriticalSectionType,
```
The type of the user object. (Used by the program to allocate memory for the object.)

```
    owner     : ObjectId ∪ {NotAnObjectId} ,
```
The thread that is currently in (owns) the critical section.

```
    recCount : Nat
```
A thread can enter (acquire) a critical section it already owns. It must leave the critical section as many times as it entered before the critical section is free. This field is the number of times the current owner has entered the critical section.

```
    waiting   : SUBSET ObjectId
```
The set of threads waiting to enter the critical section.

```
  ]
```

```
InitialMethod[id : ObjectId, os : ObjectStructure] ≜ False
```
Initially, there are no critical section objects.

```
IsOKMethod [id : ObjectId, os : ObjectStructure] ≜
```
∧ *The process field is the id of a process.*
```
     os[os[id].process].type = "Process"
```
∧ *The owner (if any) is a thread object.*
```
    (os[id].owner ≠ NotAnObjectId) ⇒
      (os[os[id].owner].type = "Thread")
```
∧ *No thread is waiting for an unowned critical section.*

```
        (os[id].waiting = NotAnObjectId) ⇒
          (os[id].waiting = { })
    ∧   The recCount is positive iff the critical section is owned.
        (os[id].recCount > 0) ≡
           (os[id].owner ≠ NotAnObjectId) )
    ∧   Every waiting object is a thread of the critical section's process.
        ∀ tid ∈ os[id].waiting :
           ∧ os[tid].type = "Thread"
           ∧ os[tid].process = os[id].process

ObjectSpec ≜ <Type, [IsOK       ↦ IsOKMethod,
                     Initial    ↦ InitialMethod   ] >
```

5 Infrastructure

MODULE W32UserObjects

User objects are regions of the user's memory in which the kernel keeps certain data structures. In our specification, a user object appears in the interface as a triple `<proc, ptr, type>` *where* `proc` *is the process identifier,* `ptr` *is a pointer to the data structure, and* `type` *is the type of data held by that structure. In our specification, each user object has a corresponding kernel object that, in this specification, contains the data that an actual implementation will keep in the process's memory, plus the following fields:*

`process` *the object id of the process containing the object.*

`pointer` *a non-nil pointer to the region in the process's memory.*

`userObjType` *the* `W32Type` *the object is stored as.*

All this is a bit silly, since it provides a general mechanism for handling user objects when the only such objects used by the threads specification are critical section objects.

IMPORT W32ObjectStructures *Defines* `NotAnObjectId`

```
UserObjId(ptr, pid, os)
```

The object id in object structure `os` *for the user object at pointer location* `ptr` *in the memory of the process with object id* `pid`*. Equals* `NotAnObjectId` *if there is none.*

```
LET P(id) ≜ ∧ id ∈ DOMAIN os
            ∧ "userObjType" ∈ DOMAIN os[id]
            ∧ os[id].process = pid
            ∧ os[id].pointer = ptr
IN  IF ∃ id : P(id) THEN CHOOSE id : P(id)
                    ELSE NotAnObjectId
```

```
UserObjects(os) ≜
```

The set of user objects described by the object structure `os`*.*

```
{ <os[id].process, os[id].pointer, os[id].userObjType> :
    id ∈ {oid ∈ DOMAIN os : "userObjType" ∈ DOMAIN os[id]} }
```

MODULE W32ObjectStructures

```
IMPORT Reals
       FiniteSets    Defines IsFinite FiniteSubset.
       W32DataTypes  Defines W32Value NotAW32Value ElementaryType
                             W32Type.
       W32Values     Defines W32ErrorValue
```

ObjectId $\triangleq$

The set of object id's. For simplicity, we assume it is infinite, so we never run out of them.

CHOOSE S : ¬IsFinite(S)

ObjValue $\triangleq$

The set of all possible values that a field of an object may contain. When defining values that may appear in objects, we must be sure that they are in `ObjValue`*. This is made easy by defining* `ObjValue` *to be huge, containing (a) all values that we can construct from values that we know about and (b) values different from all those in (a).*

LET BasicValue $\triangleq$

The set of all "basic known values".

Real ∪ ObjectId ∪ STRING ∪ W32Value ∪ W32ErrorValue
∪ {NotAW32Value}

S[n : Nat] $\triangleq$

The set of all sets of values that can be represented by an expression at most `n` *levels deep starting from the set* `BasicValue` *and using the operator* `SUBSET` *and the function-set operator* `[ ... -> ... ]`.

IF n = 0 THEN BasicValue
ELSE S[n-1] ∪ SUBSET S[n-1]
∪ UNION {[U → V] : U ∈ S[n-1], V ∈ S[n-1]}

KnownSets $\triangleq$

The set of all sets constructible by taking subsets and function sets starting from the set `BasicValue`.

UNION {S[n] : n ∈ Nat]

IN *The set of all elements in some set in* `KnownSets` *together with, for each set* `KS` *in* `KnownSets`*, an element chosen not to be in* `KS`.

(UNION KnownSets)
∪ {(CHOOSE e : e ∉ KS) : KS ∈ KnownSets}

Object $\triangleq$

The set of all objects, where an object is any record containing a `type` *field all of whose fields have values in* `ObjValue`.

```
  LET ObjectFields ≜
        The set of all possible sets of field names an object might have. We allow field
        names to be arbitrary strings.
        {S ∈ SUBSET STRING : IsFinite(S) ∧ ("type" ∈ S)}
  IN  UNION {[S → ObjValue] :  S ∈ ObjectFields}

ObjectStructure ≜
  An object structure is a function whose domain is a finite set of object ids and whose
  range is a set of objects.
  UNION {[S → Object] : S ∈ FiniteSubset(ObjectId)}

NotAnObjectId ≜
  An arbitrarily chosen object value that is not an object id.
  CHOOSE v : ∧ v ∈ ObjValue
             ∧ v ∉ ObjectId
```

MODULE W32TParameters

This module declares the constant parameters of the specification.

```
IMPORT Sequences
       W32Values     Defines W32Value
       W32DataTypes  Defines W32ArgPreSpec, ArgPreToType,
                        W32ArgPreSpec, ElementsOfType.
```

PARAMETERS

`ProcessIdent : CONSTANT`

The set of process identifiers, which is assumed to be fixed. A process's identifier is assumed to be externally visible. It should not be confused with the object id of the process object, used to represent the internal state of a process.

`ThreadIdent : CONSTANT`

The set of thread identifiers, which is assumed to be fixed. Don't confuse a thread's identifier with the object id of the corresponding thread object.

`IsCall(_, _, _, _, _, _) : BOOLEAN`

`IsCall(oldPC, newPC, thd, proc, args, argSpec)` *is true iff a change in value of* `procCalls` *from* `oldPC` *to* `newPC` *represents a kernel call by the thread with thread identifier* `thd` *of the procedure with procedure identifier* `proc` *with a sequence* `args` *of arguments for argument specifier argSpec. The argument* `args` *is a sequence of the same length as* `argSpec`*, but only the values of the input and value arguments matter.*

`IsReturn(_, _, _, _, _, _, _) : BOOLEAN`

`IsReturn(oldPC, newPC, thd, proc, val, args, argSpec)` *is true iff a change in value of* `procCalls` *from* `oldPC` *to* `newPC` *represents a return with return value* `val`*, and sequence of (return) arguments args, from a kernel call by thread with identifier* `thd` *of the procedure with identifier* `proc`*. (For procedures that don't return a value, we let* `val` *be the arbitrary value* `Void`*.) The argument* `args` *is a sequence of the same length as* `argSpec`*, but only the values of the output arguments matter.*

ASSUMPTIONS

`CallAssump ≜`

Asserts that IsCall implies that its last argument has the right "type", and that the input and value arguments of the call are elements of the right sets of values.

```
IsCall(oldPC, newPC, thd, proc, args, argSpec) ⇒
  ∧ argSpec ∈ W32ArgPreSpec
  ∧ args ∈ Seq(W32Value)
  ∧ DOMAIN args = DOMAIN argSpec
  ∧ ∀ n ∈ DOMAIN args :
       argSpec[n][1] ∈ {"Value", "In"}
         ⇒ args[n] ∈ ElementsOfType(argSpec[n][2])
```

```
ReturnAssump ≜
```

Asserts that IsReturn implies that its last argument has the right "type", and that the output arguments of the call are elements of the right sets of values.

```
IsReturn(oldPC, newPC, thd, proc, val, args, argSpec) ≜
   ∧ argSpec ∈ W32ArgPreSpec
   ∧ args ∈ Seq(W32Value)
   ∧ DOMAIN args = DOMAIN argSpec
   ∧ ∀ n ∈ DOMAIN args :
        argSpec[n][1] = "Out"
          ⇒ args[n] ∈ ElementsOfType(argSpec[n][2])
```

MODULE W32DataTypes

```
IMPORT Integers Sequences
```

Elementary Data Types

The following TLA^+ values represent elementary Win32 data types. They should not be confused with the built-in TLA^+ data types. In particular, the values `True` *and* `False` *are not the TLA^+ primitives* `TRUE` *and* `FALSE`*; and the set* `String` *of Win32 strings is a (finite???) subset of the set* `STRING` *of all of TLA^+'s strings.*

```
True        ≜ ???
False       ≜ ???
Boolean     ≜ {True, False}
DWord       ≜ ((-2^64) + 1)..(2^64 - 1)
String      ≜ {s ∈ STRING : ??? }
Long        ≜ ???
Handle      ≜ ???          The set of all possible object handles.
NilHandle   ≜ ???          The nil handle.
Pointer     ≜ ???          The set of all possible pointers.
NilPointer  ≜ ???          The nil pointer.
FieldName   ≜
```

The set of possible field names for structs. ("Struct" is C's name for record). It is provisionally defined to be the set of all strings composed of letters, digits, and underscores, that begin with a letter.

```
  LET letter ≜ {"abcdefghijklmnopqrstuvwxyz"[i] : i ∈ 1..26}
      LETTER ≜ {"ABCDEFGHIJKLMNOPQRSTUVWXYZ"[i] : i ∈ 1..26}
      digit  ≜ {"0123456789"[i] : i ∈ 1..10}
  IN  {s ∈ Seq(letter ∪ LETTER ∪ digit ∪ {"_"[1]})
         : (Len(s) > 0) ∧ (s[1] ∈ letter ∪ LETTER)}
```

Win32 Types

```
ElementaryType ≜
```

This is the set of elementary types we use in the specification. We do not bother to specify the types of pointers—that is, we do not distinguish different pointer types based on the type of the data that they point to.

```
  {"Boolean", "DWord", "Long", "String", "Handle", "Pointer",
     "Handle"}

ProtoType(S) ≜
```

In our model of Win32, we use the following types:

`<tp>`	`tp` *is an elementary type.*

`<"Array", m, tp>`	*An array consisting of* `m` *elements of type* `tp`*.*
`<"Struct", s>`	*A structure with fields specified by* `s`*, which is a sequence of elements of the form* `<nm, tp>`*, where* `nm` *is the field name (a string) and* `tp` *is the field's type. (The different fields must have distinct names.)*

C does not permit arrays whose length varies dynamically. So, in our definition of a Win32 type, the second component (the `m`*) of an array type, which specifies the length of the array, will just be a natural number. However, we will also need a more general way of specifying the length of an array. So, we define* `ProtoType(S)` *to be the set of types in which the length of an array is described by an element of the set* `S`*.*

```
LET T[n : Nat] ≜
      The set of types of "depth" at most n, where elementary types have depth 0,
      and an array or structure of types of depth at most n has depth at most n+1.
      IF n = 0
        THEN {<tp> : tp ∈ ElementaryType}
        ELSE   T[n-1]
             ∪ { <"Array", m, tp> : m ∈ S, tp ∈ T[n-1] }
             ∪ { <"Struct", tp> :
                     tp ∈ { s ∈ Seq(FieldName × T[n-1]) :
                               ∀ m, p ∈ 1..Len(s) :
                                   (m ≠ p) ⇒ (s[m][1] ≠ s[p][1]) } }
IN  UNION {T[n] : n ∈ Nat}
```

`W32Type ≜ Prototype(Nat)`

The set of Win32 types, in which an array length is just a natural number.

`ElementsOfType(tp) ≜`

The set of all TLA+ data elements that represent Win32 data of type `tp`*.*

```
LET EOT[t : W32Type] ≜
     CASE t[1] = "Boolean" → Boolean ,
          t[1] = "DWord"   → DWord   ,
          t[1] = "Long"    → Long    ,
          t[1] = "String"  → String  ,
          t[1] = "Pointer" → Pointer ,
          t[1] = "Handle"  → Handle  ,
          t[1] = "Array"   → [1..(t[2]) → EOT[t[3]]] ,
          t[1] = "Struct"  →
                   LET index ≜ 1..Len(t[2])
                       dom ≜ {t[2][i][1] : i ∈ index }
                       ran ≜ UNION {EOT[t[2][i][2]] : i ∈ index}
```

```
                          IN  {r ∈ [dom → ran] :
                                 ∀ i ∈ index : r[t[2][i][1]] ∈
                                                    EOT[t[2][i][2]] }
  IN  EOT[tp]

W32Value     ≜ UNION {ElementsOfType(T) : T ∈ W32Type}
NotAW32Value ≜ CHOOSE v : v ∉ W32Value
Void ≜ CHOOSE v : v ∈ W32Value
```

The `Void` *return value is some arbitrary element in* `W32Value`.

Some Type Definitions

```
SecurityAttributesType ≜
  <"Struct" < <"nLength", <"DWord">>
              <"lpSecurityDescriptor", <"Pointer">>,
              <"bInheritHandle", <"Boolean">>>       >

CriticalSectionType ≜ ???
```

Pseudo-Booleans

We want to use methods that are Boolean-valued functions. However, functions map values to values, and in TLA$^+$, Booleans are not values. We therefore use the values `True` *and* `False`*, defined above, to represent "Boolean" values. The operators* `BoolToW32Bool` *and* `W32BoolToBool` *convert between these ersatz Boolean values and true Booleans.*

```
BoolToW32Bool(P) ≜ IF P THEN True
                        ELSE False
W32BoolToBool(s) ≜ (s = True)
```

Operators for Specifying Procedure Arguments

```
W32ArgSpec ≜
```

The set of all argument specifiers. An argument specifier is a sequence of argument descriptors; an argument descriptor is a pair `<cl, tp>` *where* `cl` *is an argument class and* `tp` *is an argument type. Possible argument classes are:*

`Value`: *a call-by-value argument.*
`Input`: *an input argument.*
`Out`: *an output argument.*
`UserObj`: *user object. (See module* `W32UserObjects`*.)*

For all classes except `Value`*, an argument descriptor indicates an argument that is a pointer to a region containing a value of the indicated type. (We have not included "*`InOut`*" arguments because the Win32 threads procedures don't use any.)*

```
  Seq( ({"Value", "In", "Out", "UserObj"} × W32Type )

W32ArgPreType ≜
```

The set of all argument pre-types. An argument pre-type differs from an ordinary type in that the length of an array is specified not by a natural number, but by a mapping from sequences of values to naturals. It is used to specify the type of an argument to a procedure, where a function `f` *specifying the length of an array means that, if* `args` *is the sequence of arguments to the procedure, then* `f[args]` *is the length of the array argument. For example, if the length of the array argument is specified by the 2nd argument of the procedure call, then* `f` *will be defined by* `f[a : Seq(W32Value)]` $\triangleq$ `a[2]`.

```
ProtoType([Seq(W32Value) → Nat])
```

```
W32ArgPreSpec ≜
```

The set of all argument pre-specifiers, which are like argument specifiers except with argument pre-types instead of actual types.

```
Seq( ({"Value", "In", "Out", "UserObj"} × W32ArgPreType )
```

```
ArgPreToType(aps, args) ≜
```

The argument specifier corresponding to the argument pre-specifier `aps`*, for an argument sequence* `args`.

```
LET ATT[a : W32ArgPreType] ≜
      CASE  a[1] ∈ ElementaryType → a,
            a[1] = "Struct" → <"Struct", ATT[a[2]]> ,
            a[1] = "Array"  → <"Array", a[2][args], ATT[a[3]]>
IN  [ n ∈ DOMAIN aps ↦ <atp[1], ATT[atp[2]]> ]
```

MODULE W32Values

This module defines a number of Win32 constants and some sets of those constants.

```
IMPORT  Naturals  Sequences
```

```
HexVal(str)
```

Win32 constants are usually written as hexadecimal strings. This defines `HexVal(str)` *to be the natural number having the string* `str` *as its hexadecimal representation.*

```
LET charVal(c) ≜ CHOOSE i : ∧ i ∈ 0..15
                            ∧ "0123456789ABCDEF"[i+1] = c
    LastByte(ss) ≜ ss[Len(ss)]
    HighOrderPart(ss) ≜ [n ∈ 1..(Len(ss)-1) ↦ ss[n]]
    HV[s : STRING] ≜ IF s = ""
                       THEN 0
                       ELSE charVal(LastByte(s)) +
                              (16 * HexVal[HighOrderPart(s)])
IN  HV[str]
```

```
OptionSpecified(opt, val)
```

True iff argument value `val` *specifies the option described by* `opt`*. An option specifies a bit-mask value, which is conjoined with the bit-map representation of the value to determine if the option holds???*

```
LET bits[v, n : Nat] ≜
```

The sequence of `n` *bits that make up the binary representation of the number* `v`*.*

```
      LET vDiv2 ≜ CHOOSE w : ∧ w ∈ Nat
                             ∧ v ∈ {2 * w, (2 * w) + 1}
          vMod2 ≜ CHOOSE r : ∧ r ∈ {0, 1}
                             ∧ v = (2 * vDiv2) + r
      IN  IF n = 0
            THEN < >
            ELSE bits[vDiv2, n - 1] ∘ <vMod2>
    optLen ≜ CHOOSE m : (m ∈ Nat) ∧ (opt < 2^n)
IN  ∀ k ∈ DOMAIN bits[opt, optLen] :
       (bits[opt, optLen] = 1) ⇒ (bits[val, optLen] = 1)
```

```
W32ErrorValue    ≜ ???
```

A possible value returned by a call to `GetLastError`*.*

Here are some values that are used.

```
WAIT_TIMEOUT  ≜ ???
```

The return value from `WaitFor...` *on a timeout.*

`DUPLICATE_CLOSE_SOURCE` $\triangleq$ `???`

Option for `DuplicateHandle` *procedure.*

`ERROR_INVALID_HANDLE` $\triangleq$ `HexVal("6")`

The handle is invalid

`MAXIMUM_WAIT_OBJECTS` $\triangleq$ `???`

Maximum number of objects that a `WaitForMultipleObjects` *call can wait for.*

Below are some error values that may or may not appear in the final spec.

`TYPE_E_OUTOFBOUNDS` $\triangleq$ `HexVal("80028CA1")`

Invalid number of arguments.

`OR_INVALID_OID` $\triangleq$ `HexVal("1911")`

The object specified was not found.

`OLE_E_BLANK` $\triangleq$ `HexVal("80040007")`

Uninitialized object.

`STG_E_INVALIDHANDLE` $\triangleq$ `HexVal("80030006")`

Attempted an operation on an invalid object.

`STG_E_REVERTED` $\triangleq$ `HexVal("80030102")`

Attempted to use an object that has ceased to exist.