Contents

1 Summary of NT synchronization primitives

1.1 Object types

The Windows NT kernel exposes a long list of object types. Almost all of these can be involved in synchronization; that is, they can be waited on, and all using the same API, namely `NtWaitForMultipleObjects`. Some of these objects are intended and used exclusively as synchronization primitives; others are used for other purposes and tie into `NtWaitForMultipleObjects` simply by virtue of being synchronizable.

The complete list of synchronizable objects which Wine implements at this time is as follows:

- Semaphore: A classic semaphore object. It holds one 32-bit value of dynamic state, which can be decreased by one [i.e. acquire] or increased by an arbitrary amount [i.e. release]; the object is "signaled" when this value is nonzero. It also holds a second, static 32-bit value denoting the maximum count; an attempt to post past this maximum results in an error.

- Mutex (or "mutant"): A classic recursive mutex. This holds one 32-bit value denoting the current recursion count, as well as a reference to the thread which owns it. A thread which exits (normally or violently) while holding a mutex "abandons" it, and the user-space API makes a distinction as to whether a mutex is signaled normally or abandoned (much like robust pthread mutexes and `EOWNERDEAD`.)

- Event: Holds a single boolean value denoting whether the object is signaled or nonsignaled. There are two types of events, manual-reset and auto-reset; most NT APIs treat these identically, but due to their differing semantics it is often useful to think of them as separate types of objects. Auto-reset events are "consumed", that is, reset, by wait operations, much like semaphore and mutexes; manual-reset events are not.

- Timer: Like events, these hold a boolean value denoting their current signaled state. Timers can be made either non-periodic or periodic. They can be configured to call a user-space callback function when signaled (much like `alarm(2)`). They can also be canceled at any time. Like events, they can be manual-reset or auto-reset.

- Thread, process, job: These are similar to Linux's pidfd, and can similarly be waited on. They are signaled when terminated.

- File: These are designaled when I/O is queued, and signaled when it is completed, according to complex rules.

- Message queue: These are signaled when a thread has window messages that match its mask, which is set according to complex rules.

1.2 Wait functions

All synchronizable objects can be at any point described as "signaled" or "not signaled". The logic for determining this state is usually more complex than

that, however. Moreover, Windows conflates (from a POSIX perspective) the concepts of "polling" and "reading", such that many objects have their state changed by successful wait operations.

Waiting is performed by one of the following syscalls:

- `NtWaitForSingleObject`: This function takes an object of any type and a timeout (which may be "infinite"), sleeps until the object is signaled or the timeout expires, and atomically "consumes" the object if appropriate. In detail:

  - A semaphore is considered signaled if its count is nonzero, and consumed by decreasing the count by one. This is the only way to acquire a semaphore; there is no distinct semaphore-specific "down" operation.

  - A mutex is considered signaled if it is already owned by the current thread or if it is unowned. It is consumed by granting ownership to the current thread (if it is not already owned) and increasing its recursion count by one. This is the only way to acquire a mutex. The return value to this function also includes a flag (`STATUS_ABANDONED`) denoting whether the mutex was abandoned.

  - An event is considered signaled if it is in the signaled state. A manual-reset event is not affected by this function; an auto-reset event is atomically reset to the nonsignaled state. [An auto-reset event is somewhat similar to a non-recursive mutex, or a semaphore with a maximum count of one, though it is not usually used in those ways].

  Furthermore, the function can optionally be made "alertable", in which case certain APIs will cause it to return with `STATUS_ALERTED`. Alerts are mostly used for file I/O, and can be thought of as similar to POSIX signals such as `SIGIO`; in that case `STATUS_ALERTED` is itself analogous to `EINTR`.

  The caller must have `SYNCHRONIZE` access to the object, or this function fails. Note that even though the call may modify state, no other access flags are required.

- `NtWaitForMultipleObjects`: Essentially a superset of the above [in

particular, `NtWaitForSingleObject` can be trivially implemented on top of `NtWaitForMultipleObjects`], but for the sake of clarity I've separated it here. This function receives up to a fixed limit of 64 handles. It has two modes of operation:

- wait for *any* of the objects to become signaled, atomically consuming [at most] one object;

- wait for *all* of the objects to become signaled, atomically consuming all of them at once. The call times out if all of the objects are never simultaneously signaled, even if they are signaled at one point or another during the wait. Note that because the acquisition is atomic, use of this API is not vulnerable to lock inversion, regardless of the object order specified by user space.

Like `NtWaitForSingleObject`, this function can be made alertable, and its status reflects whether an acquired mutex was marked as abandoned. In a wait-any operation, the index of the acquired object is still returned in the status, combined with the `STATUS_ABANDONED` flag, but a wait-all operation returns simply `STATUS_SUCCESS` or `STATUS_ABANDONED`. If multiple mutexes are acquired in a wait-all operation, `STATUS_ABANDONED` is returned if *any* were marked abandoned; accordingly there is no way for the caller to know which or how many mutexes were actually abandoned.

- `NtSignalAndWaitForSingleObject`: This function sets an object of any type to a signaled state. Depending on the object type, this is equivalent to `NtSetEvent`, `NtReleaseSemaphore` (with a count of 1), or `NtReleaseMutant`. It then waits on another object, as with `NtWaitForSingleObject`.

  The objects may be the same or different. If either handle is invalid, or has insufficient access to the object, the call fails and neither object is affected. The entire operation is not atomic (i.e. other threads can observe a state where the first object is signaled but the second has not yet been modified by the wait call), however, if the call fails neither object can be affected. That is, if the second object cannot be waited on for any reason, the first object must not be signaled.

1.3 Events

The type of an event, and its initial state, are specified at creation time.

Events are affected by the following syscalls:

- `NtSetEvent`: This function atomically sets an event to the signaled state, and returns its previous state. If threads are waiting on the event, they are signaled, up to a maximum of one thread if the event is auto-reset. In the case of an auto-reset event the state is also reset. The caller must have `EVENT_MODIFY_STATE` access.

- `NtResetEvent`: This function atomically sets an event to the nonsignaled state, and returns its previous state. The caller must have `EVENT_MODIFY_STATE` access.

- `NtPulseEvent`: Akin to `NtSetEvent` followed by `NtResetEvent`, but performed as a single atomic operation. Hence this function wakes up at most one eligible waiter if the event is an auto-reset event, and all eligible waiters if it is manual-reset, but it leaves the event state set to nonsignaled. The caller must have `EVENT_MODIFY_STATE` access.

- `NtQueryEvent`: This function returns whether the event is currently signaled, and whether it is manual-reset or auto-reset. The caller must have "read" access. The caller must have `EVENT_QUERY_STATE` access.

1.4 Semaphores

The initial and maximum count of a semaphore are specified at creation time.

Semaphores are affected by the following syscalls:

- `NtReleaseSemaphore`: This function atomically increments the semaphore count by a given 32-bit value, and returns the previous value. If threads are waiting on the semaphore, they are signaled, and the semaphore's count is decremented accordingly. If the caller attempts to increment the count past its maximum, the call fails. The caller must have `SEMAPHORE_MODIFY_STATE` access.

- `NtQuerySemaphore`: This function returns the current and maximum semaphore count without modifying the semaphore. The caller must have `SEMAPHORE_QUERY_STATE` access.

1.5 Mutexes

The initial owner of a mutex is specified when the mutex is created, and may be set to zero, indicating that the mutex is not owned by any thread. If a mutex is created owned, the initial recursion count is always 1.

Mutexes are affected by the following syscalls:

- `NtReleaseMutant`: This function atomically decrements the mutex recursion count by one, and returns the previous count. If the count reaches zero, the call relinquishes ownership of the mutex, and causes it to become signaled. No access flags are required.

- `NtQueryMutant`: This function returns the current mutex recursion count, whether the caller owns it, and whether it is abandoned. The caller must have `MUTANT_QUERY_STATE` access.

1.6 Object names

Almost all NT objects can be named or anonymous. This provides a simple way to access objects from multiple processes. The name of an object cannot be changed, added, or removed after it is created. Names are associated with the object, not with its handles.

Most objects, including semaphores, events, and mutexes, have two different associated syscalls which can create a new object or open an existing one, e.g. `NtCreateEvent` and `NtOpenEvent`. "Create" syscalls can optionally specify create-if-exists behaviour (akin to `O_CREAT`). The name is specified at creation time.

1.7 Handles

All Windows kernel objects can be accessed from multiple processes. They are exposed to user space via "handles", which are 30-bit[1] values roughly analogous to file descriptors. Like file descriptors, multiple handles can refer to the same object, and have different access flags, which track what operations can be legally performed on the object. Unlike file descriptors, however, these access flags are more fine-grained and partially depend on the object type.

[1]The data type for a handle is pointer sized. However, the lowest two bits are always zero (and masked out by kernel APIs which take handles).

Note also that although the data type for HANDLE is pointer-sized, Microsoft documentation suggests that handle values always fit within 32 bits even on 64-bit architectures, to allow for interoperability. Since 30 bits is quite a huge limit anyway, this is difficult to confirm and not particularly important either.

Like file descriptors, handles can be duplicated within a process or across processes. Like file descriptors, handles represent a user-space reference count on an object, which is independent of its kernel-space reference count, but handles themselves are not reference-counted.

Besides functions which can open an object by its name, the following syscalls are particularly important when optimizing synchronization primitives:

- `NtDuplicateObject`: This is similar to `dup(2)`, but much more generic. It allows a caller in process A to duplicate a handle (that is, create a new handle referring to the same object) from process B into process C, and all three processes can be arbitrarily the same or different. The new handle can be given different access rights than the original. There is also an optional flag allowing to close the source handle at the same time.

- `NtClose`: This function closes a handle in the current process, analogous to `close(2)`.

The access rights that a handle has are requested when the handle is created, either by a create or open syscall or by `NtDuplicateObject`. The ACL associated with an object is specified at creation time.

2 Implementation constraints

The constraints we need to account for fall broadly into three categories: performance, correctness, and robustness.

2.1 Performance constraints

Only some objects are known to be used in hot paths, in particular, semaphores and events.

It is likely that mutexes do as well, although this has not been confirmed. Mutexes tend to need to be "accelerated" in the same way regardless, though, for reasons that will be described later.

Only some operations are known to be used in hot paths. Specifically, the following functions are known:

- `NtWaitForSingleObject`

- `NtWaitForMultipleObjects`

- `NtSetEvent`
- `NtResetEvent`
- `NtReleaseSemaphore`
- `NtReleaseMutant`

The "query" operations are rarely used by applications, probably because they are undocumented and have no corresponding `kernel32` export. So too is the ability to atomically return the previous state of an object from operations that modify it—this part of the API is optional and is not used by documented `kernel32` exports. `NtPulseEvent` is also rarely used, probably because Microsoft has publicly discouraged its use (or rather, the use of the documented equivalent). In practice, however, because these operations are so closely tied to the state of the relevant objects, it has generally been necessary to accelerate them anyway.

Wait-all is believed to be a rare operation in general, or at least in hot paths. This is stated in an interview with Microsoft programmer Arun Kishan regarding the redesign of the NT scheduler and synchronization object implementation[2]. It's also rarely useful, if the number of uses in Wine code is any indication. Existing solutions have made wait-all "mostly" optimized—that is, not as fast as wait-any, but still far faster than RPC to `wineserver`—so it is not known how many programs, if any, actually need it to be fast.

Use of objects visible to multiple processes has not been known to be common in hot paths, and one would not believe it to be common either. Most hot paths which require synchronization are attempting to synchronize access to a common resource, which due to ease of design as well as speed of access usually is better suited to live within a single process. (For example, there are known costs that the memory manager must pay for `MAP_SHARED` pages, and messaging via pipes or sockets is clearly less efficient than messaging in memory.)

Here, too, existing solutions have made cross-process access as fast as in-process access, and so it is not known whether applications depend on it. However, of late many programs (in particular HTML engines) have increasingly used multiple processes for the sake of robustness and security,

[2] `https://www.youtube.com/watch?v=OAAiOEQhsKO&t=2556s`

including to the extent of rendering in a separate process. If cross-process synchronization is not a bottleneck now, it is not unlikely that it will be in the near future.

Alertable waits are believed to be relatively rare in hot paths, partly because alerts are usually associated with file I/O, and partly because at least some versions of ntsync have used relatively slow code paths for alertable waits (i.e. involving at least one `wineserver` call even when a thread is not alerted), without any known performance problems. While the performance of file I/O is certainly important to some programs, synchronization primitives have not yet been known to be the largest bottleneck in any such program.[3]

There is also a risk of making some paths *too* performant. Some optimizations have been known to expose application bugs, by making operations faster than those applications implicitly expect them to be. This is arguably a lesser concern: it is often easy to "fix" expectations within the same design, simply by adding artificial delays, and working around application bugs is not something that upstream Wine is usually interested in.

2.1.1 Specific bottlenecks

While many factors can affect performance, one of the most important bottlenecks (once all the overhead of RPC to a single-threaded server process was bypassed) ended up being syscall context switches. During the development of esync we saw consistent performance improvements from reducing the number of syscalls made. This remains true even where no lock contention was relevant; e.g. avoiding `clock_gettime(2)` helped improve performance. Accordingly much of the performance analysis has been focused on counting the number of syscalls in a given path, and trying to find ways to avoid them.

2.2 Correctness constraints

Most of the behaviour of individual APIs is described above. Some particular problems are worth pointing out, though:

1. Cross-process usage. Many solutions which would be easy to implement (in a performant, robust manner) within a single process are much harder when it becomes necessary to synchronize between multiple processes.

[3] It seems likely that there is much less demand for running Windows versions of such I/O-heavy programs under Wine; a common example of an I/O-heavy program a database manager, running on a server, which often has better native equivalents.

This is most of the reason why `wineserver` is involved, and `wineserver` is essentially responsible for the current performance problems.

As has been described above, objects can be given a name at creation time, which allows them to be opened from other processes, but they can also be arbitrarily opened using `NtDuplicateObject`. The latter means that any solution must be prepared for *any* object to be used from another process. Notably, this makes it much more difficult to accelerate objects which will always remain in-process, while letting others go through the current (or a slower) path.

Of course, there is nothing preventing an application from using a named object in a hot path, and as described above, Wine may likely see a need to accelerate cross-process objects anyway.

2. Wait-all. This is one of the things (although not the only thing) that stands in the way of "esync" and "fsync". It is difficult if at all possible to map directly onto an API which only supports wait-any. "esync" and "fsync" approximate it by acquiring objects non-atomically, and rolling back if unable to acquire all of them. No application is known to misbehave due to this difference, but it still prevents such solutions from being acceptable upstream.

3. `NtPulseEvent`. This operation is difficult to map directly onto other synchronization APIs; like wait-all it essentially requires direct support from the underlying wait-queue implementation.

   The documented equivalent (viz. `PulseEvent()`) has been publicly deprecated by Microsoft.[4] The reasoning given is somewhat unusual: due to the design of the Windows scheduler, a thread can be temporarily removed from its wait-queue while executing a high-priority kernel procedure, and miss a pulse.[5] In practice this does not occur with any observable frequency. Moreover, since the function is most commonly used via the `timeSetEvent()` API, i.e. in order to implement a timer, the occasional missed wakeup is not really a problem anyway. However, some synchronization primitive implementations (namely, "esync" and "fsync") miss wakeups more than occasionally, and in practice this has

[4] `https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-pulseevent`

[5] One cannot help but notice that this does not inhere to the documented behaviour, and that other operating systems handle this situation just fine, e.g. by allowing multiple wait-queues to be simultaneously used, or by not stealing arbitrary threads.

been known to break applications.

4. Synchronization APIs must always wake up eligible waiting threads. That is, the wakeup essentially must be atomic. This constraint is violated by "esync" and "fsync", which allow object state to be modified between e.g. `NtReleaseSemaphore`, and acquisition of the same semaphore by another thread. In practice one specific pattern has been known to cause problems, namely, `NtSetEvent` followed immediately by `NtResetEvent` from the same thread; at least one application has been known to depend on this always waking up another thread.

Not all existing synchronization optimizations follow all correctness constraints, and in some cases this ends up breaking applications. This is often not a concern for users (or forks of Wine), as long as one can revert to an implementation which behaves correctly, but it is not acceptable for upstream Wine. An upstreamable implementation will require that all applications work without application-specific code. Even guaranteed behaviour which applications are not known to depend on should be followed (after all, new or existing applications may be found to depend on it in the future).

2.3 Robustness constraints

As far as most users (and forks of Wine) are concerned, it is sufficient to avoid crashing or misbehaving. Much like the "correctness" class of constraints, this can even be limited to "for most applications", or "for applications which have synchronization as a bottleneck", as long as one can revert to an implementation which behaves correctly.

For upstream Wine this is not sufficient. We care about the following things, and any upstreamable solution will need to follow these constraints:

- The server (`wineserver`) cannot be allowed to misbehave on client input. It should never crash or deadlock, and should never perform any incorrect behaviour, regardless of what client processes do. Wine takes extra care to ensure this in general—e.g. by aggressively validating RPC input, even though the protocol is internal to Wine.

  There are a few notable consequences that fall out of this. For one, we cannot allow object state to be locked, and for that lock to be acquired by the server, if it is possible for a client to block the server indefinitely by holding the lock. Nor can we allow clients to store object state, and

for the server to read that same object state, if it is possible for the server to misbehave if that object state is corrupted.

- Client processes should not be able to cause each other to misbehave (except to the extent that voluntarily sharing objects with each other allows). In practice this ends up having similar effects to the above. It also means that one client process cannot be allowed to write the state of an object that it does not have (and cannot by its own powers gain) a handle to.

It is currently true that a sufficiently determined process can arbitrarily break other processes anyway, including `wineserver`, by making direct calls to debug utilities such as `ptrace(2)`. However, we'd like to leave the option open for processes to run with different Unix permissions, including potentially as different users.

3 Existing implementations

3.1 Server-side synchronization

The current implementation of synchronization primitives in Wine makes use of RPC to the `wineserver` process, which controls all cross-process operations, and ultimately most operations normally handled by the Windows kernel. For synchronization APIs each NT syscall maps to one server request.

For obvious reasons, this satisfies all of the correctness and robustness constraints. However, the overhead of IPC is significant. Making matters worse, the server is single-threaded, so that multiple simultaneous requests are not handled well. Hence the impetus to find a better solution.

3.2 esync

`eventfd`-based synchronization, referred to in the code and in common parlance as "esync", maps waitable objects to `eventfd(2)` file descriptions.

`eventfd` is trivially capable of emulating the basic functionality of semaphores (via `EFD_SEMAPHORE`), both auto-reset and manual-reset events, and waiting for a single object (via blocking `read(2)`, or, in the case of a manual-reset event, `poll(2)`). By relaxing the constraint that wakeups must be atomic, one can emulate wait-any via `poll(2)`. Signal-and-wait is also trivial enough to implement. Cross-process objects are also trivial, by duplicating file

descriptors with `SCM_RIGHTS`.

This alone correctly accelerates almost all of the operations which must be fast, as described above. However, it misses the following:

- respecting the maximum count on semaphores
- mutexes
- returning the previous state (atomically) from signal or designal operations
- reading the current state of an object (`NtQuery*`)
- wait-all
- `NtPulseEvent`

In order to implement mutexes, esync needs to store extra volatile state to determine the current owner. Somewhat relatedly, it needs to be able to read the current state of semaphores, which is not possible for an `eventfd` object. Because of the cross-process nature of objects, esync uses shared memory mappings to store and retrieve this information. In order to satisfy atomicity constraints, the state in shared memory is actually treated as "canonical" to some degree (as opposed to the internal state of the `eventfd` object).

As has been described above, wait-all is implemented by ignoring the atomicity requirements. A waiting thread polls on each object, one at a time, while not actually consuming their state until all have been signaled. It then attempts to consume all objects in a tight loop, rolling back if any object became unsignaled in the meantime. This is not strictly conformant, but has not yet been known to cause problems.

Atomicity of wakeups is also violated, as has been described above. This *is* known to cause problems with real applications.

`NtPulseEvent` is implemented by setting and then immediately resetting the event. This causes problems mostly due to missed wakeups (due to, again, lack of atomicity). There are other respects in which this is not conformant (e.g. another thread which is not sleeping can observe the event as being signaled), but this has not been known to cause problems, and is not particularly likely to either.

The end result is that most of the important NT syscalls are implemented using one host syscall, and no RPC. Wait operations usually require two syscalls (specifically, `poll`, and `read` to consume the state for objects other than manual-reset events). This is fast enough to show huge performance benefits in many applications.

Because object state is stored in shared memory, it is not only possible but easy for an application to modify the state of objects in ways not provided for by the API. This is not a problem in practice. Applications which overwrite internal data generally run into problems before then, and there usually aren't other important applications in the prefix that can be corrupted anyway. And, of course, malicious applications abusing out-of-tree Wine patches is all but unheard of. Nevertheless, it is a significant barrier to upstreaming, especially considering the extent to which esync relies on shared memory.

Although esync was designed around `eventfd` (which is a Linux-specific API), it is possible to emulate `eventfd` with named pipes, and some versions of CrossOver do ship with some custom patches for this, in order to improve performance on other platforms. This is done by mapping the count of an `eventfd` to the number of bytes currently stored in the pipe.

3.3 fsync

futex-based synchronization, referred to in the code and in common parlance as "fsync"[6], maps waitable objects to futexes.

These futexes are, naturally, stored in shared memory, so as to be accessible from multiple processes. For mutexes it is necessary to store extra data (namely, they need to store both the owner thread ID and the recursion count, which together comprise 8 bytes).

The most important thing that this requires is the ability to wait on multiple futexes at once, i.e. an equivalent of `poll(2)`. This was done via a custom kernel patch introducing a new futex operation. The ABI for this has changed over the years as attempts were made to upstream this kernel patch; certain versions of the ABI have been known as `FUTEX_WAIT_MULTIPLE`, `futex2`, and `futex_waitv`. The latter syscall did eventually make it into upstream Linux release 5.16.

[6]The decision to give this patch set a name identical to a completely unrelated POSIX API was a regrettable one.

Most other simple operations proceed intuitively. Operations which change state are implemented by simply atomically writing the state (usually via atomic read-modify-write instructions) and then, depending on the operation, performing a wake operation on the futex.

Wait-all and pulse are implemented almost exactly as they are in esync, and the same characteristic problems apply here. The problems associated with the lack of atomicity of wakeups, and of the use of shared memory, affect fsync as well.

fsync was commissioned by Valve for its Proton software, on the theory that it would offer better performance than esync, due to its ability to avoid some host syscalls. In particular, unlike esync, `NtResetEvent` can be implemented without any syscalls. Wait operations can also be implemented without any syscalls if the object is uncontended, and consuming object state does not require a syscall either.

These performance benefits did pan out, and were measureable, for at least one piece of software. However, the number of games showing an improvement in performance from fsync (as compared to esync) is very low, and remains so to this day (in fact, there are only three known applications right now: Final Fantasy XV, Beat Saber, and Greedfall).

fsync does have some drawbacks. Spurious wakeups are potentially more common, meaning that more syscalls may need to be made on contended objects. fsync has been known under some conditions to cause stuttering, or less consistent performance, than esync, possibly for this reason. It has also been known to expose multiple application bugs by making operations (especially wait operations) too fast, and recent versions of Proton have in fact introduced artificial slowdown into wait operations to protect from this.

3.4 Attempts to introduce a new kernel interface

There are current plans, and even functional patch sets, to introduce a new interface into the Linux kernel, which would allow all of the exact semantics of wait operations to be followed exactly, while making exactly one (fast) syscall per NT syscall. These patch sets are still very much a work in progress (in particular, the design of the API surface is not fully settled), and have not been submitted upstream to the Linux or Wine projects, but have been variously referred to as “ntsync”, “fastsync”, and “winesync”.

The ability to have fine control over the wait-queue implementation allows all of the correctness constraints to be followed exactly, and by relegating control over all objects entirely to the kernel, ntsync can satisfy robustness constraints as well.

The question of performance is more interesting. Since each NT syscall is mapped to one host syscall, one would expect performance within measuring error from Windows in all cases.

esync, for comparison, makes one syscall for most important operations, but may make multiple syscalls for wait operations. Hence one would expect ntsync to perform at least as well as esync, and preliminary results bear this out.

fsync makes one syscall for wake operations, no syscalls for `NtResetEvent`, and zero or more syscalls for wait operations. As such one cannot conclude simply from guessing whether its performance would be better or worse than ntsync. Quantitative results, for the few known games where esync and fsync differ, are lacking, but at least one test, with Final Fantasy XV, shows that fastsync performs about as well as esync, and hence not as well as fsync. It is not yet known whether fastsync will suffer from any of the other performance-related problems associated with fsync.

One obvious disadvantage of this approach is that it requires changes to the kernel (and hence also will require a new kernel, even if accepted). As such it will also need to be adapted for any other kernels, such as BSD or Mach, and may not be deemed acceptable to any given kernel.

4 Other ideas and proposals

Some of these ideas are partial solutions, or solutions to specific problems that arise with e.g. esync. Most of them have some flaw or another. Other solutions do not have obvious flaws, but become sufficiently complex that it is difficult to tell whether they have non-obvious flaws, and ultimately approach unmaintainability.

I, personally, have come to put enough stock in fastsync (and have spent enough time fruitlessly considering other approaches) that I do not believe it particularly worthwhile to explore any other ideas.

4.1 Acceleration of in-process objects

Many difficulties result from the cross-process nature of synchronization primitives. In particular, the ability to track and modify object state from within a client process leads to robustness problems already described for esync and fsync.

One idea which has been floated for quite a while, would be to accelerate objects which can be guaranteed to live in a single process. Any object created without a name (and in practice most are) would in theory fall into this category. Unfortunately, the existence of `NtDuplicateObject` means this is impossible.

4.2 Migration

An extension of the previous idea, which has also been floated for quite a while, is to create all objects as initially accelerated, with in-process state, and then to convert them back to server-managed objects once they are accessed from another process.

This is a lot more difficult than it sounds. The main problem is avoiding missed wakeups while migrating a thread which is currently in a wait operation. In order to completely avoid that Wine would need to manage the wait-queues in user space, and these wait-queues would need to be cross-process.

This also means that cross-process objects will not be accelerated, and as described above, it may be or become necessary to accelerate them, which would leave us right back where we started.

Another problem with migration is the fact that one cannot simultaneously wait on client-managed and server-managed objects. Hence migration of one object can cascade and cause other objects to be migrated. Ensuring correct synchronization here is very difficult.

4.3 Object isolation

To some degree we can avoid letting process corrupt other processes' object state by ensuring that a process does not map any shared memory containing objects it does not have access to.

This can be simply done by allocating one page of memory to each object, but

this doesn't scale. Address space is often at a premium for 32-bit processes; memory usage can be at a premium for any process (and all of this memory would need to be mapped); and some applications leak hundreds of thousands of synchronization primitives. By contrast, the volatile state encompassed by any one object never comprises more than eight bytes.

A better approach is to carefully arrange objects in shared memory pages such that a process only maps pages containing objects that it can access. There is no show-stopping problem with this solution, although it doesn't offer as complete restrictions as a kernel or RPC-based approach. It is, however, only a very partial solution; while one can isolate objects this way, problems with wait-queues are not addressed, and wait-queues cannot be so easily isolated (from other processes, or from the server).

4.4 In-process wait-queues

Many of the problems with esync and fsync result from their inability to directly map some esoteric NT operations onto lower-level operations. Indeed, much of the above language includes phrases like "direct support from the underlying wait-queue implementation".

There is no inherent, or at least immediate, reason the wait-queue implementation has to live in the kernel, though. After all, the server itself has its own wait-queue implementation. Hence the question becomes: can management of that queue be moved to individual processes?

Here we essentially run up against robustness constraints, and these are hard if not impossible to solve. In particular, it is impossible to implement proper wait-queues without locks (after all, as has been described, we really do need many multi-step operations, such as wait-all, to be executed atomically), and because some objects need to be modified from the server as well, this means that the server can end up taking locks that clients can take as well. This means that the server can be broken by misbehaving clients, including by clients which just happen to crash while holding a lock, which is unacceptable for upstream Wine. (Robust mutexes would potentially help, but not enough to be acceptable upstream.)

4.5 Extending an existing kernel interface

This is essentially a variation of ntsync, or more generally the idea of introducing a new kernel interface, worth mentioning for completeness.

The obvious candidate for extension is `eventfd`. Some things could be relatively easily added to it, such as operations to query the current state, or atomically modify the state while retrieving the previous state. Mutexes would be more complex, but still feasible.

Harder are atomic wakeups, and wait-all. These do not fit well with the POSIX model. One would need to extend (or introduce new APIs similar to) `poll(2)` itself, as well as fundamentally modifying `struct file_operations`, which would affect *all* file descriptors, and would be of questionable utility outside of Wine. Wait-all is not clearly a good design for an API, and wait-any already has many existing and arguably superior alternatives on Linux.

Alternatively, one could design APIs that only affect `eventfd`, but at this point one is 90% of the way towards designing a new API, and the parts of `eventfd` that are still associated to traditional file I/O would be more of a burden than a help, and might potentially cause performance overhead as well.

Extending `futex(2)` is not viable. The very model for futexes does not allow wakeups to be atomic.